Diss. ETH No. 19344

Towards Many-Core Real-Time Embedded Systems: Software Design of Streaming Systems at System Level

A dissertation submitted to the

ETH ZURICH

for the degree of

Doctor of Sciences

presented by

KAI HUANG

M.S. Computer Science
Leiden University, the Netherland

born 02.10.1977
citizen of China

accepted on the recommendation of

Prof. Dr. Lothar Thiele, examiner
Prof. Dr. Peter Marwedel, co-examiner

2010

Institut für Technische Informatik und Kommunikationsnetze
Computer Engineering and Networks Laboratory

TIK-SCHRIFTENREIHE NR. 119

Kai Huang

Towards Many-Core Real-Time Embedded Systems: Software Design of Streaming Systems at System Level

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

A dissertation submitted to the
Swiss Federal Institute of Technology (ETH) Zürich
for the degree of Doctor of Sciences

Diss. ETH No. 19344

Prof. Dr. Lothar Thiele, examiner
Prof. Dr. Peter Marwedel, co-examiner

Examination date: 12. November, 2010

Abstract

Nowadays, multi-core architectures become popular for embedded systems. As VLSI technology is scaling to deep sub-micron domain, an envisioned trend is that the architectures of embedded systems are moving from multiple cores to many cores. Although state-of-art multi-core and future many-core architectures provide enormous potential, scaling the number of computing cores does not directly translate into high performance and power efficiency. To exploit the potential of a multi(many)-core platform under stringent time-to-market constraints, software development does not only need to tackle the still-valid classical requirements, e.g., memory constraints, programming heterogeneity, and real-time responsiveness, but also face new challenges stemming from the increasing number of computing cores, for instance, scalability of the technologies.

In this thesis, we focus on the class of streaming embedded systems at system level and address three important aspects of the software construction of multi/many-core embedded systems, i. e. , programming, performance, and power. To address the programmability of multi/many-core embedded systems, we present a model-of-computation based programming model which supports scalable specifications of a system in a parametrized manner. In terms of performance estimation, we present both analytic and simulation-based techniques to tackle the complex interference and correlations within multi/many-core embedded systems such that accurate estimation can be conducted. We also investigate power-efficient design and propose offline and online algorithms for dynamic power management to reduce the static power consumption under hard real-time constraints.

Zusammenfassung

Mehrkernprozessoren werden bereits heute für eingebettete Systeme verwendet. Durch die weitere Miniaturisierung im Submikrometer Bereich ist zu erwarten, dass zukünftige Architekturen für eingebettete Systeme auf Vielkernprozessoren anstatt Mehrkernprozessoren basieren werden. Obwohl Mehrkernprozessoren und zukünftige Vielkernprozessoren enorme Möglichkeiten bieten, führt die steigende Anzahl von Rechenkernen nicht direkt zu höherer Rechenleistung oder Energieeffizienz. Um die Möglichkeiten dieser Architekturen bei kurzen Produktentwicklungszeiten ausschöpfen zu können, müssen daher bei der Softwareentwicklung sowohl bereits bekannte als auch neue Aspekte berücksichtigt werden. Während beispielsweise Speichereinschränkungen, Heterogenität bei der Programmierung oder Echtzeitanforderungen weiterhin eine wichtige Rolle spielen, müssen zusätzlich die Skalierbarkeit bezüglich der Anzahl der Prozessorkerne oder technologische Einschränkungen bei der Softwareentwicklung beachtet werden.

In der vorliegenden Dissertation liegt der Schwerpunkt auf eingebetteten Systemen für Streaming Anwendungen, wobei drei Aspekte der Softwareentwicklung näher betrachtet werden: Programmierung, Analyse des Zeitverhaltens und Leistungsaufnahme. Zur Programmierung von eingebetteten Systemen mit Mehr-/Vielkernprozessoren wird ein Programmiermodell vorgestellt, das die skalierbare Spezifikation eines Systems in parametrisierter Form ermöglicht. Zur Analyse des Zeitverhaltens werden analytische und simulationsbasierte Verfahren vorgestellt, welche zur Erzielung genauer Resultate die komplizierten zeitlichen Interferenzen und Korrelationen in Mehr-/Vielkernprozessoren berücksichtigen. Schliesslich wird der Entwurf von Systemen mit effizienter Leistungsaufnahme betrachtet, wobei online und offline Algorithmen vorgestellt werden, um den statischen Leistungsverbrauch unter harten Echtzeitbedingungen zu reduzieren.

Acknowledgement

First of all, I would like to express my sincere gratitude to Prof. Dr. Lothar Thiele for offering the opportunity for a PhD and constantly patiently supervising my research. Without his support, this thesis would have not been possible.

I would like to thank Prof. Dr. Peter Marwedel for being my co-examiner in this thesis.

I would also like to thank: Prof. Dr. Jian-jia Chen and Prof. Dr. Giorgio C. Buttazzo for the fruitful research cooperation; Dr. Alexander Maxiaguine and Dr. Simon Künzli for their valuable suggestions and help in the beginning of my PhD life; Dr. Wolfgang Haid for the nice collaboration in the four-year SHAPES project and for taking his time of proofreading my thesis, and Dr. Iuliana Bacivarov and Luca Santinelli for nice research cooperation. Furthermore, I would like to thank all my former and current colleagues of the whole TEC group for their company and support, especially Dr. Clemens Moser for the great time we had while sharing our office for more than four years.

Finally, my dearest thanks go to my family for their love and support throughout all these years of my PhD study.

The work presented in this thesis was supported by the European Integrated Project SHAPES (grant no. 26825) under IST FET – Advanced Computing Architecture (ACA). This support is gratefully acknowledged.

To my wife, Yu, and
to my daughter, Wei-yi.

Contents

1

Introduction

Multi-core architectures are widely used for embedded systems nowadays. The number of computing cores is keeping on increasing as VLSI technology is scaling to deep sub-micron domain. An envisioned trend is that embedded systems are moving from multiple cores to many cores. This thesis presents a set of novel techniques for contemporary multi-core and future many-core embedded system design. In particular, we focus on solutions for the new challenges imposed by CMOS technology scaling. Section 1.1 and 1.2 survey the state-of-art platforms and software tool flows for multi-core embedded systems, respectively. Section 1.3 draws the outline and summarizes the contributions of this thesis.

1.1 Multi-Core Embedded Systems

Modern embedded systems require massive computational power due to computationally intensive embedded applications, e.g., real-time speech recognition, video conferencing, software-defined radio, and cryptography. An embedded system running all these applications demands a total performance payload of up to 10,000 SPECInt benchmark units [ABM+04]. The situation will become even worse since the demand for computation will further grow. Future embedded applications like embedded computer vision [KBC09], for instance, require computational power far beyond what can be provided by state-of-art embedded system architectures [WJM08].

Besides the computational demand, power consumption is another

first-class design concern for embedded systems. A major category of embedded systems, for instance, are hand-held mobile devices which are powered by batteries. The batteries for such devices are limited in both power and energy output. The amount of energy available thus severely limits a system's lifespan. Although research continues to develop batteries with higher energy-density, the slow growth of the energy density of batteries lags far behind the tremendous increase of demands [ITR].

To cope with the ever increasing demand of computation and stringent power constraints of modern embedded systems, multi-core architectures become the de facto choice. These are two driving forces to use multi-core architectures: On the one hand, CMOS circuits advance to the deep sub-micro domain, which will aid sustaining Moore's law in the future – the number of transistors within a chip doubles every 18 months. A recent evidence, for instance, is the newly unveiled Intel Single-chip Cloud Computer (SCC) [SCC] which integrates 1.3 billion transistors within 567 square millimeters manufactured using a 45 nm CMOS High-K metal gate process. On the other hand, increasing clock frequencies and deeper pipelines cannot sustain the performance increase under constraints of power consumption and thermal dissipation, i.e., hitting the power wall [PH09]. Experience suggests that performance only doubles with quadrupled complexity of a chip [Bor07].

The main advantage of multi-core architectures is that raw performance increases can be accomplished by increasing the number of computing cores rather than frequency, which translates into a slower growth in power consumption. For a given processor architecture in a given technology, under-clocking with lower supply voltage decreases power consumption much more than performance. On the other hand, for the same power consumption, a dual-core solution clocked at a 20 % lower frequency would bring in theory 73 % more performance than a single core [CCBB09]. Therefore, multi-core architectures embody a good trade-off between technology scaling and strict power budget requirements. Consequently, ITRS predicts that by the end of the decade consumer portable Systems-on-Chip (SoC) will contain more than 1,400 cores, as depicted in Fig. 1. Multi-core embedded systems are inevitably evolving to many-core embedded systems. In the rest of this section, we survey a few prominent cutting-edge multi-core embedded system platforms, some of which actually have many cores.

Multi-core architectures have a longer history in embedded systems than in desktop commercial products because embedded systems hit the power wall earlier [WJM08]. One of the first commercial multi-core platforms, the Lucent Daytona [AAB+00], for instance, was manufactured a decade ago. Tab. 1 lists six state-of-art multi/many-core platforms

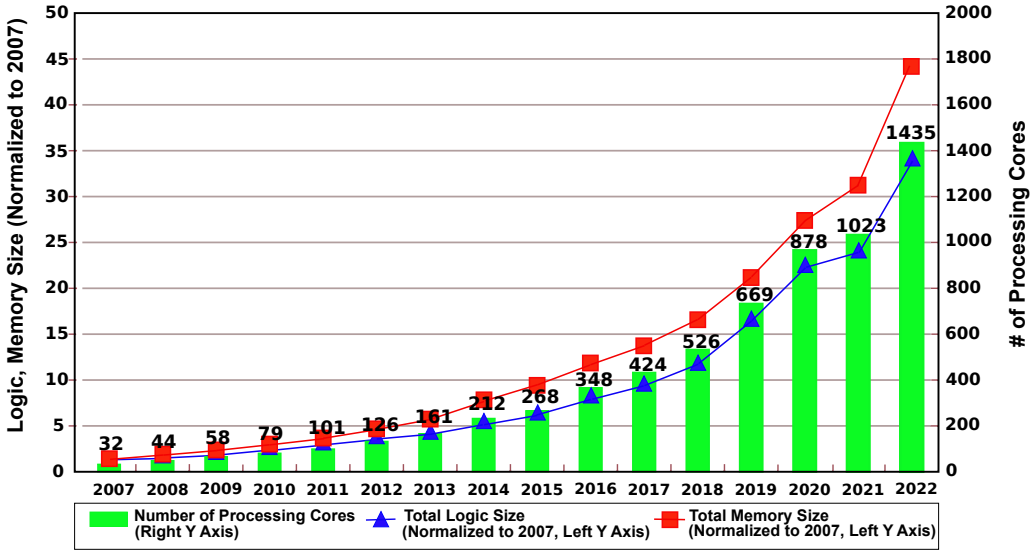


Fig. 1: ITRS 2007 [ITR] for SOC consumer portable design complexity trends.

from different vendors, the applications of which range from high-end embedded systems to portable mobile devices. The IBM Cell Broadband Engine [GHF+06], delivered at the end of 2006, implements a heterogeneous architecture with eight specialized data processing engines (SPU) for VLIW computing and a PowerPC engine (PPE) for coordination. The first major commercial application of the IBM Cell is in Sony's PlayStation 3 game console. The newly announced Intel SCC [SCC] is an experimental chip which will be delivered in 2010 for academic researches. It integrates 48 identical IA-32 cores equipped with dynamic voltage and frequency scaling, attaining power consumptions from 125 down to 25 watts. Almost the same time, TILERA claimed the world's first 100-core general-processing cores, namely TILE-Gx100 [TIL]. It pulls 55 watts at peak performance at a clock frequency of 1.5 *GHz*. This category of platforms is suitable for high-end embedded systems.

The last three rows of Tab. 1 depict another category of platforms which are suitable for mobile embedded systems. The PC205 [pic] of picoChip integrates 248 identical VLIW DSP cores running at 160 *MHz* to attain low power consumption. In contrast, the TI OMAP 4440 [TI-] is designed in a fully heterogeneous manner, being comprised of a dual-core ARM Cortex-A9, a programmable IVA 3 hardware accelerator, an image signal processor (ISP), and an SCX540 GPU. With these highly specialized cores, the OMAP 4440 can run at 1 *GHz* while its power consumption remains at only one watt. The ATMEL ShapOtto [HPB+] makes a compromise between homogeneity and heterogeneity. A tile-based architecture is chosen where each tile identically consists of an

vendor	number of cores	ISA	max. power	max. frequency
IBM CELL	9	PPE,SPU	100 W	3.2 GHz
Intel SCC	48	IA-32	125 W	3 GHz
TILERA TILE-Gx100	100	RISC-based	55 W	1.5 GHz
picoChip PC205	249	ARM,DSP	3 W	160 MHz
TI OMAP 4440	5	ARM, DSP, GPU	1 W	1 GHz
ATMEL ShapOtto	16	ARM,DSP	≤4 W	250 MHz

Tab. 1: State-of-art multi/many-core platforms.

ARM core, a VLIW DSP, and a network processor. Each tile requires a dynamic power which ranges between 360 and 460 miliwatts, while the static (leakage) consumption ranges between 8 and 23 miliwatts. The current fabrication intends to integrate eight tiles while the technology can be scaled beyond.

There are more commercial multi/many-core platforms available in the market. For extensive surveys, we refer to [BDM09, KAG+09].

1.2 System Software Design

Although state-of-art multi-core and future many-core architectures provide enormous potential, scaling the number of computing cores does not directly translate into high performance and power efficiency. To exploit the potential of a multi(many)-core platform under stringent time-to-market constraints, software development plays an essential role. Software development does not only need to tackle the still-valid classical requirements, e.g., memory constraints, programming heterogeneity, and real-time responsiveness, but also new challenges stemming from the increasing number of computing cores. We identify a few new challenges as follows:

- Due to the increased number of programmable cores, exploiting the massive parallelism offered by the abundant computing resources becomes challenging. On the one hand, it is difficult to expose and abstract the available parallelism of a multi-core platform to application programmers due to the intrinsic heterogeneous architecture of the platform. On the other hand, modern embedded applications are far more complex than their single-threaded ancestors. The reference implementation for the H.264 codec, for

instance, consists of over 120,000 lines of C code. Parallelizing it is a tedious and error-prone process. The questions are:

How to efficiently program an multi/many-core platform with respect to both performance and time-to-market? Can such a programming model scale to future many-core platforms?

- The increased number of computing cores would lead to average-case performance gains, whereas the worst-case performance might decrease because of complex interactions within a system. Furthermore, even the average case is difficult to measure because of the intrinsic heterogeneity of modern multi-core platforms. The traditional cycle/instruction-accurate simulation techniques are far too slow. The question here is:

  How to estimate the performance of large multi/many-core embedded systems with reasonable time and accuracy?

- The tremendous amount of integrated transistors within a chip incurs a new power-efficiency problem, i.e., static power consumption caused by leakage current. The leakage current originates in the dramatic increase in both sub-threshold current and gate-oxide leakage current, which is projected to account for as much as 50 percent of the total power dissipation for high-end processors in 90-nm technology [ABM+04]. The ITRS expects the static power will be much greater than its predictions due to variability and temperature effects [ITR]. The question hereby is:

  How to effectively reduce the static power under real-time constraints?

This thesis aims to give partial answers to these new challenges imposed by the steadily increasing number of programming cores of modern multi-core and future many-core embedded systems. Before presenting our contributions, we review the state-of-art related work in the literature. Due to the vast amount of related work, only a representative subset is discussed with an emphasis on multi-core embedded systems. For extensive surveys, we refer to [MJU+09, HHBT09a, wPOH09, KB09].

To the first category belong compiler-based approaches where a conventional sequential language like `C` is used as an initial specification for applications and the compiler automatically extracts parallelism from the sequential representation. Example frameworks are MAPS [CCS+08], Compaan [SZT+04], and CriticalBlue multi-core Cascade [Cri]. This approach is appreciated by application programmers because the complicated parallelism extraction is transparent and does not impose any

burden on the programmers. Automated parallelization, however, is challenging and effective parallelism extraction is rather cumbersome without domain-specific knowledge. Therefore, compiler-based approaches are normally limited to applications with evident data-parallel regions or with relatively static behavior.

To ease the job of compilers, one widely adopted approach is to use an explicit application programming interface (API) during the application development. Using APIs, application programmers can make use of their domain-specific knowledge, e.g., identifying the parallel regions, and give hints to the compiler for better parallelization. Well-known APIs are OpenMP [Ope] for shared-memory architectures, MPI [MPI] for distributed-memory architectures, and TTL [vdWdKH+04] for high-level abstraction. Industrial chip vendors adopt this approach widely within their software development kits. CUDA [BFH+04, ati] from NVIDIA and Brook+ [CUD] from AMD ATI are parallel programming solutions for GPUs, which provide specific APIs to manipulate the GPU memory and to express single-instruction multiple-data (SIMD) data parallelism. The API-based approach allows better low-level control of the parallelism by the programmer and, therefore, promises better performance if the APIs are properly used. The major drawback of compiler/API-based approaches is that an application can be written in an arbitrary form, consequently complicating quantitative analysis.

Model-driven development is recently advocated as a viable alternative. By restricting an application to a certain model of computation, quantitative analysis with respect to e.g. schedulability tests and worst-case behavior can be tackled in a reasonable manner. A well-known model-of-computation is the Kahn process network [Kah74] where an application is composed of autonomous processes which communicate asynchronously via point-to-point FIFO channels. Design flows based on KPN are e.g. Koski [KKO+06], DOL [TBHH07], SESAME/DAEDALUS [PEP06, NSD08]. Another widely used model-of-computation is the synchronous data flow (SDF) [LM87] which is a restricted version of KPN. The SDF model enables static scheduling analysis at compile time, which provides guarantees on, for instance, finite size of communication buffers and deadlock-free execution. Tool flows adopting the SDF are, for instance, SHIM [ET06], PEACE [HKL+07], and StreamIt [TKA02]. There are other frameworks supporting multiple models-of-computation simultaneously. For instance, Ptolemy [pto] supports heterogeneous modeling and Metropolis [BWH+03] defines a meta-model that can be refined into a specific model-of-computation. To compare different models-of-computation, E. Lee and A. S. Vincentelli present a framework [LSV98]. An overview of applying models-of-computation in system design is presented in [ELLSV97].

Although a vast amount of research effort has been devoted into this field, the aforementioned challenges are not completely solved. The developed techniques are not well suited for future many-core embedded systems, which motivates our work. In this thesis, we try to tackle these challenges for both contemporary multi-core and future many-core embedded systems, in particular for a specific application domain.

1.3 Thesis Outline and Contributions

A well-known methodology to guide embedded system design is the orthogonalization of concerns [KNRSV00] which proposes to separate a design process into independent but less complex parts. A corresponding flow to design an embedded system is depicted in the shadowed part of Fig. 2, now commonly referred to as the Y-chart paradigm [KDVvdW97]. One key idea underlying this paradigm is to explicitly separate application and architecture specifications and to use a mapping step to specify how the application is spatially (binding) and temporally (scheduling) executed on the architecture. During the design process of an embedded system, the application, architecture, and mapping are iteratively refined according to evaluation results at different design stages.

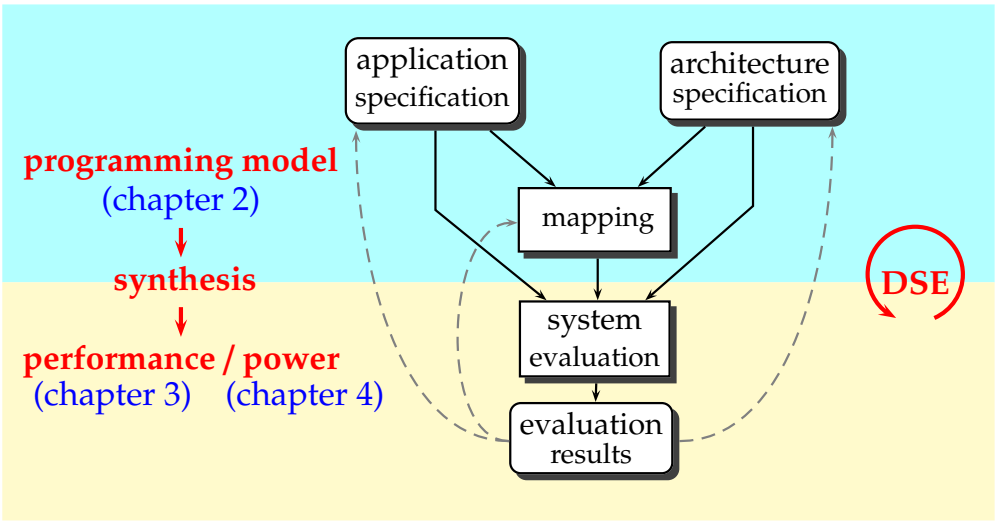


Fig. 2: Y-chart paradigm revisited.

Revisiting the Y-chart paradigm, the software design for a given embedded platform can be embodied by a few concrete procedures, depicted in the bold text of Fig. 2. First, a programming model is needed as an interface for designers to define a system, i.e., specifying the application, architecture, and mapping. The programming model

defines which model-of-computation to use and the programming syntax of the specifications. Second, to evaluate a candidate design, metrics and the corresponding evaluation model need to be defined. To analytically evaluate the performance of a system, for instance, a formal performance model for the system is needed. In addition, the programming model and the evaluation model are normally defined at different levels of abstraction. An automated synthesis is preferable to refine a system from one level of abstraction to the next in a correct manner in terms of functional and possibly non-functional properties. Last but not least, a design space exploration (DSE) procedure is also desirable in order to find the optimal design in a systematic and automated manner.

None of the procedures in the Y-chart paradigm is trivial. In this thesis, we target the domain of streaming applications and provide a set of novel techniques for the software construction for contemporary multi-core and future many-core embedded systems, trying to tackle the aforementioned design challenges. We demonstrate on the one hand how the aforementioned challenges can be tackled in a systematic manner, on the other hand the proposed solutions can be smoothly unified in a same software framework for the assessment of system-level design decisions. Specifically, we focus on the programming model (chapter 2) and two different metrics, i.e., performance (chapter 3) and energy (chapter 4), as depicted in Fig. 2. For automated software synthesis and design space exploration, we refer to [HKH+09, Hai10] and [Gri04, Kü06], respectively. The major contributions of the thesis are summarized as follows:

Chapter 2: Programming Model

In Chapter 2, we present a programming model based on KPN. Our programming model separates the application, architecture, and mapping specifications of a system. For application modeling, the KPN model-of-computation is strictly adhered. To avoid the costly communication and synchronization overheads incurred by large scale process networks, the FIFO syntax of a KPN is extended by a so-called windowed FIFO. We also develop a distributed functional simulation as a proof-of-concept runtime environment for our programming model. The detailed contributions are listed in the following:

- Following the orthogonalization-of-concerns methodology, we construct the syntax of our programming model. We define XML schemata for the system specifications, i.e., the structure of the application, the abstract architecture, and the mapping. To assist the design of large systems, a so-called iterator is developed by which a system can be arbitrarily scaled in a parametrized manner. We also define a set of C/C++ coding rules for the specification of

the functionality of individual processes in an application process network, To reduce the workloads of application programmers, Our coding rules enable the reuse of a same piece of source code for a set of iterated processes.

- We propose a syntactical extension for the FIFO communication of KPN, namely windowed-FIFO. We prove the syntactic coherency of a windowed-FIFO process network to a standard KPN. To validate this concept, we develop a hardware implementation as well as the application software interface based on Xilinx FPGAs.

- To show the effectiveness of our programming model, we develop a SystemC-based functional simulation as a runtime environment. To parallelize SystemC execution, we develop a library which enables a concurrent execution of multiple SystemC kernels. Using this library, a functional simulation of an application can execute on an arbitrary number of Linux hosts connected via TCP/IP. Furthermore, the source code for the runtime environment can be automatically generated from system specifications using our programming model.

Chapter 3: Performance Estimation

In Chapter 3, we consider the performance metric in the Y-chart paradigm. We investigate two techniques, i.e., an analytic method and a simulation-based approach, for worst and average case performance evaluation of an sysetm at system level. For the formal method, we apply real-time calculus [TCN00, CKT03] and develop new techniques for the analysis for data stream correlations, which is typical for KPN applications. For the simulation-based approach, we develop a trace-based framework which serves as a non-functional back-end of our programming model. The proposed framework can estimate the performance of large streaming multi/many-core systems within a reasonable time span with high accuracy. Specifically, the detailed contributions are listed below:

- We investigate correlations of data streams within a fork-join scenario in a KPN network and present a method to analyze such correlations based on different types of delays, e.g., splitting delay at the fork process and blocking delay at the join process. We show the applicability of the presented methods by analyzing a concrete multimedia application.

- We propose a trace-based framework to simulate timing behavior of multi/many-core embedded systems specified in aforementioned

programming model. By abstracting an application as coarse-grain traces, our framework can effectively and efficiently simulate complex systems, while considering different aspects pertaining to resource sharing, memory allocations, and multi-hop communications. We validate our framework by mapping an MPEG-2 decoding algorithm onto the ATMEL Diopsis-940 platform [Pao06]. We also demonstrate the scalability of our approach by simulating a scaled version of the MPEG-2 algorithm on 16-core platform.

Chapter 4: Power Management

Chapter 4 explores system-level dynamic power management to reduce static power consumption under real-time constraints. The KPN modeling of an application enables such an exploration at the same system level as to the performance metric. For simplicity, we consider a dual-core scenario, i.e., a processing core for data stream processing and a control core for coordination, for instance, to schedule the processing core. We propose both online and offline algorithms. To guarantee real-time requirements, We apply real-time calculus to predict future event arrivals and Real-Time Interface theory [TWS06] for the schedulability analysis. Based on the adopted worst-case interval-based abstraction, our algorithms cannot only tackle arbitrary event arrivals (even with burstiness) but also guarantee hard real-time constraints with respect to both timing and backlog constraints. The contributions of this chapter are as follows:

- We formulate the problem of finding a periodic power management scheme to minimize the average standby power consumption under hard real-time constraints. We provide optimal and approximated offline solutions for this problem. The light run-time overhead of the periodic power management scheme is particularly suitable for embedded systems with very limited power budgets.

- Alternatively, we propose online algorithms to reduce standby power consumption. Our online algorithms adaptively predict the next mode-switch moment by considering both historical and future event arrivals, and procrastinate the buffered and future events as late as possible.

- To handle multiple event streams with different characteristics, we develop solutions for two preemptive scheduling policies, i.e., earliest-deadline-first and fixed priority, for resource sharing. With respect to system backlog organization, two different scenarios, i.e., distributed and global backlog, are considered.

2

Programming Model

The multi-core architectures of modern embedded platforms are highly concurrent and software-programmable. To exploit the available concurrency and programmability, a multi-core platform has to be abstracted at a level where application programmers can efficiently develop software without digging deep into intricate hardware details. A programming model is such an abstract representation. It defines an interface by which application programmers specify an application and describe how the application runs on top of a platform. Reviewing Fig. 2, a programming model is the starting point of the software development of an embedded system. Besides the specification of a system, a programming model also affects other procedures of the software construction, for instance, the way how system performance is evaluated and how design space exploration is conducted.

The highly concurrent and software-programmable nature of multi-core platforms does not directly translate into high performance and power efficiency of a system. On the one hand, due to the heterogeneous architecture of multi-core platforms, it is difficult to expose and abstract the available parallelism in a multi-core platform in a uniform and scalable manner. On the other hand, an application needs to be parallelized into coarse-grained concurrent tasks to make use of the computing cores. However, modern embedded applications are far more complex than their single-threaded ancestors. The reference implementation for the H.264 video codec, for instance, consists of over $120,000$ lines of C code. Parallelizing an application with such a complexity is a tedious and error-prone process. Furthermore, the performance gains obtained through

this coarse-grained parallelization could be overshadowed by the cost of communication and synchronization overhead among the partitioned concurrent tasks. This effect will be accentuated as more computational cores are integrated and finer-granularity tasks are partitioned.

The questions to answer are: How can a multi-core platform be efficiently programmed with respect to both performance and time-to-market? Can such a programming model scale to future many-core platforms?

2.1 Overview

In this chapter, we present a model-of-computation based programming model, targeting stream-oriented multi/many-core embedded systems. Our programming model separates the application and architecture of a system. The application here refers to the function of the system, i.e., what the system is supposed to do. The architecture, in contrast, defines hardware resources, i.e., how the system conducts the function. To model an application, we adopt the Kahn process network (KPN) [Kah74] model of computation. To model an architecture, we define a non-functional abstraction. This non-functional abstraction describes only which hardware components the platform consists and how these hardware components are interconnected. We also define a mapping to describe how an application runs on an architecture. For specifying the application, architecture, and mapping descriptions, we design a hybrid programming syntax, i.e., `C/C++/XML`. To avoid the costly communication and synchronization overheads incurred by large scale process networks, we propose a variation of the FIFO syntax of KPN, namely windowed FIFO. We also develop a distributed functional simulation as a proof-of-concept runtime environment for our programming model.

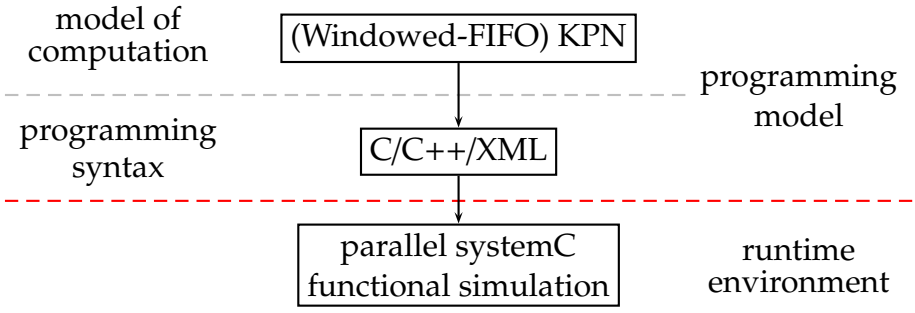


Fig. 3: An overview of this chapter.

Fig. 3 depicts an overview of this chapter. Following the orthogonalization-of-concerns methodology, we construct the syntax

of our programming model. We define XML schemata for system specifications, i.e., the structure of the application process network, the abstract architecture, and the mapping. To assist the design of large systems, a so-called iterator is developed by which a system can be arbitrarily scaled in a parametrized manner. We also define a set of C/C++ coding rules for the specification of the functionality of individual processes in an application process network. To reduce the workload of application programmers, our coding rules enable the reuse of the same piece of source code for a set of iterated processes. We propose a syntactic variation for the FIFO communication of KPN, namely windowed-FIFO. We prove the syntactic coherency of a windowed-FIFO process network to a standard KPN. To validate this concept, we also develop a hardware IP based on Xilinx FPGAs.

To functionally verify an application specified in our programming model, we develop a SystemC-based functional simulation, which serves as a prototype runtime environment. To efficiently execute a functional simulation, we parallelize SystemC execution by developing a library which enables the concurrent execution of multiple SystemC kernels. Using this library, a functional simulation of an application can be executed on an arbitrary number of Linux hosts connected via TCP/IP. Furthermore, the source code for the runtime environment can be automatically generated in a correct-by-construction manner from system specifications using our programming model.

The rest of the chapter is organized as follows: After a literature review in Section 2.2, we introduce the Kahn process network model-of-computation in Section 2.3. Section 2.4 describes the syntax of our programming model and Section 2.5 presents the windowed-FIFO communication. Section 2.6 presents a functional simulation using distributed SystemC simulation. Finally, Section 2.7 summarizes this chapter.

2.2 Related Work

In the literature, a variety of different tool-flows has been developed to program multi-core embedded systems. One category are the classical compiler-based approaches, e.g., MAPS [CCS+08], Compaan [SZT+04], CriticalBlue Cascade [Cri], CUDA [BFH+04, ati], and Brook+ [CUD]. For compiler-based approaches, a conventional sequential language, for instance, C, C++, or Matlab, is used as the initial application specification from which the compiler automatically extracts parallelism. To ease the job of compilers, explicit application programming interfaces (API), for instance, MPI [MPI], OpenMP [Ope], and TTL [vdWdKH+04], are often

used to identify parallel regions of an application with the domain-specific knowledge of programmers. The major problem of compiler-based approaches is that the level of abstraction of the underlying hardware exposed to application programmers is often too low, thereby lacking a uniform and scalable manner to specify concurrency of computation and communication of an application. The consequence is that system-level verification and software synthesis of a target system are often difficult. As a viable alternative, a model-of-computation-based approach is advocated. By restricting an application to a certain model-of-computation, the semantics of computation and concurrency can be mathematically defined. As a result, quantitative analysis pertaining to, for instance, schedulability tests and worst-case behavior, can be tackled in a reasonable manner. Furthermore, software synthesis can be applied, i.e., automatically generating implementations whose behavior is consistent with the abstract model behavior. Therefore, we focus on model-of-computation-based approaches in this chapter, in particular based on the class of process network model-of-computation.

Among other models-of-computation, the Kahn process network (KPN) [Kah74] and its ramifications are widely used because of their simple communication and synchronization mechanisms as well as coarse-grain parallelism. Besides the programming model presented in this chapter, many other tool-flows, for instance, Ptolemy [pto], Metropolis [BWH+03], Koski [KKO+06], and Artemis/SESAME/DAEDALUS [PEP06, NSD08], adopt process network. The most well-known sub-class of KPN is Synchronous Data Flow (SDF) [LM87, LP95] that enables static analysis of the specified application during compile time. Tool-flows based on SDF are, for instance, SHIM [ET06], PEACE [HKL+07], and StreamIt [TKA02]. Furthermore, Ptolemy [pto] supports heterogeneous modeling and Metropolis [BWH+03] defines a meta-model that can be refined into a specific model-of-computation. In our programming model, we adopt KPN for application modeling. We only define the semantics of the communication as point-to-point first-in first-out (FIFO) channels, and leave the implementation open for later refinements in the mapping stage. Therefore, specialized process networks semantics such as SDF can be obtained by imposing additional restrictions or semantics onto the process network.

An model-of-computation defines the semantics of a programing model. A corresponding syntax needs to be defined for the specification of a system. In Metropolis, a Java-like meta-model language is developed. In StreamIt and SHIM, custom C-like languages are exploited. Koski employs UML as its application programming interface. We argue that designing a new language is not the best option, since most of the legacy

code for embedded systems are written in C/C++. To reuse legacy code as much as possible, we define a hybrid approach, which decouples the behavior of the processes from the structure of the process network. Specifically, we propose an XML Schema to specify the structure of the process network and C/C++ coding rules for the functionality of individual processes. This decoupling enables a separation of coarse-grain and fine-grain potential parallelism. To exploit coarse-grain parallelism, for instance, at process level, system-level optimization techniques can be performed. To exploit fine-grain parallelism, for instance, at instruction-level, mature compiler techniques can be applied. Similar hybrid approaches can be found in Artemis YML [PHL+01] and Ptolemy MoML [LN00]. To assist the specifications of repetitive patterns for large systems, we propose a so-called *iterator* by which a system can be arbitrarily scaled in a parametrized manner. In the literature, only YML [PHL+01] reports a built-in scripting support but with less functionality.

The adopted KPN model-of-computation for application modeling enables the functional verification of an application before the implementation of a complete system. Therefore, we develop a functional simulation based on SystemC [Soc05, sys]. To efficiently simulate the functionality of a specified application, we develop a new technique to parallelize a SystemC simulation, enabling a geographically distributed SystemC simulation. There is related work in the literature focusing on distributing SystemC simulations. Approaches presented in [FFS01, MDC+05, Tra04] target geographically distributed IPCore verification. However, these approaches do not consider efficiency of the simulation. A synchronous data flow (SDF) extension of the simulation kernel is present in [PS05], where efficiency is gained by the concurrency of the SDF model. In [CCZ06], a functional parallel kernel has been developed by running multiple copies of the SystemC scheduler. The major drawback of the two last approaches is the modification of the SystemC kernel, which is not desired for generality and portability reasons. The work in [CCZ06], for instance, cannot support all SystemC features. A concurrency re-assignment technique is presented in [SSG02], acting as a compiler front-end of a SystemC model of a system. However, this proposed re-assignment transformation might lead to semantic unequivalence between the transformed model and the original one. Our work decouples the parallelization from the SystemC kernel, imposing no prerequisite to the execution semantics of the SystemC kernel.

2.3 Kahn Process Network

The goal of our programming model is to assist the software design of multi/many-core embedded systems, specifically, mapping a streaming application onto an multi/many-core platform. In this context, sequential languages such as `C/C++` and `Matlab`, are not effective, because they lack the semantic constructs of specifying concurrency. As a viable alternative, formal-model based design, which is often referred as model of computation, is promoted. A model-of-computation mathematically defines the semantics of computation and of concurrency. The rigid definition of model semantics enables system-level verification and software synthesis of a target system. Therefore, our programming model adopts model-of-computation-based design.

To model streaming applications, we adopt Kahn process network (KPN) [Kah74] model-of-computation. A KPN application consists of a network of concurrent *processes* that connect exclusively by point-to-point first-in first-out (FIFO) channels. Each process is autonomous and executes a sequential program. A FIFO is defined to have unlimited size. Writing to a FIFO is non-blocking and reading is blocking, i.e., a read operation will block the execution of a process if there is no data available in the FIFO buffer. Based on the preceding definitions, the execution of a process can be abstracted by three constituents:

- *Read:* a communication primitive for fetching data from a FIFO via an input port.
- *Write:* a communication primitive for sending data to a FIFO via an output port.
- *Compute:* a computation constituent, i.e., a segment of code between two communication primitives.

A KPN consisting of three processes is shown in Fig. 4. Process P_1 generates data and sends the generated data to process P_2 via FIFO C_1. Process P_2 reads data from C_1, conducts further manipulation, and sends data to process P_3 via FIFO C_2. Process P_3 is the sink of the network and it consumes data from C_2.

The KPN model-of-computation provides a rich set of properties which nicely fit the basic requirements for the design of multi/many-core embedded systems. First of all, a KPN allows explicitly specifying concurrency of an application, i.e., a network of autonomous processes. The explicit specification of concurrency allows application programmers to parallelize an application using their domain knowledge. We argue that domain knowledge is indispensable for an effective parallelization because of the high complexity of modern embedded applications.

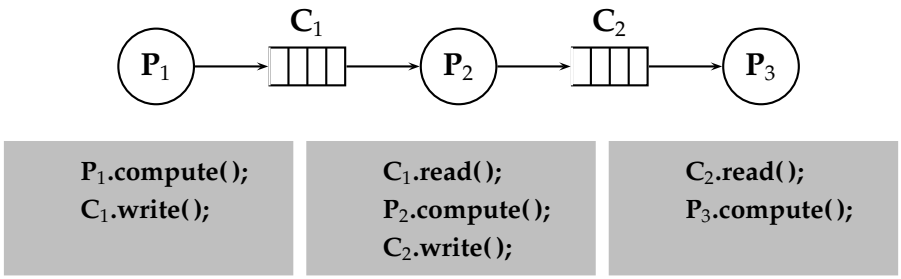


Fig. 4: An abstract view of a process network consisting of three processes.

Second, a KPN separates the communication and computation of an application, as the example in Fig. 4 shown. The separation of communication and computation allows to separately design communication and computation of a system. At an early design stage, for instance, application programmers can focus on the design of computation, while reserving the communication at system level for later refinements. One important property of KPN is that a KPN network is determinate, i.e., the functional behavior is independent from the timing of processes in the network. The determinism of KPN decouples the function of an application from the underlying architecture of a system, i.e., decoupling what a system does and how it does. This property enables separate refinements of application, architecture, and mapping during the design cycle. We will show in the rest of this thesis how we make use of these properties to assist software design of multi/many-core embedded systems.

2.4 Syntax of Programming Model

The KPN model-of-computation defines the semantics of our programming model. This section presents the corresponding syntax for the specification of a multi/many-core embedded system.

2.4.1 Basic Principles

A programming syntax provides a structural description of the various expressions that make up legal code in the programming model. To assist the design of a system, we have identified the following syntactic requirements.

- *Separation of concerns:* A key concept of the design of multi/many-core embedded systems is the separation of concerns, i.e., separating the application function and the underlying architecture, and

separating the computation and communication of the application. A programming syntax should provide means to establish these separations.

- *System-level abstraction:* Due to the high complexity of modern multi/many-core embedded systems, describing a system at system-level is an advisable starting point. A system-level abstraction allows verification of a system at an early design stage to detect bad design decisions as soon as possible. This early decision making is crucial because of the huge design space of modern embedded systems.

- *Reuse of legacy code:* In modern embedded system design, `C/C++` are still the dominant development languages to specify embedded applications. Whenever a new programming model is designed, the huge amount of legacy code should be considered. Completely re-writing an application using a new language is time-consuming and error-prone. One would expect to reuse as much legacy code as possible to reduce the development risk as well as time-to-market.

- *Scalability:* Future many-core embedded systems exhibit repetitive structures for both hardware architectures and applications, for instance, processor arrays and fast Fourier transform, respectively. Sizes and configurations of repetitive components vary for different designs. Therefore, scaling and reconfiguration of repetitive components need to be supported in a systematic way. In addition, the granularity of concurrency for an application should also be reconfigurable in order to explore the design space of a system and to find the optimal candidate design.

In the rest of this section, we present the syntax of our programming model. We separate the specifications of the application, architecture, and mapping of a system. We adopt `C/C++/XML` as the programming languages and define grammar rules for the system specifications. In addition, applications written according to this syntax are amenable for automated refinement with respect to the hardware and the operating system.

2.4.2 Application Specification

We present the syntax for the application specification in this section. The explicit differentiation between computation and communication of KPN enables the separation between the functionality and structure of an application. We use an XML Schema [MLMK05] to define the syntax for

the representation of the structure of the application, i.e., the topology of the process network. For the functionality of individual processes, we define specific C/C++ coding rules. Using an XML Schema for the structure of application allows us, on the one hand, to verify the structure of the process network prior to the final implementation and, on the other hand, to serve as an interface for other components in a software design toolchain, for instance, a graphical editor.

2.4.2.1 Process Network Specification

Instead of the verbose XML definition, we present the equivalent Backus–Naur Form (BNF) [Bac77]. Fig. 5 depicts the BNF description of the syntax for our application specification. The basic elements are *process* and *sw_channel*, which are used to define processes and FIFOs. Both processes and sw_channels have ports, i.e., *inport* and *outport*. The *connection* element defines a connection between a process port and a sw_channel port. Each process is associated with a C/C++ source file that contains the functionality of this process. The complete XML Schema is contained in the free available toolchain [dol].

```
processnetwork ::= <process>+ <sw_channel>+ <connection>+
                   <variable>* <function>* <configuration>*
                   <iterator>*
configuration  ::= <name> <value>
connection     ::= <name> <append>* <name> <name>
process  ::= <name> <append>* <source> (<iterator> <inport>+)*
             (<iterator> <outport>+)* <configuration>*
sw_channel  ::= <name> <append>* <size> <inport> <outport>
                <configuration>*
iterator ::= <name> <variable> <function>
             | <name> <variable> <function> (<iterator>+
             | <process>+ | <sw_channel>+ | <connection>+)
inport   ::= <append>* <name>
outport  ::= <append>* <name>
append   ::= <function> | <variable>
source   ::= <name>
variable ::= <name> <value>
function ::= <name> string
size     ::= digit
value    ::= digit
name     ::= string
```

Fig. 5: Application XML Schema expressed in BNF form. Items repeating 0 or more times are suffixed with an asterisk; Items repeating 1 or more times are followed by a '+'; Where items need to be grouped they are enclosed in simple parentheses.

In order to specify repetitive patterns, we design a so-called *iterator*

```
<variable value="100" name="N"/>

<process name="P1">
    <port type="output" name="10"/>
    <source type="c" location="P1.c"/>
</process>

<sw_channel type="fifo" size="10" name="C1">

<iterator name="pipeline" variable="i" range="N">
    <process name="P2">
        <append function="i"/>
        <port type="input" name="1"/>
        <port type="output" name="2"/>
        <source type="c" location="P2.c"/>
    </process>
    <sw_channel type="fifo" size="10" name="C2">
        <append function="i"/>
        <port type="input" name="0"/>
        <port type="output" name="1"/>
    </sw_channel>
</iterator>

<process name="P3">
    <port type="input" name="100"/>
    <source type="c" location="P3.c"/>
</process>
...
```

Fig. 6: Excerpt of the XML specification for the example process network in Fig. 4, where `P2` is scaled to `100` processes. The `connection` elements which connect processes and FIFOs are ignored for brevity.

element. An iterator together with an *append*, a *variable*, and a *function* element defines a scalable structure, where *append* defines which element inside the structure to iterate, *variable* defines the range of the iteration, and *function* defines how to compute the index for an iterated element. By using the iterator element, complex repetitive patterns can be specified and reconfigured in a simple way. With nested iterators, for instance, multi-dimensional structures can be constructed. Another usage for iterators is to scale the parallelism of an application. As shown in the case study in Section 2.4.5, iterators are used to define the number of concurrent sub-streams that a video stream is split into.

An iterated version of the process network in Fig. 4 is given as an example. The major part of the XML specification is depicted in Fig. 6. Process P_2 and FIFO C_2 are defined within the `pipeline` iterator with a range of $N = 100$, resulting in a pipelined network with 102 stages.

2.4.2.2 Process Specification

A C/C++ source file for each process defines a main entry of the functionality of this process. To facilitate automated software syntheses, we enforce a set of coding rules. Fig. 7 describes the definition of a process, which consists of a structure to store its local state, two function pointers, and a placeholder pointer. The functionality of a process is defined by an `init` and a `fire` procedure. The `init` procedure is called only once for initialization. Afterwards, the `fire` procedure is called repeatedly. During the execution, there might be information shared between sequences of `fire` procedures. This information is stored in the `LocalState` structure. Within the `init` and `fire` procedures, dedicated `READ` and `WRITE` communication primitives are used. To terminate a process, the `DETACH` primitive is used. By means of this design, the code for the `init` and `fire` procedures can be used in different runtime environments whereas the primitives `WRITE`, `READ`, and `DETACH` are instantiated differently according to a chosen runtime environment.

The void pointer `WPTR` is a place-holder for platform-dependent code of a final system. For example, it can be a pointer referring to the memory address of an instance of an iterated process.

```
1  // Process.h: definition of a process
2  typedef struct _local_states *LocalState; // local information
3  typedef void   (*ProcessInit)(struct _process*);
4  typedef int    (*ProcessFire)(struct _process*);
5  typedef void   *WPTR;                    // place holder
6
7  typedef struct _process {
8      LocalState     local;
9      ProcessInit    init;
10     ProcessFire    fire;
11     WPTR           wptr;
12 } Process;
13 ...
```

Fig. 7: The definition of a process in the source code.

Fig. 8 depicts the source code of the `P2` process in Fig. 6 as an example. Lines 1–8 define the local state of this process and the input/output ports corresponding to the XML definition. Lines 13–31 define the functionality of `P2`, i.e., it forwards a float from its input FIFO to its output FIFO for each `fire`. The execution ends after a certain amount of floats have been transmitted. Lines 33–41 present a sample `main` function where an instance of `P2` is instantiated.

In this manner, each process encapsulates its own state and operates independently from other processes in the system. Communications

```
1  // P2.h: header file
   typedef struct _local_states {
3      int index;
       int len;
5  } P2_State;

7  #define PORT_IN  1
   #define PORT_OUT 2
9
   // P2.c: C file
11 void P2_INIT(Process *p) {
       p->local->index = 0;
13     p->local->len = LENGTH;
   }
15
   int P2_FIRE(Process *p) {
17     float i;
       if (p->local->index < p->local->len) {
19         READ((void*)PORT_IN, &i, sizeof(float), p);
           WRITE((void*)PORT_OUT, &i, sizeof(float), p);
21         p->local->index++;
       }
23     if (p->local->index >= p->local->len) {
           DETACH(p);
25         return -1;
       }
27     return 0;
   }
29
   // main.c: sample main function
31 int main() {
       ...
33     Process p2; P2_State p2_state;
       p2.local = p2_state;
35     p2.init  = p2_INIT;
       p2.fire  = p2_FIRE;
37     ...
   }
```

Fig. 8: Source code for P2 in Fig 6.

are made explicit by FIFOs. This allows for a modular, and platform-independent application specification. Furthermore, all iterated processes from the same iterator structure reuse the same piece of source code for their functionality.

2.4.3 Architecture Specification

We also use XML Schema to define the syntax for the architecture specification of our programming model. We model the architecture of a multi/many-core platform at system-level, and thus define only the organization of the hardware in a platform, i.e., computing cores, memories, communication media, and how these hardware blocks are interconnected. We argue that a system-level specification at this abstraction level is sufficient for automated software synthesis as well as system-level design space exploration.

```
architecture ::= <processor>+ <hw_channel>+ <path>* <memory>*
                 <variable>* <function>* <configuration>*
                 <iterator>*
configuration ::= <name> <value>
processor    ::= <name> <append>* <configuration>*
hw_channel   ::= <name> <append>* <type> <configuration>*
path         ::= <processor_name> <memory_name>*
                 <hw_channel_name>+ <processor_name>
iterator ::= <name> <variable> <function>
           | <name> <variable> <function> (<iterator>+
           | <processor>+ | <hw_channel>+ | <path>+ | <memory>+)*
append   ::= <function>
variable ::= <name> <value>
function ::= <name> string
type     ::= digit
name     ::= string
processor_name   ::= string
hw_channel_name  ::= string
```

Fig. 9: Architecture XML schema expressed in BNF form.

The BNF grammar equivalent to the architecture XML Schema is depicted in Fig. 9. A *processor* element defines a computing core. Properties of the computing core, for instance, type (RISC, DSP, Hardware IP) and frequency, can be associated by the *configuration* element. Element *memory* defines a memory subsystem of a platform. Element *hw_channel* defines the used communication medium, such as buses or a network-on-chip. We use the element *path* to define how different hardware blocks interact. A path defines a route between two computing cores. By the definition of paths, we can provide a basis for communication refinement in the later on synthesis phase. To specify repetitive architectures, we apply the *iterator* element described in the previous section. With the iterator element, processor arrays, for instance, with a 3-D torus architecture, can be easily specified by three nested iterators.

2.4.4 Mapping Specification

In principle, the mapping defines how a process network is executed on an architecture. The mapping can be classified into two parts: the spatial domain that is referred to as *binding* and the temporal domain that is referred to as *scheduling*. In our programming model, the binding defines a mapping of processes to processors and FIFOs to communication paths. The scheduling defines the time-sharing policy on each shared resource and the according parameters, for instance, the cycle length and slot lengths for a time-division multiple access scheme, priorities of processes for a fixed priority scheme, and ordering of processes for a static scheduling scheme.

Again, an XML Schema is used to define the syntax for the mapping specification. The BNF description of the XML Schema is presented in Fig. 10. Besides the possibility to specify iterated mappings that are compliant to the application and architecture specifications, additional parameters, such as parameters for device drivers, can be specified by the *configuration* element.

```
mapping ::= <binding>+ <schedule>+ <variable>*
            <function>* <configuration>* <iterator>*
iterator ::= <name> <variable> <function>
             | <name> <variable> <function> (<iterator>+
             | <binding>+ | <schedule>+)*
binding  ::= <name> <append>* ((<processor_name> <process_name>)
             | (<hw_channel_name> <sw_channel_name>))
schedule ::= <name> [<append>+] <type>
type     ::= <fifo> | <fp> | <tdma>
fp       ::= <processor_name> (<process_name> <priority>)+
fifo     ::= <processor_name> <process_name>+
tdma     ::= <processor_name> <cycle_length>
             (<process_name> <slot_length>)+
append   ::= <function>
variable ::= <name> digit
function ::= <name> string
configuration   ::= <name> <value>
processor_name  ::= string          process_name    ::= string
sw_channel_name ::= string          hw_channel_name ::= string
cycle_length    ::= digit           slot_length     ::= digit
```

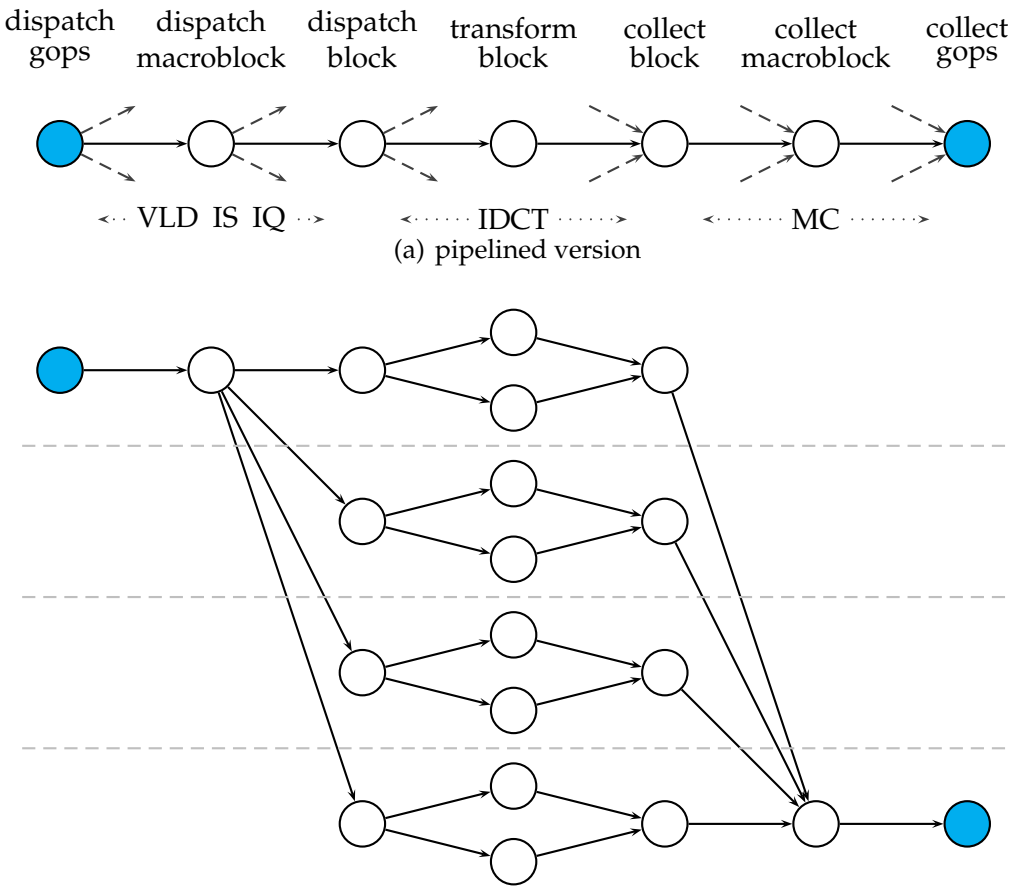
Fig. 10: Mapping XML schema in BNF form.

The mapping serves as an interface between components of a software toolchain, for instance, a design-space-exploration component generates a candidate mapping specification that is used as an input for the software synthesizer.

2.4.5 Experimental Results

To test our programming model, we parallelize a non-trivial real-life application, namely a sequential MPEG-2 decoder implementation [mpe], using the proposed programming syntax. We consider both function and data parallelism. To achieve functional parallelism, we construct three major functional components: a) process `dispatch_macroblock` for group variable length decoding (VLD), inverse scan (IS) and inverse quantization (IQ), b) process `collect_macroblock` for motion compensation (MC), and c) process `transform_block` for inverse discrete cosine transform (IDCT). These operations can be performed in a pipelined fashion on a stream of (macro) blocks, as shown by the dotted arrows in Fig. 11(a).

To achieve data parallelism, we construct three levels of data parallelism, i.e., at group of pictures (GOPS) level, macroblock level, and block level. We use processes `dispatch_gops`, `dispatch_macroblock`,



(a) pipelined version

(b) flattened version for 4 and 2 sub-streams of macroblocks and blocks, respectively

Fig. 11: A scalable MPEG-2 decoder.

and `dispatch_block` to split a stream of GOPS, macroblocks, and blocks into multiple independent sub-streams, respectively. Correspondingly, processes `collect_block`, `collect_macroblock`, and `collect_gops` are used to collect the decoded data and reassemble them into streams in the correct order. The data parallelism is indicated by the solid arrows in Fig. 11(a).

To reconfigure the granularity of each level of data parallelism, the iterator introduced in Section 2.4.2.1 is used. Fig. 12 shows a portion of the XML specification of the MPEG-2 process network, where variables N_1, N_2, and N_3 are used to specify the numbers of sub-streams for GOPS, macroblocks, and block, respectively.

Fig. 11 shows a graphical view of the process network of the MPEG-2 decoder. Fig. 11(a) depicts a pipelined version where the data parallelism is not exploited. Fig. 11(b) depicts a version where the process `dispatch_macroblock` dispatches macroblocks to four parallel branches

```
<!-- N1 is the number of gops processed in parallel -->
<variable name="N1" value="1"/>

<!-- N2 is the number of macroblocks processed in parallel -->
<variable name="N2" value="4"/>

<!-- N3 is the number of blocks processed in parallel -->
<variable name="N3" value="2"/>

<!-- instantiate processes -->
<process name="dispatch_gops">
  <iterator  variable="i" range="N1">
    <port type="output" name="out">
      <append function="i"/>
    </port>
  </iterator>
  <source type="c" location="dispatch_gops.c"/>
</process>
...
<iterator  variable="i" range="N1">
  <iterator  variable="j" range="N2">
    <process name="dispatch_blocks">
      <append function="i"/>
      <append function="j"/>
      <iterator  variable="k" range="N3">
        <port type="output" name="out">
          <append function="k"/>
        </port>
...
```

Fig. 12: Excerpt of the process network XML for the MPEG-2 decoder.

and each branch will again be dispatched by `dispatch_block` to two sub-branches `transform_block`, resulting in a network of 20 processes. Note that the source code of these two versions is identical. The only difference is the values of variables N_1, N_2, and N_3 in the XML specification.

We also report the code size in terms of lines-of-code. The reference sequential implementation [mpe] contains 5727 lines of `C` code. The parallel version using our programming model contains 4004 lines of `C` code and 319 lines of XML code. The difference can be explained by the fact that the reference code is a full-fledged compilable source code for IA-32 architecture whereas our version only contains the platform-independent portion of the decoder.

From this experiment, we conclude that our programming model is practicable. In the next two sections, we will discuss the efficiency of our programming model in terms of syntactic variation of the KPN model-of-computation as well as runtime environments.

2.5 Windowed-FIFO Communication

The KPN model-of-computation provides a set of useful properties for designing multi/many-core embedded systems. Partly, these properties are based on the FIFO communication. The rigorous FIFO communication, however, has limitations in terms of programmability and efficiency. In this section, we present a syntactic variation for the FIFO communication of KPN to increase the efficiency and programmability of FIFO communication.

2.5.1 Motivation

Although the KPN model-of-computation offers a simple interface for programming, it also has limitations that make the implementation of streaming applications difficult or inefficient. Furthermore, the communication overhead might overshadow the performance gained by the coarse-grain parallelism. A few syntactic limitations that are of interest are addressed in the following:

- *Reordering*: A KPN FIFO behaves in a strict first-in first-out manner, which does not allow reading data in an order other than the one in which the data have been written.

- *Non-destructive read*: A KPN FIFO does not allow reading the same data item more than once. After a data item is read, it will be deleted from the FIFO memory.

- *Skipping*: It is not possible to remove a data item from a FIFO without a read operation. Even if the FIFO contains unwanted data, all these unwanted data have to be read out before subsequent data can be accessed. An example for a skipping scenario is described in Section 2.5.5.2.

Ex. 1: *A simple producer-consumer example is shown in Fig. 13, where the producer generates a two-dimensional array in column order and the consumer reads the array in row order. Furthermore, the second row of the generated array will be accessed twice. The access order of the two-dimensional array is depicted in Fig. 14. Because a FIFO is one-dimensional, the two-dimensional array needs to be linearized into one dimension if FIFO communication is used. The corresponding FIFO access order is presented in Fig. 15. As shown in Fig. 15, the consumer cannot use a FIFO, because it does not support random access.*

Furthermore, from the implementation perspective, sending data across a FIFO requires two memory-copy operations, i.e., the producer copies data from its local memory to the FIFO memory and the consumer copies data from the FIFO memory to its local memory. These memory-copy operations might hamper the performance.

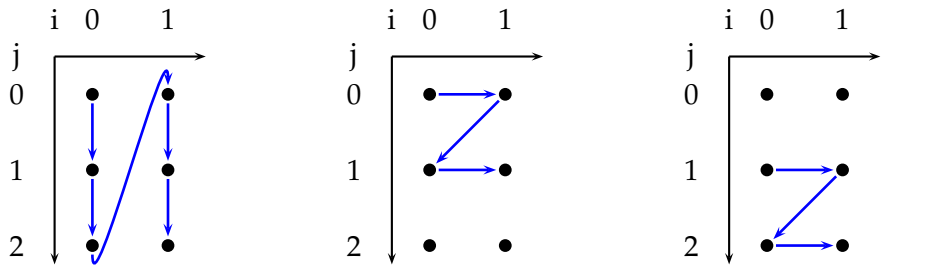
```
// producer
for (i=0; i<2; i++) {
  for (j=0; j<3; j++) {
    A[i][j]=nextValue();
  }
}
...
```

(a) Producer.

```
// consumer
for (j=0; j<2; j++) {
  process(A[0][j]);
  process(A[1][j]);
  process(A[0][j+1]);
  process(A[1][j+1]);
}
```

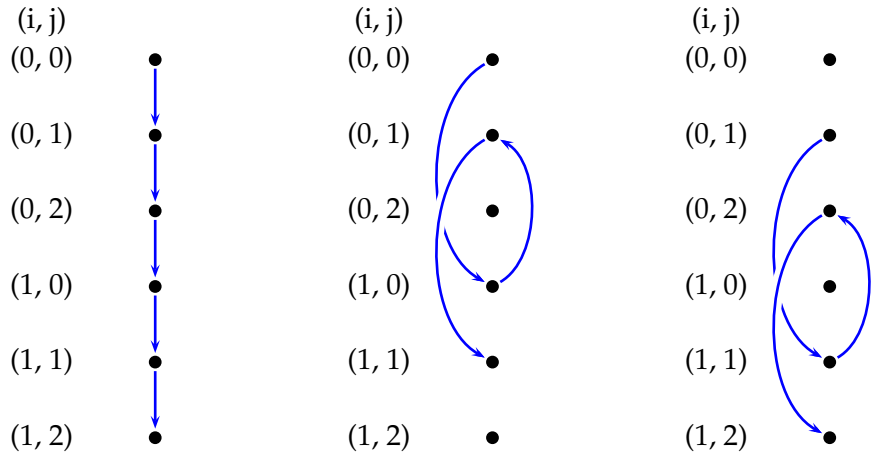
(b) Consumer.

Fig. 13: Source code for a simple producer-consumer example.



(a) Producer. (b) First iteration of the consumer. (c) Second iteration of the consumer.

Fig. 14: Access order of the array for the producer-consumer example in Fig. 13.



(a) Producer.

(b) First iteration of the consumer.

(c) Second iteration of the consumer.

Fig. 15: Access order of the linearized array for the simple producer-consumer example in Fig. 13.

To tackle the aforementioned limitations, we introduce a variation of FIFO communication, referred to as windowed FIFO (Wfifo) communication. Wfifo communication tackles in a particular way the limitations mentioned above, allowing non-destructive read, reordering, and skipping of data within a window of parameterizable length in the FIFO memory.

2.5.2 Semantics and Syntax

A Wfifo originates from a normal FIFO but offers more functionality and flexibility. Unlike a normal FIFO, a Wfifo supports out-of-order accesses within a continuous segment located at the head and tail of a normal FIFO. These two segments are called windows, which leads to the name windowed FIFO. The semantics of a Wfifo can be summarized as follows:

- A Wfifo has two access ports, i.e., a read and a write port, to support read and write operations, respectively.
- An *acquire* operation is mandatory before data transmissions. An acquire-write operation can acquire a window not larger than the available free space in the FIFO. An acquire-read operation can acquire a window not larger than the fill level of the FIFO.
- An acquire-write operation will block the calling process if the space in the Wfifo memory is smaller than the acquired window.

Similarly, an acquire-read operation will block the calling process if the number of available data is smaller than the acquired window.

- Random accesses are allowed within an acquired window.
- A *release* operation is mandatory to release an acquired window. A release-write operation will mark the data within the window available for reading whereas a release-read operation will discard the window from the FIFO.

A coarse-grained diagram of a WFIFO is depicted in Fig. 16. The gray boxes located at both ends of the FIFO can be considered as random-access memories that host the windows. At each end, a logic controls the access of the random-access memory. For instance, when the logic of the write port receives an acquire-write instruction, it allocates a piece of random-access memory that has a continuous address space and a size as required. Subsequent write instructions are directed to this memory at a position indicated by a given offset. When a release instruction is received, writing is disabled and the data in the random-access memory will be appended to the FIFO without changing their order.

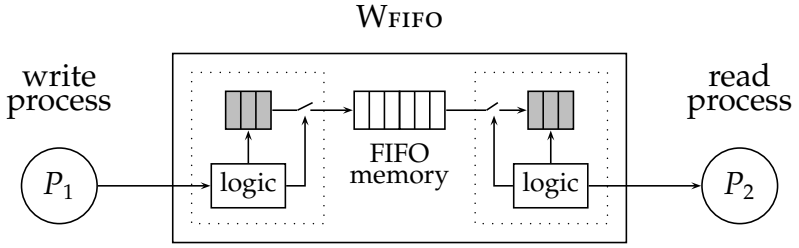


Fig. 16: A coarse-grained diagram of the WFIFO.

Based on the preceding definitions, we define the corresponding syntax. Because writing to and reading from a WFIFO are independent and symmetric, we only report the syntax for writing. Fig. 17 depicts a finite-state machine for the write port. The finite-state machine consists of two states, i.e., `idle` and `writable`. The initial state is `idle`. An *acquire-write* operation enables a state transition from `idle` to `writable`. When the write port is in state `writable`, write operations are allowed. By a *release-write* operation, the state of the write port is changed back to `idle`.

The corresponding API is defined as follows:

- **WFIFO_ACQUIRE_WRITE(port, size)**

  The acquire-write instruction allocates a window of size *size* at the write port of a WFIFO. The *port* points to the address of the WFIFO to which a process intends to write data. If the size of the free space in

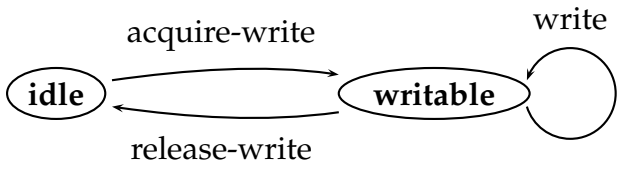


Fig. 17: The finite-state machine for the write port of a WFIFO.

the WFIFO is smaller than *size*, then this instruction blocks the calling process until enough memory is available.

- **WFIFO_WRITE(port, offset, data)**

  The write instruction writes the *data* to the write window at the position indicated by the *offset*. This instruction can be repeated an unlimited number of times. It is possible to write the same offset position more than once. In the case of multiple writing to a same offset address, the new value will overwrite the old one. If an offset position is never written, its value is undefined.

- **WFIFO_RELEASE_WRITE(port)**

  The release instruction terminates the writing phase and appends the content of the window to an internal FIFO, from which it can later be read. After releasing, no further writing to the window is possible. Before more data can be written to the WFIFO, a new write window must be acquired.

The finite-state machine and API for the read port are analogous to the write port. They contain *acquire-read*, *read*, and *release-read* operations. An *acquire-read* operation succeeds if there are enough data in a WFIFO memory for the acquired read window. After successful acquisition, data can be read from the window. The release instruction deletes the acquired window from the WFIFO.

2.5.3 Properties

A process network using WFIFO communication has properties similar to a KPN. In particular, we will show that a WFIFO process network is determinate, i.e., the functional behavior is independent on the timing of the processes in the process network.

Lem. 1: *A* WFIFO *process network can be simulated by a process network with both blocking-read and blocking-write semantics.*

Proof. The fundamental difference between a WFIFO process network and a process network with blocking semantics is the replacement of

FIFO communication with WFIFO. It must be shown that the windowed semantics can be implemented using a FIFO with blocking semantics. Given the property that the read and write windows do not overlap each other, the segment between the read and write windows is strictly first-come first-out, i.e., equivalent to a FIFO. The window mechanism can be transformed into a software implementation within the connected processes. For writing a WFIFO, when an acquire-write operation is invoked, a segment in the local memory with a size as the size of the acquired window is allocated. Subsequent write operations are redirected to this segment until a release-write operation is received. The release-write operation will move the data within this segment to the FIFO memory. Reading a WFIFO is similar. When an acquire-read operation is invoked, a segment in the local memory with a size as the size of the acquired window is allocated and data in the acquired window are moved from the FIFO memory to this segment. All subsequent read operations are redirected to this segment, i.e. reading from this piece of local memory. This segment will be freed upon a release-read operation. Under this transformation, a WFIFO process network can be simulated by a process network with blocking-read and blocking-write semantics.

□

Lem. 2: *A process network with blocking-write (and blocking-read) semantics can be simulated by a KPN.*

Lem. 2 is a known result [GB03]. Nevertheless, we show in the following an intuitive description for how to simulate a process network with blocking semantics using a KPN. A blocking-write semantics of a FIFO implies a bounded FIFO, i.e., with a finite size. A bounded FIFO, say F_i, with a size s in a process network can be replaced by a pair of unbounded FIFOs with size s, where one (forward) has the same direction as F_i and the other (backward) has the opposite direction. If F_i initially does not contain any data, i.e., an empty state, the backward FIFO is set to contain s dummy data tokens. In the source code of the calling process, a write operation to F_i is transformed into two operations: first read a dummy token from the backward FIFO, and then write the actual data to the forward FIFO. Similarly, a read operation of F_i is transformed into two operations as well: first write a dummy token to the backward FIFO, and then read the actual data from the forward FIFO. This backward-pressure mechanism guarantees that there are never more than s data tokens in the forward FIFO, which is tantamount to a bounded FIFO with size s. Under this transformation, a bounded FIFO can be implemented with a pair of unbounded FIFOs with non-blocking write and blocking-read semantics. Therefore, the lemma holds.

Thm. 1: *A* Wfifo *process network is determinate, i.e., the functional behavior is independent on the timing of the processes in the network.*

Proof. Combining Lemma 1 and Lemma 2, a Wfifo process network and a KPN are simulation equivalent [LGS+95]. Therefore a Wfifo process network preserves properties of Kpn.

□

2.5.4 Implementation Issues

This section discusses implementation issues of a Wfifo. If two processes in a process network that communicate using Wfifo are located in a same computing core, a Wfifo can be implemented in software, for instance, using a ring buffer. A reference implementation can be found in [HSH+09]. For processes located in different computing cores, it is also possible to use the Wfifo concept. We implement a hardware prototype based on FPGA, using the Xilinx Embedded Development Kit (EDK) [EDK].

Fig. 18 gives an abstract view of the architecture of the Wfifo IP. We use the Intellectual Property Interface (IPIF) in the Xilinx EDK as the interface of the Wfifo IP, such that the Wfifo IP can be connected to different types of buses, for instance, the high-speed Processor Local Bus (PLB) and the On-Chip Peripheral Bus (OPB) that are available in the EDK. The Wfifo memory is implemented with a dual-port BlockRAM (BRAM) that is an on-chip parameterizable memory module available on all newer Xilinx FPGAs. The access of the BRAM is controlled by the finite state machines explained in the previous section.

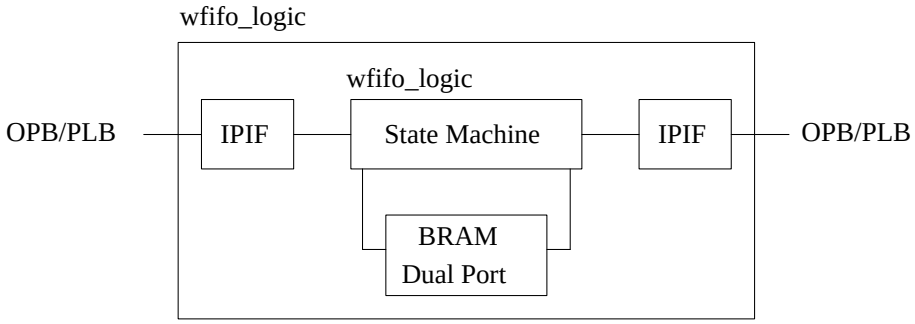


Fig. 18: An abstract view of the Wfifo IP architecture.

Using the built-in synthesis tool of EDK, the size of the Wfifo IP in terms of slices – the basic logic unit of Xilinx FPGA – is 754 slices. We consider our Wfifo IP to be practically useful as modern Xilinx FPGAs contain hundreds of thousands of slices. Write and read operations take 5 and 6 clock cycles respectively. The Wfifo IP can be clocked

at frequencies up to 173.6 *MHz*, which is the maximum frequency for the IPIF component. For a detailed hardware description and timing diagrams, the reader is referred to [Grü06].

2.5.5 Empirical Case Studies

In this section, we present proof-of-concept case studies to demonstrate the practicality of our WFIFO.

2.5.5.1 Reordering and Multiple Read

Recalling the example shown in Fig. 13, there is no unique solution for programming the producer or consumer when using a normal FIFO. Because the order in which the producer generates the array is different from the one in which the consumer read it, either the producer or the consumer has to linearize the access order of the array. Furthermore, because some entries of the array will be accessed twice, either the producer has to send these data again or the consumer has to store these data somewhere in its local memory for the second access. All these solutions are a bit ad-hoc from the programmers point of view.

```
// producer

for (i=0; i<2; i++) {
  for (j=0; j<3; j++) {
    A[i][j]=nextValue();
  }
}


// consumer

for (j=0; j<2; j++) {
  process(A[0][j]);
  process(A[1][j]);
  process(A[0][j+1]);
  process(A[1][j+1]);
}


...
```

(a) Sequential code in Fig. 13.

```
#define wf 0x1 //address of wfifo
// producer process
WFIFO_ACQUIRE_WRITE(wf,6);
for (i=0; i<2; i++) {
  for (j=0; j<3; j++) {
    WFIFO_WRITE(wf,3i+j,nextValue());
  }
}
WFIFO_RELEASE_WRITE(wf);

 // consumer process
WFIFO_ACQUIRE_READ(wf,6);
for (i=0; i<2; i++) {
  for (j=0; j<2; j++) {
    for (k=0; k<2; k++) {
      WFIFO_READ(wf,i+j+3k,tmp);
      process(tmp);
    }
  }
}
WFIFO_RELEASE_READ(wf);
```

(b) Process network using WFIFO.

Fig. 19: Source code for the sequential version and process network for the producer-consumer example in Fig. 13.

Using a WFIFO, the solution shown in Fig. 19(b) becomes straightforward. The consumer first acquires a read window from the WFIFO, and then he can randomly read the array no matter in which order the producer generated the array data. Meanwhile, as the read is non-destructive, the consumer can read eight times within the acquired window from the WFIFO although the producer only writes six times. In addition, the storage of the array shifts from the local memory of the process to the WFIFO, which reduces the memory footprint of the consumer as a side effect.

2.5.5.2 Skipping

A possible application for the *skipping* function is in the domain of parallel image and video compression [Jai81], where a sequence of image/video frames is divided into multiple slides or macroblocks which are compressed concurrently. In such a scenario, it is typically not known a priori which portions of a frame are redundant. For this type of data exchange, the WFIFO concept can be applied. Instead of a full-fledged image-compression algorithm, we present a less complex application that has similar requirements, namely a parallel version of the *game of life* [GH99].

A *world* for a game consists of a two-dimensional array of cells that are either dead or alive. The stats of cells may change every cycle. The state of a cell in the next cycle depends on its current state and on the states of its eight immediate neighbors. The rules to compute the next state of a cell are: an alive cell with fewer than two alive neighbors dies; an alive cell with more than three alive neighbors dies; a dead cell with exactly three alive neighbors becomes alive; otherwise, the state of a cell remains unchanged.

The game of life can be parallelized using a process network. A world, for instance, can be split into horizontal slides and the states of cells in each slide are computed by a separate process. If each process runs on a separate computing core, computing the next states is fully concurrent. Fig. 20 shows an example where a 28×14 world is split into two 28×7 slides. These two slides are processed by two separate processes, namely process `X` and `Y`, resulting in a process network with two processes.

For cells at the boundaries of a slide, determining their next states, in principle, requires knowledge of the states of cells located at the neighboring slide. As shown in Fig. 20, any cell in the two gray rows has three neighbors located the other slide. However, there are cases where a decision can be made with only local knowledge. Fig. 21 shows three such cases where the cell states from neighboring slide are not (completely) required. The key message here is that only the process that

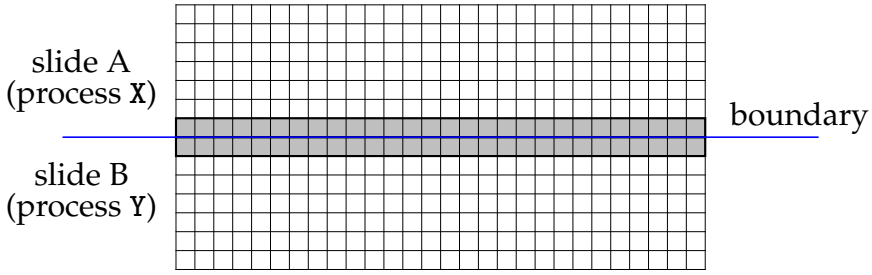


Fig. 20: A 28×14 world of a game.

computes the states of its local cells knows whether it requires the states of cells from the neighboring slide. The other process cannot know this a priori and has to send states of all cells along the boundary.

We again consider the example in Fig. 20. Process `X` has to send the states of cells at its gray row to process `Y` so that process `Y` can compute the state of cells in its gray row, and vice versa for `Y`. In the case of FIFO communication, even if some states are redundant, process `Y` has to remove these state data from the FIFO memory by read operations in order to access subsequent data. With WFIFO, only sending is mandatory and reading is more flexible. After acquiring a read window, a process is free to decide which state to read or to ignore.

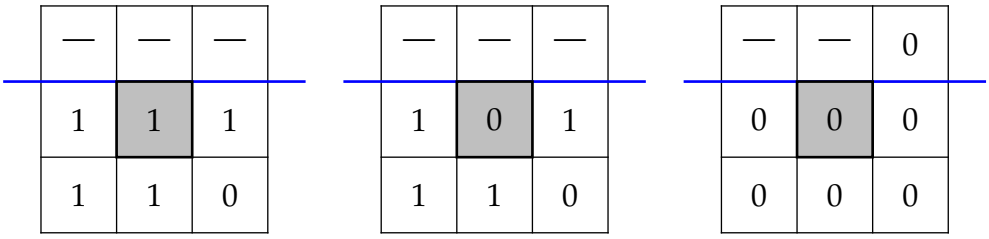


Fig. 21: Three situations where the next state of the cell indicated in gray is known without reading the states of cells beyond the boundary.

We implement this two-process network to simulate the 28×14 world in Fig. 20. At every game cycle, each process reads the states of cells in the gray row located in the other process in order to compute the next states of its local cells. We compare two cases, i.e., using FIFO and WFIFO for communication. For the case of FIFO, a process has to read 28 states data from the FIFO connected to the other process at each game cycle. When using WFIFO, on the contrary, only those states that are actually needed are read from the WFIFO memory. We run the game for 100 cycles and record the ratio of the communication reduction, i. e. , the ratio of the total number of states skipped by using WFIFO to the total number of states has been read by using FIFO in these 100 game cycles. The initial state

of being alive for a cell is randomly generated with a given probability, denoted as δ.

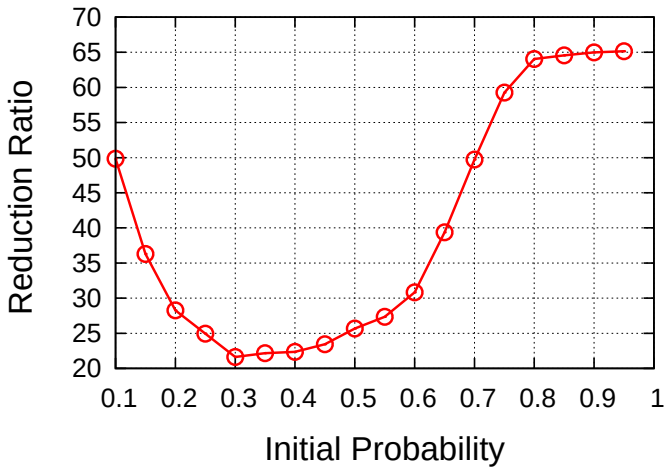


Fig. 22: Transit reduction of state data for different initial live-probability of cells.

We conduct experiments for different δ. For each δ, we report the mean value of the reduction ratios from 100 different runs. The experimental results are shown in Fig. 22. As shown in the figure, the improvement is significant. In general, we reduce 22 – 65 % state data transmission. The WFIFO mechanism conducts better for cases of large and small δ, where dead cells dominate the world in the long run. We expect applications from the domain of image/video compression can benefit more from the WFIFO concept, because data items for these applications are at block/macroblock/frame levels, where the *skipping* function can reduce a larger amount of data transmissions and I/O operations.

2.6 Parallel SystemC Functional Simulation

Modern embedded applications are far more complex than their single-threaded ancestors. The reference implementation for the H.264 codec, for instance, consists of over 120,000 lines of C code. It is preferable to verify the functionality of such complex applications before the final system is implemented.

This section presents a functional simulator, which is used to functionally verify an application specified in our programming model. Such a functional verification is possible because of the determinism of adopted KPN model in our programming model. Besides verifying the functionality of a specified application, our functional simulator is also used to obtain architecture-independent information about an

application, for instance, to extract traces for the trace-based analysis that will be presented in the next chapter.

We develop our functional simulator based on SystemC [Soc05, sys], which is a C++ library to model and simulate hardware and software systems on different levels of abstraction. The SystemC library provides a discrete-event simulation kernel, which enables a fast simulation of a process network. To further accelerate the simulation speed, we propose a new technique to parallelize a SystemC simulation. Our parallelization enables that an arbitrary number of programming cores can concurrently share the simulation workload via socket interface.

2.6.1 Simulation Framework Overview

The idea of our parallel SystemC (PSC) simulation is to simultaneously execute a set of SystemC kernels, each of which simulates portions of an application. We denote a stand-alone SystemC kernel as a *simulator*. The goal thereby is to execute a set of simulators on a single host with multiple cores or on a network of computers.

Let us revisit the KPN model used in our programming model. Each process in the process network is autonomous and the execution of processes does not require a global synchronization. In principle, a process can execute until one of its read operations blocks on an empty input FIFO (or blocks on a full output FIFO in the case that the size of a FIFO is finite). The autonomy of KPN processes enables concurrent execution of the SystemC functional simulation of a KPN application by executing a set of independent simulators, each of which simulates the functional behavior of a group of KPN processes.

We use the Y-chart paradigm for the parallelization. Fig. 23 depicts the framework. We define two layers of mappings. Layer 1 defines how many simulators are used and how to distribute application processes to these simulators. Layer 2 defines how many simulation hosts are used and how to distribute simulators to simulation hosts.

To coordinate the execution of different simulators, we develop a C++ library, which takes charge of the communication and synchronization among simulators. We also develop a software synthesizer to automate the generation of a parallel SystemC simulation. The software synthesizer generates the C++ source code of simulators from the specifications of a simulation, i.e., application process network, architecture (host name), and mapping specifications. A script file is also generated to assist the compilation of the source code to ELF binaries and distribution of the compiled binaries to specified simulation hosts.

By means of the aforementioned two-layer mapping approach and the iterator technique presented in Section 2.4.2.1, a functional simulation of

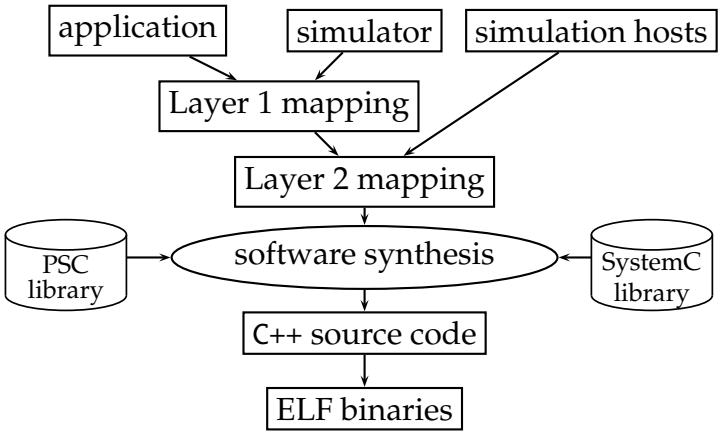


Fig. 23: Overview of the generation of a simulation.

an application can be concurrently executed on an arbitrary number of simulation hosts. Although the speedup of the simulation depends on different factors, e. g. , the amount of inter-simulator communication and granularity of the parallelism, our approach provides a convenient way to obtain an efficient mapping of the simulation workload, i. e. , adjusting the granularity of the parallelism by changing the number of processes mapped onto a simulator and the number of simulators onto a simulator host.

2.6.2 SystemC Introduction

SystemC is a C++ based library to model and simulate hardware and software systems on different levels of abstraction. The main components for modeling a system in SystemC are processes (`sc_thread`) and channels (`sc_channel`). Processes describe functionality and channels transmit communication data between processes. Processes send data to channel via ports (`sc_port`). Communication data transmitted by channels are called events. An event can be, for instance, change of a value of a signal channel.

SystemC provides an event-driven kernel to simulate a system, i.e., processes are executed in response to the occurrence of events. Fig. 24 depicts the flowchart of the SystemC kernel. A key feature of the SystemC simulation kernel is the support of the concept of a *delta cycle*, like in VHDL and in Verilog. A delta cycle consists of an evaluation phase and an update phase. The evaluation phase includes checking for runnable processes, executing runnable processes, and processing immediate notifications of an event. Processing an immediate notification will mark all processes that are sensitive to this event runnable within the same evaluation phase.

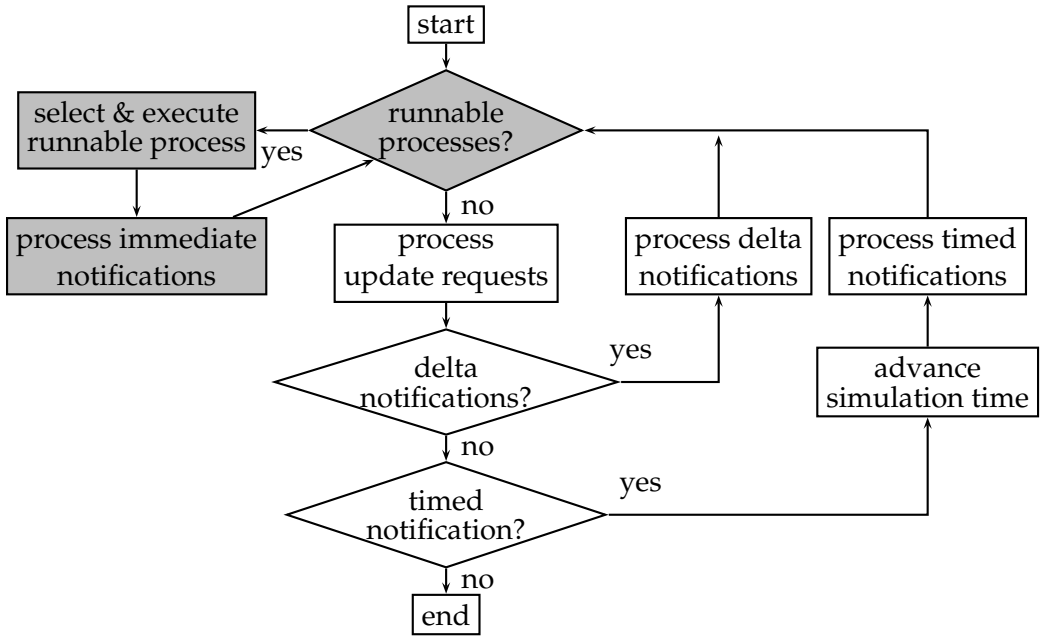


Fig. 24: The flowchart of the SystemC simulation kernel, corresponding to the `sc_start()` procedure in the source code. The gray boxes constitute the evaluation phase.

The update phase starts when there is no runnable process in the evaluation phase. The update phase will process update requests, which generate delta notifications. All processes which are sensitive to delta notifications are marked runnable and the simulation can enter the evaluation phase again. To this end, the simulation is said to advance by one delta cycle.

Simulation time does not change within a delta cycle. The simulation kernel only considers timed notifications when there is no runnable process within a delta cycle. In this case, the simulation time is advanced to the time of the earliest timed event and all processes sensitive to timed events at the current simulation time are made runnable and the evaluation phase is entered again. If no pending timed events exist, the simulation terminates.

Three possible ways to execute the SystemC simulation kernel are listed as follows: `sc_start`:

- `sc_start()` – Runs the simulation until no more event exists, as previously described.
- `sc_start(timeval)` – Run the simulation for `timeval` interval. The simulation pauses when the simulated time advances to `timeval` and processes sensitive to timed events at this time are made `runnable`.

- `sc_start(SC_ZERO_TIME)` – Runs the simulation for *one* delta cycle only. Processes sensitive to delta events of the next delta cycle are made `runnable`.

The OSCI SystemC is designed as a single-thread user-level application. Therefore, even if a simulation host has multiple cores, only a single one can be used.

2.6.3 Stand-alone SystemC Functional Simulation

Implementing a functional simulation using stand-alone SystemC (SSC) kernel is straightforward. Recalling the definition of KPN, the concept of process, port, and FIFO naturally matches the basic components of SystemC. Therefore, we use `sc_thread`, `sc_port`, and `sc_channel` to model KPN processes, ports, and FIFOs. Fig. 25 shows an example SystemC model of the process network with the three processes depicted in Fig. 4. A wrapper for each process is designed to implement the functionality of this process by means of a `sc_thread`. In this manner, the source code of each process remains unmodified.

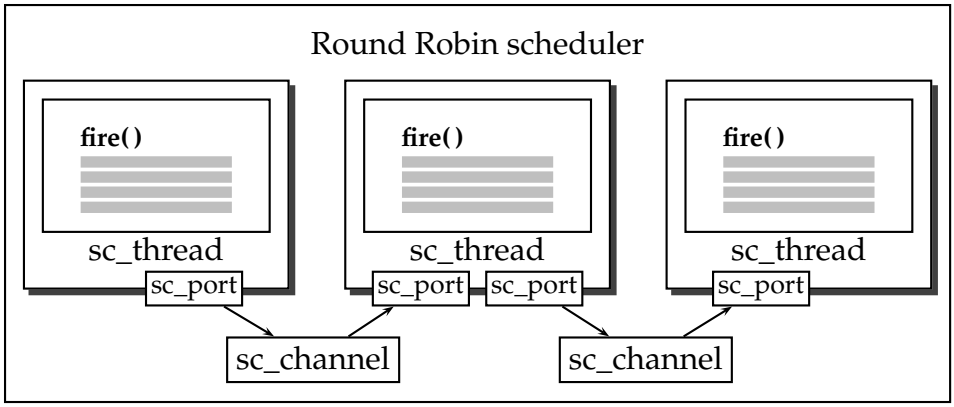


Fig. 25: Software architecture of the stand-alone SystemC simulator.

The execution order of concurrent SystemC threads is not defined by the SystemC standard. Therefore, a round-robin policy is designed to schedule the execution of KPN processes.

2.6.4 Parallel SystemC Simulation

How to parallelize a SystemC simulation is open issue. There is no definition in the IEEE SystemC standard [Soc05]. The OSCI SystemC [sys] simulation kernel is designed for single user-level thread simulation and can use only one computing core even if a simulation host contains multiple cores. Different approaches [SSG02, PS05, CCZ06] have been

proposed to parallelize a SystemC simulation. These approaches, however, require changes to the simulation kernel, which may lead to semantic differences to the standard SystemC.

In order to preserve the semantics of standard SystemC, we simulate an application by simultaneously executing a set of SystemC kernels, each of which simulates portions of the functionality of an application. To coordinate the communication and computation among simulators, we design a delta-cycle based simulation, where communication and synchronization take place between two delta cycles of the simulation. In this manner, we can keep the SystemC kernel intact while achieving parallelism. The flowchart of our approach is depicted in Fig. 26. The SystemC kernel runs the simulation for a delta cycle by calling `sc_start(SC_ZERO_TIME)` repeatedly. After each delta cycle, the simulation is temporarily paused and communication and synchronization to other simulators take place. As we consider only functional simulation, synchronization refers to stopping the whole simulation. In the rest of this section, we elaborate on the communication and synchronization mechanisms.

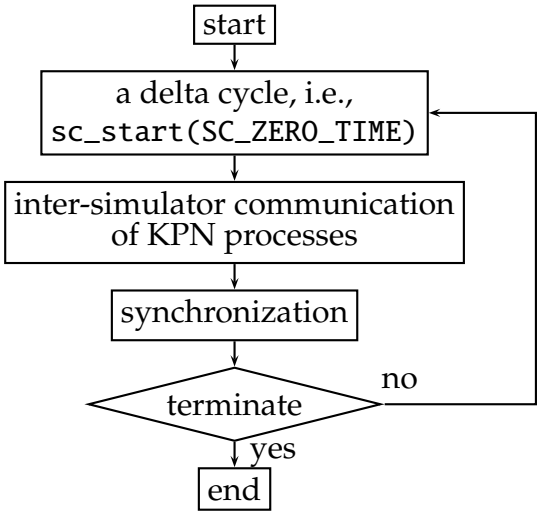


Fig. 26: The flowchart of our delta-cycle wise simulation.

2.6.4.1 Communication

To decouple the inter-simulator communication from the SystemC simulation kernel, we define a *remote channel*. A remote channel is an interface for a process to communicate to a process located in another simulator. To send out data, a *remote output channel* is defined. Similarly, *remote input channels* are defined for receiving data. A KPN FIFO that connects two processes located in two different simulators corresponds to a remote input/output channel pair. A *channel manager* is used for the

actual data transfer between remote channels.

Fig. 27 depicts our mechanism for communication. For a process which needs to send data to processes located in another simulator, remote output channels are instantiated. Similarly, remote input channels are instantiated for receiving data. Consider the remote output channel as an example. Within a delta cycle, processes write data to remote input channels. After the delta cycle finishes, the channel manager checks whether there are data available and performs the actual data transfer. For each transferred data, an immediate notification will be generated to wake up a suspended process which is waiting on a full FIFO. These suspended processes will be made runnable and executed during the next delta cycle. The channel manager will repeat this procedure until remote channels have nothing to send anymore.

A remote input channel behaves similarly. It receives data from the channel management. Once the channel manager receives a data item from the socket interface, it generates an immediate notification, denoted as *remote notification*. A remote notification will resume the execution of a suspended process at the next delta cycle.

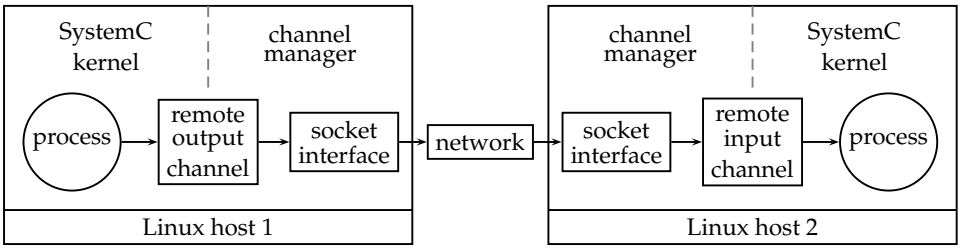


Fig. 27: The graphical view of our communication mechanism.

Our implementation of the remote channel is based on the `simple_fifo` of the OSCI library. The buffer is implemented as a ring buffer. A process writes byte-wise data to the ring buffer. If the buffer is full, the process will be suspended on the write operation. After each delta cycle the channel manager checks whether there are data in the ring buffer. If there are, the channel manager will instantiate an actual data transmission and move these data to the ring-buffer of a remote input channel. We use TCP sockets to transmit data between the two remote channel buffers, although other communication protocols could be used.

2.6.4.2 Synchronization

The synchronization is used to determine when the functional simulation completes. In the IEEE Standard [Soc05], the SystemC kernel ends a simulation when there is no pending notification. Terminating a

simulation under this condition, however, is not correct for distributed simulation, because notifications issued by a remote simulator cannot reach the local simulator immediately due to network delay. It might happen that a simulator has finished processing all pending notifications while remote notifications are still on their way. Basically, a simulator by itself cannot know when it should terminate in the case of distributed simulation.

To solve this problem, we design a master-client approach with a deferred termination mechanism. Each simulator is a client that has four states, i.e., `active`, `standby`, `idle`, and `exit`. Initially, a client is set to `active`. A client remains in `active` state as long as there are pending notifications, otherwise it changes to `standby`. Once a client changes to `standby`, it sets up a timeout. If this timeout expires, the client changes to `idle` and informs the master about its state transition. A client resumes to `active` from `standby/idle` when it receives a remote notification. It will *immediately* informs this state transition to the master. The client changes from `idle` to `exit` only when it receives a termination acknowledgment from the master. The state transition system of a client is summarized in Fig. 28.

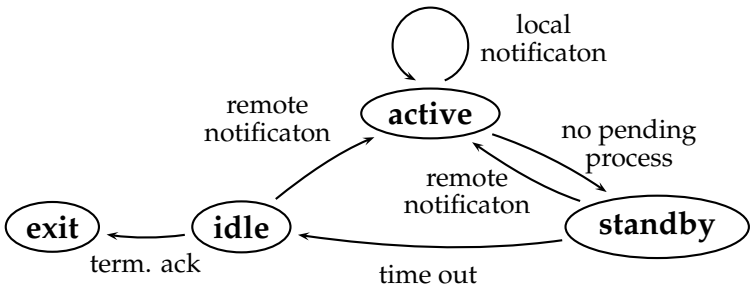


Fig. 28: The state transition diagram of a client.

The master is used to decide the termination of a simulation. It keeps a copy of the states of all clients. Only when the local copies of states for all clients are `idle`, the master will send a termination acknowledgment to all clients to notify them about the termination of the simulation.

This approach effectively prevents abnormal termination of the functional simulation if the timeout is larger than the worst-case latency of a remote notification, i.e., the time for the channel manager to transmit a data item from a remote output channel to a remote input channel. In our experiment, we empirically set the timeout to 20 *ms*.

2.6.5 Experimental Results

This section analyzes the performance of our method (denoted as PSC) by applying it to distribute the functional simulation of an MPEG-2 video decoder [Int]. The MPEG-2 video decoder application is specified as a process network with 20 parallel processes, as depicted in Fig. 11(b) in Section 2.4.5. It decodes 15 *s* of a compressed video sequence, with a resolution of 704 × 576 pixels and a frame rate of 25 *fps*.

The simulation hosts have identical configurations: each machine has two AMD Opteron 2218 dual core processors running at 2.6 *GHz*, i.e. four cores in total. The operating system is Debian Linux with kernel version 2.6.23. The simulation hosts are connected by Gigabit Ethernet where round trip time measured with `ping` was below 0.1 *ms*.

We measure two time criteria: a) simulation time (`Sim.`), corresponding to the wall-clock time from the start of the simulation until its completion, and b) accumulated computing time (`Comp.`), indicating how much accumulated computation workload the application has actually used, i. e. , it reflects the simulation time if only a single computing core would be available. All time values are rounded to seconds.

Experimental Results

Tab. 2 shows the results of six different cases, using different libraries and a different number of simulation hosts (`#Hosts`) and simulators per host (`#Sims`).

Case	Simulation	#Hosts	#Sims	Sim.	Comp.
1	**Pthread**	1	1	31 min 46 s	80 min 47 s
2	**SSC**	1	1	12 min 13 s	12 min 13 s
3	**PSC**	1	1	12 min 14 s	12 min 14 s
4	**PSC**	1	4	5 min 21 s	12 min 34 s
5	**PSC**	4	1	5 min 23 s	12 min 42 s
6	**PSC**	5	4	2 min 41 s	13 min 24 s

Tab. 2: Runtimes of the MPEG-2 example.

`Case 1` shows the performance of a POSIX Threads (Pthreads) implementation, where each process is implemented as a POSIX thread. A Pthreads implementation, in principle, can make use of all available CPU cores simultaneously. The experiment results depicted in Tab. 2 show that the Pthreads simulation is, however, the slowest one. Although the Pthreads implementation can make use of all available computing core to share the simulation workload – the simulation time is about 3 times smaller than the computation time in `Case 1`, the simulation

time is still rather slow due to the massive system calls which incur huge context-switch overhead. In contrast to Pthreads, the stand-alone SystemC (SSC) simulation is much faster because of the low switching and synchronization overheads for user-level threads. A negative consequence is that the stand-alone SystemC simulation can only use one core — `Case 2` has equal simulation and computation time. In `Case 3`, we show the overhead of the delta-cycle approach by using one simulator. As the result shows, the overhead of the delta-cycle execution is considerably small, i.e. 100 *ms*.

To demonstrate the speedup of the new method, the application is mapped onto four simulators, as depicted in Fig. 11(b). In `Case 4`, all four simulators run on the same simulation host, using all four CPU cores, whereas in `Case 5`, the same four simulators run on different simulation hosts, requiring communication through the network. In both cases, speedup by a factor of more than two is achieved. Compared to `Case 3`, the slight increase in computation time is due to additional communication and synchronization cycles. The overhead is only 2 % of the overall computation time, which is remarkable since about 3 *GB* of data is transferred during the simulation. Note that the simulation time is not reduced by a factor of four, although four times more resources are used. The reason is that the computation workload is not equally distributed to the four simulators. As shown in Tab. 3, `sim4` dominates the simulation, while the others use only about half of the time. A better distribution of the computation workload would further improve the performance of the simulation.

Simulator	Comp.	Usage
sim1	2 min 58 s	55 %
sim2	2 min 09 s	40 %
sim3	2 min 09 s	40 %
sim4	5 min 17 s	99 %

Tab. 3: Computation time of the single simulators in case 4.

The maximum speedup of 4.5 is achieved in `Case 6`, where the 20 simulators, one for each process, are distributed among five simulation hosts. In this case, each simulator owns its CPU core. As expected, the speedup is not linear to the hardware resources used. For further speedup, one needs to change the granularity of the process network of the application.

2.7 Summary

In this chapter, we present a programming model, suitable for developing streaming multi/many-core embedded systems. Our programming model follows the Y-chart paradigm, separating the concerns of application and architecture and of computation and communication. For application modeling, we adopt the Kahn process network model of computation. For specifying the application, architecture, mapping descriptions, we design a hybrid programming syntax, i.e., C/C++/XML. To assist the design of future many-core embedded systems, an iterator technique is developed by which a specified system can be arbitrarily scaled in a parametrized manner. To avoid the costly communication and synchronization overhead incurred by large scale process networks, we propose a variation of the FIFO syntax of KPN, namely windowed FIFO. We also develop a distributed functional simulation as a proof-of-concept runtime environment for our programming model. The results presented in this chapter lead to the free available embedded system software design toolchain in [dol].

Our programming model specifies an embedded system at system level, allowing a generic and versatile representation of a system. The application specified in our programming model can be synthesized to different multi/many-core platforms. Platforms currently supported are IBM Cell Broadband Engine [HSH+09] (Yellow Dog Linux [yel] and GCC), ATMEL Shapeotto [HPB+] (DNA operating system [GP09] and C99), and MPARM [HHBT09b] (RTEMS real-time operating system [RTE] and C++).

Finally, our programming model offers more than programmability and scalability. In the next chapter, we present two performance estimation techniques. Both of these techniques can be coupled to our programming model and provide quantitative verifications for systems specified in our programing model.

3

Performance Evaluation

Performance is a prime metric in embedded system design as embedded systems are often subject to tight timing constraints such as throughput, latency, and response time. Analyzing timing behavior and reliably predicting performance characteristics are essential to verify system properties and support important design decisions. Modern multi-core and future many-core (often heterogeneous) architectures of embedded systems are, however, characterized by a large design space as there is a large degree of freedom in the allocation of concurrent hardware components, the partitioning of parallel application tasks and their binding to hardware components, and the choice of appropriate resource allocation schemes. Because of the overall system complexity, inspecting the exact timing behavior of every design candidate of such an embedded system is computationally prohibitive. Therefore, fast evaluation methods in an early design stage are crucial for the exploration of the large design space.

Simulation-based methods for performance estimation are widely used in industry. There are commercial tools such as Cadence Virtual Component Co-design (VCC) [VCC] and CoWare Virtual Platform Analyzer (VPA) [CoW], which support cycle/instruction-accurate simulation of complete HW/SW systems. Besides commercial tool suites, there also exist open-source simulation frameworks that can be applied to performance estimation, for instance, SystemC [sys]. The main advantage of simulation-based approaches is their large and customizable modeling scope, which allows the modeling of systems at various abstraction levels. Simulation-based techniques, however, often suffer from long runtimes and high setup effort for each new architecture and mapping (process

allocation and scheduling discipline). Furthermore, worst-case bounds on system properties like throughput and end-to-end delay cannot be obtained, in general, because of the exhaustive corner-case coverage cannot be guaranteed in simulation.

Formal analytic methods such as symbolic timing analysis for systems (SymTA/S) [RE02], holistic analysis in the modeling and analysis for systems (MAST) [PEP+04], and modular performance analysis (MPA) [TCN00, CKT03, WT06c], are viable alternatives, which provide exhaustive corner-case coverage and have a low setup effort. Therefore, to determine guaranteed performance limits, analytic methods are preferable. The major disadvantage of analytic methods is that they typically lack of effective ways to model complex interactions and state-dependent behavior, resulting in pessimistic bounds for certain scenarios.

The increasing number of integrated cores for future embedded systems accentuates the deficiencies of both simulation-based and analytic approaches. In the case of simulations, the computing demand is super-linear to the number of cores to be simulated. In the case of analytic methods, more cores involved result in more complex interactions and correlations, which exceed the modeling scope of current analytic methods. The question hereby is how to accurately estimate the performance for large-scale many-core embedded systems within a reasonable time frame.

3.1 Overview

In this chapter, we investigate both analytic and simulation-based techniques for the performance estimation of multi/many-core embedded systems. In particular, we target streaming embedded systems that exhibit complex interferences and correlations between data streams, computation, and communication. We present techniques to tackle the complex interference and correlations within these systems such that accurate performance estimation can be conducted.

To analytically estimate the performance of a multi/many-core embedded system, we employ the modular performance analysis (MPA) framework [TCN00, CKT03, WT06c] that is based on real-time calculus (RTC). In RTC-MPA, a system is decomposed into a network of abstract computation and communication components and is analyzed by the flow of event streams through this network of abstract components. The decompositional approach for performance analysis in RTC-MPA, nevertheless, incurs an information loss among components. A typical example is that the timing correlations between event streams cannot be effectively modeled, which leads to pessimistic analysis results. To

tackle this problem, we present an extension to RTC-MPA which models correlations between event streams originated from the same source. We show the applicability of our methods by analyzing a real-life multimedia application, namely an M-JPEG encoder. Although this method is implemented within the RTC-MPA framework, the idea behind it can be applied to other modular analysis frameworks, such as SymTA/S, as well.

In comparison to analytic methods, simulation-based techniques can naturally model this kind of correlations, because the exact timing information of each sub-stream is preserved during the simulation. Therefore, we also investigate simulation-based techniques. The focus of our work here is to find a good trade-off between simulation speed and accuracy. On the one hand, the technique should provide accurate estimation close to classical cycle/instruction-accurate simulations. On the other hand, the runtime of this technique should be acceptable for modern multi-core systems and scale to future many-core systems. We propose a trace-based simulation framework at system level. By abstracting the functionality of an application into coarse-grain traces and simulating these abstract traces, our framework can evaluate complex multi/many-core embedded systems in a reasonable time, while considering different scheduling policies, memory allocations, and hierarchical communication schemes. We investigate the effectiveness of our approach by mapping an MPEG-2 decoding algorithm onto the ATMEL Diopsis-940 platform [Pao06]. We demonstrate the scalability of our approach by mapping a scaled version of the MPEG-2 algorithm onto a system with up to 16 cores.

3.2 Modular Performance Analysis

The modular performance analysis (MPA) [TCN00, CKT03, WT06c] framework that is based on real-time calculus (RTC) has been developed to model and analyze the performance of distributed real-time multimedia and digital signal processing systems. In the RTC-MPA framework, a system is decomposed into a network of abstract computation and communication components. Specifically, the performance model of a system is composed of single abstract components that model (a) resources such as buses and processors, (b) event streams that trigger resource components, and (c) resource sharing methods. The approach uses real-time calculus which itself is based on network calculus [LT01]. In particular, arrival curves $\alpha(\Delta)$, service curves $\beta(\Delta)$ and workload curves $\gamma(\Delta)$ [MKT04] model certain timing properties of event streams, the capability of architecture elements, and the execution requirement of event streams, respectively, as shown in Fig. 29. A more detailed

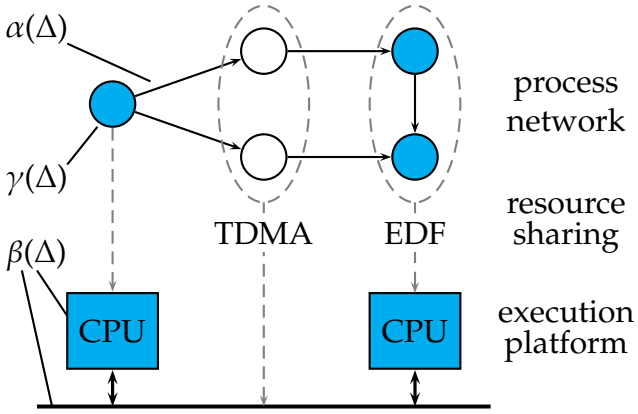


Fig. 29: Elements of RTC-MPA.

description of these elements is given in the following sections.

One drawback of using RTC-MPA for performance analysis is that the decomposition, nevertheless, leads to information loss. Considering a split-join scenario that is typical for streaming applications, an input stream is split into several sub-streams that will join again after being processed separately. Obviously, the sub-streams are often highly correlated. Due to the decomposition in RTC-MPA, these correlations cannot be effectively modeled. The phase information between the sub-streams with respect to the original input stream, for instance, is lost during the analysis. In this work, we are interested in the correlations in such a split-join scenario. We investigate techniques to model the correlations between streams that originate from the same input stream such that accurate performance analysis in terms of processing delays and buffer space usage can be conducted.

3.2.1 Related Work

Although simulation-based methods are the most commonly used techniques for performance evaluation of embedded systems, formal analytic methods are required, for instance, to determine guaranteed performance limits and provide corner-case coverage These methods provide hard bounds that are needed to verify embedded systems with respect to hard real-time requirements. In addition, performance estimation using analytic methods requires lower setup effort and smaller computational demand, which is suitable for fast design space exploration at early design stages.

Several models and methods for analytic performance analysis of embedded systems have been developed. Holistic scheduling,

for instance, extends well-known classical scheduling techniques to distributed systems [PGH98, TC94], combining the analysis of processor and bus scheduling. The modeling and analysis suite for real-time applications (MAST) [GHGGPGDM01] integrates a set of such holistic analysis techniques. A more general approach to extend the concepts of classical scheduling theory to distributed systems is presented in the symbolic timing analysis of systems (SymTA/S) framework [RE02]. In contrast to holistic approaches, this method applies existing analysis techniques in a modular manner where single components of a distributed system are analyzed with classical algorithms, and the local results are propagated through the system by appropriate interfaces relying on a limited set of event stream models. The modular performance analysis (MPA) framework [TCN00, CKT03, WT06c], which extends the basic concepts of network calculus, presents a different analytic and modular approach for performance analysis that does not rely on the classical scheduling theory. It analyzes the flow of event streams through a network of computation and communication resources.

The disadvantage of analytic methods is that they are typically not able to model complex interactions and state-dependent behavior, resulting in pessimistic bounds for certain scenarios.

To achieve a shorter runtime in comparison to simulation-based methods and tighter performance bounds in comparison to analytic methods, hybrid methods that combine simulation and analysis have been proposed. In [LRD01], a trace-based simulation method was proposed, and in [KPBT06] a method is proposed to combine a SystemC-based cycle-accurate simulation [LAB+04] with an analytic technique [TCN00]. Although these hybrid methodologies can help to shorten the runtime of simulations, the problem of insufficient corner-case coverage is still present. Recently, a compositional and hybrid approach has been presented in [LPT09], which couples RTC-MPA and state-based models in the form of timed automata [AFM+02]. By defining a pattern to convert an abstract stream model, i. e. , periodic with jitter or time-interval based models used in real-time calculus, to a network of co-operating timed-automata and vice versa, the proposed method enables a holistic analysis of a system composed of both MPA models and timed-automata.

In this section, we focus only on an analytic method, targeting the correlation between event streams. There are several approaches [JRE04, WT05] available to model event patterns and correlations *within* single event streams. In these approaches, models are developed to tackle correlations between different event types and workloads. None of these existing analytic approaches is able to model correlations between event streams, however. We present a method that can be used to analyze the correlation between different streams within a split-join scenario. By the

time this method has been developed, it was the first approach to tackle this problem. Recently, work along this direction has also been presented in [SE09, PRT+10].

3.2.2 Introduction to RTC-MPA

This section presents the basic components of the RTC-MPA framework, i.e., the event stream model, the service model, and the workload model. We also sketch how to use these components to analyze a system.

3.2.2.1 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 $[s, t)$. While any R always describes *one* concrete trace, a 2-tuple $\alpha(\Delta) = [\alpha^u(\Delta), \alpha^l(\Delta)]$ of upper and lower *arrival curves* [Cru91a] provides an abstract event stream model that characterizes a whole class of (non-deterministic) event streams. $\alpha^u(\Delta)$ 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 Δ:

$$\alpha^l(t-s) \leq R(s, t) \leq \alpha^u(t-s) \quad \forall s < t \tag{3.1}$$

with $\alpha^l(\Delta) = \alpha^u(\Delta) = 0$ for $\Delta \leq 0$. Arrival curves substantially generalize traditional event models such as sporadic, periodic, periodic with jitter, or any other arrival pattern with non-deterministic timing behavior. Therefore, they are suited to represent complex characteristics of event streams in complex multi-core embedded systems. Examples of different arrival curves are depicted in Fig. 30.

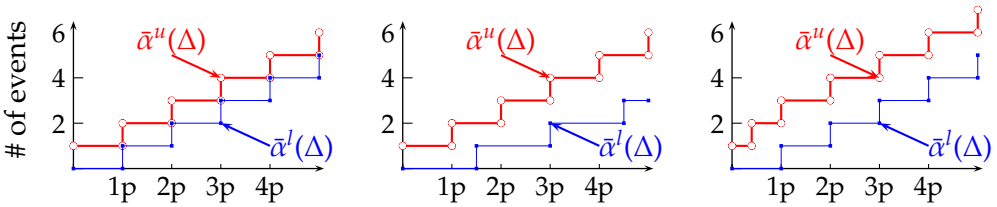


Fig. 30: Examples for arrival curves for: (a) periodic events with period p, (b) events with minimal inter-arrival distance p and maximal inter-arrival distance $p' = 1.5p$, and (c) events with period p and minimal inter-arrival distance $d = 0.4p$.

3.2.2.2 Resource Model

In a similar way, the capability of a computation or communication resource can be described by a cumulative function $C(s, t)$, defined as

the number of available resources, e. g. , processor or bus cycles, in the time interval $[s, t)$. To provide an abstract resource model for a whole set of possible resource behavior, we define a 2-tuple $\beta(\Delta) = [\beta^u(\Delta), \beta^l(\Delta)]$ of upper and lower *service curves*:

$$\beta^l(t-s) \leq C(s,t) \leq \beta^u(t-s) \quad \forall s < t \tag{3.2}$$

with $\beta^l(\Delta) = \beta^u(\Delta) = 0$ for $\Delta \leq 0$. Again, service curves substantially generalize classical resource models such as the bounded delay or the periodic resource model [SL04]. An example of the service curves for a communication resource is shown in the following.

Ex. 2: *Consider a bus with bandwidth B that implements the time division multiple access (TDMA) protocol. The length of individual TDMA slots is denoted by s_i and the TDMA cycle length is denoted by $\bar{c}$ with $\bar{c} \leq \sum s_i$. Then, the service curves representing a slot are given as:*

$$\beta_i^l(\Delta) = B \cdot \min\left\{ \lceil \Delta/\bar{c} \rceil \cdot s_i,\ \Delta - \lfloor \Delta/\bar{c} \rfloor \cdot (\bar{c} - s_i) \right\} \tag{3.3}$$

$$\beta_i^u(\Delta) = B \cdot \max\left\{ \lfloor \Delta/\bar{c} \rfloor \cdot s_i,\ \Delta - \lceil \Delta/\bar{c} \rceil \cdot (\bar{c} - s_i) \right\} \tag{3.4}$$

3.2.2.3 Workload Model

In the context of RTC-MPA, the arrival curve is event-based whereas the service curve is resource-based. To relate arrival and service curves, *workload curves* are used [MKT04]. The workload that an event stream imposes on a resource can be described by a cumulative function $W(s, t)$ defined as the number of clock cycles required to process event s (included) to event t (excluded) – here we suppose that events are successively numbered – on a computation or communication resource. We define a 2-tuple $\gamma(\Delta) = [\gamma^u(\Delta), \gamma^l(\Delta)]$ of upper and lower workload curves:

$$\gamma^l(t-s) \leq W(s,t) \leq \gamma^u(t-s) \quad \forall s < t \quad s, t \in \mathbb{N}_0 \tag{3.5}$$

Using the workload curve and its pseudo-inverse

$$(\gamma^u)^{-1}(w) = \sup\left\{ e : \gamma^u(e) \leq w \right\} \tag{3.6}$$

$$(\gamma^l)^{-1}(w) = \inf\left\{ e : \gamma^l(e) \geq w \right\} \tag{3.7}$$

arrival and service curves can be transformed from event-based to resource-based quantities and vice versa as follows:

$$\bar{\alpha}^l(\Delta) = \gamma^l\Big(\alpha^l(\Delta)\Big) \qquad \bar{\beta}^l(\Delta) = (\gamma^u)^{-1}\Big(\beta^l(\Delta)\Big) \tag{3.8}$$

$$\bar{\alpha}^u(\Delta) = \gamma^u\Big(\alpha^u(\Delta)\Big) \qquad \bar{\beta}^u(\Delta) = (\gamma^l)^{-1}\Big(\beta^u(\Delta)\Big) \tag{3.9}$$

In the simplest case, the workload of an event stream is characterized by its worst-case and best-case execution time measured in clock cycles and the workload curves would simply be:

$$\gamma^l(e) = \text{BCET} \cdot e \text{ [cycles]} \qquad (\gamma^l)^{-1}(x) = \lceil x/\text{BCET} \rceil \text{ [events]} \tag{3.10}$$

$$\gamma^u(e) = \text{WCET} \cdot e \text{ [cycles]} \qquad (\gamma^u)^{-1}(x) = \lfloor x/\text{WCET} \rfloor \text{ [events]} \tag{3.11}$$

3.2.2.4 Greedy Processing Component

The basic component of an RTC-MPA analysis model is the so-called greedy processing component (GPC). The semantics of a GPC can be described as follows: An incoming event stream, represented as a set of upper and lower arrival curves, flows into a FIFO buffer in front of the GPC. The events trigger the execution of the corresponding process while being restricted by the availability of resources that is represented as a set of upper and lower service curves. The outgoing event stream and remaining resource capacity can again be represented as a set of upper and lower curves. As has been shown in [CKT03], the output arrival curves α' can be determined as follows:

$$\alpha'^l(\Delta) = \min\left\{ \inf_{0\le\mu\le\Delta} \left\{ \sup_{\lambda>0}\{\alpha^l(\mu+\lambda) - \bar{\beta}^u(\lambda)\} + \bar{\beta}^l(\Delta-\mu) \right\}, \bar{\beta}^l(\Delta) \right\} \tag{3.12}$$

$$\alpha'^u(\Delta) = \min\left\{ \sup_{\lambda>0} \left\{ \inf_{0\le\mu<\lambda+\Delta} \{\alpha^u(\mu) + \bar{\beta}^u(\lambda+\Delta-\mu)\} - \bar{\beta}^l(\lambda) \right\}, \bar{\beta}^u(\Delta) \right\} \tag{3.13}$$

An upper bound of the maximum delay $d^{\max}$ experienced by an event and the maximal length $b^{\max}$ of the FIFO buffer of a GPC can be computed by the following relations, see also [LT01]:

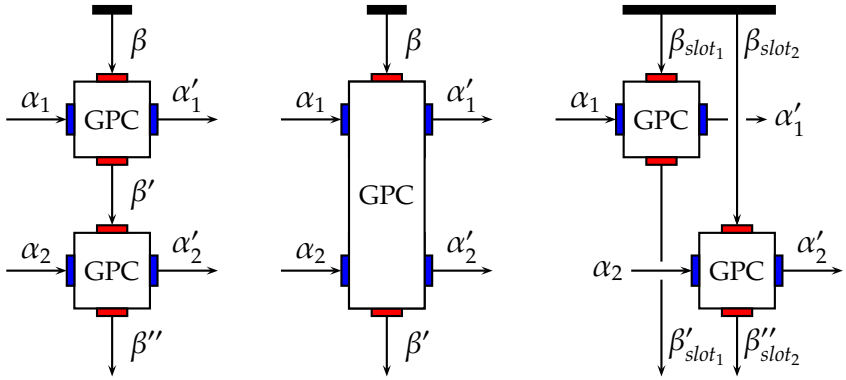
$$d^{\max} = \sup_{\lambda\ge 0} \left\{ \inf\{\tau \ge 0 : \alpha^u(\lambda) \le \bar{\beta}^l(\lambda+\tau)\} \right\} \tag{3.14}$$

$$b^{\max} = \sup_{\lambda\ge 0} \{\alpha^u(\lambda) - \bar{\beta}^l(\lambda)\} \tag{3.15}$$

3.2.2.5 System Analysis

With the aforementioned abstractions, a basic RTC-MPA model of a system can be constructed for performance analysis: Event streams are modeled by arrival curves, computation and communication resources by service curves, event processing in the system by GPCs, and the flow of event streams by interconnecting the GPCs. To complete the analysis model, the GPCs need to be connected by service curves based on the used resource sharing policies. Resource sharing policies currently supported by RTC-MPA include preemptive and non-preemptive fixed priority scheduling (FP) [WTVL06, HT07], rate monotonic scheduling

(RM) [WTVL06], time division multiple access (TDMA) [WT06b], earliest deadline first (EDF) [WT06a], and first-come first-serve (FCFS) [PRT+10]. Fig. 31 illustrates structures of the performance analysis model for three scheduling policies, which are relevant to this thesis.



(a) FP. (b) EDF. (c) TDMA.

Fig. 31: Modeling of three different scheduling policies in RTC-MPA. From left to right: preemptive fixed priority (FP), earliest deadline first (EDF), and time division multiple access (TDMA) policies.

By correctly interconnecting all service curves, the analysis model of a system can be obtained. A concrete example is depicted in Fig. 37 of Section 3.2.4. Based on the local analysis of single components, global system properties, such as end-to-end delays, buffer requirements, system throughput, and others, can be computed.

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

3.2.3 Analysis of Correlated Streams

The basic principle of RTC-MPA is to decompose a system into a network of abstract computation and communication components introduced above and analyze it by considering the flow of event streams through this network of abstract components. This decomposition incurs an information loss among components, leading to pessimistic analysis results for certain scenarios.

A typical example is the timing correlations between different event streams that originate from the same source. Consider a split-join scenario which is common in streaming applications: An input event stream is distributed to several sub-streams at a so-called `split` process and

later on sub-streams are merged again at a so-called `join` process, as shown in Fig. 32. Obviously, these sub-streams are highly correlated, for instance, the phases with respect to the original stream. Due to the compositional approach of RTC-MPA, the correlations between different sub-streams cannot be effectively modeled.

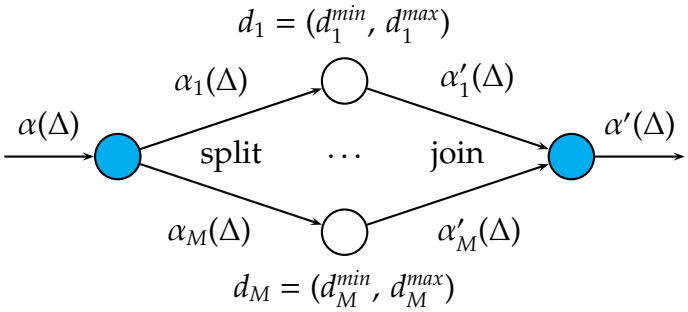


Fig. 32: RTC-MPA model representing the split-join scenario.

In order to conduct accurate performance analysis, we study stream correlations and develop methods to capture the corresponding effects in the performance analysis. In particular, we investigate stream correlations within a split-join scenario, as pictorially shown in Fig. 32. An input event stream α is split at a `split` process into M sub-streams, $M \geq 2$. Each sub-stream experiences a delay d_i (in case of not a constant delay, bounded by d_i^{min} and d_i^{max}) before it is merged at the `join` process with other sub-streams to form the output stream α'. Specifically, we study two different semantics of the `join` process, namely OR-semantics and ORDER-semantics.

3.2.3.1 Join Process with OR-Semantics

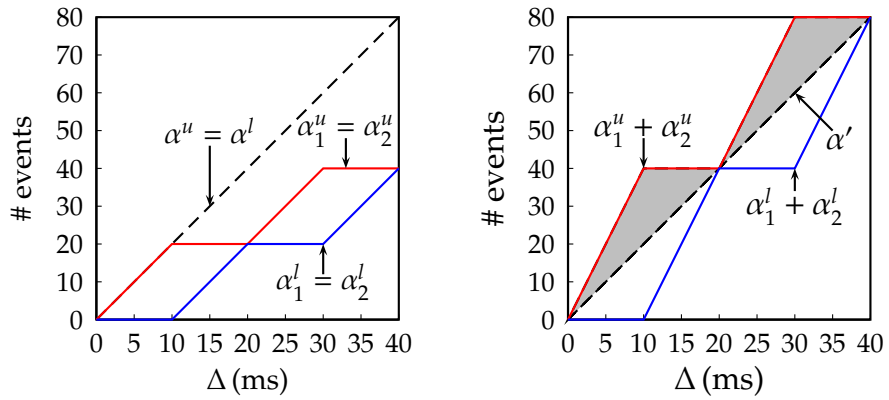
A `join` process with OR-semantics takes any event appearing on any of its input buffers and processes them in a first-come first-serve order. Without restricting the generality of our approach, we suppose that the `join` process in Fig. 32 just transfers input events to its output in zero time, i.e., without any resource usage. If there is additional processing necessary, this can easily be modeled by an additional process whose single input is connected to the output of the `join` process. At first, we define the split-join delay for a sub-stream α_i.

Def. 1: **(Split-Join Delay)** *A split-join delay d_i of α_i is the time distance between the time when an event appears at the output of the* `split` *process and the time when it reaches the input buffer of the* `join` *process and is available to be processed. In case of a non-constant delay, d_i is modeled with an upper and a lower bound, denoted as d_i^{min} and d_i^{max}.*

One can, for instance, apply (3.14) to obtain upper bounds on this delay. For more complex operation chains between the `split` and `join` processes, the usual RTC-MPA analysis can be used to determine d_i^{min} and d_i^{max}.

Current analysis methods cannot take into account the correlation between streams α_i, $i = 1, ..., M$. After being split up from α, they are considered as independent entities. As will be seen in the next example, this leads to a reduced accuracy of the performance analysis, i.e., delays and buffer sizes are considerably overestimated.

Ex. 3: *Consider a simple split-join scenario in Fig. 32 with only two sub-streams, i.e., $M = 2$. A simple TDMA scheme alternatively serves the two output streams with a fixed window size of 10ms. Consider a simple periodic input stream with two events per ms. The corresponding arrival curve α and the two output streams α_1 and α_2 are shown in Fig. 33(a). The two streams are processed separately and have delays d_1 and d_2, respectively. Even in the most simple case where $d_1 = d_2 = 0$, the derived output curve $\alpha' = \alpha_1 + \alpha_2$ is overly pessimistic in comparison to the correct result $\alpha' = \alpha$. The shadowed part in Fig. 33(b) shows the loss caused by previous methods where all subsequent subsystems assume that there are up to 40 events within a time interval of 10ms instead of just 20.*



(a) Split. (b) Join.

Fig. 33: Arrival curves for the split-join scenario in Ex. 3.

The reason for the loss in accuracy is that the information about timing correlations is lost during the analysis. The following theorem states a method, which takes into account time correlations between different sub-streams within the split-join scenario.

Thm. 2: *Assume an event stream that is constrained by arrival curve $\alpha = [\alpha^u, \alpha^l]$ is split into $M \geq 2$ sub-streams that will be combined in a `join` process with OR-semantics. The split-join delay of each sub-stream is bounded by a tuple*

$d_i = [d_i^{min}, d_i^{max}], \forall i \in M$. *Then the output of the* `join` *process is an event stream that can be bounded by the arrival curves*

$$\alpha'^u(\Delta) = \min\left\{\sum_{i=1}^{M} \alpha_i^u(\Delta + d_i^{max} - d_i^{min}),\ \alpha^u(\Delta + d^{max} - d^{min})\right\} \tag{3.16}$$

$$\alpha'^l(\Delta) = \max\left\{\sum_{i=1}^{M} \alpha_i^l(\Delta + d_i^{min} - d_i^{max}),\ \alpha^l(\Delta + d^{min} - d^{max})\right\} \tag{3.17}$$

where

$$d^{max} = \max_{i \in M}\{d_i^{max}\}$$
$$d^{min} = \min_{i \in M}\{d_i^{min}\}$$

Proof. We present the proof for the upper arrival curve α'^u as follows. The lower curve can be derived similarly.

We first consider a case with two sub-streams, i.e., $M = 2$. For any interval Δ of α'^u starting at time t_Δ with a length of l_Δ, events of α_1 that will arrive during that interval Δ at the `join` process need to be within the interval Δ_1 starting at time $t_\Delta - d_1^{max}$ with a length of $l_\Delta + d_1^{max} - d_1^{min}$. Similarly, the interval Δ_2 for α_2 that contributes events to Δ starts from time $t_\Delta - d_2^{max}$ with a length $l_\Delta + d_2^{max} - d_2^{min}$. A graphical view of these two cases is shown in Fig. 34. Combining these two cases, an upper bound of α'^u can be computed as:

$$\begin{aligned}\alpha'^u(\Delta) &= \alpha_1^u(\Delta_1) + \alpha_2^u(\Delta_2) \\ &= \alpha_1^u(\Delta + d_1^{max} - d_1^{min}) + \alpha_2^u(\Delta + d_2^{max} - d_2^{min})\end{aligned} \tag{3.18}$$

This bound is, however, not always tight, as shown in Ex. 3. If we consider the original phase information of these two sub-streams, we can compute another valid bound. We consider three different cases for the relative position of Δ_1 and Δ_2 with respect to the original stream:

- Δ_1 and Δ_2 are disjoint.

  When Δ_1 and Δ_2 are disjoint, there are two possible cases: Δ_1 is closer to Δ, i. e., $d_1^{max} + l_\Delta \leq d_2^{min}$ or the other way round, i. e., $d_1^{max} + l_\Delta > d_2^{min}$. The first case is pictorially shown in Fig. 34(a). We can compute α'^u as follows:

  - $d_1^{max} + l_\Delta \leq d_2^{min}$

$$\begin{aligned}\alpha'^u(\Delta) &= \alpha^u(\Delta + d_2^{max} - d_1^{min}) - \alpha^l(d_2^{min} - \Delta - d_1^{max}) \\ &\leq \alpha^u(\Delta + d_2^{max} - d_1^{min})\end{aligned} \tag{3.19}$$

- $d_1^{max} + l_\Delta > d_2^{min}$

$$\begin{aligned}\alpha'^u(\Delta) &= \alpha^u(\Delta + d_1^{max} - d_2^{min}) - \alpha^l(d_1^{min} - \Delta - d_2^{max}) \\ &\leq \alpha^u(\Delta + d_1^{max} - d_2^{min}) \end{aligned} \tag{3.20}$$

- Δ_1 and Δ_2 intersect.

  When Δ_1 and Δ_2 intersect, there are again two cases: Δ_1 is closer to Δ, i.e., $d_1^{min} < d_2^{min} < d_1^{max} + l_\Delta$ or the other way round, i.e., $d_2^{min} < d_1^{min} < d_2^{max} + l_\Delta$. Fig. 34(b) again shows the first case in which Δ_1 is closer to Δ. We compute α'^u as follows:

  - $d_1^{min} < d_2^{min} < d_1^{max} + l_\Delta$

$$\alpha'^u(\Delta) = \alpha^u(\Delta + d_2^{max} - d_1^{min}) \tag{3.21}$$

  - $d_2^{min} < d_1^{min} < d_2^{max} + l_\Delta$

$$\alpha'^u(\Delta) = \alpha^u(\Delta + d_1^{max} - d_2^{min}) \tag{3.22}$$

- One of Δ_1 and Δ_2 completely overlaps the other.

  For the case of a complete overlap, we again have two cases: Δ_1 overlaps Δ_2, i.e., $d_2^{min} \leq d_1^{min}$ and $d_2^{max} \geq d_1^{max}$ or the other way round, i.e., $d_1^{min} \leq d_2^{min}$ and $d_1^{max} \geq d_2^{max}$. Fig. 34(c) shows the first case in which Δ_1 overlaps Δ_2. We derive α'^u as follows:

  - $d_2^{min} \leq d_1^{min}$ and $d_2^{max} \geq d_1^{max}$

$$\alpha'^u(\Delta) = \alpha^u(\Delta + d_2^{max} - d_2^{min}) \tag{3.23}$$

  - $d_1^{min} \leq d_2^{min}$ and $d_1^{max} \geq d_2^{max}$

$$\alpha'^u(\Delta) = \alpha^u(\Delta + d_1^{max} - d_1^{min}) \tag{3.24}$$

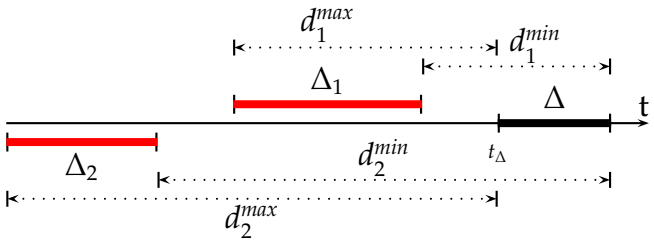
Combining (3.19)–(3.24), we have

$$\alpha'^u(\Delta) = \alpha^u(\Delta + d^{max} - d^{min}) \tag{3.25}$$

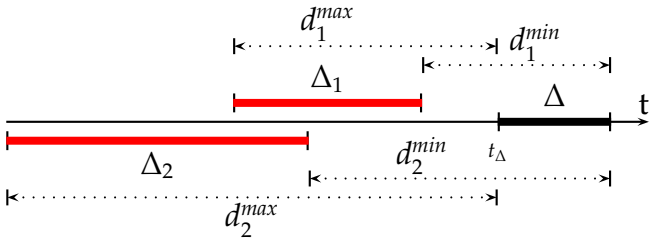
where $d^{max} = \max\{d_1^{max}, d_2^{max}\}$ and $d^{min} = \min\{d_1^{min}, d_2^{min}\}$. The computed α'^u from (3.25) is also a valid upper bound for α'^u. With (3.18) and (3.25), we have

$$\alpha'^u(\Delta) = \min\left\{\sum_{i=1}^{2} \alpha_i^u(\Delta + d_i^{max} - d_i^{min}),\ \alpha^u(\Delta + d^{max} - d^{min})\right\} \tag{3.26}$$

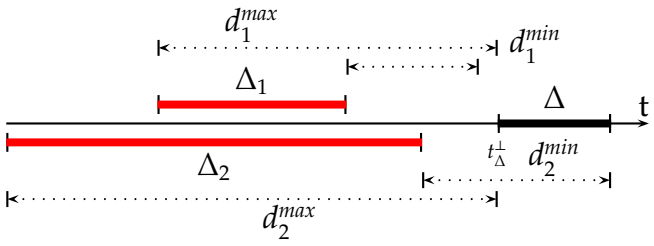
Therefore, the theorem holds for $M = 2$.



(a) Δ_1 and Δ_2 are disjoint.



(b) Δ_1 and Δ_2 intersect.



(c) Δ_2 overlaps Δ_1.

Fig. 34: Illustration for the proof of Theorem 2 in the case of $M = 2$.

For cases $M > 2$, the proof is similar to $M = 2$. For each Δ of α'^u, we can always compute Δ_i for α_i, $i \in M$. Moreover, based on enumerations similar to (3.19)–(3.24), the computed $\Delta + d^{max} - d^{min}$ defines another upper bound for the interval in which events from the original stream will arrive within Δ of α'^u. Therefore, the theorem holds.

□

3.2.3.2 Join Process with ORDER-Semantics

In OR-semantics, the `join` process greedily takes events from any of its inputs whenever events are available. Thus, the order of events with respect to the one in the original stream α is not guaranteed. Preserving the order of events in α' with respect to α is, however, necessary for certain cases of streaming applications, e. g., video codec applications.

Considering the parallel M-JPEG decoder in [HSH+09], for instance, an encoded M-JPEG video is dispatched frame-by-frame to different hardware components for decompression and the decoded frames are collected to form a video stream. In this case, preserving the frame order is mandatory in order not to scramble the resulted video.

To preserve the original order, we investigate an ORDER-semantics for the `join` process. Again, we assume that the `split` and `join` processes are infinitely fast. The split-join delay d_i for a given sub-stream α_i, however, needs more considerations. We construct d_i as follows.

Def. 2: **(Processing Delay)** *Processing delays $pd_i^{\max}$ and $pd_i^{\min}$ for events in an event stream α_i define the maximal and minimal time by which an event of α_i transfers from the output of the* `split` *process to an input buffer of the* `join` *process.*

$pd_i^{\max}$ and $pd_i^{\min}$ can be considered as the maximal and minimal accumulated processing time between the `split` and `join` processes. In the case of OR-semantics for the `join` process, they are equal to the split-join delays d_i^{max} and d_i^{min}. Again, one can apply (3.14) to obtain upper bounds on this delay. For more complex operation chains between the `split` and `join` processes, the usual RTC-MPA analysis can be used to determine lower and upper bounds on the processing delays.

Def. 3: **(Split interval)** *Split intervals $sd_{i,j}^{max}$ and $sd_{i,j}^{min}$ of sub-stream α_i are defined as the maximal and minimal relative time differences between an event in α_i and the closest recent one in sub-stream α_j.*

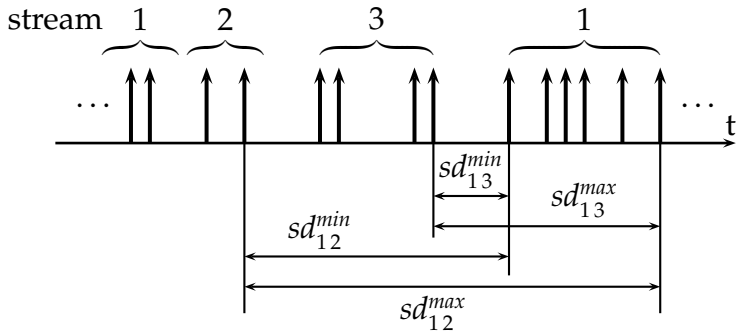


Fig. 35: Illustration of the split interval.

An example illustrating the split intervals is shown in Fig. 35. Obviously, these intervals depend on the distribution policy of the `split` process. Let us consider a periodic event stream with period p and a `split` process with a TDMA policy that alternatively distribute events to sub-streams in the order 1, 2, ... M, 1, 2 ..., for instance. The split intervals then are:

$$sd_{ij}^{max} = sd_{ij}^{min} = (i - j) \bmod M \cdot p \tag{3.27}$$

With the processing delays and split intervals, we can compute the split-join delays for sub-stream α_i. Intuitively, an event in sub-stream α_i that can be processed by the `join` process must fulfill two conditions: a) it reaches the input buffer of the `join` process, and b) all previous events with respect to the original order in the input stream α have been processed. Therefore, the maximal and minimal split-join delays are computed as:

$$d_i^{max} = \max_{j \in M \wedge j \neq i} \left\{ pd_i^{max}, pd_j^{max} - sd_{ij}^{min} \right\} \tag{3.28}$$

$$d_i^{min} = \max_{j \in M \wedge j \neq i} \left\{ pd_i^{min}, pd_j^{min} - sd_{ij}^{max} \right\} \tag{3.29}$$

These values can now be used in (3.16) and (3.17) in order to determine $\alpha'(\Delta)$. An example for (3.28) is illustrated in Ex. 4.

Ex. 4: *Again, consider the scenario in Ex. 3. The input event stream α is split into two sub-streams, namely α_1 and α_2. The event $e_{1,j}$ of α is dispatched to sub-stream α_1 at time t_2 and its maximal processing delay is pd_1^{max}. The closest previous event that has been put to α_2, namely $e_{2,i}$, has the maximum processing delay pd_2^{max}. It will reach the* `join` *process at time t_4. Then $e_{1,j}$ has to wait in the input buffer of the* `join` *process for at most $t_4 - t_3$ time units, before it can be processed by the* `join` *process.*

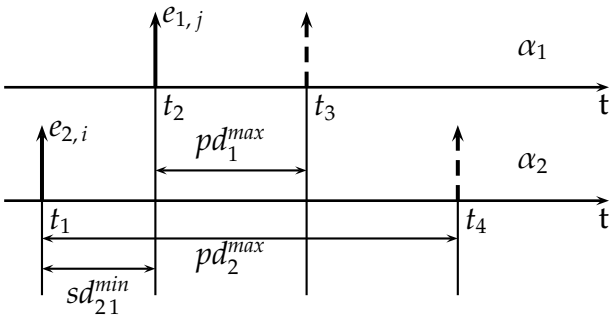


Fig. 36: Illustration of the maximum split-join delay.

Similarly, the maximal waiting time for any event of sub-stream α_i at the `join` process can be computed as:

$$bd_i^{\max} = \max_{j \in M \wedge j \neq i} \{0, pd_j^{max} - pd_i^{min} - sd_{ij}^{min}\} \tag{3.30}$$

3.2.4 Experimental Results

In this section, we apply our method to analyze a concrete system, namely a modified M-JPEG encoder [Ste04] and demonstrate the effectiveness of proposed methods.

3.2.4.1 Experimental Setup

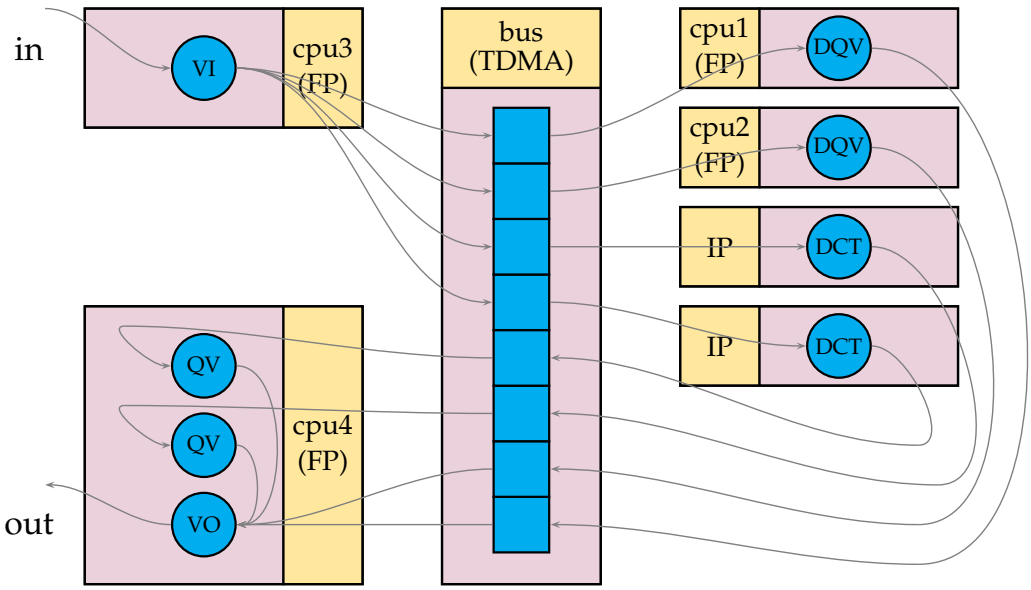
Like traditional M-JPEG encoders, the modified M-JPEG encoder compresses a sequence of frames by applying JPEG [PM93] compression to each frame. Because of the inherent parallelism in the JPEG algorithm, a frame can be split into macroblocks that can be compressed in parallel by concurrent hardware resources. The process network of the M-JPEG encoder is shown in the circular nodes in Fig. 37(a). Process `VI` splits an input image stream into macroblocks, scans macroblocks row by row, and then dispatches macroblocks to four different sub-streams. In two of the four sub-streams, processes `DQV` perform discrete cosine transform (DCT), quantization(QT), and variable length encoding (VLE). In the other two sub-streams, the DCT, QT, and VLE operations are decoupled into two processes, namely processes `DCT` and `QV`. The four sub-streams are merged in the process `VO` to form an encoded stream.

We map this process network on a 4-core platform, as graphically shown in Fig. 37(a). The `VI` process and two identical `DQV` processes execute on individual programmable cores, namely `cpu3`, `cpu1`, and `cpu2`, respectively. The two `DCT` processes use dedicated hardware IP-Cores. The two `QV` and `VO` processes run on a programmable core `cpu4` with a fixed-priority scheduling. The specifications of the involved hardware resources are listed in Tab. 4. The `IP-Core` conducts a hardware DCT with a constant delay for the processing of each macroblock. The `TDMA bus` has 8 slots with a slot length of 2048 cycles. One input frame contains 128 macroblocks, and one macroblock consists of 8×16 pixels (256 32-bit words).

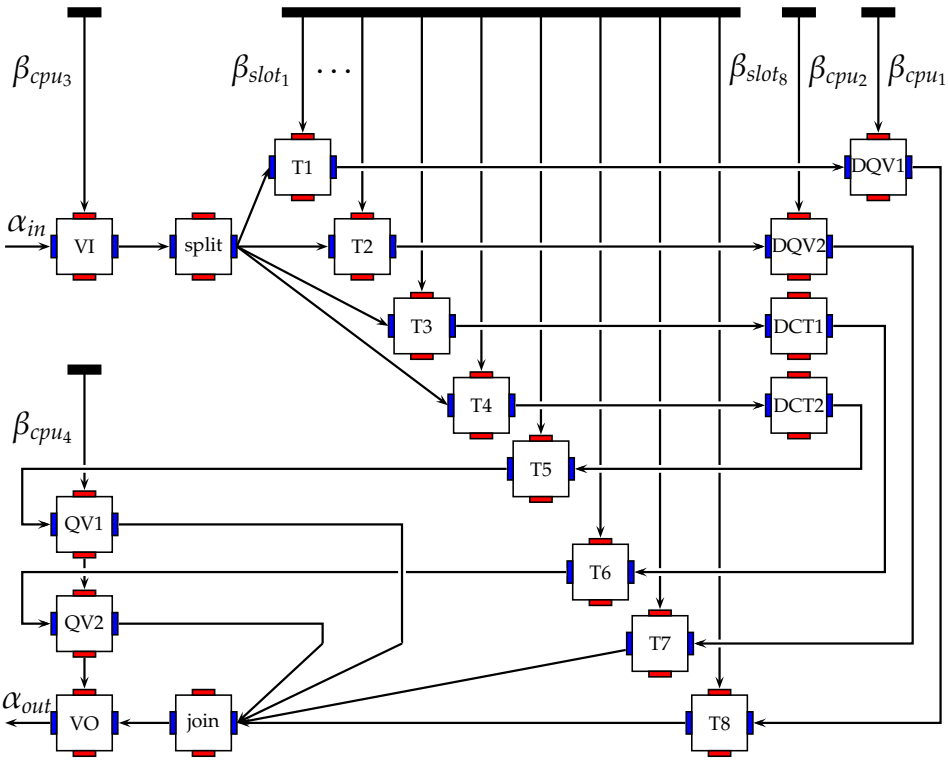
Hardware	Frequency [MHz]	Throughput [bytes/cycle]	other features
TDMA bus	100	4	8 slots, 2048 cycles each
IPCore	100	4	94-stages pipeline
cpu	100	N/A	N/A

Tab. 4: Hardware specification of the architecture.

We apply two different splitting policies to the `VI` process, namely modulo and block distributions. In the case of modulo distribution,

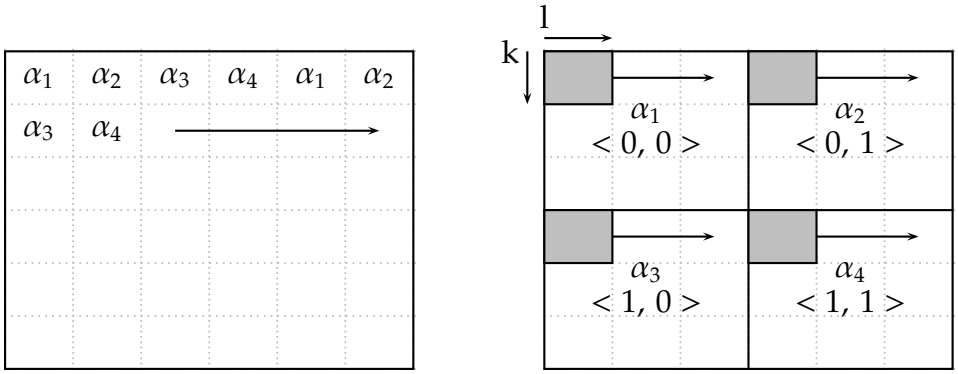


(a) System view of the M-JPEG encoder.



(b) The corresponding RTC-MPA model.

Fig. 37: M-JPEG encoder.



(a) Modulo distribution. (b) Block distribution for 2×2 grids.

Fig. 38: Macroblock distribution policies for the `VI` process, where a dotted rectangle represents a macroblock.

macroblocks are dispatched alternately to the four sub-streams, as shown in Fig. 38(a). We can compute the split intervals using (3.27) with $M = 4$.

In the case of the block distribution, each frame in the frame stream is split into $M_b \times M_b$ grids and macroblocks in grid $< k_i, l_i >$ will be dispatched to sub-stream α_i. Since the process `VI` scans macroblocks always row by row, how to compute split intervals is not obvious. For a frame stream with period p and $N_1 \times N_2$ macroblocks per frame, the split intervals for the grid $< k_i\ l_i >$ to the grid $< k_j\ l_j >$ can be computed as:

$$sd_{ij}^{max} = \begin{cases} \left(\frac{N_1}{M_b} \cdot ((l_i - l_j) \bmod M_b)\right) \cdot p & k_i = k_j \\ \left((l_i - l_j)\frac{N_1}{M_b} + (k_i - k_j)\frac{N_1 \cdot N_2}{M_b}\right) \cdot p & k_i > k_j \\ +\infty & k_i < k_j \end{cases} \tag{3.31}$$

$$sd_{ij}^{min} = \begin{cases} \left(1 + \frac{N_1}{M_b} \cdot ((l_i - l_j) \bmod M_b - 1)\right) \cdot p & k_i = k_j \\ \left(1 + (M_b + l_i - l_j - 1)\frac{N_1}{M_b} + (k_i - k_j - 1)\frac{N_1 \cdot N_2}{M_b}\right) \cdot p & k_i > k_j \\ +\infty & k_i < k_j \end{cases} \tag{3.32}$$

Intuitively, when $k_i = k_j$, i. e. , grids $< k_i, l_i >$ and $< k_j, l_j >$ are at the same horizontal row, the maximal split interval sd_{ij}^{max} is the horizontal distance between the two right-most macroblocks of $< k_j, l_j >$ and $< k_i, l_i >$ in a frame, i. e. , the number of grids between α_j and α_i, computed as $\frac{N_1}{M_b}(l_i - l_j)$. Correspondingly, the minimal split interval sd_{ij}^{min} is the horizontal distance between the last macroblock of grid $< k_j, l_j >$ and the first macroblock of $< k_i, l_i >$ located in a frame, i.e, $1 + \frac{N_1}{M_b}(l_i - l_j - 1)$. When $k_i > k_j$, we again

count the distance between the last macroblock of the grid $< k_j, l_j >$ and the last/first macroblocks of the grid $< k_i, l_i >$ for the maximal/minimal split intervals, respectively. In this case, the rows of grids between l_i and l_j needs to be counted as well. Therefore, for the maximal and minimal split intervals, there are $(k_i - k_j)\frac{N_1 \cdot N_2}{M_b}$ and $(k_i - k_j - 1)\frac{N_1 \cdot N_2}{M_b}$ more macroblocks need to be considered than the case of $k_i = k_j$, respectively. When $k_i < k_j$, we assume positive infinity, because $k_i < k_j$ represents the interval between two frames. An illustrated example for a 2×2-grid distribution is shown in Fig. 38(b).

In our experiment, an image frame contains 16×8 ($N_1 = 16$ and $N_2 = 8$) macroblocks and we set $M_b = 2$. With this information, we construct the RTC-MPA model for the M-JPEG encoder. The analytic model is shown in Fig. 37(b). As shown in the figure, a GPC component is created for each process in the process network. These GPCs are interconnected with eight GPCs modeling the communication over the TDMA bus. In addition, a `split` and a `join` component is added to model the correlations. For the `split` component, we consider the two aforementioned distribution policies. For the `join` component, we consider three scenarios, i.e., no correlation, OR-semantics, and ORDER-semantics. The results are shown in the subsequent figures.

3.2.4.2 Results

In the first experiment, we consider a case where the `split` component uses modulo distribution policy, the `join` component uses OR-semantics, and the input frame rate is set to 6 *frames/s*. We compare the output arrival curves of the `join` component for cases with and without consideration of correlations. The resulting arrival curves are shown in Fig. 39. As shown in the figure, we obtain tighter bounds (solid lines) for both upper and lower arrival curves when we consider the correlations between sub-streams. The maximal inter-arrival time between two macroblocks in α' is also depicted in Fig. 39. The maximal inter-arrival time when considering correlations, i.e., 3.46 *msec* is a factor of 2 smaller than the inter-arrival time when ignoring correlations, i.e., 7.63 *msec*.

In the second experiment, we again use modulo distribution for the `split` component and compare results for OR-Semantics and ORDER-Semantics for the `join` component. We increase the input frame rate to 9.6 *frames/s* in order to increase the computation requirements, especially for `cpu3`. The overloaded `cpu3` will cause a longer processing delay for the `QV` processes on `cpu3` than in the case of 6 *frames/s*. This longer processing delay incurs a longer waiting time for some macroblocks before they can be processed by the `VO` process. The resulting output arrival curves are shown in Fig. 40. The figure shows that the maximal inter-arrival for

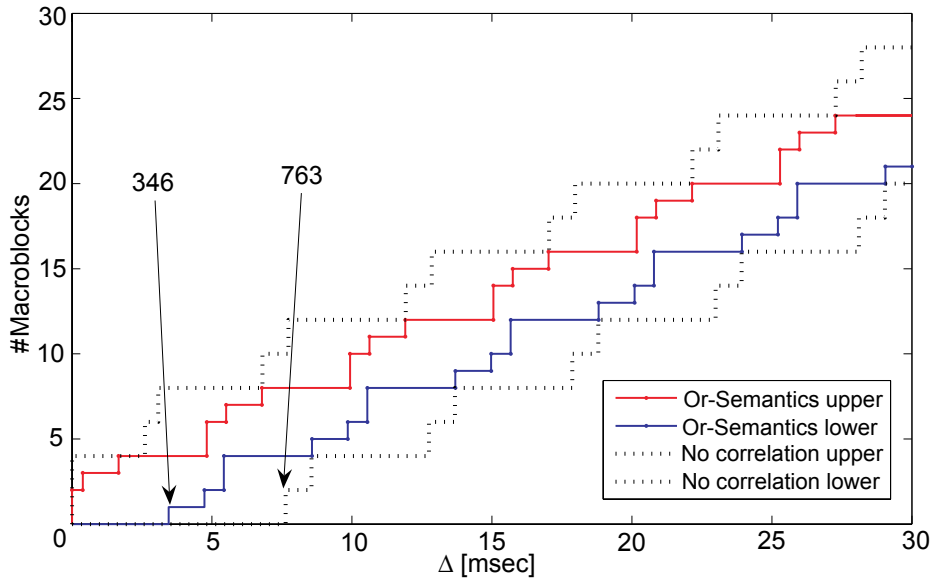


Fig. 39: The output arrival curves of the `join` process for the case of modulo distribution for the `split` process and OR-semantics and no correlations for the `join` process.

the output arrival curves corresponding to the OR-semantics (solid lines) do not change compared to the curves in Fig. 39. The reason is that the blocking effect is not considered for the OR-semantics and the `join` component will take any available macroblocks at any of its input buffers. In this case, the results derived from the OR-semantics are obviously too optimistic. On the other hand, a `join` component with the ORDER-semantics takes into account the blocking effect and derives correct results, i.e., a larger maximal inter-arrival time for output macroblocks in this case.

In the third experiment, we fix the `join` component to ORDER-semantics and compare the output arrival curves with block and modulo distributions for the `split` component. Again, we set the input frame rate to 6 *frames/s*. The output arrival curves are shown in Fig. 41. The figure shows that a block distribution policy of the `split` component results in an output stream that has large bursts and a large maximal inter-arrival time. This is because the block distribution on the one hand creates bursts for each sub-stream, i.e., eight macroblocks at once for each sub-stream. On the other hand, block distribution enlarges the split interval between sub-streams. The results shown in Fig. 40 confirm that our method correctly captures this behavior.

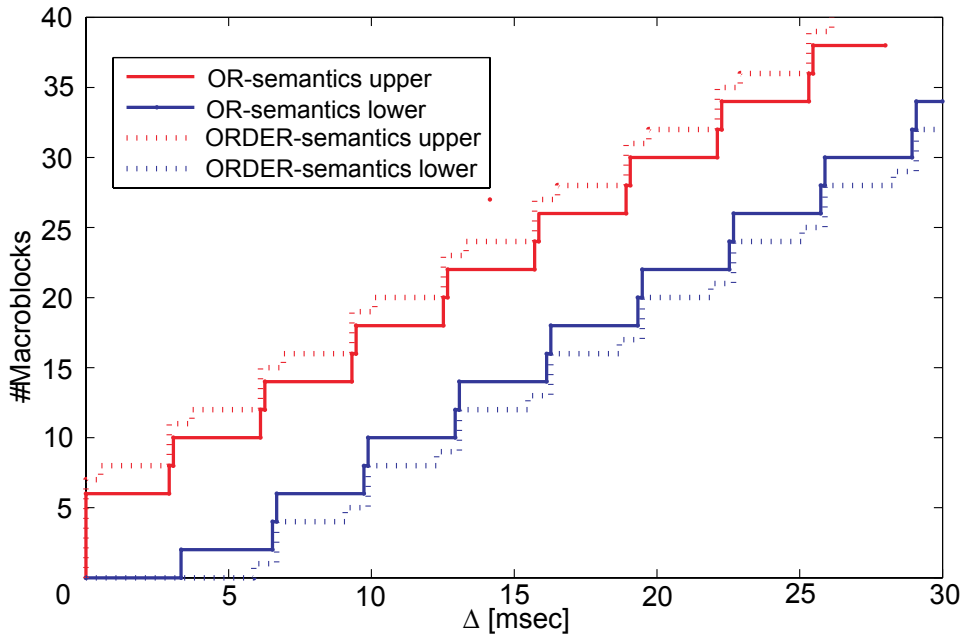


Fig. 40: The output arrival curves of the `join` process for the case of both OR-semantics and ORDER-semantics for the `join` process and modulo distribution for the `split` process.

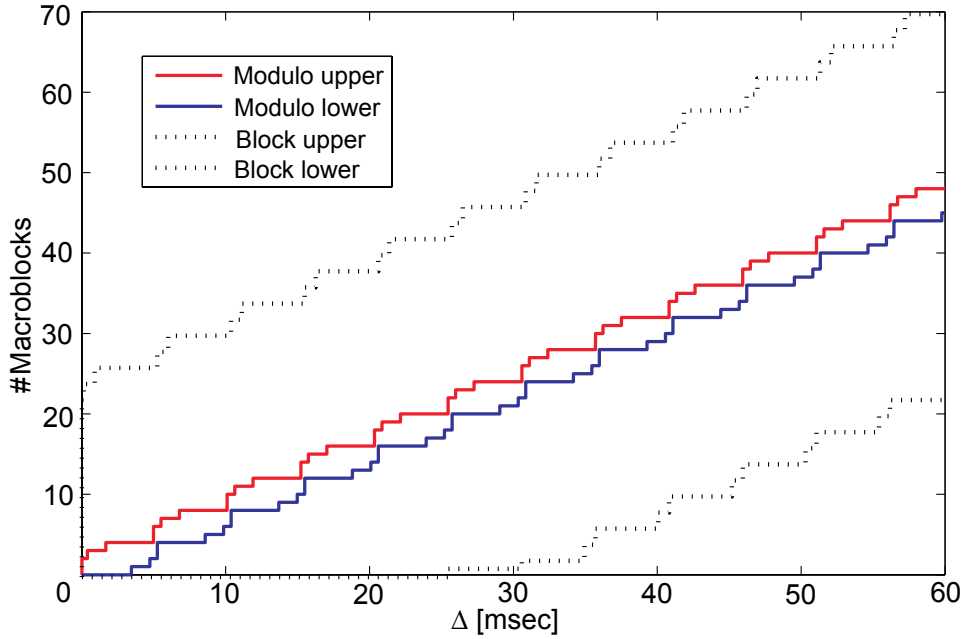


Fig. 41: The output arrival curves of the `join` component for the case of ORDER-semantics for the `join` component and block and modulo distributions for the `split` component.

3.3 Trace-Based Simulation

To model and analyze the complex interference and correlations between data streams, simulation-based approaches can be used as well. Because the exact timing information of each sub-stream is preserved during the simulation, the interference and correlations can be captured more naturally. Although simulation-based techniques cannot provide guaranteed bounds on system properties, they can be used to estimate average-case performance and to inspect properties at lower abstraction levels that analytic methods cannot capture. Therefore, we exploit simulation-based techniques in this section.

When using simulation-based techniques to estimate a multi/many-core embedded system, a major problem is the low simulation speed. Because the computing demand, in general, is super-linear to the number of cores to be simulated, inspecting the system performance at a too detailed level (at cycle-accurate level, for instance) is computational prohibitive. This problem is accentuated especially at early design stages where a large number of design alternatives need to be inspected.

The focus of our work here is to find a good trade-off between simulation speed and accuracy. On the one hand, the technique should provide accurate estimation results comparable to classical cycle/instruction-accurate simulations by considering important aspects like resource sharing and memory allocation schemes. On the other hand, the runtime of this technique should be acceptable for modern multi-core embedded system and scale for future many-core embedded systems. For these purposes, we propose a trace-based simulation framework at system level. By abstracting the behavior of an application as coarse-grain execution traces and simulating these abstract traces, our framework can evaluate large and complex embedded systems in a reasonable time frame. We investigate the effectiveness of our approach by mapping an MPEG-2 decoding algorithm onto the ATMEL Diopsis-940 platform [Pao06]. We demonstrate the scalability of our approach by mapping a scaled version of the MPEG-2 algorithm onto a scaled Diopsis-940 platform with up to 16 cores.

3.3.1 Related Work

Traditionally, simulation-based techniques are widely used for performance estimation of embedded systems. Examples are [VCC, CoW] at register-transfer level and [BZCJ02, MPND02] at system level. A drawback of these approaches is that the application of a target system has to be executed for each run of the simulation to obtain correct timing information. As the embedded applications become ever complex and

computationally demanding, these types of simulations turn out to be too slow, especially at early design stages where a large number of design alternatives need to be inspected. Furthermore, the computing demand of these techniques in general is super-linear to the number of cores to be simulated. Therefore, these techniques are mainly suitable for performance evaluation at later design stages.

Estimating system performance based on application traces is a common practice mainly for evaluating memory and cache subsystems, e. g. [GHV99, FSSZ01]. The idea of abstracting the functionality of an application as execution traces is also used to speedup the simulation of a complete embedded system. An early report on using application traces for system-level performance estimation is presented in [LRD01], which focuses on exploring on-chip communication architectures without considering the modeling of applications and different resource sharing schemes.

A system-level trace-based simulation is used in the Sesame framework [PEP06] for the performance estimation of heterogeneous embedded systems, where a trace transformation technique is introduced to refine abstract communication trace events according to the underlying architecture. This approach targets communication refinements of mixed accesses of local and shared memories via a shared bus. The approach is, however, limited to the integer-controlled dataflow model [Buc94]. A more recent work proposed by Wild et al. [WHO06] uses traces to simulate network processor architectures that consist of computation and memory modules communicating via a shared on-chip bus. A common drawback of both approaches is that the modeling scope is limited to architectures with a single shared bus. As communication schemes in modern embedded systems become more complex, modeling complex data transmission over hierarchical buses, for instance, becomes more important. New techniques are needed to cover communications via hierarchical buses.

A common limitation of the aforementioned frameworks [LRD01, PEP06, WHO06] is that none of them considers preemption, which is commonly used in modern embedded systems. Recently, a technique to support preemptive scheduling is proposed in the MESH trace-based framework [JP08]. It includes in its customized thread-based simulation kernel a speculative schedule. To correct the misspeculations triggered by preemptions, a rollback mechanism is used. The drawback of this rollback approach is that for the case of frequent preemptions, the simulation performance significantly drops. In addition, the MESH framework uses two models, i. e. , constant *penalty value* and *exponential function* to tune the accuracy of the modeling for communication. We argue that none of the two cases properly reflects the actual behavior of communication in

complex embedded systems, because the throughput of a communication resource is not always a simple function of the bandwidth of this resource, as shown in [RGB+08]. Compared to MESH, we use SystemC. By using the discrete-event simulation kernel in SystemC, our framework can precisely handle preemption without any speculation. Furthermore, we model communication resources at the same abstraction level as computation resources. This modeling allows us to inspect the effect of different communication arbitration schemes, resulting in accurate estimation results.

Moreover, in comparison to the related work mentioned above, the simulator in our framework is composed in a modular way, which offers high flexibility in terms of modeling scope and scalability in terms of the size of a target system. Coupled with the programming model presented in the previous chapter, our framework can model and simulate large-scale multi/many-core embedded systems.

3.3.2 Modeling

Our simulation framework serves as a performance estimation back-end of the programming model introduced in the previous chapter. Therefore, we separate the modeling of the application and architecture of an embedded system. This section introduces the application and architecture models.

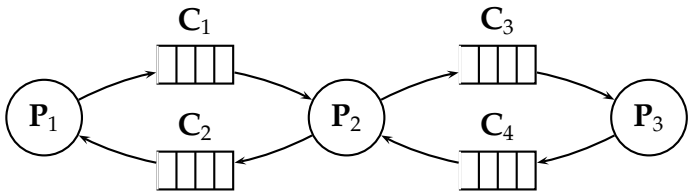
3.3.2.1 Modeling of Streaming Applications

To model an application, we use the Kahn process network (KPN) [Kah74] model of computation, as briefly introduced in Section 2.3. Using KPN to model applications is the basis of our simulation framework. On the one hand, the separation of computation and communication of a KPN application allows to group consecutive computation and communication into coarse-grain traces, as shown in Ex. 5. On the other hand, the property of being determinate of KPN, i.e., the functional behavior is independent from the timing of processes in the network, allows to reuse the same set of application traces to explore different architecture configurations and mappings.

A set of application traces can be represented as an acyclic graph $G = (V, E, E')$, where vertices in V represent computation and communication events and edges represent the functional dependencies of the trace events. A *communication event* corresponds to a read or a write operation of a FIFO in the source code. A *computation event* groups all consecutively executed instructions between two contiguous communication events within a process. We also define a source and a sink vertex for each

process to represent the starting and ending events for the traces of this process. An edge in set E connects two consecutive events of a process, the direction of which specifies the functional order of these two events. An edge in set E' connects inter-process communication events, i.e., a write/read pair, corresponding to a data transmission via a KPN FIFO. A graphical view of a set of application traces is shown in the following example.

Ex. 5: *A process network consisting of three processes and the coarse-grained source code of the processes are depicted in Fig. 42. The first iteration of the recurring pattern of the traces is shown in Fig. 43. A horizontal row in Fig. 43 represents a set of trace events for a process. The edges that connect two rows specify data dependencies between write/read pairs.*




```
P1.fire() {
   P1_1.comp();
   C1.write();
   P1_2.comp();
   C2.read();
}
```

```
P2.fire() {
   C1.read();
   P2_1.comp();
   C3.write();
   C4.read();
   P2_2.comp();
   C2.write();
}
```

```
p3.fire() {
   P3_1.comp();
   C3.read();
   P3_2.comp();
   C4.write();
}
```

Fig. 42: A process network with three processes and the coarse-grained source code of processes.

In a set of application traces, the functionality of an application is abstracted as *high-level trace events*, whereas data dependencies between processes are captured as the ordering information between trace events. The trace sequence for a single process is *totally ordered*, i. e., the chronological order of trace events is fixed. The edges in E' impose a *partial order* on the trace events between different processes. The resulting acyclic partially ordered graph is fixed for a given input of the application and *independent* of any target architecture or mapping. This property allows to reuse the same partially ordered trace graph within the loop of a design space exploration for inspecting different mappings and different architecture configurations.

To enable timing simulation, a computation event of a process is annotated with the runtime information of the corresponding code

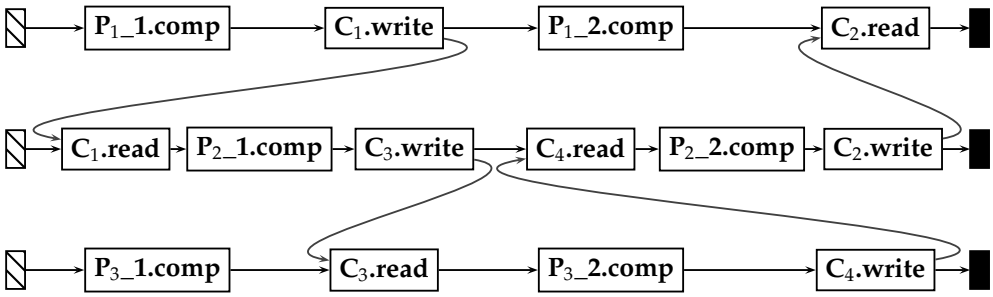


Fig. 43: First iteration of the recurring pattern of the execution traces for the process network in Fig. 42. Shadowed and black rectangles denote sources and sinks, respectively.

segment for each processor to which the process can be mapped. We will explain how to extract timing information of trace events in the follow-up sections. A communication event is annotated with the amount of data transmitted from/to a FIFO. Note that a communication event will be refined to a set of new communication events during the simulation. We define a basic unit by which a communication resource transmits data. During the simulation, a communication event will be split up into n new communication events, where n is the quotient between the amount of data in the original communication event and the basic unit.

3.3.2.2 Architecture modeling

One of our primary goals is to estimate the performance of embedded systems with multiple/many cores. In order to do so, we do not differentiate computation and communication resources in our modeling. We define *virtual machines* for timed processing of trace events and *virtual paths* for the routing of communicated data. Based on this scheme, an architecture for a target system can be easily composed. By simply varying the interconnects as well as scheduling policies of virtual machines, different design alternatives can be quickly inspected, at an abstract level.

Virtual Machine

A *virtual machine* (VM) is defined as an autonomous unit, simulating the timing behavior of a computation and communication resource. We model a VM as a 2-tuple $\mathcal{V} = (S, C)$, where S represents a scheduler and C is the capability, e.g. bus bandwidth or core frequency. The scheduler S manages the mutual exclusive processing of multiple input event queues and C decides how time advances. For a computation

VM (e.g. computing cores), there are two types of input event queues, namely process event queues and communication event queues. Each process event queue represents an application process that is mapped onto this VM. A communication event queue corresponds to a connection to a connected communication resource. A computation VM consumes trace events (both computation and communication) from its input event queues according to its scheduling policy and dispatches communication events to connected communication VMs. For a communication VM (e.g. bus and NoC), each input event queue represents a connection to a connected VM. A communication VM consumes communication events according to its arbitration policy and forwards them to the input queues of connected VMs.

A shared memory, for instance, can be modeled as a VM. The corresponding S is defined as a first-come first-serve policy and $C = \min\{B_{bus}, B_{mem}\}$ is the minimum between the bandwidths of the memory and the connected bus. In the case of $B_{bus} < B_{mem}$, the throughput of the shared memory is limited by the bus, otherwise, by the bandwidth of the shared memory.

Fig. 44 shows the model of a two-tile architecture of our experimental platform in Section 3.3.4.2. The platform contains two ATMEL Diopsis tiles connected via a NoC. The local memory of a computing core is not explicitly modeled, whereas the shared memory is explicitly modeled as a VM. Each core has a variable number of queues defined by the processes mapped onto it. Each bus has four different event queues, two for the computing cores within the same tile, one for the on-tile shared memory, and one for the NoC. The shared memory has only one queue, as it can only be accessed via the shared bus. The NoC has two queues, corresponding to the two connected buses. As shown in Fig. 44, the architecture of a system can be easily constructed by just coupling VMs through event queues. By exchanging the scheduler, different scheduling or arbitration policies can be modeled.

Virtual Path

Modern embedded systems normally consist of multiple communication resources and application data may need to be transmitted through multiple communication resources before they reach their final destination. As an example, if processes P_1 and P_2 in Fig. 42 are mapped to processing cores `ARM1` and `ARM2` in Fig. 44, respectively, a data token transmitted on the channel C_1 will need to traverse `BUS1`, `NoC`, and `BUS2`.

To capture data transmission through multiple communication resources, we define *virtual paths* (VP). A VP defines a possible route for a KPN FIFO and consists of the names of the two computation VMs

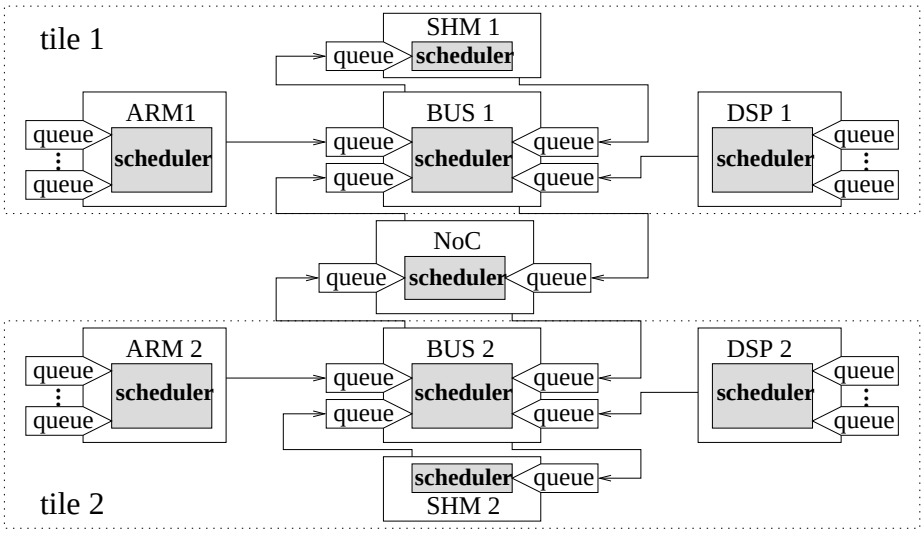


Fig. 44: The abstract model of a two-tile architecture, where the communication event queues for the computation VMs are ignored due to the space limit of figure.

that execute the producer and consumer of the FIFO, the names of the communication VMs that are used, and the name of the memory where the FIFO buffer is located. A format specification is shown in Fig. 9 in Section 2.4.3.

During the simulation, when a VM consumes a communication event, it will forward this communication event to the input queue of the next VM according to the VP onto which the FIFO is mapped. The fill level of the FIFO will be increased only after a communication event reaches the memory where the FIFO buffer is located. Similarly, the fill level of the FIFO will be decreased when the communication event is consumed by the last computation VM in the virtual path. In this manner, we can model and simulate complex communication.

3.3.3 Simulation Framework

The main inputs of our simulation framework are the application, architecture, and mapping Xml specifications specified using the programming model introduced in the previous chapter. An overview of the simulation framework is shown in Fig. 45. The application Xml specifies the application process network. The architecture Xml specifies the VMs and VPs of an target platform. The mapping Xml provides information for the binding of processes to VMs, FIFOs to VPs, and resource sharing schemes for VMs. In the case that a process is mapped onto a VM, all traces of this process are associated to this VM. In the case that a FIFO is mapped onto a VP, all communication trace events using the FIFO are

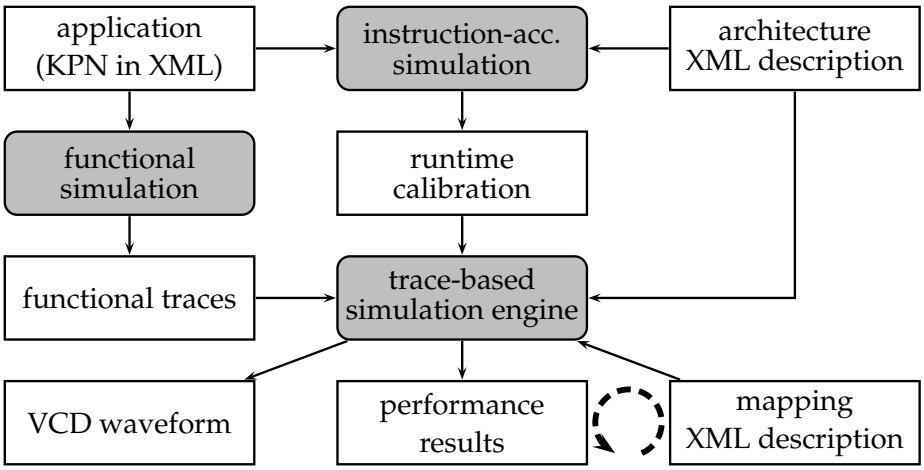


Fig. 45: An overview of the trace-based simulation framework.

associated to this VP.

We use a functional and an instruction-accurate simulation to generate application traces and timing information of the traces, respectively. Both the functional simulation and instruction-accurate simulation are executed only once. Afterwards, the trace-based simulation can run independently. We use SystemC to implement our simulation engine. The outputs of the trace-based framework are the timed traces of the application by which performance statistics such as utilization of hardware resources and FIFO buffer usage can be derived. We can also visualize timed traces as standard value change dump (VCD) waveforms.

3.3.3.1 Trace Generation

We use the SystemC-based functional simulation presented in Section 2.6 to generate the untimed traces of an application. The reason of using a functional simulation is two-fold. On the one hand, because a KPN application is determinate, the application traces for a given input are independent from the timing behavior of the processes. On the other hand, it is technically easier to extract the execution traces from functional simulation compared to low-level simulation or emulation.

To extract traces from a functional simulation, the key is to identify the start and the end of each trace event. According to the trace model in Section 3.3.2.1 and the programming syntax in Section 2.4.2.2, the starting point of a trace event can be: a) the first instruction of an `init` and a `fire` procedure, or b) the last instruction of a `READ` and `WRITE` primitive. Correspondingly, the ending point of a trace event can be: a) the last instruction of an `init` and a `fire` procedure, or b) the first instruction of a `READ`, `WRITE`, and `DETACH` primitive. Therefore, we instrument the

source code of the functional simulation to store the line numbers for all starting and ending points of trace events. Specifically, we use the GCC `__LINE__` macro to obtain the line number of the start and end of trace events during the execution of the functional simulation.

3.3.3.2 Trace Calibration

The calibration of application traces, i. e., determining the resource requirement of trace events, is more involved. A usual approach is to use cycle/instruction-accurate simulation and instrument the application source code to store the time spent from the start to the end of each trace event. One problem of using code instrumentation is that the simulator needs to execute the instrumented code. The instrumented code will distort the calibration and generate inaccurate timing information, which is undesirable especially for the timing analysis of real-time systems.

To tackle this problem, we develop a non-intrusive approach, avoiding any code instrumentation. The basic idea is to pause the simulation at the start and end of a trace event, record the simulated clock value, and then resume the simulation. In this manner, precise timing information can be obtained.

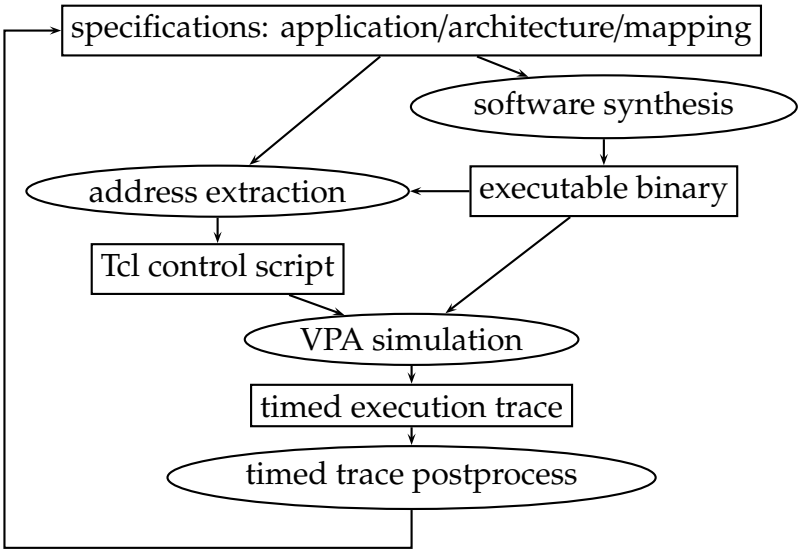


Fig. 46: An overview of the non-intrusive approach for trace calibration on the VPA.

Our approach is developed on top of the CoWare Virtual Platform Analyzer (VPA) [CoW], which is a tool to build instruction-accurate simulators. An overview of our approach is shown in Fig. 46. We use the software synthesis flow developed in [HPB+] to generate the executable

binary for a given application. This executable binary can be executed on either a simulator built by the VPA or the target hardware platform. We then analyze the generated executable binary and extract the memory addresses of the start and end of trace events. Based on the extracted addresses, a Tcl script is used to setup a breakpoint for each memory address, associate a callback function for each breakpoint at the beginning of the simulation, and control the execution of the simulation. During the simulation, the Tcl script pauses the simulation at each breakpoint, records the current simulation time, execute the callback function, and then resumes the simulation. With the recorded simulation time for each breakpoint, the actual time spent on each trace event is computed.

To extract the memory addresses of the start and end of trace events, we scan the disassembly of the executable binary with the function symbols of the start and end of trace events, i. e. , the first or last instructions of the `init` and `fire` procedures, as well as primitives `READ`, `WRITE`, and `DETACH`. A snapshot of the extraction results is depicted in Fig. 47. The core information is specified at the beginning, i.e., line 1, which includes the core name, core index, exiting function symbol, number of processes on the core, and file name of the core image. Lines 2-5 specify a set of aforementioned breakpoints, where entries are core index, function name, entry address, and leaving address. Note that the ATMEL Diopsis-940 platform supports interrupts. Therefore, we also track context switches, i.e., the procedure `scheduler_switch` in Fig. 47.

```
1 /HARDWARE/ST_0_0_0/ARM,0,DETACH,1,result/tile_0/arm/APP.x
2 BreakpointRange=0,funcA_FIRE,0x20207dd4,0x20207eb0
3 BreakpointRange=0,READ,0x20207aac,0x20207b18
4 BreakpointRange=0,scheduler_switch,0x202042bc,0x202042fc
5 BreakpointRange=0,DETACH,0x20207814,0x2020782c
```

Fig. 47: A snapshot of the extracted breakpoints for the VPA.

The control flow of the Tcl script is depicted in Fig. 48. This script controls the execution of a VPA simulation. When a simulation starts, the script first setups the aforementioned breakpoints and the corresponding callback functions. During the simulation, when a breakpoint is hit, the simulation is paused and the corresponding callback function is invoked to record the current simulation time. After executing the callback function, the script will also check the ending condition of the simulation, i. e. , whether the `DETACH` primitive of each process has been executed. If all `DETACH` primitives have been executed, i.e, the corresponding breakpoints have been hit, the script terminates the simulation. Otherwise, the script resumes the simulation.

In this manner, the performance extraction is controlled by the Tcl

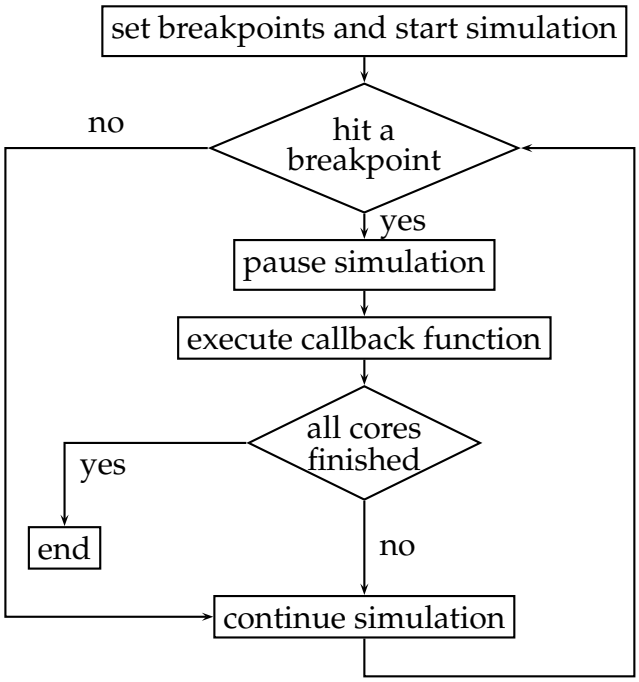


Fig. 48: The control flow of the Tcl script.

script without any user interaction. Since there is no code instrumentation in the application source code, the calibration results are accurate.

3.3.3.3 Simulation Kernel

As previously shown in Section 2.6, SystemC is a C++ based library to model and simulate hardware and software systems on different levels of abstraction. It provides a highly efficient discrete event simulation kernel, which is suitable for our trace simulation. Therefore, we use SystemC [Soc05, sys] to implement our trace-based simulation.

An abstract view of our trace-based simulation kernel is depicted in Fig. 49. Basically, we use an `sc_thread` to implement a hardware resource (i. e. , a VM) or a process in the application process network. For event queues of a VM, we use `sc_channel` . To conduct a simulation, we use immediate notifications of an `sc_event` (Section 2.6.2) to notify ready trace events and the `sc_wait` routine to advance the simulation time. We say a trace event is *ready* only when all its predecessors have been processed. When a trace event is ready, the process to which this trace event belongs will notifies the VM to which this process is mapped, by sending an immediate notification via an `sc_channel` . A VM will process ready

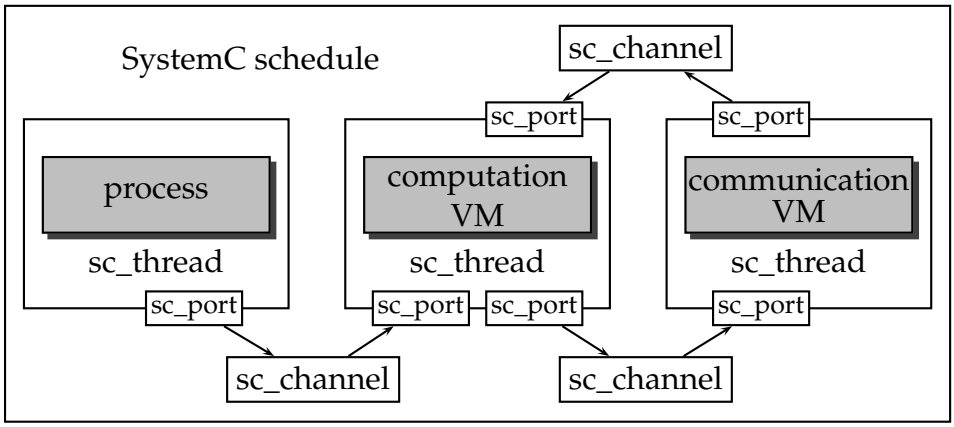


Fig. 49: An abstract view of the simulation kernel which contains a process, a computation virtual machine, and a communication virtual machine.

trace events and advance the simulation time using the `sc_wait` routine according to a specified scheduling policy.

The function of a VM is the heart of our simulation. Initially, a VM is idle waiting on empty event queues. If there arrive ready trace events, it will pick out and process one trace event according to its scheduling police. We will elaborate how to choose and process a trace event in the following paragraphs. For a computation event, the VM will mark it as processed after the time required for this event has been advanced. For a communication event, the VM will also forward it to the next connected VM according to the VP of this event.

In addition, a VM is also used to model the filled states of FIFOs. In principle, when a VM processes a communication trace event, it will increase the fill level of the FIFO to which this communication trace event belongs if it finds out (from the VP of this trace event) that the buffer of this FIFO is mapped onto it memory. Similarly, if a VM finds out that it is the last one in the VP, it will decrease the fill level of the FIFO by the size of this communication trace event.

To demonstrate how we tackle time-triggered and event-triggered scheduling, we present two representative policies, i. e., time division multiple access (TDMA) and preemptive fixed priority (FP) policies. Other policies, like first-come first-serve, static scheduling, and round robin, can be constructed in a similar way.

The pseudo code for the TDMA policy is depicted in Algorithm 1. Basically, each process mapped to a VM is assigned a time slot. All assigned time slots compose a TDMA cycle. In every cycle, a VM checks every time slot (line 1) and informs the SystemC kernel the end of a time slot even if there is no ready trace event for this slot (line 2). Routine `sc_wait` will return after the simulated time advances t_i time unit from

the current simulated time. After time for a time slot is advanced, ready trace events that belong to this time slot are processed (line 3). If a trace event cannot finish at currently slot, the unprocessed part of it will have to wait for its slot in the next cycle, see line 5-6.

Algorithm 1 TDMA scheduling of a virtual machine $\mathcal{V}$

```
1:  for each slot i do
2:      sc_wait(t_i)                                   ▷ t_i is length of slot i
3:      while ∃ e_{i,j} ∈ slot i ∨ t_i > 0 do ▷ There is a ready event e_{i,j} for slot i
4:          if t_{e_{i,j}} > t_i then  ▷ Required time longer than the length of slot i
5:              t_{e_{i,j}} ← t_{e_{i,j}} − t_i
6:              t_i ← 0
7:          else
8:              t_i ← t_i − t_{e_{i,j}}
9:              j + +        ▷ e_{i,j} is processed. Can continue to the next ready
    event
10:         end if
11:     end while
12: end for
```

The pseudo code for the FP scheduling is depicted in Algorithm 2. It works differently in comparison to the TDMA scheduling. Only when there is a ready trace event, the scheduler will inform the SystemC kernel how much time it needs in order to process this trace event, as shown in line 3. Besides the timeout $t_{e_{i,j}}$, the routine `sc_wait` will register an `sc_event` that serves as an interrupt event. If a trace event of a higher priority process comes before the timeout $t_{e_{i,j}}$, the interrupt event `IRQ_event` will be activated and routine `sc_wait` will return with the actual advanced time, as shown in line 5–7. The scheduler will continue with the new highest priority trace event.

Algorithm 2 FP scheduling of a virtual machine $\mathcal{V}$

```
1: while ∃ e_{i,j} do              ▷ Get the highest ready event from all queues
2:     t_old ← sc_simulation_time( )
3:     sc_wait(t_{e_{i,j}}, IRQ_event)                ▷ Interrupt event IRQ_event
4:     t_new ← sc_simulation_time( )
5:     if t_new − t_old < t_{e_{i,j}} then                ▷ The actual advanced time
6:         t_{e_{i,j}} ← t_{e_{i,j}} − (t_new − t_old)
7:     end if
8: end while
```

3.3.4 Experimental Results

To demonstrate the capabilities of the proposed approach, we map a parallel MPEG-2 video decoder presented in Section 2.4.5 onto an ATMEL Diopsis platform and compare performance results of our trace-based simulation to an instruction-accurate simulation implemented using the CoWare Virtual Platform Analyzer (VPA) [CoW]. Both VPA and the trace-based simulation execute on a 2 *GHz* AMD Athlon 2800+ Linux machine.

3.3.4.1 Experimental Setup

The VPA simulator consists of a heterogeneous ATMEL Diopsis 940 tile [Pao06], as graphically shown in Fig. 50. This tile is composed of a shared memory, a VLIW DSP core, and an ARM9 core, working at a clock frequency of 100 *MHz*. Both cores as well as the shared memory connect through an in-tile bus that provides a maximum throughput of 400 *MByte/s*. The simulated MPEG-2 clip has a frame-rate of 25 *fps*, a bit-rate of 8 *Mbps*, and a resolution of 704×576 pixels. For our experiments, we use a video clip with a duration of 15 seconds.

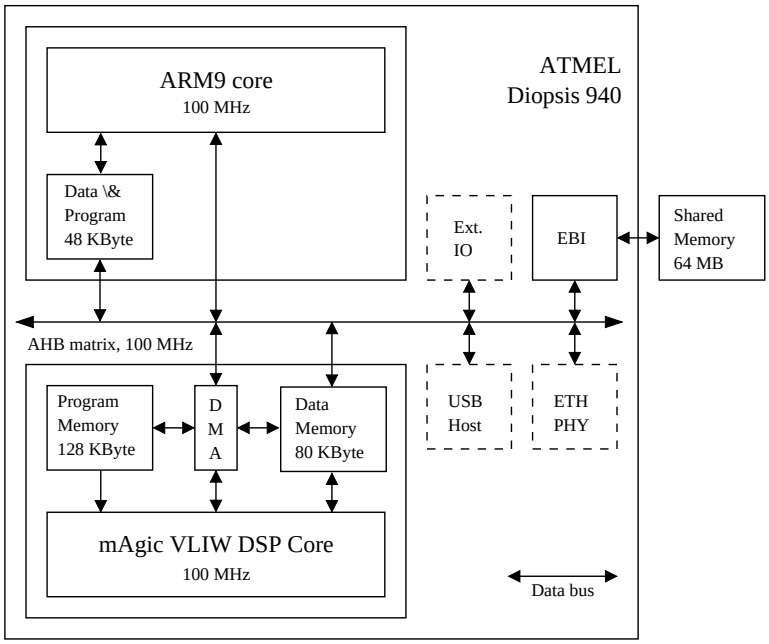


Fig. 50: Block diagram of the experimental architecture.

We measure two different timing criteria: a) simulation runtime (`Sim.`), corresponding to the wall-clock time from the start of the simulation until its completion, and b) estimated execution time (`Est.`), which indicates

how long the decoder needs to finish the decoding of the 15 second video. All time values are rounded to seconds.

3.3.4.2 Experimental Results

For the first experiment, we map a pipelined version of MPEG-2 decoder (shown in Fig. 11(a) in Section 2.4.5) onto a single Diopsis 940 tile. We investigate three different mappings. `Mapping 1` uses only the ARM core of a tile, i.e., all processes are mapped onto an ARM core and all FIFO buffers are located in the local memory of the ARM core. `Mapping 2` uses the same process mapping as in `Mapping 1` whereas FIFO buffers are mapped to the shared memory. `Mapping 3` maps the most computation intensive process, i.e., `transform_block`, onto the DSP core and the buffers for those FIFOs connecting to `transform_block` are set to the shared memory.

Case	Est. time (hh:mm:ss)			Sim. time (hh:mm:ss)		
	TSim	VPA	Error	TSim	VPA	Speedup
1	1:36:51	1:38:26	-3%	0:03:28	30:34:00	611
2	1:54:13	1:52:20	+2%	0:03:30	31:30:00	630
3	0:17:17	0:16:46	+3%	0:03:30	74:19:00	1466

Tab. 5: Estimated execution time and simulation time for the considered mappings, when decoding a 15 *s* MPEG-2 video.

The experimental results are presented in Tab. 5. From the table, we can find out that our trace-based simulation obtains a high accuracy for all three mappings. The error rate in comparison to the VPA instruction-accurate simulation is within 3 %. The deviation is mainly due to the cache effect in the platform, which is currently not modeled in our trace-based simulation. In terms of simulation speed, our trace-based simulation is considerably faster than the VPA simulation. We obtain two and three orders of magnitude speedup for the single core and dual cores cases, respectively. The reason is that our simulation does not execute the application, which is computationally expensive. In addition, a hardware resource in our model, i.e., a virtual machine, works only as a trace event dispatcher and does not simulate instruction-level behavior, which further reduces the simulation runtime.

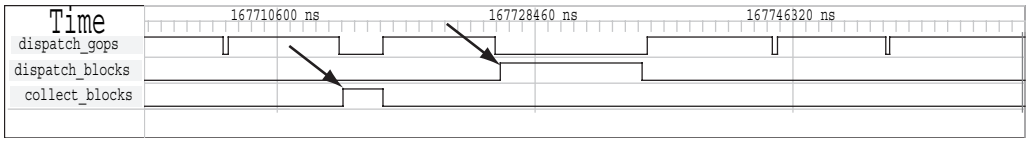
In the second experiment, we simulate `Mapping 1` with different sizes of the FIFOs in the process network. The simulation results are shown in Tab. 6. From the table, we can find out that the error rate is again about 3 %. Note that the `Est.` time for MPEG-2 decode increases as the size of

the FIFO buffers decreases. The reason is that with a smaller FIFO size, more context switches are triggered.

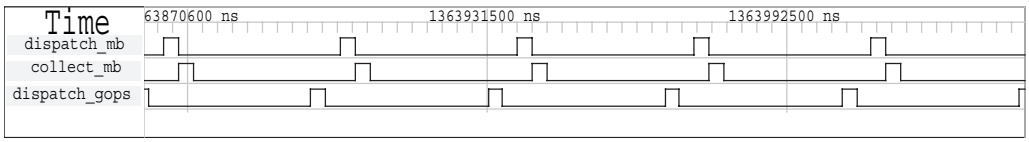
	Est. runtime (hh:mm:ss)			
Chan. size	**600 bytes**	**1 Kbyes**	**8 Kbyes**	**16 Kbyes**
VPA	1:56:38	1:44:25	1:38:52	1:38:26
TSim	1:53:03	1:45:35	1:37:24	1:36:51
Error	-3%	+1%	-2%	-2%

Tab. 6: Estimated execution time for different buffer sizes of FIFOs in `Mapping 1`.

The third feature that we investigate is the capability of modeling event/time-triggered scheduling policies. Since our VPA implementation currently supports only first-come first-serve scheduling, we show the waveforms of the trace-based simulation. We reuse the settings of `case 3` and replace the scheduling of the ARM core with FP and TDMA policies. The corresponding results are shown in Fig. 51. As shown in Fig. 51(a), the lower priority task `dispatch_gops` has been preempted by higher priority tasks `collect_block` and `dispatch_block`, respectively. In the TDMA case, each time slot is 3 μs and the cycle length is 18 μs, which is directly reflected in the waveform in Fig. 51(b).



(a) FP policy.



(b) TDMA policy.

Fig. 51: Trace-based simulation waveforms with FP and TDMA scheduling policies.

We demonstrate the scalability of our approach in the last experiment. We use a scaled version of the MPEG-2 encoder process network, where the process `dispatch_gop` will dispatch groups of pictures to four parallel sub-streams and each sub-stream will again dispatch data to two concurrent `transform_block` processes, resulting in a process network with totally 26 processes. To scale the experimental platform, we connect multiple ATMEL Diopsis 940 tiles with a network-on-chip which has a maximum throughput of 400 *MByte/s*. We simulate different mappings

with up to eight tiles. The simulation time is shown in Tab. 7. The simulation time of our simulations increases only by 18 %, even when eight times more hardware resources are simulated. The reason for the increment of the simulation time is that with more tiles added, more communication trace events need to be transmitted through multiple communication media. These multi-hop communication trace events consume additional simulation time.

Tiles	Simulation time (hh:mm:ss)
1	0:05:08
2	0:05:29
4	0:05:43
8	0:06:04

Tab. 7: Simulation time for 1, 2, 4, 8 tiles.

3.4 Summary

In this chapter, we investigate techniques to estimate the timing behavior of streaming embedded systems. We describe two different methods, i.e., a formal analytic method and a simulation-based technique.

For the analytic approach, we inspect timing correlations between data streams and present a new method to analyze correlated data streams that originate from the same source. By means of the proposed method, we can obtain considerably tighter performance bounds than those using previous methods. We show the applicability of the presented method by analyzing a concrete multimedia application. Although this method is implemented within the RTC-MPA framework, the idea behind can also be applied to other modular analysis frameworks, such as SymTA/S.

For the simulation-based approach, we propose a trace-based framework, which can simulate the timing behavior of complex multi/many-core embedded systems. By abstracting application functionality into coarse-grained traces and simulating a system at system level, our simulation is orders of magnitude faster than instruction-accurate simulations. On the other hand, by capturing essential system properties, e. g. scheduling and buffer location, together with the intrinsic characteristics of streaming system, e. g. stable pipelined stage and small cache effect, we obtain high accuracy compared to instruction-accurate simulations.

Another advantage of our techniques is that both the analytic method and the trace-based simulation can serve as a non-functional back-end for

the programming model defined in the previous chapter. Both techniques can thereby be embedded into an automated design space exploration to assist the software design of embedded systems.

In the next chapter, we will inspect another design metric, i.e., the power metric. In particular, we investigate system-level power-efficient design under certain performance constraints.

4

Power Management

Power dissipation has been an important design issue in a wide range of computer systems in the past decades. Power management with energy efficiency considerations is helpful for server systems to reduce power bills. Efficient power management is even more crucial for embedded systems. A major category of embedded systems, for instance, are hand-held mobile devices that are powered by batteries. The batteries for such devices are limited in both power and energy output. The amount of energy available thus severely limits a system's lifespan. Although research continues to develop batteries with higher energy-density, the slow growth of the energy density of batteries lags far behind the tremendous increase of demands [ITR]. Because of these facts, power consumption becomes one of the first-class design concerns for modern computer systems, in particular for multi-core and future many-core embedded systems.

Two major sources of power consumption of a CMOS circuit are dynamic power consumption due to transistor switching activities and static power consumption due to leakage current [JPG04]. For micrometer-scale semiconductor technology, dynamic power dominates the power consumption of a processor. However, as modern VLSI technology is scaling down to the deep sub-micron domain, chips consume significantly more static power. The leakage current that originates from the dramatic increase in both sub-threshold current and gate-oxide leakage current is projected to account for as much as 50 % of the total power dissipation for high-end processors in 90 *nm* technologies [ABM+04]. The International Technology Road-map for

Semiconductors (ITRS) expects that the percentage of static power of the total power dissipation in the future will be much greater due to variability and temperature effects [ITR].

On the other hand, the increasing number of integrated cores of an embedded platform provides opportunities to efficiently harness the static power consumption, e.g., switching those idle cores to a sleep mode to reduce leakage current. An interesting research question is how to schedule the mode switches of those idle cores under real-time requirements.

4.1 Overview

In the previous chapter, we investigated the performance metric of multi/many-core embedded systems and presented techniques for system-level performance estimation. This chapter studies another metric, i. e. , power metric. We focus on streaming embedded systems at system level and study efficient power management under certain performance constraints. Specifically, we make use of the worst-case analysis method, i.e., real-time calculus and explore how to apply dynamic power management to reduce static power consumption while satisfying real-time constraints.

For simplicity, we consider a dual-core scenario, i.e., a processing core for data stream processing and a control core for coordination, e.g., scheduling. The processing core has three power modes, i.e., `active`, `standby`, and `sleep` modes, with different power consumptions. The control core decides when to change the power modes of the processing core. Intuitively, the processing core can be switched off to `sleep` mode to reduce the power consumption when it becomes idle and switched to `active` mode again upon the arrival of an event. These switching operations, however, need more careful consideration. On the one hand, the sleep period of the processing core after switching-off should be long enough to recuperate mode-switch overheads. On the other hand, when to activate the processing core is even more involved due to the possible burstiness of future event arrivals. For every switching-on operation, sufficient time has to be reserved to serve the possible burstiness of future events in order to prevent deadline violation of events or overflow of system backlog.

To resolve these concerns, we propose both online and offline algorithms that are applicable particularly for embedded systems. We apply real-time calculus to predict future event arrival and real-time interface theory [TWS06] for schedulability analysis, trying to procrastinate the buffered and future events as late as possible. Our offline

algorithms compute optimal and approximate schemes for periodic power management. Alternatively, our online algorithms adaptively predict the next mode-switch moment by considering both past and future event arrivals. Based on the adopted worst-case interval-based abstraction, our algorithms can not only tackle arbitrary event arrivals, e.g., with burstiness, but also guarantee hard real-time constraints with respect to both timing and backlog constraints. To handle multiple event streams, we propose solutions for two preemptive scheduling policies, i.e., earliest-deadline-first and fixed priority. Although our algorithms are developed for a dual-core scenario, the principle behind can be applied to multi/many-core platforms, in particular the class of platforms using tile-based architectures, e. g. , Intel SCC [SCC] and ATMEL ShapOtto [HPB+].

The rest of this chapter is organized as follows: The next section reviews the related work in the literature. Section 4.3 provides system models. Section 4.4 presents a set of new routines which will be used throughout this chapter. Section 4.5 presents our proposed online algorithms for one event stream, while Section 4.6 copes with multiple event streams. The offline periodic power management approach is presented in Section 4.7. Simulations results are presented within each section and Section 4.8 summarizes the chapter.

4.2 Related Work

Power estimation techniques are important for energy-efficient embedded system designs. To estimate the power consumption of a microprocessor at abstract level, analytic methods [BFSS00, MPS98] based on statistical or information-theoretic techniques and simulation-based approaches [BTM00, YVKI00] based on instruction set simulators (ISSs) are widely used. To leverage the trade-off between accuracy and estimation speed, trace-based power simulation [vSP10] is also proposed to evaluate the power consumption of a system.

To analyze power-aware real-time behavior of an embedded system, event-stream based models are commonly adopted. Baptiste [Bap06] proposes an algorithm based on dynamic programming to control when to turn on/off a device for aperiodic real-time events with the same execution time. For multiple low-power modes, Augustine et al. [AIS04] determine the mode that a processor should enter for aperiodic real-time events and propose a competitive algorithm for online use. Swaminathan et al. [SC05] explore dynamic power management and develop offline algorithms to find the exact starting time of real-time events and to compute an approximation in polynomial time. To aggregate the idle time for energy reduction, Shrivastava et al. [SEDN05] propose a framework

for code transformations. By considering platforms with both DPM and dynamic voltage scaling (DVS), Chen and Kuo [CK07] propose to execute tasks at a certain speed and to control the procrastination of real-time events. By turning the device to the sleep mode, the execution of the procrastinated real-time events is aggregated in a busy interval to reduce energy consumption. Heo [HHLA07] et al. explore how to integrate different power management policies in a server farm. Alternatively, Devadas and Aydin [DA08, DA10] consider the interplay between DVS and DPM on a system with a DVS-capable processor and multiple devices. Based on the concept of forbidden regions, the authors propose algorithms to determine the optimal processor speed as well as transition decisions of device states to minimize overall system energy for periodic real-time tasks.

Most of the above approaches require either precise information of event arrivals, such as periodic real-time events [CK07], or aperiodic real-time events with known arrival time [Bap06, AIS04, ISG03]. We argue that the abstraction is, however, too coarse. In practice, the precise timing information of event arrivals might not be known in advance since the arrival time depends on many factors. When the precise timing of event arrivals is unknown, to our best knowledge, the only known approaches are to apply the online algorithms proposed by Irani et al. [ISG03] and Augustine et al. [AIS04] to control when to turn on the device. However, since the online algorithms in [AIS04, ISG03] greedily stay in the sleep mode as long as possible without referring to incoming events in the near future, the resulting schedule might make an event miss its deadline. Such algorithms are not applicable for hard real-time systems.

To model irregular arrival patterns of event streams, real-time calculus, based on Network Calculus [Cru91b], was proposed by Thiele et al. [TCN00, WTVL06] to characterize events with arrival curves. The arrival curve of an event stream describes the upper and lower bounds of the number of events arriving at the system for a specified interval. Therefore, schedulability analysis can be done based on the arrival curves of event streams. In [MCT05], Maxiaguine et al. apply real-time calculus within the DVS context and compute a safe frequency at periodical intervals with predefined length to prevent buffer overflow of a system. Chen et al. [CST09] explore the schedulability for on-line DVS scheduling algorithms when the event arrivals are constrained by a given upper arrival curve. In contrast to these closely related approaches, we focus on dynamic power management. We propose offline algorithms to find periodic time-driven patterns to turn on/off devices for energy savings without sacrificing timing guarantees. The small run-time overhead of the periodic schemes is very suitable for embedded systems that only have limited computational power. We also propose online algorithms

where the on/off decisions are dynamic and adaptively vary according to the actual arrivals of events. Furthermore, we provide solutions on multiple event-stream scenarios where event streams with different characteristics can be tackled with both earliest-deadline-first and fixed-priority scheduling policies.

4.3 System Model and Problem Definition

This section presents in detail the system model as well as the problem definition.

4.3.1 System Model

We consider a dual-core system with a processing core and a control core. The processing core is responsible for event processing. The control core conducts three tasks: a) handling event arrivals, e.g., storing unprocessed events into system backlog, b) controlling the power mode of the processing core to serve arrived events, e.g., turning on and off the processing core, and c) dispatching events in system backlog to the processing core with a certain scheduling policy. An abstract view of this model is shown in Fig. 52. Parameters S, α, and D in Fig. 52 will be introduced later. Note that the model we consider here is only a principle model, i. e. , an abstraction to simplify the analysis.

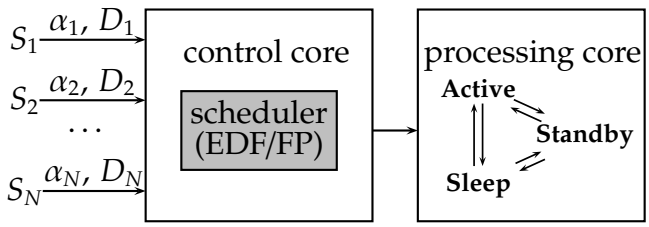


Fig. 52: The system model of the studied problem.

The processing core has three power modes, namely `active`, `standby`, and `sleep`. The power consumption in `sleep` mode is P_σ. To serve an event, the processing core must be in `active` mode with power consumption P_a, where $P_a > P_\sigma$. When there are no events to serve, the processing core can enter `sleep` mode. However, switching from `sleep` mode to `active` mode and back takes time, denoted by $t_{sw,on}$ and $t_{sw,sleep}$, and incurs an energy overhead, denoted by $E_{sw,on}$ and $E_{sw,sleep}$, respectively. To prevent the processing core from frequent mode switches, the processing core can also stay in `standby` mode. The power

consumption P_s in `standby` mode lies, by definition, between P_a and P_σ, i.e., $P_a \geq P_s > P_\sigma$. Like in [ZC05, YCHK07], it is assumed that switching between `standby` mode and `active` mode has negligible overhead. Fig. 53 illustrates the state chart for the power model of the processing core.

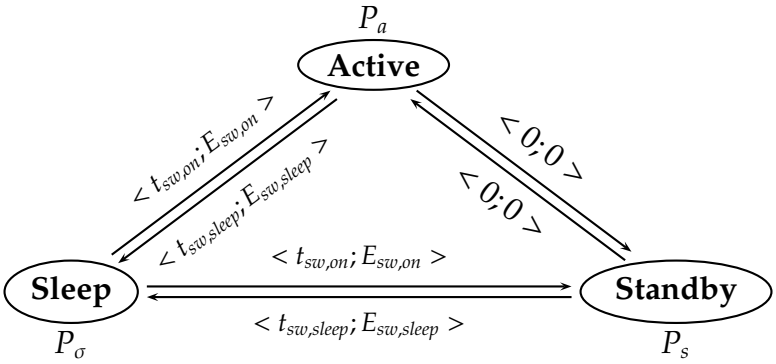


Fig. 53: The state transition diagram of the processing core, where the tuple on each edge is the timing and energy overheads.

We abstract a data stream in a KPN as an event stream. Therefore, different event streams serve as inputs of the system. We denote $\mathcal{S}$ as a stream set containing N event streams with different characteristics. To buffer incoming events of the streams in $\mathcal{S}$, the control core maintains a backlog. Buffering more events than the size of the backlog incurs a backlog overflow and causes a system failure. We discuss two types of backlog management, i.e., *global backlog* where all event streams in $\mathcal{S}$ share a common backlog and *individual backlog* where each event stream has its own backlog. The size of the backlog in either case is assumed to be given. How to decide a proper size of the system backlog is a different research problem and is not in the scope of this chapter.

4.3.2 Worst-Case Interval-Based Streaming Model

To model irregular arrival of events, we adopt the arrival curves $\bar{\alpha}(\Delta) = [\bar{\alpha}^u(\Delta), \bar{\alpha}^l(\Delta)]$ from real-time calculus, in which $\bar{\alpha}_i^u(\Delta)$ and $\bar{\alpha}_i^l(\Delta)$ are the upper and lower bounds on the number of events for a stream S_i in any time interval of length Δ. For instance, for an event stream with period p, jitter j, and minimal inter arrival distance d, the upper arrival curve is $\bar{\alpha}^u(\Delta) = \min\{\left\lceil \frac{\Delta+j}{p} \right\rceil, \left\lceil \frac{\Delta}{d} \right\rceil\}$. The concept of arrival curves unifies many other timing models of event streams. Analogous to arrival curves that provide an abstract event stream model, a tuple $\beta(\Delta) = [\beta^u(\Delta), \beta^l(\Delta)]$ defines an abstract resource model which provides upper and lower bounds on the available service in any time interval Δ. For further details about $\bar{\alpha}$ and β, we referred to Section 3.2.2.

Note that an arrival curve $\bar{\alpha}_i(\Delta)$ specifies the number of events of stream S_i whereas a service curve $\beta(\Delta)$ specifies the available amount of services in terms of time for an interval of length Δ. Therefore, $\bar{\alpha}_i(\Delta)$ has to be transformed to $\alpha_i(\Delta)$ to indicate the amount of computation time required for the arrived events in intervals. Suppose that the execution time of any event in stream S_i is w_i. Then the transformation can be done by $\alpha_i^u = w_i \cdot \bar{\alpha}_i^u$, $\alpha_i^l = w_i \cdot \bar{\alpha}_i^l$ and back by $\bar{\alpha}_i^u = \lceil \alpha^u / w_i \rceil$, $\bar{\alpha}_i^l = \lfloor \alpha^l / w_i \rfloor$. For variable workloads in an event stream, our algorithms can be revised slightly by adopting the variable workload model in Section 3.2.2.3. Moreover, the response time of an event in event stream S_i must not exceed its specified relative deadline D_i, where the response time of an event is its finishing time minus the arrival time of the event. On the arrival of an event of stream S_i at time t, the absolute deadline is $t + D_i$.

4.3.3 Problem Definition

We explore how to effectively minimize the power consumption to serve a stream set $\mathcal{S}$ by dynamic power management (DPM). Intuitively, static power can be reduced by a) turning the device to `sleep` mode when there is no event to process, and b) staying at `sleep` mode as long as possible by postponing the processing of arrived events. However, switching from/to `sleep` mode incurs both a timing and an energy overhead. To model these overheads, we define a break-even time.

Def. 4: **(Break Even Time)** *Suppose that a) switching the processing core to* `sleep` *mode takes $t_{sw,sleep}$ time and switching back to* `active/standby` *mode takes $t_{sw,on}$ time, b) the corresponding energy consumption for the switching activities are $E_{sw,sleep}$ and $E_{sw,on}$, and c) P_s and P_σ are the power consumption in the* `standby` *and* `sleep` *modes, respectively. The break-even time T_{BET} is defined as:*

$$T_{BET} \stackrel{def}{=} \max\left\{t_{sw,on} + t_{sw,sleep},\ \frac{E_{sw,on} + E_{sw,sleep}}{P_s - P_\sigma}\right\} \tag{4.1}$$

Consider the case that the processing core is switched to `sleep` mode. If the interval that the processing core can stay in `sleep` mode is shorter than T_{BET}, the mode-switch overhead is larger than the energy saving. In this case, a mode switch does not pay off. On the other hand, retaining the processing core in `sleep` mode might incur backlog overflow or deadline misses for (burst) event arrivals in the near future. The question is what is a proper moment to conduct a mode switch of the processing core.

We use the following notation: A schedule decides when to perform a mode-switch of the processing core. A schedule is *feasible* if it is always possible to meet the timing and backlog constraints of the system. An algorithm is *feasible* if it always generates feasible scheduling decisions.

Therefore, the problem studied in this chapter is to determine a feasible schedule a) *when to turn the processing core to* `sleep` *mode to reduce the static power*, and b) *when to turn the processing core from* `sleep` *mode to* `active` *mode to serve events*.

4.4 Real-Time Calculus Routines

To compute a feasible schedule for the processing core, real-time calculus and real-time interface are applied. Within this context, the processing core is said to provide a guaranteed output service $\beta^G(\Delta)$. Correspondingly, a stream S_i requests service demand $\beta^A(\Delta)$. To obtain a feasible schedule of the processing core that serves stream S_i, the condition

$$\beta^G(\Delta) \geq \beta^A(\Delta),\ \forall \Delta \geq 0 \tag{4.2}$$

has to be fulfilled. In this section, we present how to construct proper service guarantees and demands such that (4.2) leads to a feasible schedule.

4.4.1 Bounded Delay

Suppose that a service curve $\beta(\Delta)$ can be constructed as a bounded delay function.

Def. 5: **(Bounded Delay Service Curve)** *A bounded delay service curve is defined as follows:*

$$\boldsymbol{bdf}(\Delta,\ \tau) \stackrel{def}{=} \max\Big\{0,\ (\Delta - \tau)\Big\},\ \forall \Delta \geq 0,\ \tau \geq 0 \tag{4.3}$$

where τ denote the sleep interval.

We say that a *sleep interval* is the accumulated time for which the processing core retains in `sleep` mode. In the case that the processing core provides full service, i. e. , $\tau = 0$, the processing core never switches to `sleep` mode.

Def. 6: **(Longest Feasible Sleep Interval)** *The longest feasible sleep interval τ^* with respect to a given service demand $\beta^A(\Delta)$ is thereby defined as:*

$$\tau^* = \max\Big\{\tau : \boldsymbol{bdf}(\Delta, \tau) \geq \beta^A(\Delta),\ \forall \Delta \geq 0\Big\} \tag{4.4}$$

Def. 7: **(Deadline Service Demand)** *Suppose events in event stream S_i are constrained by a given relative deadline D_i. To satisfy the required relative deadline D_i, the minimal service demand $\beta^\flat$ of stream S_i is*

$$\beta^\flat(\Delta) \stackrel{def}{=} \alpha_i^u(\Delta - D_i) \tag{4.5}$$

Def. 8: **(Backlog-size Service Demand)** *Suppose a system has a backlog of size Q_i to buffer unprocessed events for stream S_i. To prevent overflow of the system backlog, the minimal service demand $\beta^{\dagger}$ of stream S_i is*

$$\beta^{\dagger}(\Delta) \stackrel{def}{=} \alpha_i^u(\Delta) - w_i \cdot Q_i \tag{4.6}$$

Combining both deadline and backlog service demands, the longest feasible sleep interval τ^* in (4.4) can be refined as

$$\tau^* = \max\Big\{\tau : \boldsymbol{bdf}(\Delta, \tau) \geq \max\{\beta^{\flat}(\Delta), \beta^{\dagger}(\Delta)\}\Big\} \tag{4.7}$$

Fig. 54 illustrates Def. 5–8 for the case of a single event-stream. Based on these definitions, we state the following lemma.

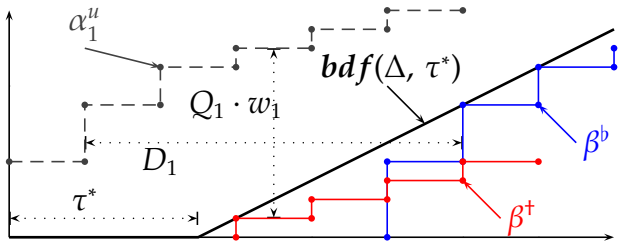


Fig. 54: An example for the bounded delay function for event stream S_1 with upper arrival curve $\alpha_1^u(\Delta)$.

Lem. 3: *Assume a τ^* computed from (4.7) that is larger than T_{BET}. At any time instant t when the processing core is active and there are no events to process, it is feasible to switch the processing core to* `sleep` *mode for $[t, t + \tau^*)$ interval without violating the deadline and backlog-size requirements of any event in stream S_i, if the processing core provides full service from time $t + \tau^*$.*

Proof. We prove the lemma as follows: The service demand $\beta^{\flat}$ in (4.5) is constructed by horizontally right-shifting α_i^u by distance D_i, which represents the tightest bound that guarantees the deadline constraint. Similarly, the service demand $\beta^{\dagger}$ in (4.6) is obtained by vertically shifting α_i^u down by $(w_i \cdot Q_i)$ distance, being the tightest bound that prevents backlog overflow. Therefore, $\max\{\beta^{\flat}, \beta^{\dagger}\}$ is an upper bound on the service demand that guarantees deadline and backlog-size constraints. On the other hand, if one considers time t as the starting point, the service that starts from $t + \tau^*$ can be seen as a special case of $\boldsymbol{bdf}(\Delta, \tau^*)$, i. e. , all accumulated time of $\boldsymbol{bdf}(\Delta, \tau^*)$ in `sleep` mode is at the beginning interval $[t, t + \tau^*)$. According to Def. 5, we have that $\boldsymbol{bdf}(\Delta, \tau^*)$ is a lower bound for this special case. Because of (4.7), we know that $\boldsymbol{bdf}(\Delta, \tau^*)$ bounds the service demand, i. e. , $\max\{\beta^{\flat}, \beta^{\dagger}\}$. Consequently, the service provided by

this special case can satisfy the service demand. In addition, the sleep interval is longer than the mode-switch overhead, i. e. , $\tau^* > T_{BET}$, which guarantees that the processing core can provide service no later than $t+\tau^*$. Therefore, the lemma holds.

□

4.4.2 Future Prediction with Historical Information

For a given event stream S_i, the corresponding upper arrival curve α_i^u can be used to predict the upper bound on future event arrivals from any time instant t, i. e. , $[t, +\infty)$. Because this bound is the worst-case bound for all possible arrival patterns of S_i, it may be too pessimistic for certain cases. For example, if a burst of events has been recently observed, events will only sparsely arrive in the near future. To obtain a tighter bound during the runtime, we keep track of event arrivals in the past as a history.

Def. 9: **(History Curve)** *Suppose t is the current time and $R_i(t)$ is the accumulated number of events of stream S_i in interval $[0, t)$. The length of the history window that can be maintained is Δ^h, i.e., historical information for only Δ^h time units is recorded. At time t, a history curve for stream S_i is defined as*

$$H_i(\Delta, t) \stackrel{def}{=} \begin{cases} R_i(t) - R_i(t-\Delta), & \text{if } \Delta \le \Delta^h, \\ R_i(t) - R_i(t-\Delta^h), & \text{otherwise} \end{cases} \tag{4.8}$$

The maximal number of event arrivals in the time interval $[t, t+\Delta)$, denoted as $\bar{\alpha}_i^u(\Delta, t)$, is constrained by

$$\bar{\alpha}_i^u(\Delta, t) = \inf_{\lambda \ge 0} \left\{ \bar{\alpha}_i^u(\Delta + \lambda) - H_i(\lambda, t) \right\} \tag{4.9}$$

and the corresponding computation-based arrival curve, denoted as $\alpha_i^u(\Delta, t)$, is $\alpha_i^u(\Delta, t) = w_i \cdot \bar{\alpha}_i^u(\Delta, t)$. A graphical example of above definitions are shown below:

Ex. 6: *Assume an event stream where events come with a period $1\,\mathrm{ms}$ with a possible burst of two events. We keep track of the historical arrivals for at most $2\,\mathrm{ms}$. At time instant t, we know that in the past events e_1, e_2, and e_3 arrived at time $t-2.5$, $t-1.5$, and t, respectively, as depicted in Fig. 55. The history curve based on (4.8) is shown in the dotted line. Because of e_3, we know that at most one more event will come in the future interval $(t, t+1)$, for instance. Based on this historical information, we obtain the history-aware arrival curve $\bar{\alpha}(\Delta, t)$ depicted in the straight in Fig. 55.*

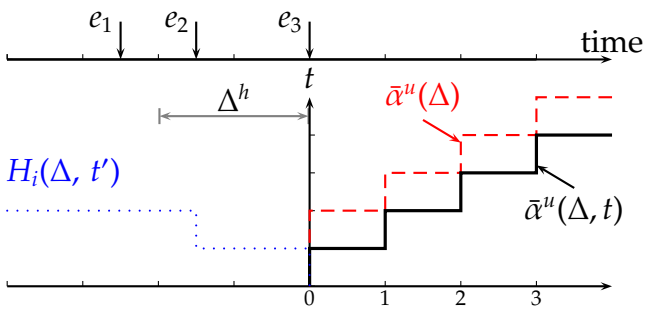


Fig. 55: An example for a history curve and the corresponding history-aware arrival curve for a given time instant.

4.4.3 Backlogged Demand

The deadline service demand in Def. 7 is defined for the case of no backlogged events. In the case that there are events in the system backlog waiting to be processed, the service demand from these backlogged events need to be considered as well. Note that although the absolute deadlines for these backlogged events remain the same, the relative deadlines of these backlogged events however have changed. For example, suppose that an event with deadline D arrived in the past at time t_a, the relative deadline with respect to a time instant t_c is $D - t_c + t_a$. Based on this observation, we define backlog demand.

Def. 10: **(Backlogged Demand)** *Suppose that the set of unprocessed events of an event stream S_i at time t is $\boldsymbol{E_i}(t)$. Events in $\boldsymbol{E_i}(t)$ are indexed as $e_{i,1}, e_{i,2}, \ldots, e_{i,|\boldsymbol{E_i}(t)|}$ according to their arrived time, i.e., $e_{i,1}$ is the earliest arrived event. A backlog demand curve for S_i at time t is defined as:*

$$B_i(\Delta, t) \stackrel{def}{=} w_i \cdot \sum_{j=1}^{|\boldsymbol{E_i}(t)|} f(\Delta + t) \tag{4.10}$$

where

$$f(\Delta + t) = \begin{cases} 1 & \text{if } \Delta + t \geq D_{i,j} \\ 0 & \text{otherwise} \end{cases}$$

and $D_{i,j}$ is the absolute deadline of event $e_{i,j}$.

An example of above definition is illustrated in Fig. 56 where there are four unprocessed events in the system backlog at time t. As shown in the figure, we can precisely define the service demand of backlogged events by using (4.10).

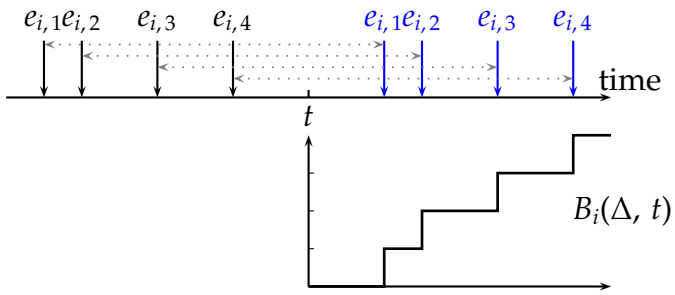


Fig. 56: An illustrated example for formula (4.10).

4.5 Online DPM for Single Stream

For online dynamic power management schemes, in general, the control core has to decide when to turn the processing core to `active` mode to serve events from `sleep` mode, and when to turn it back to `sleep` mode to reduce static power. Therefore, we have to deal with *deactivation decisions* and *activation decisions* to switch safely and effectively. An overview of our approach is illustrated in Fig. 57.

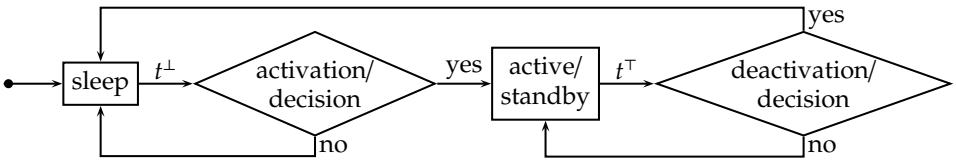


Fig. 57: The control flow of our approach.

For deactivation decisions, when the processing core is in `active` mode and there is no event in the backlog, we have to decide whether the processing core has to change to `sleep` mode instantly or it should remain active/idle [1] for a while to serve the next incoming event. For brevity, for the rest of this chapter, time instants for deactivation decisions are denoted by $t^{\top}$.

After the processing core is switched to `sleep` mode, it has to be switched to `active` mode again for event processing. The activation decision is evaluated at the time instant upon the arrival of an event or expiration of the sleep interval that the control core previously set. The control core has to decide whether the processing core has to change to `active` mode instantly to serve events, or it should remain in `sleep` mode for a while to aggregate more events and prevent unnecessary

[1] We assume that the processing core will conduct automated mode-switch in the following two cases: a) switch to `standby` mode when it is in `active` mode and there are no events to be processed, and b) switch to `active` mode when it is in `standby` and there are events to be processed.

mode switches. For brevity, for the rest of this chapter, time instants for activation decisions are denoted by $t^{\perp}$.

In this section, we present our online algorithms that are applied to the control core and minimize static power consumption of the processing core. The assumption of our approach is that the timing and backlog constraints of system can always be guaranteed if the processing core provides full service all the time, i.e., the processing core never turns to `sleep` mode. For simplicity, we first consider only the case of a single event stream S_1. We present how to deal with the deactivation decision and then propose two methods for activation decision. The solutions for multiple event streams are presented in the subsequent two sections.

4.5.1 Deactivation Algorithm

The History-Aware Deactivation (HAD) algorithm analyzes whether the processing core should be turned to `sleep` mode from `active` mode. The principle is to switch the processing core only when energy savings are possible. One obvious fact is that as long as there are events in the system backlog, the processing core can be kept busy in `active` mode until all backlogged events are processed. In this case, no static power is wasted and any mode-switch will introduce an additional time and energy overhead. In order to reduce the mode-switch overhead, the deactivation decision thereby makes sense only when the processing core is in `active` or `standby` mode while there is no new arrival of events as well as no event in system backlog. Suppose that $t^{\top}$ is such a time instant.

Turning the processing core instantly at time $t^{\top}$ to `sleep` mode, however, does still not always help. The reason is that we pay a time and energy overhead for each mode switch. In the case that there are events arriving in the very near future, the processing core has to be switched to `active` mode again to process these events. If the energy savings obtained from a short sleep interval cannot outweigh the switching overhead, this mode switch only introduces additional energy consumption. Therefore, the idea is firstly to compute the maximal possible sleep interval τ^* and check whether this τ^* is sufficient to cover the break-even time. Specifically, we calculate the arrival curve $\bar{\alpha}_1^u(\Delta, t^{\top})$ at time $t^{\top}$ by (4.9) and refine the service demands in (4.5) and (4.6) as

$$\beta^{\flat}(\Delta) = \alpha_1^u(\Delta - D_1, t^{\top}) \tag{4.11}$$

$$\beta^{\dagger}(\Delta) = \alpha_1^u(\Delta, t^{\top}) - Q_1 \cdot w_1 \tag{4.12}$$

By applying (4.11) and (4.12) to (4.7), the maximal sleep interval τ^* is computed. If τ^* is larger than T_{BET}, the processing core is switched to `sleep` mode at time $t^{\top}$. Otherwise, the processing core is retained in

`active/standby` mode. The pseudo code of the algorithm is shown in Algorithm 3.

Algorithm 3 HAD deactivation

procedure *at time instant* $t^\top$*:*

1: compute τ^* of (4.7) by $\beta^\flat$ and $\beta^\dagger$ in (4.11) and (4.12);
2: **if** $\tau^* > T_{BET}$ **then**
3: deactivate the processing core;
4: **end if**

The algorithm leads to the following theorem:

Thm. 3: *Algorithm HAD guarantees a feasible scheduling upon a deactivation decision at any time $t^\top$ for a single event-stream system if the processing core provides full service starting from time $t^\top + \tau^*$, where τ^* is computed from line 1 of the algorithm.*

Proof. We prove this theorem by contradiction. At any time instant $t^\top$ at which Algorithm HAD decides to deactivate the processing core, the latest activation time to prevent constraint violations is $t^\top + \tau^*$. Suppose at a later time instant $t^\top + \lambda$, the deadline of an event which comes within the interval $[t^\top, t^\top + \lambda)$ is missed. We denote the number of events arrived within this interval as u. Because of the deadline missing, the service demand $u \cdot w_1$ in this interval is larger than our constructed service supply $\boldsymbol{bdf}(\lambda, \tau^*)$ which actually bounds the service demand of the maximum number of events that can arrive, i.e., $w_1 \cdot \bar{\alpha}_1^u(\lambda, t^\top)$. The inequality $u > \bar{\alpha}_1^u(\lambda, t^\top)$ contradicts the definition in (4.9), however. Therefore, the theorem holds.

□

4.5.2 Activation Algorithms

Once the processing core is in `sleep` mode, the control core needs to switch it back to `active` mode at a later moment for event processing. How to decide the actual switch moment needs more consideration. On the one hand, it is preferable to aggregate as many events as possible for each switch operation to not only reduce the standby period but also minimize the number of switch operations. On the other hand, the real-time constraints of the aggregated and future events need to be respected. In addition, a polling mechanism is not desirable which will overload the control core. In this section, we present two algorithms, namely worst-case-greedy (WCG) algorithm and event-driven-greedy (EDG) algorithm, for activation scheduling decisions. The differences between these two algorithms are:

- Algorithm WCG is time-triggered. It conservatively assumes worst-case event arrivals and predicts the earliest switching moment. If the worst case does not occur when the predicted moment comes, a new prediction is conducted and the switch decision is deferred to a later moment.

- Unlike WCG, Algorithm EDG works in an event-triggered manner. It optimistically assumes the lowest number of event arrivals and predicts the latest time for mode switches. Upon the new arrival of an event before the predicted time, the decision is reevaluated and shifted to an earlier moment if necessary.

Therefore, the time instant $t^{\perp}$ for the activation decisions can be evaluated at either event arrivals or the predicted activation time. We refer to these two cases by *event arrival* and *wake-up alarm arrival*, at time instant $t_e^{\perp}$ and $t_w^{\perp}$, respectively.

4.5.2.1 Worst-Case-Greedy (WCG) Activation

Algorithm WCG works in a time-triggered manner. It reacts to each wake-up alarm and performs two tasks: a) check whether the processing core has to be switched to `active` mode for the current alarm, b) if not, determine the moment of the next wake-up alarm. In the case that a worst-case burst happens, the previous prediction is correct and the mode switch has to be carried out at the current wake-up alarm. If the number of arrived events is smaller than the worst case, switching the processing core to `active` at the current time $t_w^{\perp}$ will result in service more than actual needs. The processing core can stay in `sleep` mode for a longer period.

An illustrated example is shown in Fig. 58. Suppose after the processing core is in `sleep` model, an event, denoted as $e_{1,1}$, arrives at time $t_{w_1}^{\perp}$. We assume the worst case burst, i. e. two events (dashed arrows in the figure), takes place immediately after time $t_{w_1}^{\perp}$. To guarantee the deadlines of $e_{1,1}$ and the potential burst, the processing core has to be switched to `active` mode at time $t_{w_2}^{\perp} = t_{w_1}^{\perp} + \tau_1^*$, as shown in Fig. 58(a). When $t_{w_2}^{\perp}$ comes, the assumed burst does not happen, i. e. , no events arrive between time interval $[t_{w_1}^{\perp}, t_{w_2}^{\perp})$, as shown in Fig. 58(b). We then assume the burst will take place at time $t_{w_2}^{\perp}$, and the mode-switch decision is postponed to $t_{w_3}^{\perp} = t_{w_2}^{\perp} + \tau_2^*$. Again, if no events comes between time interval $[t_{w_2}^{\perp}, t_{w_3}^{\perp})$, we can postpone the decision to $t_{w_3}^{\perp} + \tau_3^*$, as depicted in Fig. 58(c).

To evaluate the activation decision and predict a new wake-up alarm, we again apply the bounded-delay function (4.7) to find the next sleep interval. To obtain tight results, the deadline and backlog service demand in (4.5) and (4.6) can be refined. At the current wake-up alarm time $t_w^{\perp}$, the

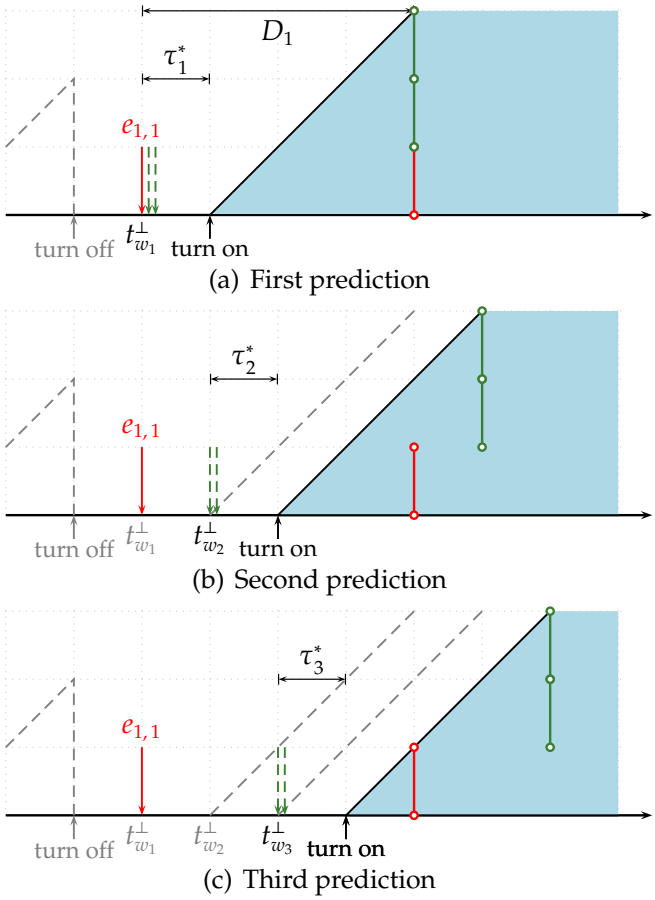


Fig. 58: An illustrated example for the Algorithm WCG, where *turn off* represents switching the processing core from `active/standby` mode to `sleep` mode and *turn on* represents switching back to `active` mode from `sleep` mode.

deadline service demand $\beta^{\flat}$ includes the events that are already stored in the system backlog, i.e., $B_1(\Delta, t_w^{\perp})$ defined in (4.10), together with the history-refined worst-case event arrival $\alpha_1^u(\Delta - D_1, t_w^{\perp})$. Similarly, the current size of the available backlog is the original size minus the number of backlogged events, i.e., $|\boldsymbol{E}(t_w^{\perp})|$ defined in Section 4.4.3. The deadline service demand $\beta^{\flat}$ and the backlog-size service demand $\beta^{\dagger}$ are refined as

$$\beta^{\flat}(\Delta) = \alpha_1^u(\Delta - D_1, t_w^{\perp}) + w_1 \cdot B_1(\Delta, t_w^{\perp}) \tag{4.13}$$

$$\beta^{\dagger}(\Delta) = \alpha_1^u(\Delta, t_w^{\perp}) - \left(Q_1 - |\boldsymbol{E}(t_w^{\perp})|\right) \cdot w_1 \tag{4.14}$$

Using (4.13) and (4.14), the next sleep interval τ^* is computed. If $\tau^* > 0$, the next wake-up alarm time is set to $t_w^{\perp} + \tau^*$. Otherwise, the processing core is switched to `active` mode. The pseudo code of Algorithm WCG is listed in Algorithm 4.

The constructed $\beta^{\flat}$ in (4.13) bounds the future arrival demands from

Algorithm 4 WCG activation

procedure *event arrival at time* $t_e^{\perp}$*:*
1: do nothing;
procedure *wake-up alarm arrival at time* $t_w^{\perp}$*:*
1: compute τ^* of (4.7) with $\beta^{\flat}$ and $\beta^{\dagger}$ by (4.13) and (4.14);
2: **if** $\tau^* > 0$ **then**
3: new wake-up alarm at time $t_w^{\perp} \leftarrow t_w^{\perp} + \tau^*$;
4: **else**
5: switch processing core to `active` mode;
6: **end if**

$t_w^{\perp}$ on and $\beta^{\dagger}$ in (4.14) guarantees the system backlog from overflowing, leading to following theorem:

Thm. 4: *Algorithm WCG guarantees a feasible scheduling upon an activation decision at any wake-up alarm time* $t_w^{\perp}$ *for single event-stream system, if the processing core provides full service from time* $t_w^{\perp}$ *on.*

We omit the proof due to the similarity to Theorem 1. Algorithm WCG is effective in the sense that it greedily extends the sleep period as long as a scheduling decision is feasible. It is also efficient when actual event arrivals are close to the worst case, where the reevaluation of the wake-up alarm time does not take place often. Furthermore, the number of reevaluation is bounded by $\alpha_1^u(D_1 - w_1)$.

The last question is when to set the first wake-up alarm. There are two possibilities: a) at the time of the first arrived event after the processing core is deactivated, and b) at the deactivation time instant $t^{\top} + \tau^*$ (τ^* is computed by (4.11), (4.12), and (4.7) in the HAD algorithm). Both approaches will result in feasible schedule. For consistency, we adopt the second approach and thus Algorithm WCG is purely time-driven.

4.5.2.2 Event-Driven-Greedy (EDG) Activation

In contrast to Algorithm WCG that predicts the earliest wake-up alarm time $t_w^{\perp}$, Algorithm EDG predicts the latest one. It computes the latest moment by assuming the lowest number of event arrivals in the near future. Unlike Algorithm WCG where the evaluation of the activation decisions takes place upon each wake-up alarm arrives, the decision here is refined upon event arrivals.

At time $t_{e_{1,i}}^{\perp}$ at which an event $e_{1,i}$ arrives, it is not obvious when the corresponding latest wake-up alarm $t_w^{\perp}$ will be. One intuitive guess is $t_{e_{1,i}}^{\perp} + D_1 - w_1$. This time instant is however too optimistic except for the first event $e_{1,1}$ after the processing core has been deactivated. Our EDG

algorithm works in the following manner. For the first arrived event $e_{1,1}$, the wake-up alarm is set to $t^{\perp}_{e_{1,1}} + D_1 - w_1$. For any subsequent event $e_{1,i}$, the wake-up alarm time is set to the minimum of the previous $t^{\perp}_w$ and $t^{\perp}_w - (w_1 - (t^{\perp}_{e_{1,i}} - t^{\perp}_{e_{1,i-1}}))$. An illustrated example for this approach is shown in Fig. 59, where the predictions for the first three arrived events are shown in Fig. 59(a)–Fig. 59(c). This new $t^{\perp}_w$ is still not always a feasible activation time instant. If τ^* computed from this time instant is not larger than 0, the activation is set to an earlier time, i.e., the earliest activation time as if the worst-case event arrival happens at $t^{\perp}_{e_{1,1}}$.

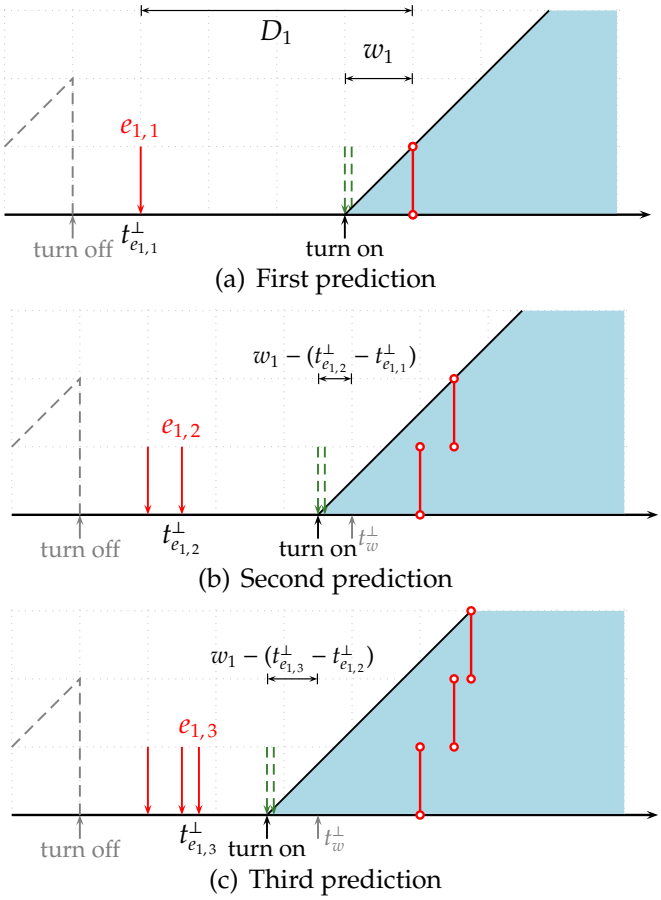


Fig. 59: An illustrated example for the Algorithm EDG, where *turn off* represents switching the processing core from `active/standby` mode to `sleep` mode and *turn on* represents switching back to `active` mode from `sleep` mode.

For an event $e_{1,i}$ arrived at time $t^{\perp}_{e_{1,i}}$, the service demand for the newly computed wake-up alarm time $t^{\perp}_w$ includes a) the possible burst from $t^{\perp}_w$ on, which is bounded by $\bar{\alpha}^u_1(\Delta, t^{\perp}_w)$, b) the backlog until $t^{\perp}_w$, and c) the estimated least event arrival between $[t^{\perp}_{e_{1,i}}, t^{\perp}_w)$, constrained by $\bar{\alpha}^l_1(\Delta)$. To compute a precise $\bar{\alpha}^u_1(\Delta, t^{\perp}_w)$, we first revise the historical information $H_1(\Delta, t^{\perp}_w)$ by advancing the time from $t^{\perp}_{e_{1,i}}$ to $t^{\perp}_w$ to include those events that definitely

have to come between $[t^{\perp}_{e_{1,i}}, t^{\perp}_{w})$. We denote such a trace as $H'(\Delta, t^{\perp}_{w})$:

$$H'_1(\Delta, t^{\perp}_{w}) = \begin{cases} \bar{\alpha}^l_1(\epsilon) - \bar{\alpha}^l_1(\epsilon - \Delta), & \text{if } \Delta < \epsilon, \\ H_1(\Delta, t^{\perp}_{e_{1,i}}) + \bar{\alpha}^l_1(\epsilon), & \text{if } \epsilon < \Delta < \Delta^h - \epsilon, \\ H_1(\Delta^h - \epsilon, t^{\perp}_{e_{1,i}}) + \bar{\alpha}^l_1(\epsilon), & \text{otherwise,} \end{cases} \tag{4.15}$$

where $\epsilon = t^{\perp}_{w} - t^{\perp}_{e_{1,i}}$ for brevity and Δ^h is length of the history window. The curve H'_1 can be considered as the concatenation of the historical information H_1 until $t^{\perp}_{e_{1,i}}$ and the time inversion of $\bar{\alpha}^l_1$ in the interval $[0, \epsilon)$. The worst-case arrival curve after time $t^{\perp}_{w}$ with the new historical information H'_1 is

$$\bar{\alpha}^u_1(\Delta, t^{\perp}_{w}) = \inf_{\lambda \geq 0} \left\{ \bar{\alpha}^u_1(\Delta + \lambda) - H'_1(\lambda, t^{\perp}_{w}) \right\} \tag{4.16}$$

and $\alpha^u_1(\Delta, t^{\perp}_{w}) = w_1 \cdot \bar{\alpha}^u_1(\Delta, t^{\perp}_{w})$.

The corresponding backlog demand curve that expresses the estimated lowest number of arrival events within the interval $[t^{\perp}_{e_{1,i}}, t^{\perp}_{w})$ is

$$B'_1(\Delta, t^{\perp}_{w}) \stackrel{\text{def}}{=} w_1 \cdot \sum_{j=1}^{|\boldsymbol{E_1}(t)| + \bar{\alpha}^l_1(\epsilon)} f(\Delta + t^{\perp}_{w}) \tag{4.17}$$

where

$$f(\Delta + t^{\perp}_{w}) = \begin{cases} 1 & \text{if } \Delta + t^{\perp}_{w} \geq D_{1,j} \\ 0 & \text{otherwise} \end{cases}$$

and $D_{1,j}$ is the absolute deadline of event $e_{1,j}$ and $\epsilon = t^{\perp}_{w} - t^{\perp}_{e_{1,i}}$.

With the refined historical information and backlog demand, the two service demands $\beta^{\flat}$ and $\beta^{\dagger}$ are refined as:

$$\beta^{\flat}(\Delta) = \alpha^u_1(\Delta - D_1, t^{\perp}_{w}) + w_1 \cdot B'_1(\Delta, t^{\perp}_{w}) \tag{4.18}$$

$$\beta^{\dagger}(\Delta) = \alpha^u_1(\Delta, t^{\perp}_{w}) - \left(Q - |\boldsymbol{E}(t^{\perp})| - \bar{\alpha}^l_1(\epsilon) \right) \cdot w_1 \tag{4.19}$$

By applying (4.18) and (4.19), the sleep interval τ^* in (4.7) is computed for event $e_{1,i}$. If $\tau^* > 0$, the wake-up alarm is valid. Otherwise, the new wake-up alarm is set to an earlier moment. The pseudo code of the algorithm is depicted in Algorithm 5.

Thm. 5: *Algorithm EDG guarantees a feasible scheduling upon an activation decision at any wake-up alarm time $t^{\perp}_{w}$ for single event-stream system, if the processing core provides full service starting from time $t^{\perp}_{w}$.*

Algorithm 5 EDG activation

procedure *event arrival at time* $t^{\perp}_{e_{1,i}}$*:*
1: **if** $t^{\perp}_{e_{1,i}} - t^{\perp}_{e_{1,i-1}} < w_1$ **then**
2: $\quad t^{\perp}_{w} \leftarrow t^{\perp}_{w} - (w_1 - (t^{\perp}_{e_{1,i}} - t^{\perp}_{e_{1,i-1}}))$
3: **end if**
4: calculate τ^* at time $t^{\perp}_{w}$ by (4.15–4.19);
5: **if** $\tau^* \leq 0$ **then**
6: $\quad t^{\perp}_{w} \leftarrow t^{\perp}_{e_{1,1}} + \tau^{\perp}$, where $\tau^{\perp}$ computed from (4.5) – (4.7)
7: **end if**

procedure *wake-up alarm arrival at time* $t^{\perp}_{w}$*:*
1: activate the processing core;

Proof. We differentiate between two mode switch decisions: when the condition in line 5 is fulfilled and when this is not the case. When the condition is not fulfilled, the feasibility of the decision is guaranteed by (4.15)–(4.19) where the actually arrived events before time instant $t^{\perp}_{w}$ and the potential burst afterwards are both considered in each evaluation. In the case that the condition in line 5 is fulfilled, we need to prove that the arrival time $t^{\perp}_{e_{1,i}}$ of event $e_{1,i}$ is always earlier than $t^{\perp}_{e_{1,1}} + \tau^{\perp}$. For time instant $t^{\perp}_{e_{1,1}}$ at which $e_{1,1}$ arrives, the maximum number of events that can be stored in the system backlog is $\min\{Q_1, \bar{\alpha}^u_1(\tau^{\perp}) - 1\}$, denoted as u. Based on the construction in line 1, $\tau^* \leq 0$ is only fulfilled if the number of arrived events reaches this maximum. According to the subadditivity of an upper arrival curve and the linearity of the bounded delay service curve, the time interval to generate u events is bounded by $\tau^{\perp}$, i.e., the inequality $(\bar{\alpha}^u)^{-1}(\tau^{\perp}) > u$ always holds[2]. Therefore, $t^{\perp}_{e_{1,1}} + \tau^{\perp} > t^{\perp}_{e_{1,i}}$. Because $t^{\perp}_{e_{1,1}} + \tau^{\perp}$ assumes that the worst-case event arrivals happen at the time instant of the arrival of the first event after the processing core is switched to `sleep` mode, switching the processing core to `active` mode at this time always results in a feasible scheduling decision. As the scheduling decisions are feasible for both cases, the theorem holds.

□

According to Thm. 5, Algorithm EDG results in a feasible schedule. The algorithm is efficient as well. It is designed for scenarios where events come sparsely and the worst case seldom occurs. In such scenarios, the pessimistic decision, i.e., the condition in line 5 is fulfilled, takes place seldom. In addition, the theoretical upper bound of the number on reevaluations is $\min\{Q_1, \bar{\alpha}^u_1(\tau^{\perp})\}$. In practice, the number of reevaluations is approximately equal to the number of actually arrived

[2]Symbol $(\bar{\alpha}^u)^{-1}$ represents the inverse function of $\bar{\alpha}^u$.

events. Furthermore, $\tau^{\perp}$ can be computed offline as it is a constant given the specification of a stream. Note that it is possible to refine $t_w^{\perp}$ when $\tau^* \leq 0$ is fulfilled, instead of pessimistically setting the prediction back to $t_{e_{1,1}}^{\perp} + \tau^{\perp}$. However, such a refinement demands more computation.

4.5.3 Experimental Results

This section provides simulation results for the proposed online dynamic power management schemes. The simulator is implemented in MATLAB by applying MPA and RTS tools from [WT06c].

4.5.3.1 Experimental Setup

We take the event streams studied in [HSC+09, HE05] for our case studies. The specifications of these streams are shown in Tab. 8. The parameters `period`, `jitter`, and `delay` are used for generating arrival curves defined in Section 4.3.2 and `wcet` represents the worst-case execution time of an event. The relative deadline D_i of a stream S_i is defined as the product of its period and a *deadline factor*, denoted by χ. In our simulations, we consider the processing core as a peripheral device. We adopt the power profiles for four different devices in [CG06], shown in Tab. 9.

	S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8	S_9	S_{10}
period (msec)	198	102	283	354	239	194	148	114	313	119
jitter (msec)	387	70	269	387	222	260	91	13	302	187
delay (msec)	48	45	58	17	65	32	78	-	86	89
wcet (msec)	12	7	7	11	8	5	13	14	5	6

Tab. 8: Event stream setting according to [HSC+09, HE05].

Device Name	P_a (W)	P_s (W)	P_σ (W)	t_{sw} (S)	E_{sw} (mJ)
Realtek Ethernet	0.19	0.125	0.085	0.01	0.8
Maxstream	0.75	0.1	0.05	0.04	7.6
IBM Microdrive	1.3	0.5	0.1	0.012	9.6
SST Flash	0.125	0.05	0.001	0.001	0.098

Tab. 9: Power profiles for devices according to [CG06].

We simulate different event streams. To compare the impact of different algorithms, we simulate traces with a 10 sec time span. The traces are generated by the RTS tools [WT06c] and conform to the arrival curve specifications. The length of the history window Δ^h is five times the

period of an event stream. We evaluate two DPM schemes, i.e., switching to sleep with the HAD algorithm and switching back with the WCG or EDG algorithm, denoted as WCG-HAD and EDG-HAD. To show the effects of our schemes, we report the average idle power that is computed as the total idle energy consumption divided by the time span of the simulation. The average idle power is formally defined as follows:

Def. 11: (Average Idle Power Consumption) *Suppose that:* $E_{sw,on}$ *and* $E_{sw,slepp}$ *are the energy overheads for the switches to* `active` *and* `sleep` *modes, respectively;* C_{on} *and* C_{sleep} *are the total numbers of switches to* `active` *and* `sleep` *modes, respectively;* $T_{on,sum}$ *the sum of all the time intervals in which the device stays in* `active/standby` *modes; and* P_{σ} *is the power consumption in* `standby` *mode. The average idle power consumption is defined as:*

$$\frac{E_{sw,on} \cdot C_{on} + E_{sw,sleep} \cdot C_{sleep} + T_{on,sum} \cdot P_{\sigma}}{total\_time\_span} \tag{4.20}$$

For comparison, two other power management schemes described in 4.7 are measured as well, i.e., a periodic scheme (OPT) and a naive event-driven scheme (ED). The OPT scheme is a periodic power management (PPM) scheme which controls the processing core with a fixed on-off period. The lengths of the on and off periods are optimally computed with respect to the average idle power by an offline algorithm presented in the upcoming Section 4.7. The ED scheme turns on the processing core whenever an event arrives and turns off when the processing core becomes idle. Note that OPT does not consider the size of the system backlog. For a fair comparison, we smooth out the effect of a small backlog-size by setting the backlog size to a relatively large number, i.e., 60 events for this experiment.

4.5.3.2 Results

First, we show the effectiveness of the proposed WCG-HAD and EDG-HAD schemes comparing to the OPT and ED schemes. Fig. 60 shows the *normalized* values of average idle power with respect to OPT for the streams in Tab. 8 individually processed on the four devices in Tab. 9. As depicted in the figure, both schemes proposed in this section outperform the pure event-driven scheme as well as the OPT periodic scheme for all cases. On average, 25 % of the average idle power with respect to OPT is saved for the deadline factor $\chi = 1.6$.

In general, a small `wcet/period` ratio results in higher idle power savings, as the cases for event streams S_3, S_4, S_5, and S_9. Counter-examples are event streams S_2, S_7, and S_8. Event stream S_8 has the largest `wcet/period` ratio, conducting the worst idle power savings for

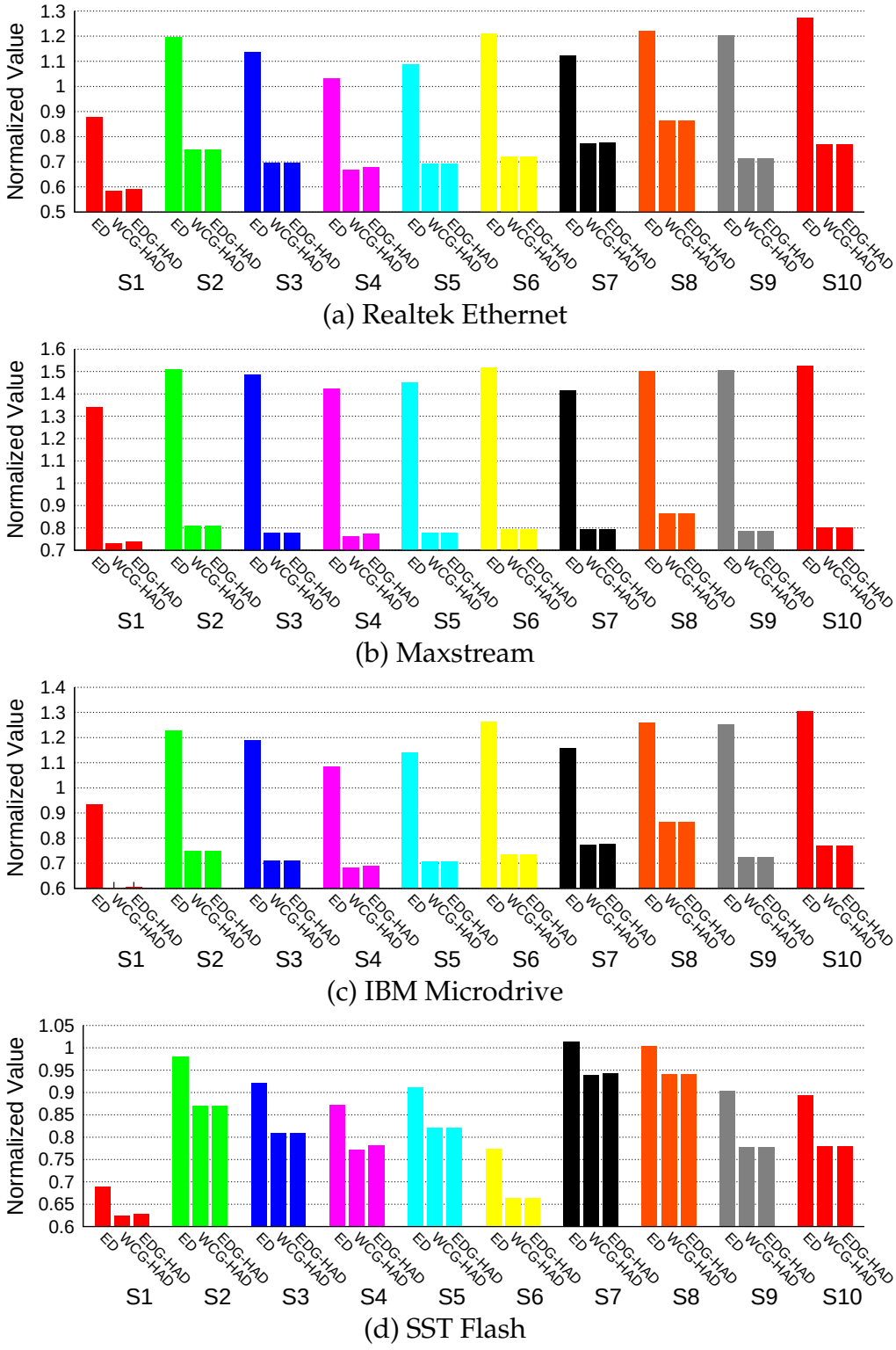
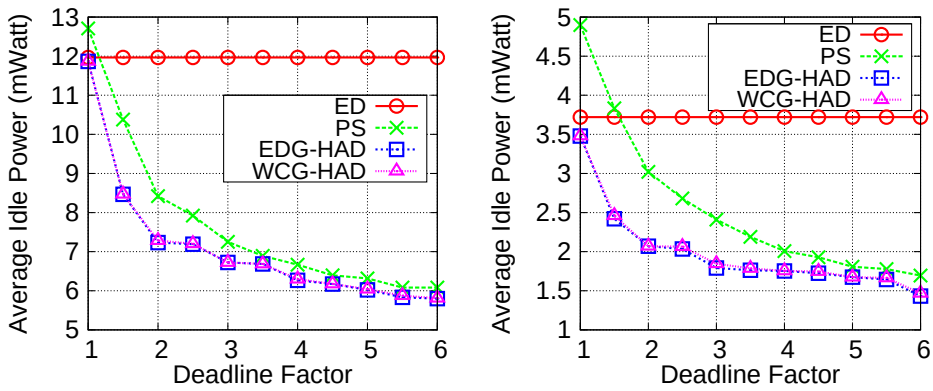


Fig. 60: Idle power consumption for single stream cases individually running on four different processing cores with deadline factor $\chi = 1.6$. The values are normalized to the OPT periodic scheme, that is, power consumption for the OPT periodic scheme is 1 in all cases.

all four devices. In addition, a smaller `delay/period` ratio also results in higher idle power savings, as the cases for event streams S_1 and S_6. Another observation is that the overhead caused by the break-even time does not really affect the optimization of the average idle power. As shown in Fig. 60, the normalized values for a given stream do not change significantly for different processing cores, although the break-even time is considerably different for the four processing cores, e.g., 18.2 *ms* for the SST Flash and 152 *ms* for the Maxstream.



(a) Stream S_8 (b) Stream S_4

Fig. 61: Average idle power consumption of different deadline settings on Realtek Ethernet.

We also outline how the average idle power changes when the relative deadline of a stream varies. Fig. 61 compares the four schemes when varying the deadline factor χ for streams S_8 and S_4. As shown in the figure, our online schemes again outperform the other two. Another observation is that OPT can achieve good results only when the relative deadline is large. For the cases of small relative deadlines, it can be worse than ED. Our online schemes, on the contrary, can smoothly handle different deadlines. The reason is that our online schemes consider the actual arrivals of event, resulting in a more precise analysis of the scheduling decision. Note that ideally our two online schemes, i.e., WCG-HAD and EDG-HAD, should produce identical results, because the WCG and EDG algorithms should theoretically converge to the same actual mode-switch decision, given the same trace. The slight deviations depicted in these two figures are due to the pessimistic activation decision in line 6 of Algorithm EDG. This deviation is expected to become larger in case of multiple streams.

4.6 Online DPM for Multiple Streams

When tackling multiple event streams, an essential problem is to compute the total service demand for a stream set $\mathcal{S}$. The total service demand for $\mathcal{S}$ does not only depend on the service demand of individual streams but also the scheduling policy and the system backlog organization. The scheduling in this section refers to a resource contention scheme, i. e., which backlogged event is chosen to be processed in the case that events of different streams are available in the system backlog at the same time. In this section, we consider two preemptive scheduling policies, i. e., earliest-deadline-first (EDF) and fixed-priority (FP). With respect to the backlog organization, two different schemes are investigated, referred to as individual and global backlog. In the case of individual backlog, each event stream S_i owns its private backlog with size Q_i. In the case of global backlog, all event streams in $\mathcal{S}$ share the same system backlog.

In this section, we present solutions for computing the total service of $\mathcal{S}$ by applying real-time interface theory. Again, the basic assumption of our solutions is that the deadline and backlog requirements for all event streams can be guaranteed if the processing core always provides full service, i.e., the processing core never turns to `sleep` mode. Without loss of generality, we consider a stream set $\mathcal{S}$ with N event streams, where $N \geq 2$. We present solutions for Algorithm EDG. Solutions for Algorithms HAD and WCG are similar to Algorithm EDG and can be easily adapted in the same manner. Note that the refinements of the history curve and backlog demand in (4.15) and (4.17) can be applied to every individual stream, denoted as H'_i and B'_i for brevity, respectively.

4.6.1 FP Scheduling with Individual Backlog

Unlike a system with a single event stream where the bounded delay is applied directly to the computed service demand of an event stream, we compute first the individual service demand of every stream, denoted as β^A_i, then derive the total service demand of the set $\mathcal{S}$, denoted as β^A_{total}. With the computed β^A_{total}, the bounded delay is applied to calculate the feasible sleep interval τ^*.

Without loss of generality, the event streams $S_1, S_2, \ldots, S_N$ in $\mathcal{S}$ are ordered according to their priorities, where the priority of stream S_i is higher than that of S_k when $i < k$. The processing of event streams in a FP scheduling policy domain can thereby be modeled as a chain of processing components ordered according to their priorities whereby a low priority stream can only make use of the resource left from the high priority streams. To compute the service demand of a high priority stream, a backward approach is applied by considering the service demand from

the low priority streams, as shown in Fig. 62.

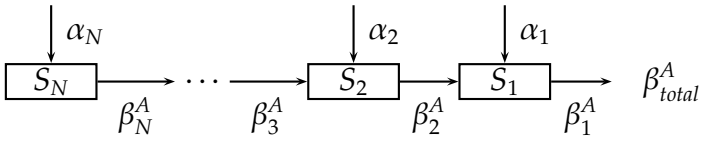


Fig. 62: The computation of the total service demand β^A_{total} for the FP scheduling with distributed backlog.

For the activation scheduling decision of the arrival of an event $e_{1,i}$, we first compute the service demand β^A_N of stream S_N at time $t^{\perp}_w$. We use the same approach presented in Section 4.5.2.2. To to compute β^A_N, formulas (4.15)–(4.19) are refined as follows:

$$\beta^A_N(\Delta, t^{\perp}_w) = \max\left\{\beta^{\flat}_N(\Delta, t^{\perp}_w), \beta^{\dagger}_N(\Delta, t^{\perp}_w)\right\}, \text{where} \tag{4.21}$$

$$\beta^{\flat}_N(\Delta, t^{\perp}_w) = \alpha^u_N(\Delta - D_N, t^{\perp}_w) + w_N \cdot B'_N(\Delta, t^{\perp}_w) \tag{4.22}$$

$$\beta^{\dagger}_N(\Delta, t^{\perp}_w) = \alpha^u_N(\Delta, t^{\perp}_w) - \left(Q_N - |\boldsymbol{E}_N(t^{\perp}_w)| - \bar{\alpha}^l_N(t^{\perp}_w - t^{\perp}_{e_{1,i}})\right) \cdot w_N \tag{4.23}$$

$$\alpha^u_N(\Delta, t^{\perp}_w) = w_N \cdot \left(\inf_{\lambda \geq 0}\left\{\bar{\alpha}^u_N(\Delta + \lambda) - H_N(\lambda, t^{\perp}_w)\right\}\right) \tag{4.24}$$

To derive β^A_1, we have to compute the service bounds $\beta^A_{N-1}, \beta^A_{N-2}, \ldots, \beta^A_2$, sequentially. Suppose that β^A_k has been derived, the resource constraint is that the remaining service curve $\beta^{\sharp}_{k-1}$ after serving S_{k-1} should be guaranteed to be no less than β^A_k, i.e.,

$$\beta^{\sharp}_{k-1}(\Delta) \geq \inf\left\{\beta : \beta^A_k(\Delta, t^{\perp}_w) = \sup_{0 \leq \lambda \leq \Delta}\{\beta(\lambda) - \alpha^u_{k-1}(\lambda, t^{\perp}_w)\}\right\} \tag{4.25}$$

By inverting (4.25), i. e. , deriving the minimum β in (4.25), we have:

$$\begin{aligned}\beta^{\sharp}_{k-1}(\Delta) &= \beta^A_k(\Delta - \lambda) + \alpha^u_{k-1}(\Delta - \lambda, t^{\perp}_w) \\ &\text{where } \lambda = \sup\left\{\tau : \beta^A_k(\Delta - \tau, t^{\perp}_w) = \beta^A_k(\Delta, t^{\perp}_w)\right\}\end{aligned} \tag{4.26}$$

To guarantee the timing constraint of event stream S_{k-1}, we also know that β^A_{k-1} must be no less than its own demand, i. e. , its deadline service demand $\beta^{\flat}_{k-1}$ and backlog-size service demand $\beta^{\dagger}_{k-1}$. The $\beta^{\flat}_{k-1}$, $\beta^{\dagger}_{k-1}$, and $\alpha^u_{k-1}(\Delta, t^{\perp}_w)$ can computed in a similar way as (4.15)–(4.19) in Section 4.5.2.2. Therefore, we have:

$$\beta^A_{k-1}(\Delta) = \max\left\{\beta^{\sharp}_{k-1}(\Delta), \beta^{\flat}_{k-1}(\Delta, t^{\perp}_w), \beta^{\dagger}_{k-1}(\Delta, t^{\perp}_w)\right\} \tag{4.27}$$

where

$$\beta^{\flat}_{k-1}(\Delta,\, t^{\perp}_{w}) = \alpha^{u}_{k-1}(\Delta - D_{k-1},\, t^{\perp}_{w}) + w_{k-1} \cdot B'_{k-1}(\Delta,\, t^{\perp}_{w}) \tag{4.28}$$

$$\beta^{\dagger}_{k-1}(\Delta,\, t^{\perp}_{w}) = \alpha^{u}_{k-1}(\Delta,\, t^{\perp}_{w}) - \Big(Q_{k-1} - |\boldsymbol{E_{k-1}}(t^{\perp}_{w})| - \bar{\alpha}^{l}_{k-1}(t^{\perp}_{w} - t^{\perp}_{e_{1,i}})\Big) \cdot w_{k-1} \tag{4.29}$$

$$\alpha^{u}_{k-1}(\Delta,\, t^{\perp}_{w}) = w_{k-1} \cdot \Big(\inf_{\lambda \geq 0}\Big\{\bar{\alpha}^{u}_{k-1}(\Delta + \lambda) - H_{k-1}(\lambda, t^{\perp}_{w})\Big\}\Big) \tag{4.30}$$

By applying (4.27) for $k = N-1, N-2, \ldots, 1$, the service demand β^{A}_{1} of stream S_1 is derived.

Based on this approach, the computed service demand for the highest priority stream S_1 can also be seen as the total service demand β^{A}_{total} for stream set $\mathcal{S}$ under FP scheduling. Therefore, the timing as well as backlog constraints for all streams in $\mathcal{S}$ can be guaranteed by a sleep interval τ^* such that $\boldsymbol{bdf}(\Delta,\, \tau^*)$ bounds β^{A}_{1}:

$$\tau^* = \max\Big\{\tau : \boldsymbol{bdf}(\Delta, \tau) \geq \beta^{A}_{1}(\Delta),\ \forall \Delta \geq 0\Big\} \tag{4.31}$$

4.6.2 EDF Scheduling with Individual Backlog

Again, we present the revision of the EDG algorithm as an example. For EDF scheduling, the total service demand β^{A}_{total} for all N streams can be bounded by the sum of their service demands like in FP scheduling. The β^{A}_{total} computed in this manner, however, is not sufficient to guarantee the backlog constraint of any stream in $\mathcal{S}$. When an event of a stream S_j happens to have the latest relative deadline, events in any stream of $\mathcal{S} \setminus \{S_j\}$ will be assigned a higher priority. S_j might suffer from backlog overflow in this case.

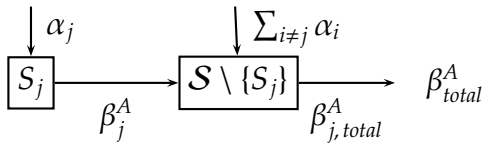


Fig. 63: The computation of the total service demand for EDF scheduling with distributed backlog.

To compute a correct service demand that satisfies the backlog constraint of stream S_j, S_j is considered as the stream with the lowest priority. Therefore, a backward approach similar to the previous section is applied, as shown in Fig. 63. Instead of tracing back stepwise, the service demand needed for high-priority streams is the sum of all streams from $\mathcal{S} \setminus \{S_j\}$.

The service $\beta^{\sharp}_{j}$ to guarantee the timing constraint of the lowest priority

stream S_j should be more than the demand β_j^A of S_j, i.e.,

$$\beta_j^\sharp(\Delta) \geq \inf\left\{\beta : \beta_j^A(\Delta, t_w^\perp) = \sup_{0\leq\lambda\leq\Delta}\{\beta(\lambda) - \sum_{i\neq j}^{N} \alpha_i^u(\lambda, t_w^\perp)\}\right\} \tag{4.32}$$

By inverting (4.32), we can derive $\beta_j^\sharp(\Delta)$ as:

$$\beta_j^\sharp(\Delta) = \beta_j^A(\Delta - \lambda, t_w^\perp) + \sum_{i\neq j}^{N} \alpha_i^u(\Delta - \lambda, t_w^\perp) \tag{4.33}$$

where $\lambda = \sup\left\{\tau : \beta_j^A(\Delta - \tau, t_w^\perp) = \beta_j^A(\Delta, t_w^\perp)\right\}$, and

$$\beta_j^A(\Delta, t_w^\perp) = \max\left\{\beta_j^\flat(\Delta, t_w^\perp), \beta_j^\dagger(\Delta, t_w^\perp), \beta_j^\sharp(\Delta)\right\} \tag{4.34}$$

where $\beta_j^\flat$ and $\beta_j^\dagger$ are from (4.28) and (4.29). To guarantee the timing constraint of all higher-priority streams, we also know that $\beta_{j,total}^A$ must be no less than the demand of $\mathcal{S} \setminus \{S_j\}$ as well. Therefore, we know that at time $t_w^\perp$,

$$\beta_{j,total}^A(\Delta) = \max\left\{\beta_j^\sharp(\Delta), \sum_{i\neq j}^{N} \beta_i^\flat(\Delta, t_w^\perp)\right\} \tag{4.35}$$

Applying (4.35) to each steam in $\mathcal{S}$, the service demand for each steam is computed. Because each stream could be the one with the lowest priority in the worst case, only the maximum of them can be seen as the total service demand for stream set $\mathcal{S}$. Therefore, a feasible sleep interval τ^* that guarantees both deadline and backlog-size constraints can be computed from the bounded delay function that bounds the maximum of individual streams:

$$\tau^* = \max\left\{\tau : \boldsymbol{bdf}(\Delta, \tau) \geq \max_{i\in N}\{\beta_{i,total}^A(\Delta)\}, \forall\Delta \geq 0\right\} \tag{4.36}$$

4.6.3 EDF Scheduling with Global Backlog

The approach to get the total service demand of $\mathcal{S}$ for global backlog is different from the approach for individual backlog. Without loss of generality, we assume that a backlog with size Q is shared by all event streams in $\mathcal{S}$.

For the HAD algorithm, because there is no backlog for each evaluation, the relative deadline for each event $e_{i,j}$ in every stream S_i

remains D_i. Therefore, the service demand to guarantee the deadline requirements of all streams is

$$\beta^{\flat}(\Delta) = \sum_{i=1}^{N} \alpha_i^u(\Delta - D_i, t^{\top}) \tag{4.37}$$

In the case of (4.13) of the WCG algorithm, the backlogs of different streams need to be considered. We apply the backlog demands for all streams:

$$\beta^{\flat}(\Delta) = \sum_{i=1}^{N} \Big(\alpha_i(\Delta - D_i,\, t^{\perp}) + w_i \cdot B_i(\Delta,\, t^{\perp})\Big) \tag{4.38}$$

The same applies to (4.18) of the EDG algorithm at time $t_w^{\perp}$.

Now we consider the backlog-size constraint. Besides the sum of all arrival curves, the constraint in (4.12) additionally needs to account for events with the longest execution time, i.e., $\max_{i \in N}\{w_i\}$. Therefore, it is revised as

$$\beta^{\dagger}(\Delta) = \sum_{i=1}^{N} \alpha_i^u(\Delta) - Q \cdot \max_{i \in N}\{w_i\} \tag{4.39}$$

The backlog constraint in (4.14) is more complex, because the backlog is not empty and contains events from different streams. The remaining capacity of the backlog is

$$\max_{i \in N}\{w_i\} \cdot Q - \sum_{j=1}^{|E(t^{\perp})|} \sum_{i=1}^{N} x_{i,j} \cdot w_i \tag{4.40}$$

where $x_{i,j} = 1,\ \forall j$ for Stream S_i, otherwise 0. Therefore, it is revised as

$$\beta^{\dagger}(\Delta) = \sum_{i=1}^{N} \alpha_i^u(\Delta, t^{\perp}) - \Big(\max_{i \in N}\{w_i\} \cdot Q - \sum_{j=1}^{|E(t^{\perp})|} \sum_{i=1}^{N} x_{i,j} \cdot w_i\Big) \tag{4.41}$$

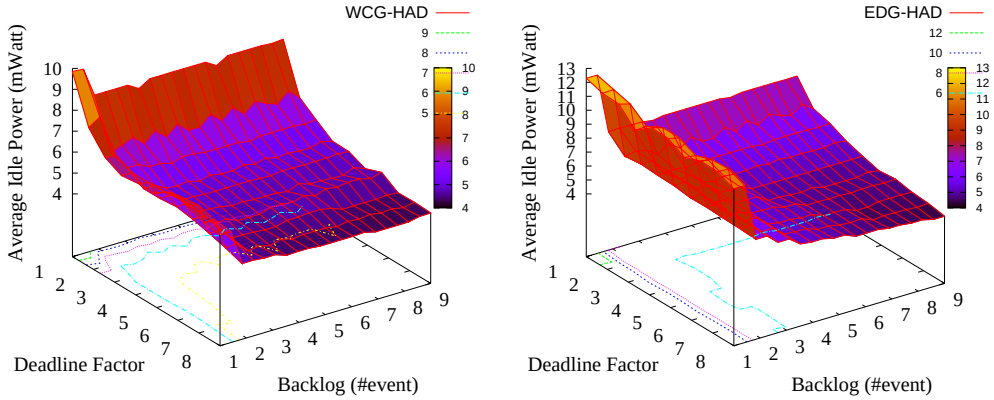
The last revision is (4.19) of the EDG algorithm, where the estimated future events of all streams need to be counted. Therefore it is revised as

$$\beta^{\dagger}(\Delta) = \sum_{i=1}^{N} \alpha_i^u(\Delta, t_w^{\perp}) - \Big(\max_{i \in N}\{w_i\} \cdot Q - \sum_{j=1}^{|E(t_w^{\perp})|} \sum_{i=1}^{N} x_{i,j} \cdot w_i - \sum_{i=1}^{N} \alpha_i^l(\epsilon)\Big) \tag{4.42}$$

4.6.4 Experimental Results

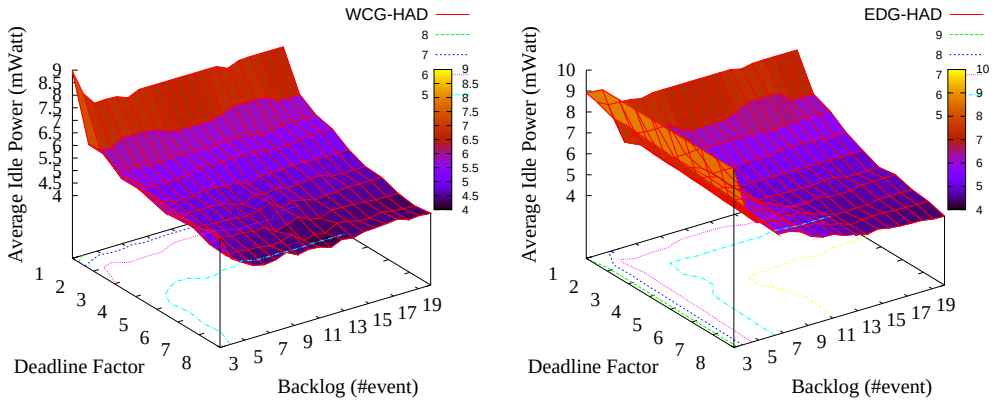
We present results for multiple event streams in this section. We report results only for random subsets of the stream set in Section 4.5.3.1. $\mathcal{S}(3,4)$, for instance, represents a case considering only the streams S_3 and S_4. For FP scheduling policy of a multiple-stream set, the stream index defines the priority of a stream. Considering again the stream set $\mathcal{S}(3,4)$, for instance, S_3 has a higher priority than S_4. Note that the history window Δ^h is set to five times of the longest period in a stream set.

Fig. 64 depicts simulation results for stream set $\mathcal{S}(6,9,10)$ running on Realtek Ethernet with individual and global backlog allocation schemes, respectively. Note that the smallest backlog sizes of Fig. 64(c) and 64(d) are set to three events, because the stream set used in this experiment



(a) Individual Backlog & EDF Scheduling

(b) Individual Backlog & FP Scheduling



(c) Global Backlog & EDF Scheduling

(d) Global Backlog & EDF Scheduling

Fig. 64: Average idle power consumption with respect to different deadline and backlog settings on Realtek Ethernet for the stream set $\mathcal{S}(6,9,10)$ under individual and global backlog allocations.

contains exactly three event streams. A backlog size less than three will result in no sleep interval for any relative deadline setting.

The results from Fig. 64 demonstrate the effectiveness of our solutions for multiple event streams. These results confirm the following statements: a) When the relative deadline and backlog size are small, the average idle power is large, where the chances to turn off the processing core are small. b) Increasing the relative deadline or backlog size individually helps reducing the idle power to only a certain degree. At a certain point, only marginal improvements can be achieved by further increasing the relative deadline or backlog size. c) Increasing both relative deadline and backlog size can effectively reduce the idle power, where more arrived events can be procrastinated and accumulated for each activation of the processing core.

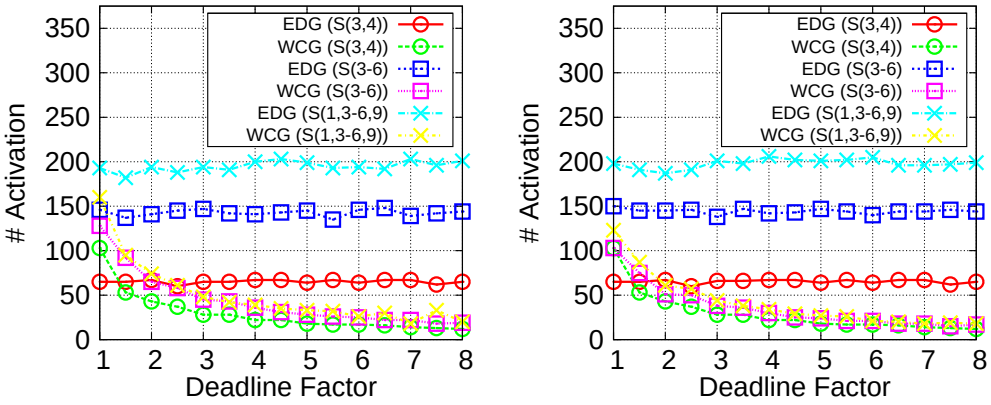
Another observation is that EDG is more sensitive than WCG on small backlog sizes for both global and individual backlog organization. As Figures 64(b) and 64(d) shown, when the backlog sizes increase from 1 to 2 and from 3 to 4 for individual and global backlog organization, respectively, the idle power drops significantly. The reason is the pessimistic activation decision in line 6 of Algorithm EDG.

We also demonstrate the computational efficiency of our schemes. Fig. 65 shows the numbers of activations of our algorithms over the 10 *sec* time span and Fig. 66 depicts the worst, best, and average case computational time for an activation.

From Fig. 65, one can conclude that given a single stream set the number of activations for Algorithm EDG does not much vary depending on the relative deadline, which can be expected due to the principle of the algorithm. The fluctuations are caused by event arrivals when the processing core is in `active` mode. Such events do not activate the algorithm. The second observation is that the number of activations for EDG depends on the number of streams running on the processing core while the number of activations for WCG quickly converges even when a stream set contains 3 times more event streams. The reason is that the activations of WCG are determined by the predicted turn-on moments that depend on the backlog size and relative deadline. When the backlog size and relative deadline are large enough, the number of activations is one, no matter how many streams are added to the system.

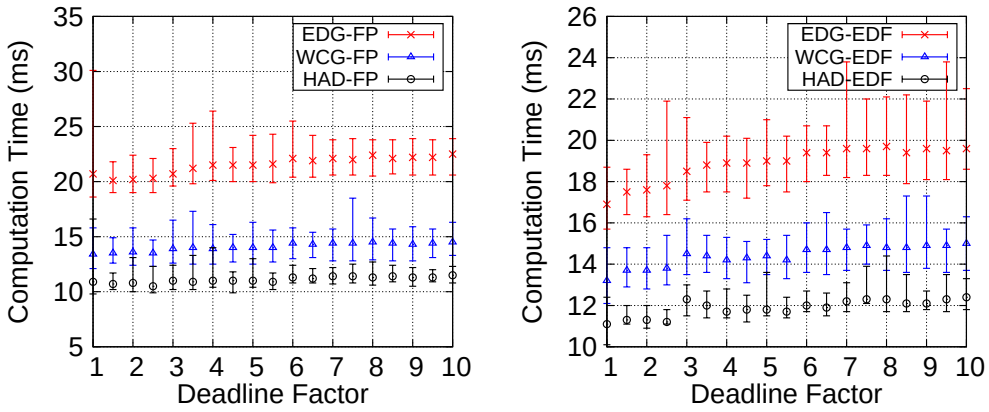
From the above facts, one might conclude that WCG is better. EDG is, however, meaningful in the case when event arrivals are sparse. In such cases, the number of activations of EDG will be less than that of WCG. The results shown in the figure are caused by the dense-event traces generated by the RTS tools.

Fig. 66 presents the worst, best, and average case computational time of an activation of the proposed algorithms with respect to different deadline



(a) EDF scheduling (b) FP scheduling

Fig. 65: Numbers of activations of different deadline settings for the stream sets $\mathcal{S}(3,4)$, $\mathcal{S}(3\text{–}6)$, and $\mathcal{S}(1,3\text{–}6,9)$ running on Realtek Ethernet under individual backlog organization with backlog size of 10 events for each event stream.



(a) individual backlog & FP scheduling (b) Global backlog & EDF scheduling

Fig. 66: Worst, best, and average case computation time of an activation of the proposed algorithms with respect to different deadline factors for three 4-stream sets $\mathcal{S}(1\text{–}4)$, $\mathcal{S}(3\text{–}6)$, and $\mathcal{S}(2,4,6,8)$ individually running on Realtek Ethernet.

factors for stream sets $\mathcal{S}(1\text{–}4)$, $\mathcal{S}(3\text{–}6)$, and $\mathcal{S}(2,4,6,8)$ individually running on Realtek Ethernet. Results for FP scheduling coupled with individual backlog scheme and EDF scheduling coupled with global backlog organization are shown in Fig. 66(a) and Fig. 66(b), respectively. We do not report the results for EDF scheduling with individual backlog organization due to the similarity to the FP case.

From the figure, we can conclude that our algorithms are efficient. The worst, best, and average case computation expenses of each activation are within the range of milliseconds and are acceptable for the stream set in

Tab. 8. Specifically, the time to evaluate a decision is almost constant even with large relative deadlines. In general, EDG is more expensive than WCG and HAD, which can be expected from the definition in Section 4.5. The last observation is that the computation time is not negligible. There are also means to tackle this problem, for instance, setting these computation overheads as a safe margin for the computed sleep period or making the activation itself the highest priority event stream of the system.

4.7 Offline Periodic DPM

Distinct from the online adaptive algorithms in the previous sections, we propose offline algorithms to derive optimal and approximated schemes for periodic dynamic power management (PPM) in this section. Unlike the online algorithms for which the activation/deactivation scheduling decisions heavily depend on the complexity of the arrival curves, the offline approaches off-load computation overhead to design time. The light run-time overhead of the periodic power management schemes is particularly suitable for embedded systems that only have limited power on computation. By simply using a hardware timer, an offline computed periodic power management scheme can be applied to the processing core. The control core is not necessary for the PPM in this section.

4.7.1 System Model and Problem Definition

An abstract model of our periodic power management (PPM) is illustrated in Fig. 67, in which power management schemes are derived by analyzing the arrival curves of event streams $\mathcal{S}$ statically. Specifically, we first decide a period $T = T_{on} + T_{off}$, then switch the system to `active/standby` mode for T_{on} time units, followed by T_{off} time units in `sleep` mode.

Based on model in Fig. 67, the energy consumption for a time interval L, where $L \gg T$ and $\frac{L}{T}$ is an integer, consists of mode-switch overheads, the energy in `standby` and `sleep` modes, and the energy for processing arrived events. Supposing that $\gamma_i(L)$ is the number of events of event stream S_i served in interval L and all the served events finish in time interval L, the energy consumption $E(L, T_{on}, T_{off})$ can be computed as

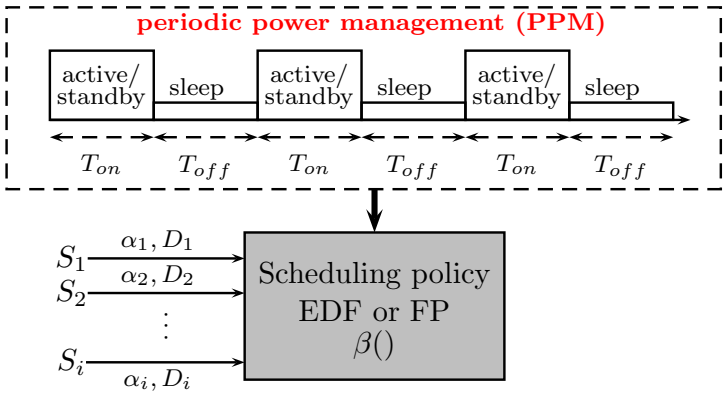


Fig. 67: The abstract model of the periodic power management problem.

follows:

$$\begin{aligned} E(L, T_{on}, T_{off}) &= \frac{L}{T_{on}+T_{off}}(E_{sw,on}+E_{sw,sleep}) \\ &\quad + \frac{L\cdot T_{on}}{T_{on}+T_{off}}P_s + \frac{L\cdot T_{off}}{T_{on}+T_{off}}P_\sigma \\ &\quad + \sum_{S_i\in\mathcal{S}} w_i\cdot\gamma_i(L)\cdot(P_a-P_s) \\ &= \frac{L\cdot E_{sw}}{T_{on}+T_{off}} + \frac{L\cdot T_{on}\cdot(P_s-P_\sigma)}{T_{on}+T_{off}} \\ &\quad + L\cdot P_\sigma + \sum_{S_i\in\mathcal{S}} w_i\cdot\gamma_i(L)\cdot(P_a-P_s) \end{aligned} \tag{4.43}$$

where E_{sw} is $E_{sw,on}+E_{sw,sleep}$ for brevity.

Based on (4.43), we define average idle power consumption.

Def. 12: (PPM Average Idle Power Consumption) *Given a sufficiently large L, the average idle power consumption is computed as:*

$$\begin{aligned} P(T_{on}, T_{off}) &\stackrel{def}{=} \frac{\frac{L\cdot E_{sw}}{T_{on}+T_{off}} + \frac{L\cdot T_{on}\cdot(P_s-P_\sigma)}{T_{on}+T_{off}}}{L} \\ &= \frac{E_{sw}+T_{on}\cdot(P_s-P_\sigma)}{T_{on}+T_{off}} \end{aligned} \tag{4.44}$$

According to (4.43), $L\cdot P_\sigma + \sum_{S_i\in\mathcal{S}} w_i\cdot\gamma_i(L)(P_a-P_s)$ is a constant for a given L. Therefore, for an L that is sufficiently large, without changing the scheduling policy, the minimization of energy consumption $E(L, T_{on}, T_{off})$ of a PPM scheme requires to find T_{on} and T_{off} so that the average idle power consumption $P(T_{on}, T_{off})$ is minimized. We now define the PPM problem studied in this section as follows:

Given a set of event streams $\mathcal{S}$ under real-time requirements, the objective of the studied problem is to find a periodic power management characterized by T_{on} and T_{off} that minimizes the average idle power consumption $P(T_{on}, T_{off})$, in which the response time of any event of event stream S_i in $\mathcal{S}$ must be no more than D_i.

4.7.2 Motivational Example

Consider a system processing a single stream S_1. The goal of our algorithms is to find a pair $(T_{on}, T_{off}) \in \mathbb{R}^+ \times \mathbb{R}^+$ so that on the one hand the average idle power consumption of the processing core is minimized, on the other hand the constructed β^G based on (T_{on}, T_{off}) bounds the service demand of the event stream S_1, i.e., $\beta^G \geq \beta^A = \alpha_1^u(\Delta - D_1)$. Note that for a given T_{on} and T_{off} pair, the guaranteed service of the processing core is given by:

$$\beta^G(\Delta) = \max\Big(\Big\lfloor \frac{\Delta}{T_{on} + T_{off}} \Big\rfloor \cdot T_{on},\ \Delta - \Big\lceil \frac{\Delta}{T_{on} + T_{off}} \Big\rceil \cdot T_{off}\Big) \tag{4.45}$$

By the service guarantee curve β^G in (4.45), the service demand curve of S_1, i.e., $\beta^A = \alpha_1^u(\Delta - D_1)$, and the schedulability definition in (4.2), the minimal T_{on} to fulfill the schedulability requirement in terms of a given T_{off} can be defined as:

$$T_{on}^{\min} = \min\Big\{T_{on} : \beta^G(\Delta) \geq \beta^A(\Delta),\ \forall \Delta \geq 0\Big\}. \tag{4.46}$$

To our best knowledge, there is no explicit form to compute T_{on}^{min}. Furthermore, due to the complex shape of the arrival curves, exhaustively testing (4.2) is the only way to determine the minimal average idle power from all possible T_{on}. Fig. 68 presents an example for illustrating the irregular pattern of the average idle power consumption by solving (4.46).

As a result, exhaustively checking all possible (T_{on}, T_{off}) pairs is the only way to find the exact minimum of the average idle power $P(T_{on}, T_{off})$. To reduce computational overhead, we propose a method to find an approximation with smaller complexity.

4.7.3 Bounded Delay Approximation

Again, we first consider a single event stream. The solution for multiple event streams is presented later on.

Reviewing the average idle power consumption in Def. 12, there are two cases for $P(T_{on}, T_{off})$: a) If $\frac{E_{sw}}{P_s - P_\delta} \geq T_{off}$, we know that $P(T_{on}, T_{off})$ is minimized when T_{on} is set to $+\infty$. b) If $\frac{E_{sw}}{P_s - P_\delta} < T_{off}$, the minimal T_{on} under the service constraint $\beta^G(\Delta)$ minimizes the average idle power

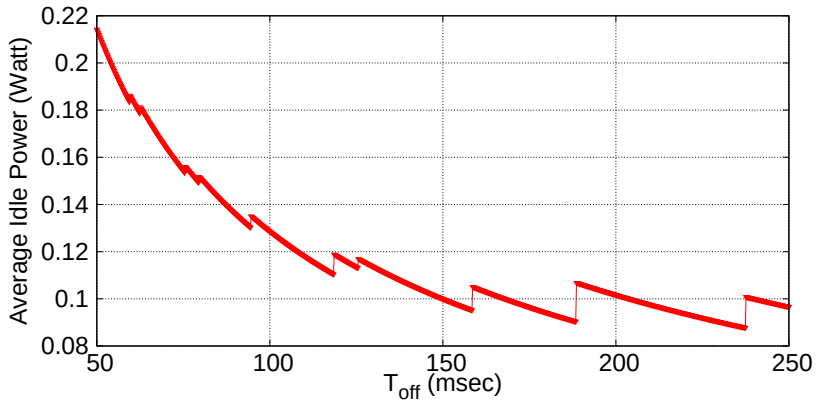


Fig. 68: The relation of the minimal average idle power consumption and T_{off} for an IMB Microdrive processing stream S_1, see Tab. 8 and 9 in Section 4.5.3.1.

consumption $P(T_{on}, T_{off})$. In this sense, $\frac{E_{sw}}{P_s - P_\delta}$, can be seen as the break-even time of the system.

Our proposed approaches are based on a) finding the minimal T_{on} by which the constructed β^G bounds a given β^A, provided that T_{off} is given, and b) the exploration of the best T_{off}. One could also derive solutions in another direction by searching the best T_{off} for a specified T_{on} along with the exploration on T_{on}, but the procedure would be more complicated.

4.7.3.1 Feasible Region of T_{off}

Before presenting how to find the optimal T_{off}, we will first discuss the feasible region of T_{off}. Intuitively, if T_{off} is smaller than the break-even time, i.e., $\frac{E_{sw}}{P_s - P_\delta}$, turning the processing core to `sleep` mode consumes more energy than keeping it in `active/standby` mode. In this case, a mode-switch does not pay off. Therefore, for searching the optimal T_{off}, the region $\left[0, \frac{E_{sw}}{P_s - P_\delta}\right]$ can be safely discarded. Moreover, as T_{off} must also satisfy the timing overheads for mode switches, we also know that T_{off} must be no less than t_{sw}, where $t_{sw} = t_{sw,sleep} + t_{sw,on}$.

There is also an upper bound for T_{off}. On the one hand, T_{off} should be smaller than $D_1 - c_1$. Otherwise, no event can be finished before its deadline. On the other hand, as the processing core provides no service when it is off, a maximum service $\beta^G_\top(\Delta) = \max\{0, \Delta - T_{off}\}$ is imposed. According to (4.2), we know that predicate

$$\beta^G_\top(\Delta) = \max\{0, \Delta - T_{off}\} \geq \beta^A_1(\Delta) = \alpha_1(\Delta - D_1) \tag{4.47}$$

must hold in order to satisfy the timing constraint. By inverting (4.47), we can compute the maximum T_{off} as

$$T^{max}_{off} = \max\left\{T_{off} : \beta^G_\top(\Delta) \geq \beta^A_1(\Delta),\ \forall \Delta \geq 0\right\}. \tag{4.48}$$

In summary, to find an optimal PPM, the feasible region of $T_{off} \in [T^l_{off}, T^r_{off}]$ can be bounded as follows:

$$T^l_{off} = \max\left\{t_{sw}, \frac{E_{sw}}{P_S - P_\delta}\right\} \tag{4.49}$$

$$T^r_{off} = \min\left\{D_1 - c_1, T^{max}_{off}\right\} \tag{4.50}$$

4.7.3.2 Approximation Construction

Instead of calculating the exact $T^{\min}_{on}$, we propose an alternative approach, namely a bounded-delay approximation, to find an approximated minimum, denoted as $\tilde{T}_{on}$. The basic idea of this approach is to compute a minimal *bounded-delay service curve* by which the minimal T_{on} is derived.

Def. 13: (Bounded Delay Service Curve) *A bounded-delay service curve* $\boldsymbol{bdf}(\Delta, \rho, T_{off})$*, defined by the slope* ρ *and the bounded-delay* T_{off} *for interval length* Δ*, is*

$$\boldsymbol{bdf}(\Delta, \rho, T_{off}) \stackrel{def}{=} \max\{0, \rho \cdot (\Delta - T_{off})\} \tag{4.51}$$

This definition is a generalized case of Def. 5 where the slope ρ is always 1. For a given bounded-delay function with slope ρ and bounded-delay T_{off}, we can construct an approximated T_{on} by

$$\tilde{T}_{on} = \frac{\rho \cdot T_{off}}{1 - \rho} \tag{4.52}$$

so that the resulting service curve β^G (constructed by 4.45) built from this $(\tilde{T}_{on}, T_{off})$ pair is no less than the minimal $\mathbf{bdf}(\Delta, \rho, T_{off})$ for any $\Delta \geq 0$. Fig. 69 illustrates an example for deriving $\tilde{T}_{on}$. Using the aforementioned definitions, the following lemma can be stated:

Lem. 4: *For specified* $T_{off} > 0$ *and* $0 < \rho \leq 1$*:*
(1) If $\boldsymbol{bdf}(\Delta, \rho, T_{off}) \geq \alpha^u_1(\Delta - D_1)$*, then,* $\boldsymbol{bdf}(\Delta, \rho', T_{off}) \geq \alpha^u_1(\Delta - D_1)$ *for any* $\rho' > \rho$*.*
(2) If $\boldsymbol{bdf}(\Delta, \rho, T_{off}) < \alpha^u_1(\Delta - D_1)$*, then,* $\boldsymbol{bdf}(\Delta, \rho', T_{off}) < \alpha^u_1(\Delta - D_1)$ *for any* $\rho' < \rho$*.*

Proof. Given a T_{off}, $\mathbf{bdf}(\Delta, \rho, T_{off})$ can be seen as a straight line in the Cartesian coordinate system, starting from point $(T_{off}, 0)$ with slope ρ to $+\infty$. Moreover, α^u_1 is sub-additive, according to the definition of an arrival curve. The lemma holds because of the monotonicity of both the bounded delay curve and arrival curves. Furthermore, the monotonicity also guarantees the existence and uniqueness of a tangent point of these two curves.

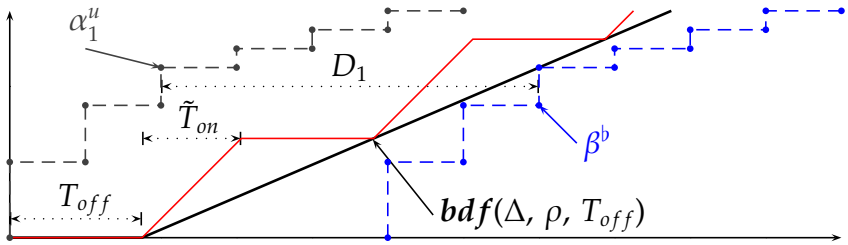


Fig. 69: An example for the bounded delay approximation, in which only part of the upper arrival curve $\alpha_1^u(\Delta)$ is presented for simplicity.

□

By (4.52) and Lemma 4, finding the minimal ρ, i.e., $\rho_{\min, T_{off}}$, under the constraint of a service demand β^A, is equivalent to the derivation of the minimal $\tilde{T}_{on}$ in the bounded-delay approximation, where

$$\rho_{\min, T_{off}} = \inf\left\{\rho : \mathbf{bdf}(\Delta, \rho, T_{off}) \geq \alpha_1^u(\Delta - D_1), \forall \Delta \geq 0\right\}.$$

Now we can formally define $\tilde{T}_{on}$ as follows.

Def. 14: *The minimal T_{on} obtained from the bounded-delay approximation is a function of T_{off}:*

$$\tilde{T}_{on} = \frac{T_{off} \cdot \rho_{\min, T_{off}}}{1 - \rho_{\min, T_{off}}} \stackrel{def}{=} f(T_{off}) \tag{4.53}$$

Based on Lemma 4, we can simply apply a binary search of ρ in the range of $[0, 1]$ to compute $\rho_{\min, T_{off}}$. Suppose that there are n possible values of ρ, the complexity of deriving $\rho_{\min, T_{off}}$ is $O(\log n)$. Therefore, the complexity to compute $\tilde{T}_{on}$ for a given T_{off} is $O(\log n)$ as well, instead of $O(n)$ in (4.46). This construction is practically useful as well, because verifying whether $\beta^G(\Delta) \geq \beta^A(\Delta)$ for all $\Delta \geq 0$ requires complex numerical computation which is time-consuming. Moreover, the derived $\tilde{T}_{on}$ has the nice property of being strictly increasing, which will be used to further reduce the time complexity for deriving the optimal PPM.

Lem. 5: *Given β^A, the function $f(T_{off})$ defined in (4.53) is strictly increasing and $\frac{T_{off}}{f(T_{off})} > \frac{(1+\epsilon)T_{off}}{f((1+\epsilon)T_{off})}$ for any $\epsilon > 0$.*

Proof. From the definition it follows that $\rho_{\min, T_{off}} < \rho_{\min, (1+\epsilon)T_{off}}$, and, thereby, $f(T_{off}) < f((1+\epsilon)T_{off})$ that proves the property when f is strictly monotonically increasing. Because $\rho_{\min, T_{off}} < \rho_{min, (1+\epsilon)T_{off}}$, we can derive $\frac{1}{1+\frac{T_{off}}{f(T_{off})}} < \frac{1}{1+\frac{(1+\epsilon)T_{off}}{f((1+\epsilon)T_{off})}}$, then, $\frac{T_{off}}{f(T_{off})} > \frac{(1+\epsilon)T_{off}}{f((1+\epsilon)T_{off})}$.

□

Based on the monotonicity of $f(T_{off})$, we claim that the objective function $P(T_{on}, T_{off})$ obtained from the application of bounded-delay approximated $\tilde{T}_{on}$ defined in Def. 14 is convex.

Thm. 6: *Using the bounded-delay algorithm approach to compute $\tilde{T}_{on} = f(T_{off})$ as depicted in (4.53), $P(\tilde{T}_{on}, T_{off}) = P(f(T_{off}), T_{off})$ is a convex function.*

Proof. The objective function $P(\tilde{T}_{on}, T_{off})$ can be split into two parts: $\frac{E_{sw}}{T_{on}+T_{off}}$ and $(P_s + P_\delta) \cdot \frac{T_{on}}{T_{on}+T_{off}}$. For the first part $\frac{E_{sw}}{T_{on}+T_{off}} = \frac{E_{sw}}{f(T_{off})+T_{off}}$, $f(T_{off}) + T_{off}$ is strictly increasing according to Lemma 5. Therefore $\frac{E_{sw}}{T_{on}+T_{off}}$ is a monotonically decreasing convex function. For the second part $(P_s + P_\delta) \cdot \frac{T_{on}}{T_{on}+T_{off}} = \frac{P_s+P_\delta}{1+\frac{T_{off}}{f(T_{off})}}$, according to Lemma 5, we know that $\frac{1}{1+\frac{T_{off}}{f(T_{off})}}$ is monotonically increasing and is a convex function as well. As a linear combination of convex functions is also a convex function, the original function $P(\tilde{T}_{on}, T_{off})$ is a convex function of T_{off}.

□

Thm. 6 allows to efficiently compute the optimal PPM, rather than exhaustively searching for every possible T_{off}. The complexity, for instance, is reduced to $O(\log n \cdot \log m)$ by applying a bisection search to the feasible region of T_{off}. The pseudo code of the algorithm is described in the Algorithm 6.

4.7.3.3 Optimal PPM

For comparison, we also present a brute-force algorithm, denoted as OPT, to find the minimal average idle power consumption $P(T_{on}, T_{off})$, by exhaustively searching all possible (T_{on}, T_{off}) pairs. The pseudo code of OPT is depicted in Algorithm 7.

Due to the irregular shape of $P(T_{on}, T_{off})$ as previously shown in Fig. 68, OPT is the only way to find the exact optimum for our PPM problem. Suppose that there are m possible T_{off} within the region $[T^l_{off}, T^r_{off}]$, the complexity of OPT is thereby $O(n \cdot m)$.

4.7.3.4 Multiple Streams Extension

When tackling multiple event streams, an essential problem is to compute the total service demand for a stream set $\mathcal{S}$. The total service demand for $\mathcal{S}$ does not only depend on the service demand of individual streams but also the scheduling policy of the processing core. For fixed priority

Algorithm 6 BDA

Input: $\alpha_1, D_1, T^l_{off}, T^r_{off}, \epsilon$
Output: T'_{on}, T'_{off}

if $T^r_{off} - T^l_{off} < \epsilon$ **then**
  if $P\Big(T^l_{off}, f(T^l_{off})\Big) < P\Big(T^r_{off}, f(T^r_{off})\Big)$ **then**
    return $\{T'_{on} \leftarrow f(T^l_{off}); T'_{off} \leftarrow T^l_{off}\}$
  else
    return $\{T'_{on} \leftarrow f(T^r_{off}); T'_{off} \leftarrow T^r_{off}\}$
  end if
end if
$\rho_l \leftarrow P'(T^l_{off}, f(T^l_{off}))$ ▷ P' is the derivative of P with respective to T_{off}
$\rho_m \leftarrow P'\Big(\frac{T^l_{off}+T^r_{off}}{2}, f(\frac{T^l_{off}+T^r_{off}}{2})\Big)$
if $\rho_l \cdot \rho_m > 0$ **then**
  $T^l_{off} \leftarrow T^m_{off}$
else
  $T^r_{off} \leftarrow T^m_{off}$
end if
recursively call BDA with the new T^l_{off} and T^r_{off}

Algorithm 7 OPT

Input: $\alpha_1, D_1, T^l_{off}, T^r_{off}, P_{min} = \infty, \epsilon$
Output: T'_{on}, T'_{off}

for $T_{off} = T^l_{off}$ to T^r_{off} step ϵ **do**
  exhaustively find T^{min}_{on} by testing (4.46)
  if $P(T^{min}_{on}, T_{off}) < P_{min}$ **then**
    $T'_{on} \leftarrow T^{min}_{on}; T'_{off} \leftarrow T_{off}$
    $P_{min} \leftarrow P(T^{min}_{on}, T_{off})$
  end if
end for

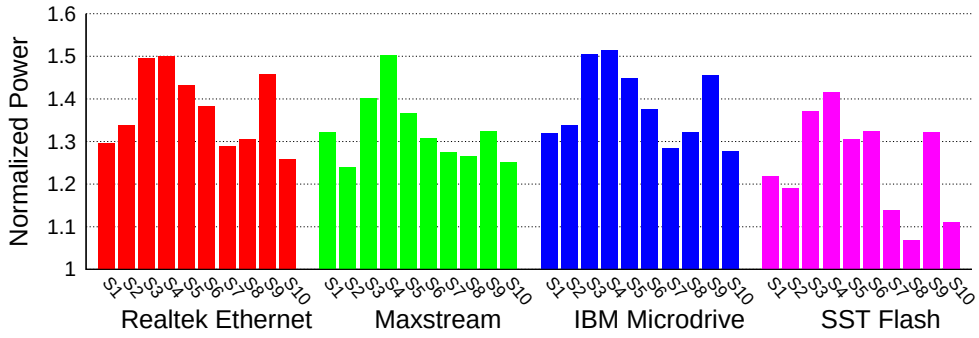
scheduling, we use the same approach as in Section 4.6.1 to compute the service demand of the event stream with the highest priority as the service demand for the stream set. For EDF scheduling, the total service bound is simply $\sum_{S_i \in \mathcal{S}} \alpha^u_i(\Delta - D_i)$, as we do not consider system backlog. For first-come first-serve (FCFS) scheduling, the service bound is $\sum_{S_i \in \mathcal{S}} \alpha^u_i(\Delta - D_{\min})$, where $D_{\min}$ is the minimal relative deadline of all event streams in $\mathcal{S}$.

4.7.4 Experimental Results

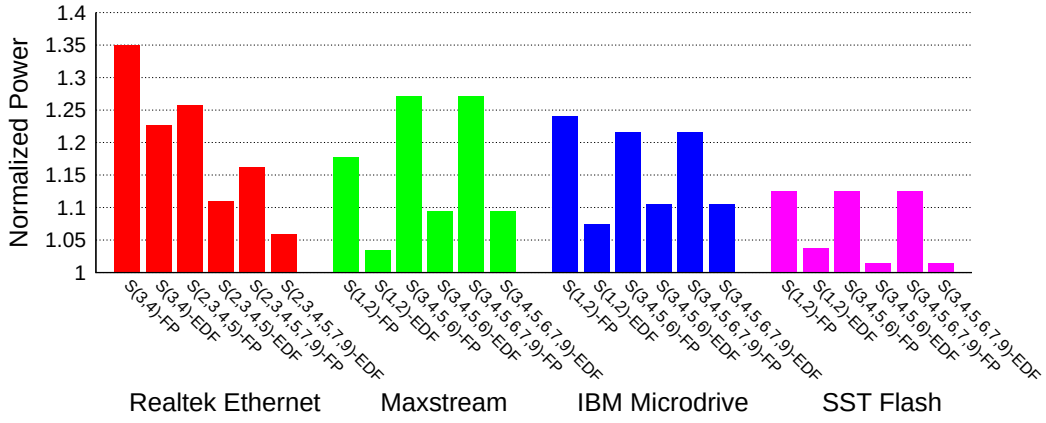
This section provides simulation results for the PPM schemes derived from the proposed BDA algorithm. All results are obtained from a simulation host with an Intel 1.7 *GHz* processor and 1 *GB* RAM.

We use the same stream set and devices as described in Tab. 8 and Tab. 9 in Section 4.5.3.1. As all PPM schemes derived from both OPT and BDA have the same energy consumption for event processing, we compare the average idle power defined in (4.44). We also report the computation time required to derive PPM schemes for both BDA and OPT.

First, we show the quality of the BDA approximation, i. e. the deviation of a BDA solution to the optimal solution. Fig. 70 shows the *normalized average idle power* of the PPM schemes derived by BDA with respect to those by OPT. We compare cases for a single event stream as well as for multiple event streams on four devices and different scheduling policies. As shown in the two figures, the PPM scheme derived by BDA reasonably approximates the optimal scheme obtained from OPT with



(a) Single-stream scenarios.

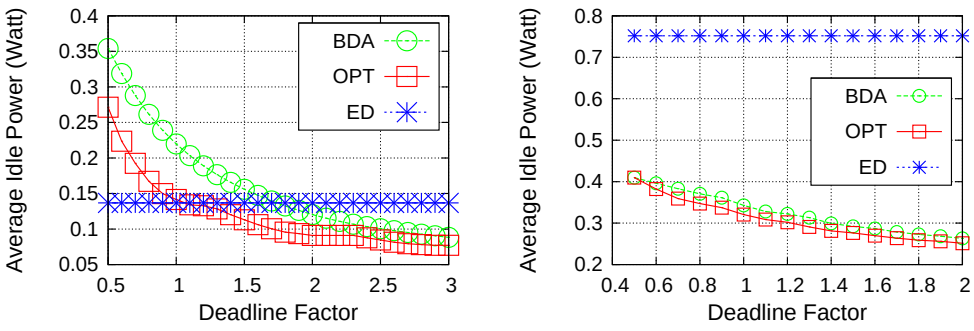


(b) Multiple-stream scenarios.

Fig. 70: Normalized average idle power of PPM schemes derived by BDA with respect to OPT with $\chi = 2$.

respect to different subsets of the stream set, devices, and scheduling policies. In general, BDA derives better schemes for multiple stream scenarios than for single stream scenarios. The reason is that with more streams involved, the computed service demand curve regresses to a more linear form, resulting in a closer match of the bounded delay function. Consequently, the PPM scheme derived from the bounded delay function better approximates the optimum.

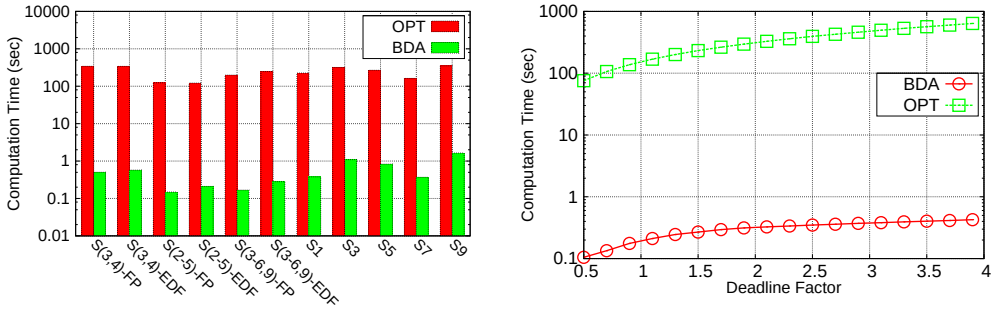
We also investigate how the average idle power changes as the relative deadlines of event streams vary. As our PPM schemes are *time-driven*, we use an event-driven (ED) scheme as a reference, where the device is turned to `sleep` mode when there is no event to be processed, and is awaken for event processing whenever an event arrives. Fig. 71(a) and Fig. 71(b) depict results for single and multiple stream scenarios, respectively. As shown in these figures, BDA effectively approximates the optimum. We also observe that the average idle power decreases as the relative deadline increases for both single and multiple stream cases. The reason is that for longer relative deadlines, more arrived events can be accumulated for each activation of the device. Another observation is that ED is effective for the single stream scenarios and short relative deadlines. As more streams are involved and the relative deadline increases, the PPM schemes derived from BDA outperform the ED scheme due to the reduction of mode-switches.



(a) stream S_8 (b) 10-stream set with EDF scheduling

Fig. 71: Average idle power of S_8 and a 10-stream set scenarios for the IBM Microdrive.

Second, we demonstrate the efficiency for deriving a minimal PPM scheme for BDA and OPT by reporting the computation time. Fig. 72(a) depicts the computation time for different stream combinations for a deadline factor $\chi = 2$, showing that BDA is about two orders of magnitude faster than OPT. Fig. 72(b) shows the relation of computation time and the deadline factor for the 10-stream scenario. As the figure shows, the computation time for OPT increases as the relative deadline increases, whereas the time remains in the same order of magnitude for BDA. Note



(a) $\chi = 2$ (b) 10-stream set with EDF scheduling

Fig. 72: Computation time of different scenarios for IBM Microdrive, where ϵ in OPT is set to 0.5.

that OPT can be much slower for a smaller granularity of ϵ in Algorithm 7. We can conclude that BDA is efficient.

4.8 Summary

This chapter explores system-level dynamic power management to reduce the static power consumption for hard real-time embedded systems. Considering both timing and backlog constraints, both offline and online algorithms are proposed that are applicable for embedded systems. Our offline algorithms optimally and approximately compute periodic power management schemes. Our online algorithms adaptively control the power mode of a system based on the actual arrival of events, tackling multiple event streams with irregular event arrival patterns under both earliest deadline first and fixed priority preemptive scheduling. Extensive simulation results demonstrate the effectiveness of our approaches.

Although we use a dual-core scenario to explain our algorithms, the approach itself can be applied to embedded systems with more computing cores. Considering the popular state-of-art tile-based platforms, for instance, ATMEL ShapOtto [HPB+] and Intel SCC [SCC] where each tile consists of two computational cores, our algorithms can be directly applied on every tile of the platforms. By the intrinsic compositionality of RTC, correlations between different tiles can be tackled by deriving output arrival curves of an event stream from each tile.

One might notice that the computing costs of our approach are not negligible, especially for Algorithm EDG. These computation expenses are caused by the expensive numerical (de)convolution for curve operations, for instance, for the derivation of history-aware arrival curves. To tackle this problem, fast mechanisms that are used in [LPT09],

for instance, can be exploited to replace the expensive numerical computation.

5

Conclusions

5.1 Main Results

The aim of this thesis is to address new challenges stemming from future many-core technologies for real-time embedded system. We categorize the challenges into a few major topics and provide corresponding solutions. The main contributions are summarized in the following:

- We present a programming model, which is suitable for streaming embedded systems. We adopt the Kahn process network for application modeling and design a hybrid programming syntax for the specification of a target system. To assist the design of many-core embedded systems, an iterator technique is developed by which a specified system can be arbitrarily scaled in a parametrized manner. To avoid the costly communication and synchronization overhead incurred by large scale process networks, we propose a variation of the FIFO syntax of Kahn process network. We also develop a parallel functional simulation as a proof-of-concept runtime environment for this programming model. The presented results lead to the free available software design toolchain in [dol].

- We investigate both analytic and simulation-based techniques for the performance evaluation of multi/many-core embedded systems at system level. For the analytic approach, we investigate modular performance analysis based on real-time calculus. We inspect timing correlations between data streams and present new methods to analyze correlated data streams that originate from a same source.

For simulation-based method, we develop a trace-based framework which can estimate the performance of large-scale embedded systems with reasonable time as well as high accuracy. Both the analytic method and the trace-based simulation can be served as a non-functional back-end for the aforementioned programming model. They can be embedded into an automated design space exploration to assist the software design of embedded systems.

- we also investigate system-level power-efficient design in particular under performance constraints. We propose offline and online algorithms for dynamic power management, targeting the reduction of static power consumption under hard real-time constraints. Our offline algorithms compute periodic power management scheme which switches the power mode of a system by a fixed period during the execution of the system. Alternatively, our online algorithms adaptively control the power mode of a system during the execution of the system. We also develop methods to tackle multiple event streams under both preemptive earliest-deadline-first and fixed-priority scheduling policies.

5.2 Future Perspectives

The contributions presented in this thesis provide partial solutions for a few topics for future many-core real-time embedded systems. Revisiting Fig. 2 in Section 1.3, more things deserve further investigation in order to complete the software design cycle. A few important future perspectives are listed in the following:

- The modern multi-core and future many-core (often heterogeneous) architecture of embedded systems is characterized by a large design space as there is a large degree of freedom in the partitioning of parallel application tasks, the allocation of concurrent hardware components, their binding to application processes, and the choice of appropriate resource allocation schemes. Because of the overall system complexity, inspecting all design alternatives of such a system, even analytically, is computationally prohibitive. Therefore, design space exploration techniques are critical to support important design decisions. The key for an efficient design space exploration is efficient search methods. An efficient search method requires not only fast convergence to the optimum but also fair distribution of the search results in the case of multi-objective optimization. To tackle this problem, evolutionary algorithm would be a viable candidate as a starting point.

- The model-based design methodology opens a gap between the system-level specifications and the actual implementation of a system, sometimes referred to as the implementation gap [HHBT09a]. Bringing this gap is challenging because on the one hand the semantics of a model-of-computation has to be correctly preserved on the other hand desired performance of the complete HW/SW system has to be achieved. In the case of performance analysis at system level, this gap is also valid between the system specification and the formal model of a system. Because the complexity of modern embedded systems increases, manually deriving the formal model of a system becomes error-prone. Therefore, an automated synthesis is desirable to close the gap.

- The energy issue is always valid. In this thesis, we consider only dynamic power management to reduce the static power consumption of a system. One interesting topic would be to combine dynamic power management and dynamic voltage/frequency scaling to reduce both dynamic and static power consumption.

- Last but not least, an important topic is the thermal issue. As more computing cores are integrated into a single chip, the power density of a chip is rapidly increased. The high temperature and heat generated by the high power density affect vital aspects of the multi/many-core embedded system pertaining to timing, reliability, fault-tolerance, and packaging and cooling costs. Therefore, thermal-aware design becomes crucial. A starting point at this direction could be to apply the techniques in Chapter 4 to consider thermal issue.

Bibliography

[AAB+00] B. Ackland, A. Anesko, D. Brinthaupt, S.J. Daubert, A. Kalavade, J. Knobloch, E. Micca, M. Moturi, C.J. Nicol, J.H. O'Neill, J. Othmer, E. Sackinger, K.J. Singh, J. Sweet, C.J. Terman, and J. Williams. A single-chip, 1.6-billion, 16-b mac/s multiprocessor dsp. *IEEE Journal of Solid-State Circuits*, 35(3):412–424, March 2000.

[ABM+04] Todd Austin, David Blaauw, Scott Mahlke, Trevor Mudge, Chaitali Chakrabarti, and Wayne Wolf. Mobile supercomputers. *IEEE Computer*, 37:81–83, 2004.

[AFM+02] T. Amnell, E. Fersman, L. Mokrushin, P. Petterson, and W. Yi. Times - a tool for medlling and implementation of embedded systems. In *Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 460–464, 2002.

[AIS04] John Augustine, Sandy Irani, and Chaitanya Swamy. Optimal power-down strategies. In *45th Symposium on Foundations of Computer Science (FOCS)*, pages 530–539, October 2004.

[ati] Ati steam computing–technical overview. `http://developer.amd.com/gpu_assets/Stream_Computing_Overview.pdf`.

[Bac77] John Backus. Can programming be liberated from the von neumann style?: a functional style and its algebra of programs. *ACM Turing award lectures*, 1977.

[Bap06] Philippe Baptiste. Scheduling unit tasks to minimize the number of idle periods: A polynomial time algorithm for offline dynamic power management. In

Proceedings of the 17th annual ACM-SIAM symposium on Discrete algorithm (SODA), pages 364–367, 2006.

[BDM09] G. Blake, R.G. Dreslinski, and T. Mudge. A survey of multicore processors. *IEEE Signal Processing Magazine*, 26(6):26–37, November 2009.

[BFH+04] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for gpus: stream computing on graphics hardware. In *ACM SIGGRAPH*, pages 777–786, New York, NY, USA, 2004. ACM.

[BFSS00] C. Brandolese, W. Fornaciari, F. Salice, and D. Sciuto. An instruction-level functionally-based energy estimation model for 32-bits microprocessors. In *Proceedings of the 37th Annual Design Automation Conference (DAC)*, pages 346–351, 2000.

[Bor07] Shekhar Borkar. Thousand core chips: a technology perspective. In *Proceedings of the 44th annual Design Automation Conference (DAC)*, pages 746–749, New York, NY, USA, 2007. ACM.

[BTM00] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: a framework for architectural-level power analysis and optimizations. In *Proceedings of the 27th International Symposium on Computer Architecture*, pages 83–94, 2000.

[Buc94] J.T. Buck. Static scheduling and code generation from dynamic dataflow graphs with integer-valued control streams. In *Conference Record of the Twenty-Eighth Asilomar Conference on Signals, Systems and Computers*, volume 1, pages 508–513, 1994.

[BWH+03] Felice Balarin, Yosinori Watanabe, Harry Hsieh, Luciano Lavagno, Claudio Passerone, and Alberto Sangiovanni-Vincentelli. Metropolis: An Integrated Electronic System Design Environment. *Computer*, 36(4):45–52, April 2003.

[BZCJ02] Amer Baghdadi, Nacer-Eddine Zergainoh, Wander O. Cesário, and Ahmed Amine Jerraya. Combining a Performance Estimation Methodology with a Hardware/Software Codesign Flow Supporting

Multiprocessor Systems. *IEEE Trans. on Software Engineering*, 28(9):822–831, 2002.

[CCBB09] Y.-K. Chen, C. Chakrabarti, S. Bhattacharyya, and B. Bougard. Signal processing on platforms with multiple cores: Part 1 - overview and methodologies. *IEEE Signal Processing Magazine*, 26(6):24–25, November 2009.

[CCS+08] J. Ceng, J. Castrillon, W. Sheng, H. Scharwachter, R. Leupers, G. Ascheid, H. Meyr, T. Isshiki, and H. Kunieda. Maps: An integrated framework for mpsoc application parallelization. In *Proceedings of the 45th ACM/IEEE Design Automation Conference(DAC)*, pages 754–759, June 2008.

[CCZ06] B. Chopard, P. Combes, and J. Zory. A conservative approach to systemc parallelization. In *International Conference on Computational Science (ICCS) Part IV*, pages 653–660, 2006.

[CG06] Hui Cheng and Steve Goddard. Online energy-aware I/O device scheduling for hard real-time systems. In *Proceedings of the 9th Design, Automation and Test in Europe (DATE)*, pages 1055–1060, 2006.

[CK07] Jian-Jia Chen and Tei-Wei Kuo. Procrastination determination for periodic real-time tasks in leakage-aware dynamic voltage scaling systems. In *Int'l Conf. on Computer-Aided Design (ICCAD)*, pages 289–294, 2007.

[CKT03] S. Chakraborty, S. Künzli, and L. Thiele. A general framework for analysing system properties in platform-based embedded system designs. In *Proc. 6th Design, Automation and Test in Europe (DATE)*, pages 190–195, March 2003.

[CoW] CoWare Virtual Platform Analyzer. `http://www.coware.com`.

[Cri] Criticalblue multicore cascade. `http://www.criticalblue.com/criticalblue_products/multicore.html`.

[Cru91a] R.L. Cruz. A calculus for network delay. *IEEE Trans. Information Theory*, 37(1):114–141, 1991.

[Cru91b] R.L. Cruz. A calculus for network delay. I. network elements in isolation. *IEEE Transactions on Information Theory*, 37(1):114–131, Jan 1991.

[CST09] Jian-Jia Chen, Nikolay Stoimenov, and Lothar Thiele. Feasibility analysis of on-line dvs algorithms for scheduling arbitrary event streams. In *Proceedings of the 30th IEEE Real-Time Systems Symposium (RTSS)*, 2009.

[CUD] nvidia cuda.
`http://www.nvidia.com/cuda`.

[DA08] Vinay Devadas and Hakan Aydin. On the interplay of dynamic voltage scaling and dynamic power management in real-time embedded applications. In *Proceedings of ACM International Conference on Embedded software (EMSOFT)*, pages 99–108, 2008.

[DA10] Vinay Devadas and Hakan Aydin. DFR-EDF: A unified energy management framework for real-time systems. *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 121–130, 2010.

[dol] Distributed Operation Layer.
`http://www.tik.ee.ethz.ch/~shapes`.

[EDK] Xilinx Platform Studio and the Embedded Development Kit, EDK version 7.1i edition.
`http://www.xilinx.com/ise/`.

[ELLSV97] Stephan Edwards, Luciano Lavagno, Edward A. Lee, and Alberto Sangiovanni-Vincentelli. Design of Embedded Systems: Formal Models, Validation, and Synthesis. *Proceedings of the IEEE*, 85(3):366–390, March 1997.

[ET06] Stephan A. Edwards and Olivier Tardieu. SHIM: A Deterministic Model for Heterogeneous Embedded Systems. *IEEE Trans. VLSI Syst.*, 14(8):854–867, August 2006.

[FFS01] Alessandro Fin, Franco Fummi, and Denis Signoretto. The use of SystemC for design verification and integration test of IP-cores. In *Proceedings of the 14th Annual IEEE International ASIC/SoC Conference*, pages 76–80, September 2001.

[FSSZ01] William Fornaciari, Donatella Sciuto, Cristina Silvano, and Vittorio Zaccaria. A Design Framework to Efficiently Explore Energy-Delay Tradeoffs. In *Proc. of the international symposium on Hardware/software codesign (CODES)*, pages 260–265, 2001.

[GB03] Marc Geilen and Twan Basten. Requirements on the execution of Kahn process networks. In *Proceedings of the 12th European conference on Programming (ESOP)*, pages 319–334, Berlin, Heidelberg, 2003. Springer-Verlag.

[GH99] B. Gennart and R. Hersch. Computer-aided synthesis of parallel image processing applications. In *Proceedings Conf. Parallel and Distributed Methods for Image Processing III, SPIE Int. Symp. on Opt. Science, Denver, SPIE Vol-3817, 48-61*, 1999.

[GHF+06] Michael Gschwind, H. Peter Hofstee, Brian Flachs, Martin Hopkins, Yukio Watanabe, and Takeshi Yamazaki. Synergistic processing in cell's multicore architecture. *IEEE Micro*, 26:10–24, 2006.

[GHGGPGDM01] Michael González Harbour, José Javier Gutiérrez García, José Carlos Palencia Gutiérrez, and José María Drake Moyano. MAST: Modeling and Analysis Suite for Real Time Applications. In *Proc. Euromicro Conference on Real-Time Systems*, pages 125–134, Delft, The Netherlands, June 2001.

[GHV99] Tony D. Givargis, Henkel Henkel, and Frank Vahid. Interface and Cache Power Exploration for Core-Based Embedded System Design. In *Proc. of the IEEE/ACM int'l conf. on Computer-Aided Design (ICCAD)*, page 270, 1999.

[GP09] Xavier Guerin and Frédéric Petrot. A system framework for the design of embedded software targeting heterogeneous multi-core socs. In *Proceedings of the*

2009 20th IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP), pages 153–160, Washington, DC, USA, 2009. IEEE Computer Society.

[Gri04] Matthias Gries. Methods for evaluating and covering the design space during early design development. *Journal of Integrated VLSI*, 38(2):131–183, 2004.

[Grü06] David Grünert. Window based fifos for communication in on-chip multiprocessor systems. Master's thesis, ETH Zürich, Switzerland, 2006.

[Hai10] Wolfgang Haid. *Design and Performance Analysis of Multiprocessor Streaming Applications*. PhD thesis, ETH Zurich, 2010.

[HE05] Arne Hamann and Rolf Ernst. Tdma time slot and turn optimization with evolutionary search techniques. In *Proceedings of the 8th Design, Automation and Test in Europe (DATE)*, pages 312–317, 2005.

[HHBT09a] Wolfgang Haid, Kai Huang, Iuliana Bacivarov, and Lothar Thiele. Multiprocessor soc software design flows. *IEEE Signal Processing Magazine*, 26(6):64–71, 2009.

[HHBT09b] Kai Huang, Wolfgang Haid, Iuliana Bacivarov, and Lothar Thiele. Coupling MPARM with DOL. Technical Report 314, D-ITET, TIK, ETH Zürich, Nov. 2009.

[HHLA07] Jin Heo, Dan Henriksson, Xue Liu, and Tarek Abdelzaher. Integrating adaptive components: An emerging challenge in performance-adaptive systems and a server farm case-study. In *Proceedings of the 28th IEEE Real-Time Systems Symposium (RTSS)*, pages 227–238, 2007.

[HKH+09] Wolfgang Haid, Matthias Keller, Kai Huang, Iuliana Bacivarov, and Lothar Thiele. Generation and calibration of compositional performance analysis models for multi-processor systems. In *Proc. Intl Conference on Systems, Architectures, Modeling and Simulation (SAMOS)*, pages 92–99, Samos, Greece, 2009.

[HKL+07] Soonhoi Ha, Sungchan Kim, Choonseung Lee, Youngmin Yi, Seongnam Kwon, and Young-Pyo Joo. Peace: A hardware-software codesign environment for multimedia embedded systems. *ACM Trans. Des. Autom. Electron. Syst.*, 12(3):1–25, 2007.

[HPB+] Wolfgang Haid, Pier Paolucci, Iuliana Bacivarov, Kai Huang, Francesco Simula, Piero Vicini, Xavier Guerin, Alexandre Chagoya-Garzon, Stefan Kraemer, and Rainer Leupers. Shapes: A scalable HW/SW platform for embedded systems. *Unpublished*.

[HSC+09] Kai Huang, Luca Santinelli, Jian-Jia Chen, Lothar Thiele, and Giorgio C. Buttazzo. Periodic power management schemes for real-time event streams. In *the 48th IEEE Conf. on Decision and Control (CDC)*, pages 6224–6231, Shanghai, China, 2009.

[HSH+09] Wolfgang Haid, Lars Schor, Kai Huang, Iuliana Bacivarov, and Lothar Thiele. Efficient execution of kahn process networks on multi-processor systems using protothreads and windowed fifos. In *Proc. IEEE Workshop on Embedded Systems for Real-Time Multimedia (ESTIMedia)*, pages 35–44, Grenoble, France, 2009.

[HT07] Wolfgang Haid and Lothar Thiele. Complex Task Activation Schemes in System Level Performance Analysis. In *Proc. Int'l Conf. on Hardware/Software Codesign and System Synthesis (CODES/ISSS)*, pages 173–178, Salzburg, Austria, October 2007.

[Int] ISO/IEC 13818-2: Information technology — Generic Coding of moving pictures and associated audio information — Part 2: Video.

[ISG03] Sandy Irani, Sandeep Shukla, and Rajesh Gupta. Algorithms for power savings. In *Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 37–46, 2003.

[ITR] International technology roadmap for semiconductors 2007 edition: System drivers. `http://www.itrs.net/Links/2007ITRS/2007_Chapters/2007_SystemDrivers.pdf`.

[Jai81] A.K. Jain. Image data compression: A review. *Proceedings of the IEEE*, 69(3):349–389, march 1981.

[JP08] F. Ryan Johnson and Joann M. Paul. Interrupt Modeling for Efficient High-Level Scheduler Design Space Exploration. *ACM Trans. Des. Autom. Electron. Syst.*, 13(1):1–22, 2008.

[JPG04] R. Jejurikar, C. Pereira, and R. Gupta. Leakage aware dynamic voltage scaling for real-time embedded systems. In *ACM/IEEE Design Automation Conference (DAC)*, pages 275–280, 2004.

[JRE04] M. Jersak, R.Henai, and R. Ernst. Context-aware performance analysis for efficient embedded system design. In *Proc. Design, Automation and Test in Europe (DATE)*, March 2004.

[KAG+09] L.J. Karam, I. AlKamal, A. Gatherer, G.A. Frantz, D.V. Anderson, and B.L. Evans. Trends in multicore dsp platforms. *IEEE Signal Processing Magazine*, 26(6):38–49, November 2009.

[Kah74] Gilles Kahn. The Semantics of a Simple Language for Parallel Programming. In *Proceedings of IFIP Congress*, North Holland Publishing Co, 1974.

[KB09] Hahn Kim and Robert Bond. Multicore software technologies. *IEEE Signal Processing Magazine*, 26(6):80–89, November 2009.

[KBC09] Branislav Kisačanin, Shuvra S. Bhattacharyya, and Sek Chai, editors. *Embedded Computer Vision*. Springer London, 2009.

[KDVvdW97] Bart Kienhuis, Ed Deprettere, Kees Vissers, and Pieter van der Wolf. An Approach for Quantitative Analysis of Application-Specific Dataflow Architectures. In *Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP)*, pages 338–349, July 1997.

[KKO+06] Tero Kangas, Petri Kukkala, Heikki Orsila, Erno Salminen, Marko Hännikäinen, and Timo D. Hämäläinen. UML-Based Multiprocessor SoC Design Framework. *ACM Transaction on Embedded Comp. Systems*, 5(2):281–320, May 2006.

[KNRSV00] K. Keutzer, A.R. Newton, J.M. Rabaey, and A. Sangiovanni-Vincentelli. System-Level Design: Orthogonalization Of Concerns and Platform-Based Design. *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, 19(12):1523–1543, Dec 2000.

[KPBT06] S. Künzli, F. Poletti, L. Benini, and L. Thiele. Combining simulation and formal methods for system-level performance analysis. In *Proc. Design, Automation and Test in Europe (DATE)*, March 2006.

[Kü06] Simon Künzli. *Efficient Design Space Exploration for Embedded Systems*. PhD thesis, ETH Zurich, 2006.

[LAB+04] Mirko Loghi, Federico Angiolini, Davide Bertozzi, Luca Benini, and Roberto Zafalon. Analyzing on-chip communication in a MPSoC environment. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition Volume II (DATE)*, pages 752–757, 2004.

[LGS+95] C. Loiseaux, S. Graf, J. Sifakis, A. Bouajjani, and S. Bensalem. Property preserving abstractions for the verification of concurrent systems. *Form. Methods Syst. Des.*, 6(1):11–44, 1995.

[LM87] Edward Ashford Lee and David G. Messerschmitt. Synchronous Data Flow. In *Proceedings of the IEEE*, volume 75, pages 1235–1245, January 1987.

[LN00] Edward A. Lee and Steve Neuendorffer. MoML: A Modeling Markup Language in XML. Version 0.4. Technical Report UCB/ERL M00/12, University of California at Berkeley, March 2000.

[LP95] Edward A. Lee and Thomas M. Parks. Dataflow Process Networks. *Proceedings of the IEEE*, 83(5):773–799, May 1995.

[LPT09] Kai Lampka, Simon Perathoner, and Lothar Thiele. Analytic real-time analysis and timed automata: A hybrid method for analyzing embedded real-time systems. In *Proceedings of ACM international conference on Embedded software (EMSOFT)*, pages 107–116, Grenoble, France, 2009. ACM.

[LRD01] K. Lahiri, A. Raghunathan, and S. Dey. System level performance analysis for designing on-chip communication architectures. *IEEE Transactions on Computer Aided-Design of Integrated Circuits and Systems*, 20(6):768–783, 2001.

[LSV98] Edward A. Lee and Alberto Sangiovanni-Vincentelli. A Framework for Comparing Models of Computation. *IEEE Transaction on Computer-Aided Design of Integrated Circuits Systems*, 17(12):1217–1229, December 1998.

[LT01] J.Y. Le Boudec and P. Thiran. *Network Calculus - A Theory of Deterministic Queuing Systems for the Internet*. LNCS 2050, Springer Verlag, 2001.

[MCT05] Alexander Maxiaguine, Samarjit Chakraborty, and Lothar Thiele. DVS for buffer-constrained architectures with predictable QoS-energy tradeoffs. In *the International Conference on Hardware-Software Codesign and System Synthesis (CODES+ISSS)*, pages 111–116, 2005.

[MDC+05] Samy Meftali, Anouar Dziri, Luc Charest, Philippe Marquet, and Jean-Luc Dekeyser. SOAP based distributed simulation environment for System-on-Chip (SoC) design. In *Forum on Specification and Design Languages (FDL)*, December 2005.

[MJU+09] M. Mehrara, T. Jablin, D. Upton, D. August, K. Hazelwood, and S. Mahlke. Multicore compilation strategies and challenges. *IEEE Signal Processing Magazine*, 26(6):55–63, November 2009.

[MKT04] Alexander Maxiaguine, Simon Künzli, and Lothar Thiele. Workload characterization model for tasks with variable execution demand. In *Proceedings of the 7th Design, Automation and Test in Europe (DATE)*, 2004.

[MLMK05] Makoto Murata, Dongwon Lee, Murali Mani, and Kohsuke Kawaguchi. Taxonomy of xml schema languages using formal language theory. *ACM Trans. Internet Technol.*, 5(4):660–704, 2005.

[mpe] Reference MPEG-2 Decoder. `http://www.mpeg.org/MPEG/video/mssg-free-mpeg-software.html`.

[MPI] Message passing interface. `http://www.mpi-forum.org`.

[MPND02] S. Mohanty, V. K. Prasanna, S. Neema, and J. Davis. Rapid Design Space Exploration of Heterogeneous Embedded Systems Using Symbolic Search and Multi-Granular Simulation. In *Proc. of the Joint Conference on Languages, Compilers and Tools for Embedded Systems (LCTES/SCOPES)*, pages 18–27, June 2002.

[MPS98] E. Macii, M. Pedram, and F. Somenzi. High-level power modeling, estimation, and optimization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(11):1061–1079, nov 1998.

[NSD08] Hristo Nikolov, Todor Stefanov, and Ed Deprettere. Systematic and Automated Multiprocessor System Design, Programming, and Implementation. *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, 27(3):542–555, March 2008.

[Ope] Openmp: Api specification for parallel programming. `http://openmp.org`.

[Pao06] P.S. Paolucci. The Diopsis Multiprocessor Tile of SHAPES. *MPSOC'06, 6th Int. Forum on Application-Specific MPSoC*, 2006.

[PEP+04] Paul Pop, Petru Eles, Zebo Peng, Viacheslav Izosimov, Magnus Hellring, and Olof Bridal. Design Optimization of Multi-Cluster Embedded Systems for Real-Time Applications. In *Design, Automation and Test in Europe (DATE)*, pages 1028–1033, 2004.

[PEP06] Andy D. Pimentel, Cagkan Erbas, and Simon Polstra. A Systematic Approach to Exploring Embedded System Architectures at Multiple Abstraction Levels. *IEEE Trans. Comput.*, 55(2):99–112, February 2006.

[PGH98] J.C. Palencia and M. Gonzalez Harbour. Schedulability analysis for tasks with static and dynamic offsets. In *19th IEEE Real-Time Systems Symposium (RTSS)*, pages 26–37, 1998.

[PH09] David A. Patterson and John L. Hennessy. *Computer Organization and Design, Fourth Edition*. Morgan Kaufmann, 2009.

[PHL+01] Andy D. Pimentel, Louis O. Hertzberger, Paul Lieverse, Pieter van der Wolf, and Ed F. Deprettere. Exploring Embedded-Systems Architectures with Artemis. *Computer*, 34(11):57–63, 2001.

[pic] picochip pc205. `http://www.picochip.com/page/76/Multi-core-PC205`.

[PM93] W.B. Pennebacker and J.L. Mitchel. *JPEG Still Image Data Compression Standard*. Van Nostrand Reinhold, New York, 1993.

[PRT+10] Simon Perathoner, Tobias Rein, Lothar Thiele, Kai Lampka, and Jonas Rox. Modeling structured event streams in system level performance analysis. In *ACM SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems (LCTES)*, pages 37–46, Stockholm, Sweden, 2010. ACM.

[PS05] H.D. Patel and S.K. Shukla. Towards a heterogeneous simulation kernel for system-level models: a systemc kernel for synchronous data flow models. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 24(8):1261–1271, August 2005.

[pto] Ptolemy Project Home Page. `http://ptolemy.eecs.berkeley.edu`.

[RE02] K. Richter and R. Ernst. Event model interfaces for heterogeneous system analysis. In *Proc. 5th Design, Automation and Test in Europe (DATE)*, pages 506–513, March 2002.

[RGB+08] Martino Ruggiero, Alessio Guerri, Davide Bertozzi, Michela Milano, and Luca Benini. A Fast and Accurate Technique for Mapping Parallel Applications on Stream-Oriented MPSoC Platforms with Communication Awareness. *Int'l Journal of Parallel Programming*, 36(1):3–36, February 2008.

[RTE] RTEMS Steering Committee. RTEMS Home Page.

[SC05] Vishnu Swaminathan and Krishnendu Chakrabarty. Pruning-based, energy-optimal, deterministic I/O device scheduling for hard real-time systems. *ACM Transactions in Embedded Computing Systems*, 4(1):141–167, 2005.

[SCC] Single-chip cloud computer. http://techresearch.intel.com/articles/Tera-Scale/1826.htm.

[SE09] Simon Schliecker and Rolf Ernst. A recursive approach to end-to-end path latency computation in heterogeneous multiprocessor systems. In *Proc. 7th International Conference on Hardware Software Codesign and System Synthesis (CODES-ISSS)*, Grenoble, France, October 2009. ACM.

[SEDN05] Aviral Shrivastava, Eugene Earlie, Nikil Dutt, and Alex Nicolau. Aggregating processor free time for energy reduction. In *Proceedings of the 3rd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis (CODES+ISSS)*, pages 154–159, 2005.

[SL04] Insik Shin and Insup Lee. Compositional Real-Time Scheduling Framework. In *Proceedings of the Real-Time Systems Symposium (RTSS)*, pages 57–67. IEEE Press, 2004.

[Soc05] IEEE Computer Society. IEEE 1666 Standard SystemC Language Reference Manual. http://standards.ieee.org/getieee/1666/index.html, December 2005.

[SSG02] N. Savoiu, S.K. Shukla, and R.K. Gupta. Automated concurrency re-assignment in high level system models for efficient system-level simulation. *Proceedings of Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pages 875–881, 2002.

[Ste04] Todor Stefanov. Converting weakly dynamic programs to equivalent process network specifications, September 2004. Ph.D. dissertation book, Leiden University, Leiden, The Netherlands, September 2004, ISBN: 90-9018629-8.

[sys] The Open SystemC Initiative (OSCI). http://www.systemc.org.

[SZT+04] T. Stefanov, C. Zissulescu, A. Turjan, B. Kienhuis, and E. Deprette. System design using khan process networks: the compaan/laura approach. In *Proceedings*

of the Design, Automation and Test in Europe (DATE), pages 3400–345 Vol.1, February 2004.

[TBHH07] Lothar Thiele, Iuliana Bacivarov, Wolfgang Haid, and Kai Huang. Mapping applications to tiled multiprocessor embedded systems. In *Proc. 7th Intl Conference on Application of Concurrency to System Design (ACSD)*, pages 29–40, Bratislava, Slovak Republic, 2007.

[TC94] Ken Tindell and John Clark. Holistic schedulability analysis for distributed hard real-time systems. *Microprocessing and Microprogramming*, 40(2–3):117–134, 1994. Parallel Processing in Embedded Real-time Systems.

[TCN00] Lothar Thiele, Samarjit Chakraborty, and Martin Naedele. Real-time calculus for scheduling hard real-time systems. In *Proc. IEEE International Symposium on Circuits and Systems (ISCAS)*, volume 4, pages 101–104, 2000.

[TI-] TI OMAP 4440. `http://focus.ti.com/lit/ml/swpt034/swpt034.pdf`.

[TIL] Tilera tile-gx100. `http://www.tilera.com/products/TILE-Gx.php`.

[TKA02] William Thies, Michal Karczmarek, and Saman Amarasinghe. StreamIt: A Language for Streaming Applications. In *Proc. Int'l Conf. on Compiler Construction*, pages 179–196, Grenoble, France, April 2002.

[Tra04] Mario Trams. Conservative distributed discrete event simulation with SystemC using explicit lookahead. `http://digital-force.net/publications`, 2004.

[TWS06] Lothar Thiele, Ernesto Wandeler, and Nikolay Stoimenov. Real-time interfaces for composing real-time systems. In *International Conference On Embedded Software (EMSOFT)*, pages 34–43, 2006.

[VCC] The Cadence Virtual Component Co-design (VCC). `http://www.cadence.com/products/vcc.html`.

[vdWdKH+04] Pieter van der Wolf, Erwin de Kock, Tomas Henriksson, Wido Kruijtzer, and Gerben Essink. Design and Programming of Embedded Multiprocessors: An Interface-Centric Approach. In *Proceedings of the 2nd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis (CODES+ISSS)*, pages 206–217, Stockholm, Sweden, 2004.

[vSP10] Peter van Stralen and Andy D. Pimentel. A high-level microprocessor power modeling technique based on event signatures. *Journal of Signal Processing Systems*, 60(2):239–250, 2010.

[WHO06] T. Wild, A. Herkersdorf, and R. Ohlendorf. Performance Evaluation for System-on-Chip Architectures Using Trace-Based transaction fLevel Simulation. In *Proc. Design, Automation and Test in Europe (DATE)*, pages 248–253, 2006.

[WJM08] W. Wolf, A.A. Jerraya, and G. Martin. Multiprocessor system-on-chip (mpsoc) technology. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(10):1701–1713, October 2008.

[wPOH09] Hae woo Park, Hyunok Oh, and Soonhoi Ha. Multiprocessor soc design methods and tools. *IEEE Signal Processing Magazine*, 26(6):72–79, November 2009.

[WT05] E. Wandeler and L. Thiele. Characterizing Workload Correlations in Multi Processor Hard Real-Tim e Systems. In *11th IEEE Real-Time and Embedded Technology and Applications Sympo sium (RTAS)*, pages 46–55, March 2005.

[WT06a] Ernesto Wandeler and Lothar Thiele. Interface-Based Design of Real-Time Systems with Hierarchical Scheduling. In *Proc. Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 243–252, San Jose, USA, 2006.

[WT06b] Ernesto Wandeler and Lothar Thiele. Optimal TDMA Time Slot and Cycle Length Allocation for Hard Real-Time Systems. In *Proc. Asia and South Pacific*

Conf. on Design Automation (ASP-DAC), pages 479–484, Yokohama, Japan, January 2006.

[WT06c] Ernesto Wandeler and Lothar Thiele. Real-Time Calculus (RTC) Toolbox. `http://www.mpa.ethz.ch/Rtctoolbox`, 2006.

[WTVL06] Ernesto Wandeler, Lothar Thiele, Marcel Verhoef, and Paul Lieverse. System Architecture Evaluation Using Modular Performance Analysis: A Case Study. *Int'l Journal on Software Tools for Technology Transfer (STTT)*, 8(6):649–667, November 2006.

[YCHK07] Chuan-Yue Yang, Jian-Jia Chen, Chia-Mei Hung, and Tei-Wei Kuo. System-level energy-efficiency for real-time tasks. In *the 10th IEEE International Symposium on Object/component/service-oriented Real-time distributed Computing (ISORC)*, pages 266–273, 2007.

[yel] Yellow Dog Linux. `http://www.yellowdoglinux.com`.

[YVKI00] W. Ye, N. Vijaykrishnan, M. Kandemir, and M.J. Irwin. The design and use of simplepower: a cycle-accurate energy estimation tool. In *Proceedings of the 37th Annual Design Automation Conference (DAC)*, pages 340–345, 2000.

[ZC05] Jianli Zhuo and Chaitali Chakrabarti. System-level energy-efficient dynamic task scheduling. In *Proceedings of the 42nd ACM/IEEE Design Automation Conference(DAC)*, pages 628–631, 2005.

List of Publications

The following list summarizes the publications on which this thesis are based. The pertinent chapters of this thesis are given in brackets.

K. Huang, L. Santinelli, J. J. Chen, L. Thiele, and G. C. Buttazzo. Adaptive Dynamic Power Management for Hard Real-Time Streams. In *Real-Time Systems Journal* (accepted)
(Chapter 4)

K. Huang, L. Santinelli, J. J. Chen, L. Thiele, and G. C. Buttazzio. Adaptive power management for real-time event streams. In *the 15th IEEE Conf. on Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 7–12, 2010.
(Chapter 4)

K. Huang, L. Santinelli, J. J. Chen, L. Thiele, and G. C. Buttazzio. Adaptive dynamic power management for hard real-time systems. In *the 30th IEEE Real-Time Systems Symposium (RTSS)*, pages 23–32, Washington D.C. U.S., 2009.
(Chapter 4)

K. Huang, L. Santinelli, J.-J. Chen, L. Thiele, and G. C. Buttazzio. Periodic power management schemes for real-time event streams. In *the 48th IEEE Conf. on Decision and Control (CDC)*, pages 6224–6231, China, 2009.
(Chapter 4)

K. Huang, I. Bacivarov, J. Liu, and W. Haid. A modular fast simulation framework for stream-oriented mpsoc. In *IEEE Symposium on Industrial Embedded Systems (SIES)*, pages 74–81, Ecole Polytechnique Fédérale de Lausanne, Switzerland, 2009.
(Chapter 3)

K. Huang, L. Thiele, T. Stefanov, and E. Deprettere. Performance analysis of multimedia applications using correlated streams. In *Design, Automation and Test in Europe (DATE)*, pages 912–917, Nice, France, 2007.
(Chapter 3)

K. Huang, I. Bacivarov, F. Hugelshofer, and L. Thiele. Scalably distributed systemc simulation for embedded applications. In *IEEE Symposium on Industrial Embedded Systems (SIES)*, pages 271–274, 2008.
(Chapter 2)

K. Huang, D. Gruenert, and L. Thiele. Windowed fifos for fpga-based multiprocessor systems. In *IEEE 18th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 36–42, Montreal, Canada, 2007.
(Chapter 2)

L. Thiele, I. Bacivarov, W. Haid, and K. Huang. Mapping Applications to Tiled Multiprocessor Embedded Systems. In *Proceedings of International Conference on Application ofConcurrency to System Design (ACSD)*, pages 29–40, Bratislava, Slovak Republic, July 2007.
(Chapter 2)

It follows a list of publications that are not fully covered in this thesis.

K. Huang, W. Haid, I. Bacivarov, M. Keller and L. Thiele. Embedding performance analysis into the design cycle of mpsocs for real-time multimedia. *ACM Transactions on Embedded Computing Systems*. (to appear)

J. J. Chen and K. Huang and L. Thiele Power Management Schemes for Heterogeneous Clusters under Quality of Service Requirements Streams *26th ACM Symposium On Applied Computing (SAC) 2011*

I. Bacivarov, W. Haid, K. Huang, and L. Thiele. Methods and Tools for Mapping Process Networks Onto Multi-Processor Systems. In Bhattacharyya, Shuvra S. and Deprettere, Ed F. and Leupers, Rainer and Takala, Jarmo, editors *Handbook of Signal Processing Systems*. Springer, pages 1007—1040, Oct. 2010.

W. Haid, K. Huang, I. Bacivarov, and L. Thiele. Multiprocessor soc software design flows. *IEEE Signal Processing Magazine*, 26(6):64–71, 2009.

W. Haid, L. Schor, Kai Huang, I. Bacivarov, and L. Thiele. Efficient execution of kahn process networks on multi-processor systems using protothreads and windowed fifos. In *Proc. IEEE Workshop on Embedded Systems for Real-Time Multimedia (ESTIMedia)*, pages 35–44, Grenoble, France, 2009.

K. Huang, W. Haid, I. Bacivarov, and L. Thiele. Coupling MPARM with DOL. Technical Report 314, D-ITET, TIK, ETH Zürich, Nov. 2009.

W. Haid, M. Keller, K. Huang, I. Bacivarov, and L. Thiele. Generation and calibration of compositional performance analysis models for multi-processor systems. In *Proc. Intl Conference on Systems, Architectures, Modeling and Simulation (SAMOS)*, pages 92–99, Samos, Greece, 2009.

J. Liu, Z. h. Li, I. Bacivarov, and K. Huang. Scheduling mechanisms in high-level simulation framework. *Microelectronics & Computer*, 25(8):64–68, 2008.

T. Sporer, A. Franck, I. Bacivarov, M. Beckinger, W. Haid, K. Huang, L. Thiele, P. Paolucci, P. Bazzana, P. Vicini, J. J. Ceng, S. Kraemer, and R. Leupers. Shapes - a scalable parallel hw/sw architecture applied to wave field synthesis. In *Proc. 32nd Intl Audio Engineering Society (AES) Conference*, pages 175–187, Hillerod, Denmark, 2007. Audio Engineering Society.

K. Huang, J. Gu, T. Stefanov, and E. Deprettere. Automatic Platform Synthesis and Application Mapping for Multiprocessor Systems On-Chip. Technical Report 05-30, LIACS, Leiden University, Aug. 2005.

Curriculum Vitae

Name	Kai Huang
Date of Birth	2 October 1977
Nationality	China
Website	www.tik.ee.ethz.ch/~khuang

Education:

10/2005–11/2010	ETH Zurich, Computer Engineering and Networks Laboratory, Switzerland Doctor Thesis under the Supervision of Prof. Dr. Lothar Thiele
01/2004–08/2005	Leiden University, Leiden Institute of Advanced Computer Science, the Netherlands Master of Computer Science
09/1995–06/1999	Fudan University, Dept. Computer Science, China Bachelor of Computer Science

Professional Experience:

10/2005–now	Research & Teaching Assistant at ETH Zurich
10/2001–12/2003	Senior Software Engineer at Redsonic Inc, Shanghai, China
07/2001–10/2001	Program Analyzer at National Computer Systems Pte Ltd (NCS), Shanghai, China
07/1999–05/2001	Software Engineer at Fudan Network Ltd., Shanghai, China

Honors:

2010	Student Travel Grant for the 15th IEEE Conf. on Asia and South Pacific Design Automation Conference (ASP-DAC)
2009	General Chairs' Recognition Award for Interactive Papers for the 48th IEEE Conf. on Decision and Control (CDC)
2009	Co-recipient for the Best Paper Award Nomination for the 7th IEEE Workshop on Embedded Systems for Real-Time Multimedia (ESTIMedia)
2009	Co-recipient for the Best Paper Award for the Int'l Symposium on Systems, Architectures, Modeling and Simulation (SAMOS)