ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

TIK Institut für Technische Informatik und Kommunikationsnetze

Master Thesis
at the Department of Information Technology
and Electrical Engineering

Generation of Accurate Performance Analysis Models for Embedded Systems

Matthias Keller

Advisors: Wolfgang Haid, Kai Huang
Professor: Prof. Dr. Lothar Thiele

Zürich, April 2008

Abstract

Formal-analysis-based methods are a powerful method for performance analysis of heterogeneous distributed systems. Due to their capability to analyze best- and worst-case behavior, analysis-based methods are particularly useful for the analysis of embedded real-time systems. Unfortunately, the effort to create an analysis model for a specific system is considerable.

In this thesis, an approach is presented that seamlessly integrates the generation of an analysis model into a design flow for embedded real-time systems. In particular, the approach is targeted at a multi-processor system-on-chip (MPSoC) design flow and uses modular performance analysis for system performance analysis. The model generation is based on the specification that is used for system synthesis and all the required application and architecture model parameters are automatically obtained.

While the fully automatic generation of an analysis model for a specific system implementation itself is useful, the proposed approach allows to generate analysis models for alternative system implementations as well. This enables the efficient design space exploration of MPSoC systems based on modular performance analysis.

Zusammenfassung

Formale Analysemethoden sind ein mächtiges Werkzeug für die Performanzanalyse von heterogenen verteilten Systemen. Da das Verhalten eines Systems sowohl für den günstigsten als auch für den ungünstigsten Fall analysiert werden kann, eignen sich formale Analysemethoden besonders für die Untersuchung von eingebetteten Realzeitsystemen. Der Aufwand für die Erstellung eines geeigneten Modells ist allerdings erheblich.

Diese Arbeit beschreibt einen Ansatz, der die Erstellung von Modellen zur Systemananalyse nahtlos in einen Entwurfsablauf für eingebettete Realzeitsysteme einbindet. Der Fokus liegt hierbei auf dem Entwurf von Multiprozessor Systemen. Für die Analyse von Charakteristiken auf Systemebene wird Modular Performance Analysis verwendet. Dabei basiert die Modellerstellung auf der selben Beschreibung, die auch für die Synthese verwendet wird. Alle relevanten Parameter im Bezug auf die Applikation und die eingesetzte Architektur werden automatisch ermittelt.

Hierbei ist die vollautomatische Modellerstellung nicht nur auf eine konkrete Implementierung beschränkt, Modelle für alternative Konfigurationen können ebenso erstellt werden. Dies ermöglicht eine effiziente Entwurfsraumexploration für Multiprozessor Systeme unter der Verwendung von Modular Performance Analysis.

Acknowledgements

I would like to thank

- Prof. Dr. Lothar Thiele for giving me the opportunity to write this thesis in his research group at the Computer Engineering and Network Lab of the Swiss Federal Institute of Technology (ETH) Zürich.
- Wolfgang Haid and Kai Huang for their constant support during the whole project that made this work possible.
- Prof. Dr. Arndt Bode and Dr. Carsten Trinitis for supporting me in writing this thesis abroad.
- The Center for Digital Technology and Management (CDTM) and especially the Lothar-und-Sigrid-Rohde Foundation for financially supporting my time abroad.
- Prof. Dr.-Ing. Klaus Diepold for supporting my wish to write my thesis in the truly challenging environment at the Swiss Federal Institute of Technology (ETH).
- My parents for supporting my studies in Zürich and also my studies in general.

Contents

1

Introduction

An embedded system is a special-purpose information processing system that is closely integrated into its environment. Embedded systems are ubiquitous nowadays, their existence is not perceived in most of the cases. Furthermore, embedded systems are often required to meet real-time constraints.

One popular example for the deployment of embedded systems is the control of technical or chemical processes in arbitrary environments that range from home appliances and cars to nuclear power plants.

Many recent applications can be found in consumer electronics devices like mobile media players or broadband routers. Here, lightweight operating systems with multitasking capabilities are necessary to allow the concurrent processing of multiple tasks and sporadic user inputs.

The dedication to a certain task of a specific application domain enables tremendous savings in costs, power consumption and size compared to general-purpose computer systems. The designer can select both hardware and software components to optimize a system for its specific task.

Modern electronic technology allows to integrate different functionalities on a single chip. An example of a system-on-chip (SoC) design locates a processor core, memories and peripheral devices like communication controllers on the same chip. The development time of a SoC design can be massively shortened by using components from existing intellectual property (IP) libraries. In general, SoC design facilitates further reductions in costs, power consumption and system size.

Recent multi-processor system-on-chip (MPSoC) designs employ multiple heterogeneous computing resources on a single chip. Complex

on-chip networks are used to integrate different components such as CPUs, field programmable gate arrays (FPGAs), digital signal processors (DSPs), memories and other peripheral devices.

MPSoC architectures are the key technology for building distributed parallel embedded systems that are able to cope with the demand of future applications in the area of multimedia and digital signal processing in general.

Design Space Exploration

On the design level, MPSoC systems can be characterized by their large design space. For instance, the existence of multiple, heterogeneous processor cores leads to several choices in the binding of application tasks to processing elements. Furthermore, there is also a large degree of freedom in the temporal domain where the designer has to be concerned about selecting and configuring schemes for resource sharing.

Suitable configurations can be found by analyzing different design points with respect to the design requirements. These requirements can be expressed by certain characteristics of a system design, such as end-to-end delays, throughput, or buffer requirements. The estimation of this parameters is generally referred to as system-level performance analysis.

Efficient methods for system-level performance analysis are crucial for the exploration of the large design space. This applies especially to the early phases of the exploration where a large number of implementation choices is investigated, often in an automated or semi-automated way.

Unfortunately, the analysis of heterogeneous distributed systems is difficult. For instance, small local changes in a mapping can massively influence the overall system performance of a specific implementation: Firstly, the execution time of a task with special requirements changes, if the task is mapped to a processing element with fitting capabilities, such as floating-point units, vector units or programmable logic arrays. Secondly, the impact of on-chip and off-chip connection delays on the execution time is affected by the spatial location of communicating tasks. In a more general way, Wandeler introduces six categories of challenges that appear in the system-level performance analysis of complex distributed embedded systems [46, p. 3-4]. In the context of this thesis, especially the categories resource sharing, interferences between different applications and workload correlations are of interest.

There exist two main classes of methods for system-level performance analysis, namely simulation-based and formal-analysis-based methods [46, p. 4]. Depending on the used layer of abstraction, there is a trade-off between evaluation time and result accuracy. More precisely, simulation techniques on a low level of abstraction, such as cycle-accurate simulation,

offer a high accuracy for the price of a long evaluation time. Higher level practices include instruction-accurate simulation, trace-based simulation and lastly formal analysis methods. The latter deliver rather coarse-grained results because of the high layer of abstraction they are applied to.

Different methods are used in certain stages of state-of-the-art design space exploration. The design space is iteratively narrowed by subsequent explorations at decreasing levels of abstractions. An example state-of-the-art design space exploration setup might use a hierarchy of formal analysis methods for fast performance estimation at the beginning, trace-based simulation with different abstraction levels for the analysis of selected candidates, and finally low-level simulation techniques for further selection and system verification.

The main drawback of simulation-based methods can be found in the long runtime and insufficient corner-case coverage [46]. It becomes apparent, that the result of a simulation is always depending on the data that was used to drive the simulation. The expressiveness of the input data is difficult to measure, usually it is more likely that only the average-case behaviour is covered [19]. Furthermore, manual corner-case identification and input pattern selection is not practical for complex embedded systems. The latter criticism on simulation-based methods is for instance highlighted in [33].

In contrast, formal-analysis-based methods can be used to analyze the upper and lower bounds of certain characteristics. These bounds cover both best-case and worst-case behaviour as they include observations of varying time intervals.

Overall, three desirable characteristics of system-level performance analysis methods in the context of design space exploration can be named:

- Rapidness: A short evaluation time is crucial for a viable exploration of the large design space. Preferably, a fast method should be available during the whole design space exploration.

- Accuracy: Wide ranges of the whole design space exploration can only be covered with the same method if the provided accuracy is competitive to methods that work on a low level of abstraction.

- Completeness: Observations of the average-case behaviour are not sufficient for raising critical design decisions.

1.1 Aim of this Thesis

It has been shown that formal-analysis-based methods are well-suited for design space exploration [43, 47, 14].

The problem of existing approaches is that the analysis models need to be generated by hand from either abstract system specifications or also manually performed benchmark measurements. Additionally, most of them are only considerable during the early phases of a design space exploration.

This thesis aims to show an approach for automatically generating performance analysis models that can be used in all phases of a design space exploration. Consequently, both the analysis model generation as well as the analysis with a formal method can be fully integrated into a design tool flow.

The following sections give an introductory overview of the research environment in which this thesis is embedded. Some more detailed aspects of the following topics will be mentioned in the later chapters that deal with the implementation phase.

1.2 The SHAPES Project

Scalable Software Hardware Architecture Platform for Embedded Systems (SHAPES) [26] is a joint project of several European top universities and industry leaders. The list of partners includes, but is not limited to, the Swiss Federal Institute of Technology, RWTH Aachen University, TIMA Grenoble Laboratory and ATMEL Roma. The SHAPES project started in January 2006 and will end mid of 2009.

The goal of the SHAPES project is to develop a hardware platform and a software design flow that copes with the challenges in the design of scalable embedded systems.

After introducing the hardware architecture (see 1.2.1), the programming model (see 1.2.2) as well as a functional simulation (see 1.2.3) are described. Next, an instruction-accurate simulator that supports the software development process is shown (see 1.2.4). The distributed operation layer (DOL) that is described afterwards deals with the optimization of application to architecture mappings (see 1.2.5). After mentioning the software stack generation (see 1.2.6), the introduction of the SHAPES project ends with the presentation of an example application (see 1.2.7).

1.2.1 Tiled Architecture

The main challenge in building future computer architectures that utilize nanoscale CMOS technologies with billions of gates is the wiring [26] as the delay of interconnects is rapidly increasing on smaller CMOS technologies [15].

A tiled architecture is proposed as the solution for this problem. The idea are "small" processing tiles that are connected by "short" wires. A distributed packet switching network is used for connecting on-chip and off-chip tiles. The system density is maximized by adopting 3D next-neighbours engineering methodologies for off-chip networking [27].

The current SHAPES tile is the evolution of Atmel Diopsis [30], a MPSoC which includes an ARM9 RISC processor and a floating-point VLIW mAgic DSP [28]. Additionally, a tile includes a distributed network processor, on-tile memories and a set of on-tile peripherals. Each tile can be equipped with a distributed external memory.

1.2.2 Programming Model and Application Specification

The programming model defines how applications can be programmed in a platform-independent manner. It is based on the process network model of communication. In short, an application exists of several processes that exchange messages over communication channels. Each communication channel is equipped with a FIFO buffer for storing unprocessed messages.

On an abstract level, the application specification defines the process network layout by defining processes and their communication paths that are implemented as software channels. The application specification allows the use of any network semantics, e. g. Kahn process network [16], by imposing additional restrictions or semantics onto the elements [42].

1.2.3 Functional Simulation

A functional simulation can be derived automatically from the application specification. The underlying event-based simulation control is implemented on the SystemC kernel [39].

The resulting execution trace basically contains the number of activations per task and the amount of transferred data per software channel. This information can be used for debugging and testing as well as for obtaining mapping relevant parameters at the application level.

1.2.4 Virtual SHAPES Platform

The Virtual SHAPES Platform (VSP) is the simulation environment for the SHAPES architecture. The VSP is distributed as a CoWare Virtual

Platform [7], a proprietary model specification format. The VSP model can be analyzed with the CoWare Virtual Platform Analyzer (VPA) application.

For a good tradeoff between performance and accuracy, instruction-accurate simulation is used for the SHAPES platform. This means that the concrete instruction set architecture (ISA) is simulated. Thus, the hardware dependent software like drivers and operating systems can be developed concurrently with the hardware and tested before the actual hardware prototype is available [17].

Simulation is not only helpful for shortening the development cycles, it also plays a decisive role in the design space exploration. Trace and profiling information are necessary in the mapping procedure to evaluate the quality of different mappings. These information can be collected much easier by a simulation model than from the hardware [27].

Furthermore, the VSP can also be used to support the high-level architecture exploration. The goal of this process is the system level evaluation of different combinations of SHAPES tiles [17].

Virtual Platform Analyzer

Virtual Platform Analyzer (VPA) is a tool for running simulations on Virtual Platforms.

A simulation can be interrupted by breakpoints on arbitrary single addresses or address ranges. VPA offers access to all information that are needed to collect trace and profiling data. These information include the program counter for each core, the local clock of each processing core, the platform clock, the values of all hardware registers and the contents of all memory regions.

VPA offers a graphical user interface (see Fig. 1) as well as a scripting interface. All features of the VPA plus all features of the underlying Tcl programming language [41] can be exploited by the scripting interface.

Section 5.4.1 deals with the implementation of a script for fully automatic gathering of profiling data.

1.2.5 Distributed Operation Layer

Complex MPSoC systems offer a large degree of freedom in the binding of application processes and communication paths to processing elements and communication resources, respectively. The additional choice between different resource allocation and arbitration schemes also leads to a large design space. Established design methodologies and tools for the design of single processor systems are not suited for handling such complex systems. Thus a new design methodology is necessary to keep

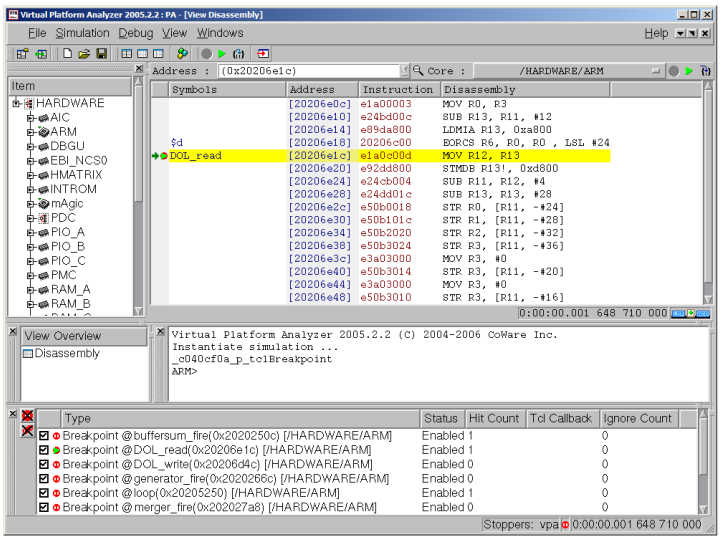


Fig. 1: Virtual Platform Analyzer

the time and effort that has to be spent for the development of scalable and efficient MPSoC systems at a reasonable amount [42].

The Distributed Operation Layer (DOL) framework [8] is proposed to face that challenge. The idea is to find an optimal mapping of an already parallelized application onto a MPSoC architecture in an automated fashion. System-level performance analysis and multi-objective application-architecture mapping optimization based on evolutionary algorithms [48] are the two main services that are offered for this task [42].

1.2.6 Software Stack Generation

A software stack contains the different layers that form the final software binary. The existence of architecture-specific layers requires an adapted software stack for each hardware architecture on which it should be executed.

Application tasks, the top layer of a software stack, are based on a specific API that is provided by the Hardware dependent Software (HdS). Basically, the HdS implements typical functions of an operating system, e. g. drivers and a scheduling service. It is in turn based on the Hardware Abstraction Layer (HAL), the bottom layer, which is unique for each hardware platform.

A software stack is generated out of an application that is specified with the programming and specification model of DOL (see 1.2.2). The processing steps include the generation of an application initialization routine, the generation of build scripts and finally the compilation and

linking of the executable binary [12].

1.2.7 Application Example: Wave Field Synthesis

Wave Field Synthesis (WFS) is a digital audio application that deals with the reproduction of acoustical scenes. The basic concept is to create an accurate representation of the recorded, original wave field with its natural temporal and spatial properties by the superposition of the signals of a large number of loudspeakers. This accurate reproduction can take place in a listening area that has different properties than the recording room [38].

Due to its high computational complexity, WFS has not found broad application until today. Decreasing costs of computation power enabled the first application in the professional market, but the execution speed is still limited by the available computation power. A shift from former homogeneous digital signal processor systems to highly parallel heterogeneous MPSoC architectures like SHAPES is proposed to satisfy this demand [38].

1.3 Modular Performance Analysis with Real-Time Calculus

Modular performance analysis with real-time calculus (MPA-RTC) [47, 25] is a formal method for system-level performance evaluation of distributed heterogeneous embedded systems. The theory behind MPA-RTC and the development of a Matlab toolbox for MPA-RTC is a current research topic at the Computer Engineering and Networks Laboratory (TIK) [45] of the Swiss Federal Institute of Technology (ETH).

1.3.1 Real-Time Calculus

Real-time calculus [6] is based on network calculus [21] which is in turn based on max-plus algebra [2]. Using RTC, it is possible to determine worst-case bounds on system properties, such as buffer memory requirements or end-to-end delays. In particular, MPA-RTC is suited for communication-centric designs like a communication network of connected processing elements.

1.3.2 Arrival and Service Curves

The processing capabilities of a resource are described by service curves, while the workload in terms of arriving events of a computation or

communication task is given by arrival curves, respectively (see Fig. 2). The upper and lower arrival curve of an event stream give the upper and lower bound on the number of events that arrive in any time interval. Similarly, the upper and lower bound on the number of events that can be processed by a resource in any time interval are given by the upper and lower service curve.

MPA-RTC offers a set of event and resource models that are modeled as special shaped arrival and service curves, respectively.

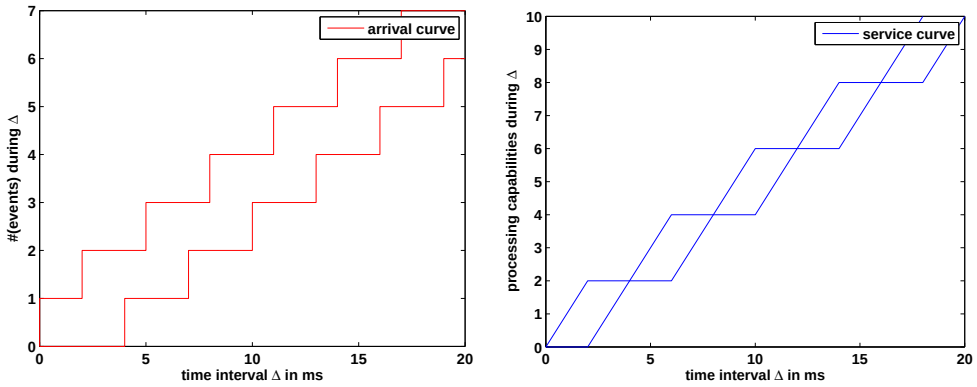


Fig. 2: Arrival curve of a periodic event source with jitter (left) and service curve of a TDMA resource (right)

1.3.3 Abstract Modeling of Tasks

A computation or communication task with a specific execution demand and a single incoming event stream complies with the abstract greedy processing component (GPC) in MPA-RTC. This component stands for the mathematical transformation of incoming arrival and service curves into outgoing arrival curves and remaining service curves, respectively.

The worst-case and best-case execution demands can be given in an arbitrary corresponding real unit like the needed computation time per task activation or the amount of transferred data per task activation. The corresponding real unit of the execution demands and the real unit of the processing capabilities of the used resource have to match, of course.

For the case of multiple, independent event streams and a single resource, more complex components with an underlying scheduling strategy for resource sharing are available.

1.3.4 Features of MPA-RTC

Currently, MPA-RTC supports the following scheduling policies: preemptive and non-preemptive fixed-priority scheduling, time division

multiple access scheduling (TDMA), first-in first-out scheduling (FIFO) and earliest deadline first scheduling (EDF).

Furthermore systems with complex task activation schemes can be modeled by combining multiple event streams with binary logic functions [13].

1.4 Related Work

Over the past decades, many models for formal scheduling analysis have been developed. Popular examples are rate-monotonic scheduling, earliest deadline first (EDF) and time-division multiple access (TDMA). Instead of dealing with single events, these techniques are usually designed for the analysis of input event streams with certain characteristics. In particular, most of the existing models are limited to the analysis of event streams that correspond to periodic or sporadic event sources with jitter.

This limitation makes existing methods inapplicable for the analysis of complex systems. The reason can be found in the fact, that the characteristics of an input event stream, such as periodicity, change tremendously while the stream is propagated through the network of different components.

To overcome this problem, MPA-RTC introduces arrival curves as a new event stream presentation plus new scheduling analysis techniques for local components.

Gresser [10] follows a similar idea by introducing an event stream model based on event vectors and event dependence matrices for deadline verification. The main step consists of finding the so-called C(I)-function which states the maximum requested computation time for any time interval. Setting up this function out of the given event streams and dependencies is NP-hard, thus this method is not suitable for analyzing large systems without further improvements. Additionally, the model has only been proven for the analysis of systems that use EDF scheduling, so far.

Event model interfaces and event adaption functions are used in SymTA/S [33] to resolve input-output event stream incompatibilities. More precisely, event model interfaces and adaption functions are abstracting complex event streams by simple event streams. Latter can be described either by periodic or sporadic event stream models with jitter. From there falling back to already existing models for scheduling analysis is possible.

Künzli and Wandeler use MPA-RTC for the design space exploration of a network processor [43] and an in-car navigation system [47],

respectively. In both cases, information from system specifications is used for estimating system parameters, only the early phases of a design space exploration are covered. The analysis models are generated manually.

Henia et al. [14] propose a whole design space exploration loop based on SymTA/S by interfacing a multi-objective optimization based on evolutionary algorithms. The process for extracting the system parameters that are used in their analysis model is not explicitely covered in their work.

Based on the Polis [3] framework and Ptolemy [5], Lahiri et al. [20] are also using formal analysis methods for the whole design space exploration. Trace-based simulation is carried out once to extract so-called communication analysis graphs (CAG). This approach is limited to communication scheduling and does not include strategies for the sharing of processing resources.

SPADE [23] and its successor Sesame [32] are two projects that use trace-based simulation during the whole design trajectory. Both projects are following the Y-chart paradigm which means that application and architecture description are independent from each other and only merged in a concrete mapping.

In Sesame, the trace generation includes a manual instrumentation of the application sources. The used language offers three operations, namely read, write and execute. Depending on the phase of the design space exploration and the corresponding desired layer of abstraction, these operations are implemented in a varying level of details. The process of refining the architecture model is referred to as trace transformation. Apart from the fact, that an instrumentation can only be carried out with knowledge of the application domain, the work of Thompson et al. [44] reveals a high manual effort that is necessary to set up a simulation.

Pimentel et al. [31] perform a one-time calibration of an architecture model that is used in a trace-based simulation. Performance data is gathered from instruction-accurate simulation with SimpleScalar ISS [36]. High manual effort is necessary for performing their calibration approach.

Concluding, a comparison of several projects and frameworks for design space exploration of embedded systems can be found in [11].

1.5 Thesis Outline and Contributions

In this thesis, an approach for fully automatically generating accurate performance analysis models based on a one-time calibration is presented. The proposed design space exploration loop that is based on this approach is outlined in Fig. 3.

Chapter 2 starts with the introduction of the so-called abstract

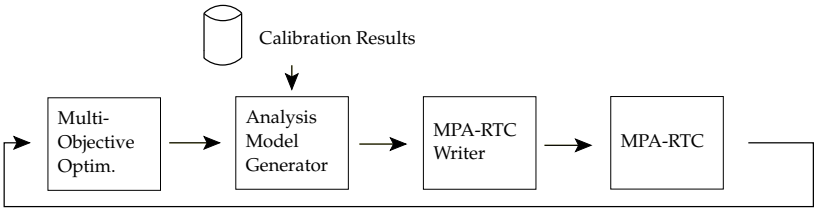


Fig. 3: Design space exploration loop that uses performance data from a one-time calibration for generating accurate performance analysis models. Firstly, an instance of the abstract model is generated for each design point that was selected by the optimization controller. Then, MPA-RTC is employed for formal-based system-level analysis. The results of the analysis are returned to the optimization controller.

model, a new abstract specification language for modeling complex embedded systems. The also presented MPA-RTC writer can be used for automatically generating a Matlab script that employs the MPA-RTC toolbox from an instance of the abstract model. In the envisioned design space exploration tool flow, the abstract model is used as the interface between a project-specific specification model and a specific method for formal system-level analysis, here MPA-RTC.

The general concept of employing instruction-accurate simulation during a one-time calibration is presented in Chapter 3.

Chapter 5 describes a reference implementation of the proposed tool flow for an automatic generation of performance analysis models. Here, SHAPES is used as the example environment.

Finally, the results that can be obtained with the current approach and reference implementation are shown in Chapter 6.

For a better understanding of the latter two chapters, a short introduction into internals of the tool support of the SHAPES project is given in Chapter 4.

2

Modeling Systems for Performance Analysis

Complex embedded systems consist of several shared computation and communication resources. Varying schemes for resource sharing might be applied. These schemes are no longer limited to standard concepts, such as fixed-priority or first-in first-out (FIFO) scheduling, it is also possible to schedule a resource by a hierarchy of several concepts. In addition, tasks with multiple inputs and outputs may have to be activated by complex patterns.

Modeling such systems for carrying out formal analysis is elaborate. Usually, a high manual effort is necessary to transform the abstract system model into a model, or rather a script, that matches the specification of the analysis tool.

Thus, an automated way for the generation of analysis models is desirable, especially for a growing number of components. Furthermore, interchangeability between different system description formats on the input and varying methods for performance analysis on the output can only be established by an intermediate description model. Independence between system and analysis model is also essential concerning the software development effort that has to be spent for automatically analyzing and generating both parts, respectively.

2.1 Overview

The proposed abstract model is designed to be as unrestrictive as possible on a high level of abstraction. Limitations should only appear in connection with a particular technique for formal analysis.

An instance of the abstract model can be created by using a reader. By now, there exist two readers, namely the XML reader and the SHAPES reader. The latter converts architecture, application and mapping description of a SHAPES application into an instance of the abstract model.

In turn, a writer is necessary for converting an instance of the abstract model into the input format of an analysis framework. While the XML writer is mainly used for debugging purposes, Matlab scripts that employ the MPA-RTC toolbox can be created with the MPA-RTC writer.

Fig. 4 highlights the proposed idea of multiple reader and writer implementations that can be interchanged.

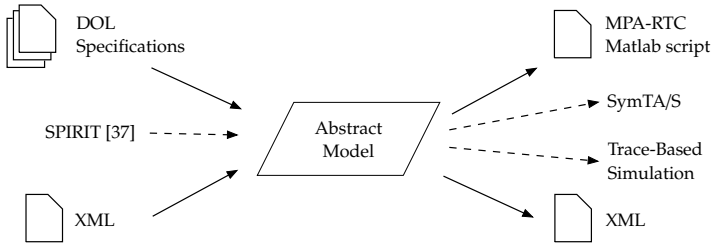


Fig. 4: Available and imaginable readers and writers for the abstract model

The following sections describe the introduced model as well as the implementation of the MPA-RTC writer. Insights into the SHAPES reader are given in 5.7.

2.2 Abstract Model

Basically, an instance of the abstract model includes typical parts of a process network, such as connected event sources and processes, as well as a description of the available resources with the related schemes for sharing. Optional observation targets, i. e. end-to-end delays, complete the description. Fig. 5 gives a more detailed overview of all mandatory components on the top layer.

2.2.1 Process Network

A minimal process network in the context of this work has to include an event source, a process and an event sink. One connection element is

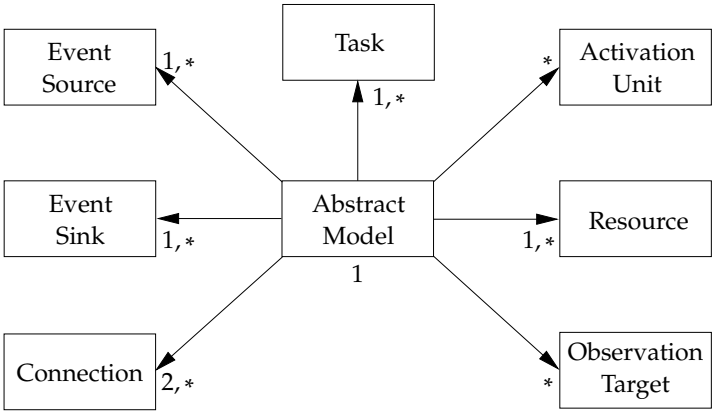


Fig. 5: Entity relationship diagram of the abstract model. Numbers adjacent to blocks indicate the frequency in which each entity must occur in a valid instance of the abstract model. "1,*" denotes that the specific entity can occur in an arbitrary frequency starting from at least one occurence.

necessary for linking each pair of two components, thus two connections are needed to put this small network together.

Both process and event sink components are only accepting a single input. However, activation units can be used to extend a single input port to a gate with an arbitrary number of inputs. The output port of the activation unit can be activated by using either a disjunction or a conjunction of the input ports.

On the outgoing side, an event stream can be partitioned by defining multiple, weighted connections that start at the same output port.

Event Sources

In many cases, it is sufficient to use a periodic event source with jitter. This component takes the period, the jitter and the minimum distance between two subsequent events as input parameters.

Regarding the preferred usage with MPA-RTC, it is also possible to define an event source by arbitrary upper and lower curves. Here, a curve is specified by passing a list of segments for its periodic and its aperiodic part.

2.2.2 Resource Model

A resource offers a service that can be characterized by the bandwidth that corresponds to its processing capabilities. There exist three options for the availability of a service, namely full, periodic and delayed availability.

Resource Sharing

Several scheduling schemes are available for resource sharing. Currently fixed-priority scheduling, preemptive EDF scheduling, preemptive FIFO scheduling and TDMA scheduling are supported.

All scheduling schemes are splitting the service of a resource into a number of slices. Depending on the used scheme, each slice needs to be parameterized, e. g. with the length of the corresponding TDMA slot. The capabilities of a slice can either be consumed by a process or be split up again by another scheduling scheme. Without further limitation, a hierarchy of any depth is supported by the abstract model. An example hierarchy of several resource sharing schemes is shown in Fig. 6.

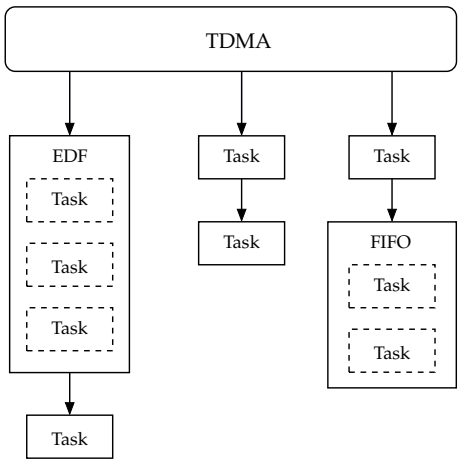


Fig. 6: Hierarchy of several resource sharing schemes. The processing capabilities of each TDMA slot are again shared by other scheduling schemes.

2.2.3 Observation Targets

Observation targets can be useful, if the value of a certain characteristic should be available at a specific location in the generated analysis model, i. e. for automated post-processing of the results. By now the abstract model allows to watch the backlog of a task's input buffer as well as the end-to-end delay between two tasks.

2.3 MPA-RTC Writer

Systems for analysis with MPA-RTC are modeled as Matlab scripts. The source code example in Lst. 2.1 describes a system with two tasks that

share a full service. Preemptive fixed-priority scheduling is used for resource sharing.

Lst. 2.1: Two tasks example in MPA-RTC

```
% Fully available resource, bandwidth = 1
b1 = rtcfs( 1 );
% Periodic event source with jitter
% period = 3, jitter = 1, min. distance = 2
a1 = rtcpjd( 3, 1, 2 );
% First task, highest priority, WCED = 5, BCED = 3
[ a2 b2 ] = rtcgpc( a1, b1, [ 5 3 ] );
% Second task, WCED = 4, BCED = 2
[ a3 b3 ] = rtcgpc( a2, b2, [ 4 2 ] );
% The resulting arrival curves are now available in
% a2 and a3. In turn, the resulting service curves
% are available in b2 and b3.
```

2.3.1 Limitations

The set of systems that can be analyzed with MPA-RTC is smaller than the set of systems that can be described by the abstract model. Thus, analysis with MPA-RTC can only be carried out for systems that meet a set of limitations that restrict the resource model.

TDMA Scheduling

In MPA-RTC, TDMA scheduling is implemented as a resource model. This means that TDMA scheduling can only be used to share a fully available resource, but not for sharing remaining service or service that is provided by another scheme.

EDF and FIFO Scheduling

Both, EDF and FIFO schedulers components in MPA-RTC allow only to share the available service between a set of processes. It is not possible to embed other components that implement scheduling techniques.

Covered Resource Sharing Hierarchies

In conclusion, TDMA scheduling can be applied on top for splitting a fully available service into several slices.

The available service on the next layer can either be shared by EDF, FIFO or fixed-priority scheduling. Latter is modeled as chain of components that passes the remaining service top down. Within this chain, processes as well as EDF and FIFO scheduler components can be included.

2.3.2 Command Order Constraint

Apparently, a certain sequential order of the used commands in Lst. 2.1 is necessary, such that all corresponding input parameters are available before the next command is issued. Finding a valid order is the major issue when generating scripts that employ the MPA-RTC toolbox.

Event and Service Flow

A command set can be ordered to reflect the event flow or the service flow, respectively. Finding a direct solution is only possible if both sequences are not mutually exclusive. An iterative solution for this problem is covered later (see 2.3.8.2).

Complex Components

In MPA-RTC, there exist two complex components that embed a number of processes. More precisely, only one single command is issued to define an EDF or FIFO scheduler including all managed tasks. Thus, the input arrival curves of all embedded tasks need to be available in advance. A direct solution for the ordering problem is no longer available if an event path between two or more embedded tasks of the same complex component can be found (see Fig. 7). In this case, the problem also needs to be solved with an iterative solution (see 2.3.8.3).

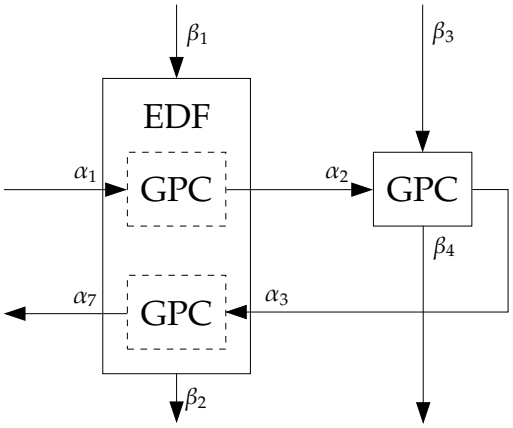


Fig. 7: EDF component with an event path between two embedded tasks in MPA-RTC

2.3.3 Component Graph

Satisfying the ordering constraint is difficult. Therefore, building a simple converter for MPA-RTC is not feasible.

The presented approach introduces an extra internal graph representation for MPA-RTC models. A component graph is a directed graph that describes both event and service flows.

This concept is necessary for two reasons. Firstly, an algorithm must be able to decide if a certain process network can be modeled in MPA-RTC. Solving this problem is related to the analysis of dependency relations in this approach. Secondly, an order of the given components that satifies the command order constraint must be found. Here, the basic idea is to perform sort operations on subsets of all given components.

2.3.3.1 Graph Structure

A vertex of a component graph corresponds to a component of the abstract MPA-RTC model. The list of components includes event sources, event sinks, activation units, resources and all kinds of processing components such as greedy processing components (GPCs) and schedulers.

Directed horizontally and vertically aligned edges are connecting the vertices of a component graph.

A vertical edge points from a service parent to its single service child that receives the remaining service. Likewise, a horizontal edge denotes that an event stream is passed from a parent to its child. The weight of an horizontal edge stands for the scaling factor of the arrival curve. For instance, a factor of two means that twice as much events arrive in any time interval.

Each component is limited to one incoming and one outgoing edge in the service domain. In contrast, the number of outgoing horizontal edges is not limited for any component. Furthermore, an activation unit has to receive the arrival curves of exactly two parents.

Service Paths

The term service path refers to a special subgraph of a component graph. Only vertices that are connected by a specific vertically aligned path are included in this subgraph. The order of a service path is explicitly given by the sequence that is described by the graph.

On the abstract system level, all included components share the service of the same resource. Here, the sequence of the components corresponds to the scheduler priorities of the underlying fixed-priority scheduler (see 2.3.1).

2.3.3.2 Graph Generation

Event and service flows are described by the process network (see 2.2.1) and the resource model (see 2.2.2) of the abstract model, respectively.

Extracting the vertices of the corresponding component graph is a simple task. Basically, an instance of the matching MPA-RTC component is created for each entity of the abstract model.

Special Case: Converting Activation Units

As the only exception, activation units cannot be converted without further modifications. The problem is that the abstract model is not limiting the number of inputs of activation units, while activation units in MPA-RTC are only defined for exactly two inputs.

The solution is to replace a single activation unit by several activation units in the MPA-RTC model. The used procedure starts with a list that includes all input ports. Each pair of two ports is replaced by the single output port of an added activation unit during iterative runs over the changing list. In the final state, the list contains only one single entry that refers to the output port of the whole activation unit.

To the end, a network that contains u activation units on l layers is used to model one activation unit with n input ports in MPA-RTC (see Eq. 2.1 and Eq. 2.2). Fig. 8 shows how an activation unit with five incoming ports is modeled in MPA-RTC.

$$l = \lceil \log_2 n \rceil \tag{2.1}$$

$$u = \left(\sum_{i=1}^{l} 2^{i-1} \right) - \left(2^l - n \right) = \left(2^l - 1 \right) - \left(2^l - n \right) = n - 1 \tag{2.2}$$

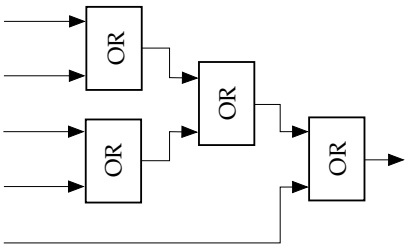


Fig. 8: Activation based on the disjunction of five incoming event streams in MPA-RTC

Connecting the Graph

After the vertices have been extracted, at least two solutions exist for connecting the graph in both dimensions.

The first approach is to create a separate graph for each dimension. Both graphs need to be merged afterwards by comparing matching components.

Based on so-called event stream connectors, the second and finally used approach needs to process each component only once.

An event stream connector corresponds to a process network connection. All connectors can be generated at once by parsing the list of process network connections. This step is actually performed in the initialization phase of the component graph generation.

As the proper order of any service path is already accessible in the abstract model, links in the service domain can be established immediately while MPA-RTC components are created along the given sequence.

Concerning connections in the event domain, it is not guaranteed that the component which refers to the connection peer has been instantiated, yet. But, the event source connector that corresponds to the given process network connection already exists. So, any component can be linked in both involved dimensions immediately after it has been instantiated.

Concluding, splitting any process network connection in two sections, that can be established asynchronously, leads to an efficient algorithm for the generation of component graphs.

2.3.4 Dependency Relations

There exist two types of dependencies, namely service and event dependencies. Here, dependency between two MPA-RTC components is regarded as a transitive binary relation.

Let C denote a set of MPA-RTC components. In the following, the two relations $R_{service} \subseteq C \times C$ and $R_{event} \subseteq C \times C$ are introduced.

Def. 1: **(Service dependency)** $\forall a, b \in C : (a, b) \in R_{service}$, *if a directly or indirectly consumes the remaining service of b. In words, this means that a is depending on b in terms of service.*

Def. 2: **(Event dependency)** $\forall a, b \in C : (a, b) \in R_{event}$, *if there exists at least one path that starts at a and ends at b. In words, this means that a is depending on b in respect of events.*

For simplification, $R \subseteq C \times C$ corresponds to one of $R_{service}$ and R_{event} in the further explanations. Apparently, both relations are independent and cannot be mixed.

2.3.5 Component Interrelations

Eq. 2.3 describes the case of two MPA-RTC components that are linked in only one direction.

$$a, b \in C, a \neq b : (a, b) \in R \wedge (b, a) \notin R \tag{2.3}$$

In contrast, the relation between two components that are not linked at all can be described by Eq. 2.4.

$$a, b \in C, a \neq b : (a, b) \notin R \wedge (b, a) \notin R \tag{2.4}$$

A complex component, e. g. an EDF scheduler, may have links between two or more embedded components. The term self-dependency is used for this setting that is characterized by Eq. 2.5.

$$a \in C : (a, a) \in R_{event} \tag{2.5}$$

Symmetric relations between two commands a and b as stated in Eq. 2.6 are only allowed if the dependency between a and b is not strict. In other words, the emerging cycle must be connected to at least one other component $c \in C$ outside the cycle for which Eq. 2.7 holds. More precisely, cyclic event dependencies are only allowed if the cycle is activated by a disjunction of the feedback of the cycle itself as well as an event stream that is produced outside the cycle. For illustration, a valid event stream cycle configuration is shown in Fig. 9.

$$a, b \in C, a \neq b : (a, b) \in R_{event} \wedge (b, a) \in R_{event} \tag{2.6}$$

$$(b, c) \in R_{event} \wedge (a, c) \in R_{event} \wedge (c, a) \notin R_{event} \wedge (c, b) \notin R_{event} \tag{2.7}$$

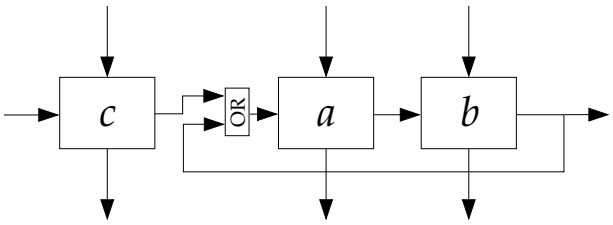


Fig. 9: Event stream cycle in MPA-RTC

2.3.6 Graph-based Dependency Evaluation

In the trivial case, $(a, b) \in R$ is obvious when both components are explicitly joined by an edge in the component graph.

The membership $(a, b) \in R$ of a not directly linked pair $a, b \in C$ can be checked by exploiting the transitivity of the relation. Starting from a it is recursively checked if $(p, b) \in R$ holds for any directly connected parent

$p \in C$. If $(a, b) \notin R$, the search ends when a component that has no parents, such as an event source or a resource, is reached.

2.3.7 Sorting a Set of MPA-RTC Components

A set of MPA-RTC components can be sorted by analyzing its mutual dependencies.

There are no limitations on how a set of components for sorting has to be composed. For instance, a set can include components that are part of two subsets that are not related in the considered dimension (see Eq. 2.4). Thus, some elements of a set of components may not be comparable. As standard sort algorithms like merge sort expect data sets that can be totally ordered, these algorithms are not applicable for the given problem.

Def. 3: **(Dominated element)** *A component $c \in S \subseteq C$ is dominated by the subset $\{S \setminus c\}$ if $\exists d \in \{S \setminus c\} : (c, d) \in R \wedge (d, c) \notin R$.*

Let $S \subseteq C$ denote the list of all components that are not part of the ordered sequence, yet. The proposed sort algorithm chooses always the first element $c \in S$ that is not dominated by the subset $\{S \setminus c\}$. This is done by iteratively choosing one candidate $c \in S$ that is individually compared with each component $d \in \{S \setminus c\}$. If c is not dominated by any component $d \in \{S \setminus c\}$, c is added to the ordered sequence.

Eq. 2.8 highlights the upper and lower bounds for the number of comparisons c that are necessary to sort a set with a cardinality of $n \geq 2$.

$$\sum_{i=1}^{n-1} i = \frac{n \cdot (n-1)}{2} \leq c < \sum_{i=0}^{n-2} (n-i) \cdot (n-i-1) \tag{2.8}$$

It becomes apparent, that the proposed algorithm is not scaling for large sets. For instance, more efficient standard sort algorithms could be applied if a set of components is split into independent subsets beforehand.

Special Case: Sorting a Service Path

However, it is still desirable to use this strategy for a specific reason. Concretely, it offers a better sort order when a service path (see 2.3.3.1) has to be sorted in both event and service dimensions.

As already mentioned, the two sequences that result from sorting the same set concerning event and service dependencies, respectively, may not match. Because inconsistencies in the service dimension can be fixed in the later Matlab script generation (see 2.3.8.2), top priority is given to the order that refers to the event flow when a service path is sorted. So, the

goal is to always ensure the correct order in the event dimension, and the best achievable order in the service dimension. Here, the best achievable order of a service path refers to the order with the smallest number of permutations compared to the correct order in the service dimension.

For illustration, the components of a service path are already ordered properly concerning the service flow after the graph generation. The proposed sort algorithm is then used to guarantee the correct order in the event dimension. As the algorithm always selects the first eligible element of the presorted list, only the least necessary number of permutations is introduced.

In contrast, splitting the service path in the event dimension for the utilization of standard sort algorithms would destroy the sequence of the presorting step.

Furthermore, it turns out that the current implementation of the MPA-RTC toolbox is more likely to be the bottleneck for the analysis of large systems with a high number of components.

2.3.8 Matlab Script Generation

A component graph contains all information that are necessary for finally generating a Matlab script.

Concerning the syntax, the corresponding toolbox command for any component is given by the type of the component. Variable names can be implicitly determined by either the incrementing edge index or the name of the modeled process, respectively. Finally, special properties like the weight of an edge can directly be converted into Matlab code by using predefined macros.

2.3.8.1 Finding the Next Command

Without any limitation, all commands that refer to event sources can be written immediately as they are not dependent on any other command. A similar rule applies to event sinks and latency observations, the corresponding commands are always displayed at the end of a script.

All remaining components are organized in a number of sets. From these sets, all except one are related to an ordered service path (see 2.3.7). There is one set of this kind for each involved resource, resources that are scheduled by TDMA have a separate set for each TDMA slot, respectively.

The order of the underlying sorted service path is preserved in the corresponding set. Thus, each set starts with a resource as the first element. Again, the corresponding commands of all resources can be written out immediately without caring about the order.

As activation units are only linked in the event dimension, they are

not included in any service path. Therefore, activation units are treated separately by a dedicated, unordered set.

Iterative Command Selection

In general, a component is removed from its set after it has been written out. For an ordered set this implies that all remaining components are shifted upwards by one position.

After removing the resources, there are n components that are arbitrarily distributed over m sets. Each of this m sets refers to an ordered service path. Additionally, a activation units can be found in the extra set.

The iterative selection algorithm picks exactly one component per step. Thus, $n + a$ iterations are necessary to get a valid command order of $n + a$ toolbox commands, or MPA-RTC components, respectively.

Any component demands and offers a number of variables that correspond to arrival curves. In the basic case, a component can be picked if all demanded event variables are already available in the script. The list of already known event variables is updated after each issued command. Section 2.3.8.3 deals with an expansion of this procedure for resolving complex dependencies.

As all service path sets are already ordered concerning the event flow, only the top most element of each set is eligible in all steps. Obviously, this does not hold for the extra set of unordered activation units. Here, all remaining activation units have to be considered as candidates in all iterations.

Fig. 10 illustrates the described method.

The bounds for the number of needed tests c for determining the command order of $n + a$ components are stated in Eq. 2.9. Again, m denotes the number of sets over which the n components are arbitrarily distributed.

$$n + a \leq c < m \cdot n + (m + 1) \cdot a \tag{2.9}$$

2.3.8.2 Computing the Service Flow

By now, the service flow always came second. Similar to the event variables, components also offer and demand certain variables that correspond to a service. Again, there is a list of already provided variables that is updated after each displayed command.

Because breaking the order in the service dimension is allowed when sorting a service path, the demanded service variable of a recently picked command might not be available, yet. In this case, the value of the missing variable has to be determined by iterative calculations. This is done by

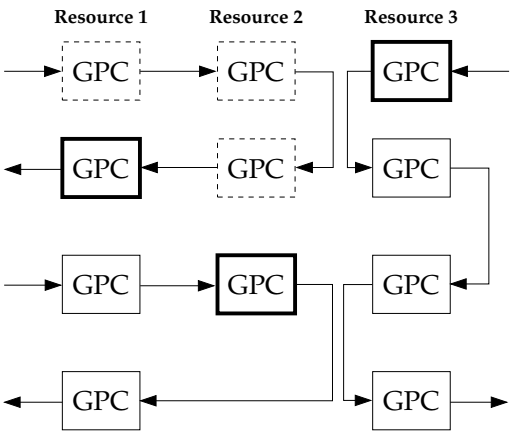


Fig. 10: Snapshot of an iterative command selection. Each column of components corresponds to a service path that has been sorted in the event dimension. All dashed components have already been issued. The three components with the bold border refer to the only three candidate components that have to be evaluated in the current iteration.

adding extra code to the Matlab script.

Concretely, an iterative analysis that represents a fixed-point problem is used. For the case that after an iteration all calculated output service curves stay unmodified, convergence is reached and the last calculated output service curves are valid.

First of all, the missing variable is initialized with the value of the lastly written service variable of the same service path (see Lst. 2.2, line 7). The loop header that follows (see Lst. 2.2, line 13) marks the beginning of a block of iterated commands. The command that demands the missing service variable is the first command within this block (see Lst. 2.2, line 15). The block is basically closed after the command that provides this variable has been written (see Lst. 2.2, line 18 and line 31).

In the execution of the generated script, a solution is found if the iteratively calculated value for the missing variable converged. This means that the upper and lower service curve did not change between two sequential calculation steps (see Lst. 2.2, line 23). A source code example for iteratively estimating the value of a service variable is given in Lst. 2.2.

Lst. 2.2: Computing the service flow by iterative calculations

```
01 % Fully available resource, bandwidth = 10
02 b1 = rtcfs( 10 );
03 % period = 3, jitter = 1, min. distance = 2
04 I1_out = rtcpjd( 3, 1, 2 );
05 % b2 is missing before the second task has been analyzed. Here,
06 % b2 is initialized with value of the already known variable b1.
07 b2 = b1;
08 b2_upper = b2(1);
```

```
09 b2_lower = b2(2);
10 b2_upper_last = rtcuplus(b2_upper);
11 b2_lower_last = rtcuplus(b2_lower);
12 conv_b2 = false;
13 for it_b2 = [1:20]
14     % First task in event flow order, demands b2
15     [ T2_out b3 T2_delay T2_buf ] = rtcgpc( I1_out, ...
16         b2, [ 4.0 1.0 ] );
17     % Second task in event flow order, provides b2
18     [ T1_out b2 T1_delay T1_buf ] = rtcgpc( T2_out, ...
19         b1, [ 1.0 1.0 ] );
20     b2_upper = b2(1);
21     b2_lower = b2(2);
22     % Both upper and lower service curve must converge.
23     if ((b2_upper == b2_upper_last) & (b2_lower == b2_lower_last))
24         conv_b2 = true;
25         break
26     end
27     % Save the current values for the comparisons in
28     % the next iteration.
29     b2_upper_last = rtcuplus(b2_upper);
30     b2_lower_last = rtcuplus(b2_lower);
31 end
32 if (conv_b2)
33     disp(['Fixed point for b2 reached.'])
34 else
35     disp(['Fixed point for b2 NOT reached.'])
36 end
```

2.3.8.3 Solving Event Dependencies

During the iterative command selection (see 2.3.8.1) it is checked if a candidate component has a self-dependency in the event dimension. So, the candidate component either refers to a complex component with an event dependency between embedded tasks, or to a component that is part of an event stream cycle.

If a candidate component has a self-dependency in the event dimension, it is selected. But, instead of directly issuing the corresponding command to the Matlab script, the component is pushed on a temporary stack. This stack refers to the so-called dependent commands stack.

After each command selection, it is checked if the overall demand for defined variables of the dependent commands stack can be fulfilled by already really issued commands and commands that are on the stack. Any next selected command is pushed onto the dependent commands stack until this conditions holds.

Flushing the Stack

If all variable requests of the dependent commands stack are fulfilled, it can be flushed. First of all, this always refers to performing iterations until a fixed-point is reached. Here, all calculated output event streams must stay unmodified after an iteration so that convergence is reached.

For calculating the start values of the first iteration, another temporary stack is created. Here, complex components are not added as a whole, but as set of several components of which each refers to an embedded task (see Lst. 2.3, line 7 and 15).

Additionally, a disjunctive activation unit is replaced by a variable assignment if only one of both input variables is available in the first iteration. This method refers to feeding an event stream cycle with a first event from outside.

After all components have been copied to the new stack, the stack contents get sorted in the event dimension. After the resulting sequence of commands of this sort operations has been issued to the Matlab script for initializing all variables, all components of the dependent commands stack are also issued in the same sequence as they were pushed onto the stack. Here, the latter commands are surrounded by a loop that corresponds to the fixed-point iterations that need to be performed (see Lst. 2.3, line 20 to 26).

Finally, the contents of both stacks can be flushed. Normal operation with directly issuing commands takes place as long as no further component with a self-dependency in the event dimension is detected.

Lst. 2.3: Resolving a self-dependency within a FIFO component

```
01 % Fully available resource, bandwidth = 10
02 b1 = rtcfs( 10 );
03 % period = 3, jitter = 1, min. distance = 2
04 I1_out = rtcpjd( 3, 1, 2 );
05 % The FIFO component can not be issued before
06 % T1_out has been initialized.
07 [ T1_out b2 ] = rtcgpc( I1_out, b1, [ 1.0 1.0] );
08 T1_out_upper = T1_out(1);
09 T1_out_lower = T1_out(2);
10 T1_out_upper_last = rtcuplus(T1_out_upper);
11 T1_out_lower_last = rtcuplus(T1_out_lower);
12 % The values of the second GPC are also calculated
13 % here for having a previous value in the convergence
14 % check during the first iteration.
15 [ T2_out b2 ] = rtcgpc( T1_out, b2, [ 4.0 1.0] );
16 T2_out_upper = T2_out(1);
17 T2_out_lower = T2_out(2);
18 T2_out_upper_last = rtcuplus(T2_out_upper);
19 T2_out_lower_last = rtcuplus(T2_out_lower);
20 for it_horizontal = [1:20]
21     [ T1_out T1_delay T1_buf T2_out T2_delay ...
22         T2_buf b2 ] = rtcfifo( I1_out, [ 1.0 ...
23         1.0 ], T1_out, [ 4.0 1.0 ], b1 );
24     % Removed for illustration:
25     % Check convergence of T1_out and T2_out.
26 end
```

2.3.8.4 Modeling Observations Targets

Mainly due to resource sharing, events may have to wait in the buffer of the communication channel before they can be processed from the

following task.

Backlog Observation Targets

MPA-RTC has no prior limits on the size of communication buffers, a communication buffer is always available for writing. Furthermore, it is assumed that any task is immediately writing an event to the buffer of the outgoing communication channel after an arriving event has been processed.

Depending on the processing capabilities of the receiving task, its input buffer might get filled up more and more by arriving events. The upper bound of the size of the emerging backlog is necessary for calculating the minimum buffer space requirements.

The minimum buffer requirements of any single task are directly available in the result vector of the corresponding toolbox commands. Examining standalone task-specific buffer space requirements is sufficient when analyzing systems that allocate a dedicated memory region of a certain size for each communication channel.

For systems that allocate one memory region that is shared between all buffers, there exist two ways for estimating the required size of the shared memory region. Firstly, it is possible to sum up all task-specific buffer space requirements. But, this method is too pessimistic as it assumes, that all buffers may reach the maximum buffer fill level at the same time. Here, MPA-RTC offers an extra way that is called “pay bursts only once” [21] for estimating tighter bounds.

In SHAPES, a separate buffer is allocated for each communication channel. Thus, the current implementations of the abstract model as well as the MPA-RTC writer are only supporting task-specific backlog observations targets.

Event Delays

After an event has arrived in the input buffer of a task, a certain time elapses until the event has been processed. Firstly, a task might not be able to process the arriving event immediately due to the availability of the service. Secondly, the processing time of an event is depending on its execution demand. The maximum event delay refers to the maximum possible time that might be necessary to process one event due to both described effects.

Again, the standalone task-specific maximum event delay is part of the result vector of all toolbox commands that refer to tasks.

End-to-End Delays

The needed time amount for processing an event by a chain of connected tasks is called end-to-end delay. In the abstract model, this refers to a latency observation. A latency observation can either be defined for one single task or two tasks that are connected by at least one path in the event dimension. The maximum latency contains all occurring waiting and processing times, including those that appear at the two ends of the observation target.

Once more, an additive approach or "pay bursts only once" can be used. Although "pay bursts only once" leads to tighter bounds, the current implementation uses the additive approach.

Multipath Latency Observations

Using the additive approach means that all local delays that occur by walking along a connecting event path need to be added up. Again, all task-specific event delays are already given in the result vector of all toolbox commands that are related to tasks.

Due to components that support multiple event outputs, the two given ends might be connected by more than one path. Fig. 11 shows an example configuration with two event paths between two components. All possible paths need to be evaluated for determining the worst-case delay.

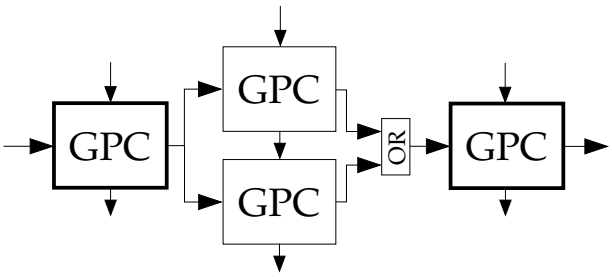


Fig. 11: Example for multiple event paths between two components in MPA-RTC

Graph-based Tracing of Multiple Event Paths

The proposed algorithm for determining all paths is based on the component graph (see 2.3.3) of the analyzed model. The iterative search procedure starts at the end that is closer to the event source.

Only one path is investigated at the same time. An internal data structure saves the sequential order of the already visited components on the investigated path.

In each iteration, all event children of the component that was most recently added to the path history are checked. Here, event dependency relations (see 2.3.4) are used for finding child components that are connected with the target component. The first eligible child component is added to the history of the current path.

If there are more than one suited children, a copy of the current path history is made for each remaining child. The currently evaluated child is added to the copy, the copy is pushed on an extra queue that contains partial paths that need to be evaluated one after another.

A path is complete and thus removed from the queue when the target component is reached. The full procedure terminates when the path queue is empty.

2.4 XML Reader and Writer

For any instance of the abstract model, an equivalent instance of the introduced XML document type exists.

There is basically no difference between the data structures of both representations. Each entity type of the abstract model is directly related to an XML element type. Hierarchical data structures in the abstract model correspond to nested sets in the XML representation.

The current implementation of both XML reader and writer is based on SAX and JDOM. An XML instance verification step is part of the XML reader.

2.5 Model Verification

The validity of a given instance of the abstract model can be checked by the model validator. As a part of the concept, a model instance has always to be verified before it is passed to a writer. This makes the overall approach more robust as any implementation of a writer is only required to work on valid models.

Currently, the following malfunctions are checked by the model validator:

- **Minimum Components Constraint**
  Any instance of the abstract model must at least contain one event source, one process, one event sink and one resource.

- **No Unused Ports Constraint**
  All input and output ports need to be connected.

- **Single Input Constraint**
  Except for activation units, only one connection is allowed per input port.

- **Invalid Process References**
  Connection elements, latency observations and backlog observations contain references to processes. All referenced processes must be existent.

- **Unused Resources**
  At least one task has to be bound to any resource. In detail, this means that a scheduling strategy with at least one scheduled task must be defined for each resource.

Although it is still possible to specify erroneous models, it turned out that the current implementation detects the most common errors.

2.6 Related Work

A tool-independent XML system description format called tool-independent system description for real-time embedded systems (TiDRES) is proposed in the semester thesis of Christoph Rüegg [34]. Here, the main motivation is to have a single system specification that can be converted into tool-specific formats. Currently, there exist XSLT transformations for interfacing SymTA/S, PESIMDES [29] and MPA-RTC.

On the abstract view, TiDRES is limited to a smaller class of describable systems than the new abstract model. For instance, TiDRES does not allow to share a resource by more than one scheduling strategy. Additionally, activation units are limited to only two inputs.

Concerning the implementation, TiDRES is currently not able to create MPA-RTC models that employ EDF or FIFO schedulers. There is a first approach for coping with cyclic dependencies, but TiDRES is not able to deal with complex dependencies in a way as the proposed MPA-RTC writer does. Additionally, multipath latency observation is one of the features that are also not supported by the current implementation of TiDRES.

Apparently, there exist more abstract description languages for system modeling, such as Ptolemy [5]. However, TiDRES and the proposed abstract model are the only two known description languages that can be used for interfacing MPA-RTC.

2.7 Moses

A modeling language specification for Moses [24] that corresponds to the abstract model has been created. Now, Moses can be used for graphically creating instances of the abstract model and therefrom Matlab scripts that employ the MPA-RTC toolbox.

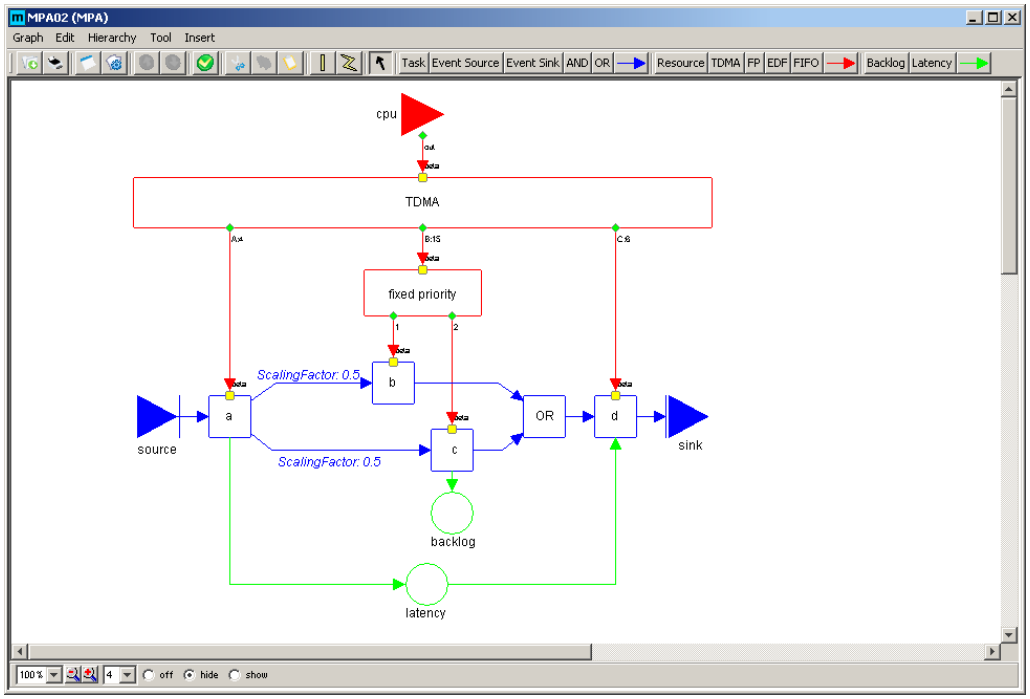


Fig. 12: Screenshot of Moses application. Instances of the abstract model and therefore MPA-RTC models can be generated with a graphical user interface.

A screenshot of the resulting application is given in Fig. 12.

2.8 Implementation Environment

Java 1.5 has been used for implementing the described concepts.

3

Analysis Model Calibration

The proposed design space exploration tool flow uses formal-based analysis methods for the evaluation of application to architecture mappings. In contrast to existing approaches, accurate performance data are used for specifying execution demands.

Here, instruction-accurate simulation offers the desired level of result accuracy. Apparently, it is not reasonable to carry out a time-consuming instruction-accurate simulation for each design point. Instead, the idea is to use instruction-accurate simulation only during an initial analysis model calibration. In this calibration, a small set of design points, so-called calibration mappings, is evaluated.

After the one-time calibration, the necessary performance data for each evaluated analysis model instance is loaded from the calibration results.

This chapter presents an approach for performing an analysis model calibration by analyzing execution traces of instruction-accurate simulation. At first, the class of supported systems is described. Then, required performance parameters are identified. After presenting how the whole design space can be covered in an economic way, the extraction of the desired performance parameters is described.

Chapter 5 describes a reference implementation of the presented approach for SHAPES.

3.1 System Model

The described approach is suited for the analysis of distributed systems that can be represented as a Kahn process network (KPN) [16].

To form a process network, a group of processing units are connected by communication channels. The unbounded FIFO communication channels are used for transferring atomic data units. Here, an event refers to an atomic data unit.

All processes are iteratively invoked, a firing corresponds to an invocation.

3.2 Performance Parameters

In an analysis model, a platform is basically represented by a set of resources that refer to processors and communication busses.

Any resource offers the capability to process a certain workload during a given time interval. The measurement units for describing a processing capability as well as for characterizing an execution demand must match specific to the addressed resource.

Basically, any artificial unit, such as processed video frames, can be used for describing demands and capabilities.

However, it is desirable to use standard units for two reasons. Firstly, the capabilities of a hardware component can be directly extracted from the data sheet without carrying out extra benchmark measurements. Secondly, most generalization is reached by sticking to standard units.

Capabilities of a Communication Bus

Concerning a communication bus, it is obvious to specify capabilities and workload in bytes per second and bytes.

Capabilities of a Processor

The capabilities of a processor could be specified in cycles per time unit, instructions per time unit or processing time per time unit.

Using instructions per time unit is only reasonable if any instruction corresponds to the same number of cycles. This is not given for all instruction set architectures.

The remaining options cycles and processing time per time unit offer the same accuracy. But, using the latter is more desirable as this unit demands the lowest level of implemented architectural details in the used measurement tool.

Above all, it becomes apparent that any processor offers one second of processing time per second and processing core. Thus, workload measurements specific to different processors can be compared easily.

Needed Performance Parameters

The workload of a computation has to be given in the necessary processing time specific to a certain processor. In turn, the workload of a bus transaction is measured in the amount of transferred data in bytes. All measurements are subject to a firing of the observed process.

3.3 Measuring Amounts of Transferred Data

Obviously, the amount of transferred data during a bus transaction is neither specific to the used platform nor to an exact design point.

By restricting the system model to synchronous data flow [22], the amount of transferred data per firing and channel is known a priori from the application specification.

For the more general case, functional simulation is sufficient for getting the upper and lower bounds of the amount of transferred data per firing and channel.

3.4 Measuring Processing Times

Purely event-based methods such as functional simulation are not suited for measuring processing times. Here, instruction-accurate simulation is used as this method also models the time behavior of the platform.

3.4.1 Influences

There are mainly three influences that are affecting a measured processing time of a certain application source code.

Processor

Firstly, the processing time is specific to the used processor as well as the related architecture-specific compiler that was used to generate the executable code.

Communication

In general, a processor has to read and write data from and to buffers for transferring data. Here, a buffer refers both to dedicated memory resources as well as send and receive buffers of communication resources, such as DMA or ethernet controllers.

A processing time measurement is influenced both by the characteristics, such as latency, of the accessed buffer as well as the characteristics of the connection. The combination of a buffer and the used connection specific to the used processor refers to a communication path.

Resource Sharing

Lastly, extra waiting times can occur during communication operations due to the employed resource schedule and resource sharing in general. For instance, a full buffer has to be emptied by another process before data can be written again. Thus, a process that wants to access a full buffer for writing has to wait until the buffer is accessible again.

As the analysis of the impact of resource sharing is an objective of the later formal-based analysis, the influence of resource sharing does not have to be addressed in the calibration process.

In particular, formal-based analysis methods require that the specified execution demands reflect the necessary workload without any influence of resource sharing such as blocking.

3.4.2 Purpose

Apparently, the initial one-time calibration has to cover all possible design points that might be chosen for an evaluation in the later formal analysis.

This can be achieved most economically concerning the usage of time-consuming instruction-accurate simulation, if the outcome of each simulated calibration mapping can be re-used for calculating the performance data of as many design points as possible.

Dividing the Program Path

The idea is to divide the program path that is executed during a firing of a specific process into several sections that can be assigned to categories. Each category is treated separately concerning the measurement of the processing time. The goal is a modular system that allows to generate missing values by combining partial measurements.

On the first level, there exist two categories of sections, namely communication and computation. Here, communication is not only limited to memory accesses, it includes all actions that need to be taken

for reading or writing data, respectively. For instance, looking up the proper communication device by a given device name is also a part of a program path section that refers to communication.

For processes that communicate over different channels during one firing, a communication section can also be assigned to a specific process network channel in the second available dimension.

Calibration Result

Recapitulatory, the goal of the calibration process is to extract a result set that contains the following data in terms of processing times.

Firstly, it contains the totaled processing time of all sections that refer to a computation section of the program path. This number has to be given for all processors to which the process can be bound to.

Secondly, it includes the added processing time of all sections that refer to a communication section specific to the used process network channel. This measurement has to be given for any possible combination of a process, a process network channel and a communication path to which the channel is bound to. Here, the processor to which the process is bound to is implicitly given by the used communication path.

Depending on the concrete implementation, the processing times of multiple measured firings can either be averaged or given in a worst-case and best-case sense.

The overall execution demand for one firing of any process for a particular design point is given by adding the corresponding processing times for computation and communication.

Obviously, all measurements must be subject to the same input data to ensure the validity of the modular approach. The additionally needed determinism between input and output data is already given by the system model.

3.4.3 Calibration Mapping Generation

A set of calibration mappings is complete if both ends of any communication channel are mapped at least once to any available communication path. As a communication path is always subject to a specific processor, any process gets implicitly also bound to all available processors at least once.

While there are no restrictions on the employed resource sharing strategies in a calibration mapping, it is desirable to avoid waiting times as well as to ensure that communication paths are not used simultaneously by processes that run concurrently on different processors. As this might be difficult for complex systems, it may be necessary to post-process

Tab. 1: Example calibration result contributions

Usage	Totaled processing time
Computation	31 ms
Communication, Channel A	10 ms
Communication, Channel B	10 ms
Communication, Channel OUT	15 ms

measured processing times.

3.4.4 Execution Trace Analysis

Execution traces of an instruction-accurate simulation are used for measuring processing times.

3.4.4.1 General Procedure

For any calibration mapping, the corresponding software stack is generated and executed on the instruction-accurate simulator. An execution trace is generated during each simulation run.

Basically, an execution trace contains an entry for each function call during a program execution. Besides of the name of the called function, the processing time that was spent during the specified function call is given.

Listing 3.1 shows an example source code of an application that is written corresponding to the DOL API. An example execution trace with already calculated processing times and annotated channel names is given in Fig. 13. Table 1 shows the contributions of the assumed calibration mapping to the calibration results.

The given results are specific to an arbitrary design point that refers to a calibration mapping. While it is desirable to evaluate a number of firings of the same process and a specific calibration mapping, only one firing is evaluated here for illustration purposes.

Lst. 3.1: Example application code

```
01 int example_fire(DOLProcess *p) {
02     double c, d;
03     DOL_read((void*)PORT_A, &c, sizeof(double), p);
04     DOL_read((void*)PORT_B, &d, sizeof(double), p);
05     double sum = c + d;
06     DOL_write((void*)PORT_OUT, &sum, sizeof(double), p);
07     return 0;
08 }
```

Fig. 13: Annotated example execution trace

```
example_fire   //  5 ms, computation
DOL_read       // 10 ms, communication, channel A
example_fire   //  1 ms, computation
DOL_read       // 10 ms, communication, channel B
example_fire   // 20 ms, computation
DOL_write      // 15 ms, communication, channel OUT
example_fire   //  5 ms, computation
```

3.4.4.2 Identifying Process and Channel Context

In the given example, the two occurrences of `DOL_read` were assigned to different process network channels. Furthermore, all three mentioned symbols `DOL_read`, `DOL_write` and `example_fire` could appear in the process context of different instances of the implementation that is given in Lst. 3.1. Intentionally, information about the current process and channel context is not part of an execution trace.

Here, four sources for identifying process and channel contexts are presented.

Application Source Code Modification

The application sources could be modified in a way that process context and channel identifiers are written to a console. The recorded output can be used to extract process context and channel sequences that can be matched with the execution trace. However, this way should only be chosen as the last possible option as it modifies the processing times by adding extra instructions.

Source Code Analysis

If static scheduling is used, process context sequences can be extracted from a source code analysis. For a known input data set, the sequence of used communication channels can also be extracted from a source code analysis.

Functional Simulation

In the scope of a specific task, the sequence of used communication channels for given input data is invariant to the used simulation method. This method is desirable if a functional simulation is available that offers the desired details.

Extended Execution Traces

Specific to features of the instruction-accurate simulator, an execution trace may include additional parameters such as memory and register contents that contain information about the current process and channel contexts.

3.4.4.3 Post-Processing

Due to effects of the resource sharing scheme that is used in the evaluated calibration mappings, it might be necessary to post-process the measured processing times. An example approach is given in 5.6.4.

4

SHAPES Internals

This chapter aims to give an introduction about several SHAPES internals that are necessary for understanding the following chapters dealing with a concrete tool flow implementation for SHAPES.

4.1 VSP Architecture

The currently used version of the Virtual Shapes Platform models one single tile that consists of one ARM9 RISC and one mAgic VLIW DSP core. Both cores are integrated on one single Atmel Diopsis 940 chip. Fig. 14 shows the simplified layout of the Atmel Diopsis 940 chip.

Floating-Point Support

Only the mAgic core features a floating-point unit, a special math library has to be used for carrying out floating-point calculations on the ARM9 core.

Memory Resources

Tab. 2 gives an overview about the main memory resources of the VSP architecture.

While the DXM is directly mapped to the address space of the ARM9 core, a DMA controller must be used for transferring data between DDM and DXM.

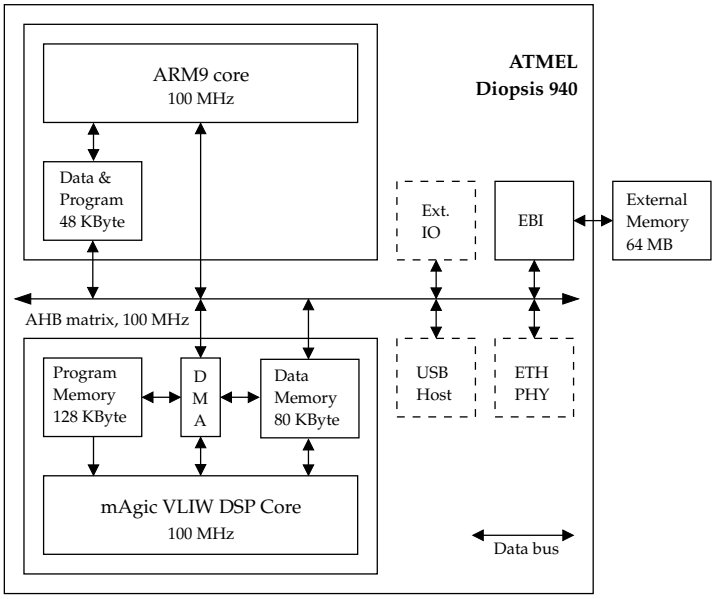


Fig. 14: Atmel Diopsis 940 with external memory

Tab. 2: VSP memory configuration

Resource	Shortcut	Size
ARM9 Internal RAM (Program and Data)	RDM	48 KByte
mAgic DSP Internal Data Memory	DDM	80 KByte
mAgic DSP Internal Program Memory		128 KByte
External RAM	DXM	64 MB

4.2 Distributed Operation Layer

While there exist more detailed resources about DOL, this section intends only to explain a set of basics that are necessary to understand further chapters.

In particular, detailed information can be found in the documents “Semantics of the DOL XML Schemata”, “C/C++ Coding Guide” and “Tool Description” [26].

4.2.1 Platform Specification

Concerning real hardware components, the platform specification states the available processors, memory resources and communication bus resources.

Communication Paths

Beside of real hardware components, the platform specification describes virtual communication paths.

According to the application model (see 1.2.2), processes communicate over channels that are equipped with FIFO buffers. The buffer of a communication channel corresponds to a dedicated region of a memory resource in the real implementation. Apparently, a shared memory resource has to be used if the two communicating tasks are located on different processors.

A write path describes how a specific processor can access a particular memory resource that hosts a channel buffer. In turn, a read path describes how a specific processor can read the channel buffer. In both cases, the declaration of the communication path states the employed communication bus resources.

4.2.2 Mapping Specification

Where and how components of an applications are executed on the platform is defined in an instance of the mapping specification. In the spatial domain, bindings define how processes and software channels are mapped to processors and communication paths, respectively. More precisely, a software channel is mapped to a combination of a write path and a read path. Apparently, a combination is only reasonable if both parts expect the channel buffer on the same memory resource.

Mappings in the temporal domain are referred to as schedules. A scheduling policy including its according parameters is assigned to each resource in this domain.

In connection with the HdS, it is currently not possible to directly configure the schedules of a system. Concretely, a static sequence of task activations is automatically generated during the software stack generation. Detailed information on this process is missing.

4.2.3 Programming Model

In general, a DOL application has to be implemented in the C programming language. The process execution model expects two specific functions that need to be implemented for any DOL application process.

The initialization function is executed once for each instance of a process when the application is started. After the initialization, the execution control is reiteratively calling the so-called fire function of any process instance.

Lst. 4.2 shows an example instance of the DOL process handler

definition that is given in Lst. 4.1. Here, an application developer can describe the corresponding names of the implemented functions that refer to the initialization and the fire function, respectively. Besides the two function names, a reference to a data structure that corresponds to the local state of a process instance has to be specified.

Lst. 4.1: DOL process handler definition

```
01 typedef struct _process {
02     LocalState     local;
03     ProcessInit    init;
04     ProcessFire    fire;
05 } DOLProcess;
```

Lst. 4.2: Example DOL process handler

```
01 DOLProcess consumer = {
02     &consumer_state,
03     consumer_init,
04     consumer_fire
05 };
```

Although the DOL programming model allows an arbitrary naming of the initialization and fire functions, this work demands that the corresponding function names can always be resolved by adding the suffixes `_init` and `_fire` to the name of the process handler. In turn, the name of the process handler must be equal to the base name of the implemented process.

While the base name of a process is specific to the implementation, the name of a process refers to a certain instance of the implementation. For example, the identifiers `consumer_1` and `consumer_2` refer to two specific instances of the implementation of the `consumer` process. Both instances get activated by calling `consumer_fire` in the proper task context.

4.2.4 DOL API

Without giving an introduction into the whole DOL API, Lst. 4.3 describes all functions that will appear in the following chapters. In particular, `DOL_read` (Lst. 4.3, line 2) and `DOL_write` (Lst. 4.3, line 4) form the API for communication, while `DOL_detach` (Lst. 4.3, line 6) belongs to the task management API.

Lst. 4.3: Extract of DOL API

```
01 // Read len bytes from port into buf
02 int DOL_read(void *port, void *buf, int len, DOLProcess *process);
03 // Write len bytes from buf to port
04 int DOL_write(void *port, void *buf, int len, DOLProcess *process);
05 // Detach process p
06 void DOL_detach (DOLProcess *p);
```

4.3 Functional Simulation

As an additional detail of the functional simulation (see 1.2.3), the information that is given by the output of the functional simulation is shortly described here.

Firstly, the resulting trace file contains a header section where all modeled communication channels are specified. This information is necessary for matching entries of the main section with process network channel names. In turn, the main section contains two entries for each firing of a process and one entry for each communication channel access. The number of transferred bytes is included in the latter case.

Fig. 15 and Fig. 16, respectively, show example extracts of a trace file of a functional simulation.

Fig. 15: Extract of functional simulation output. Header section

```
c C1 10 o generator 0xbf9247a8 i square_2 0xbf924ba8
c C2 10 o square_1 0xbf9249bc i buffersum_1 0xbf924dbc
c C3 10 o buffersum_1 0xbf924de4 i merger 0xbf9251e4
```

Fig. 16: Extract of functional simulation output. Main section

```
10 merger started.
11 merger r 0xbf9251e4 5000
12 generator w 0xbf9247a8 5000
13 square_1 w 0xbf9249bc 5000
14 generator stopped.
```

4.4 Hardware Dependent Software

The HdS provides an operating system kernel as well as architecture-specific C libraries for both core types that are used in the SHAPES platform. Furthermore, the HdS implements the DOL API that provides methods for task management and communication operations.

4.4.1 Scheduling

There are two operating system kernels that differ in the support for concurrently running tasks. While the OS kernel that is designed for the

ARM9 processor supports multiple tasks, it is only possible to run one application task on the mAgic processor.

A non-preemptive static scheduler is used for sharing the processing power of the ARM9 core. Generally, a task only returns from the processing state if a communication operation cannot be finished immediately. Thus, the scheduling dynamics are implicitly determined by the size of the used communication buffers.

4.4.2 Driver Modules

Communication buffers are implemented as device drivers, a device node is created for each buffer. Both available implementations store the data in a circular buffer.

The so-called generic software FIFO driver allocates the buffer memory at any free address on the heap during boot up. The corresponding device is only accessible on the same processor.

In turn, the D940 external FIFO driver allocates a channel buffer at a user-defined location on a shared memory resource. Both processors can concurrently access the buffer, mutual exclusive access is guaranteed by spinlocks.

Lastly, the HdS offers a third driver module that provides a debug output console.

4.5 Software Stack Generation

After providing the application, platform and mapping specifications as well as the sources of a DOL application, the software stack generation can be used for generating a runnable binary image for each processor.

4.5.1 Code Generation

Generally, the mapping-independent source tree of a DOL application contains only task-specific C source and header files. At the beginning of the software stack generation, a code generator is used for creating a mapping-specific main control routine. This generated routine acts as the application entry point, it is basically responsible for allocating communication channels as well as initializing application processes.

4.5.2 Compilation

Based on the GNU ARM-ELF toolchain and a mAgic toolchain from TARGET, scripts from the HdS distribution can be used for finally

generating the software stacks.

Chapter A.2 gives a list of pitfalls in the configuration of the software stack generation. These points need to be considered for generating usable software stacks.

5

Analysis Model Generation for SHAPES

The overall goal envisions to integrate the proposed concept in a design space exploration loop. In particular, MPA-RTC shall be used for evaluating design points that are chosen by a multi-objective optimization controller, namely EXPO [9]. As EXPO outputs DOL mapping specifications, an analysis model based on abstract SHAPES specifications has to be generated for each design point before MPA-RTC can be used.

At first, this chapter describes the implementation of a one-time calibration process for SHAPES. Afterwards, it is explained how SHAPES specifications can be converted into an instance of the abstract model (see Chapter 2).

5.1 Restrictions

Two restrictions apply to ensure the validity of the implementation that is presented here.

Firstly, the approach is only suited for applications whose process network is limited to synchronous data flow (see 5.7.4).

Furthermore, the size of each channel buffer must be a multiple of the fixed amount of data that is transferred during one access to the specific channel (see 5.6.4).

5.2 Calibration Procedure Overview

In general, the analysis model calibration for SHAPES is based on the approach that is described in Chapter 3.

For SHAPES, the results of functional simulation and instruction-accurate simulation are combined for extracting the desired performance parameters (see 3.2).

5.2.1 Required Input Data

Here, information about the application, the used platform and finally the results of an initially performed functional simulation have to be supplied.

Concerning the application, the application source codes as well as the SHAPES application specification (see 1.2.2) must be provided.

For information about the platform, the platform specification needs to be supplied. Both platform and application specifications must be given in their XML representation.

Lastly, the result of the functional simulation has to be provided explicitly in the form of the corresponding execution trace as well as implicitly in the form of an extra annotated application specification.

After providing the needed input data in the initial setup, the calibration process runs fully automated.

5.2.2 Functional and Instruction-Accurate Simulation

While a functional simulation is carried out only once per application and input data set, instruction-accurate simulation is used for evaluating several calibration mappings per application and input data set. The same input data set is used during the single run of the functional simulation run as well as all runs of the instruction-accurate simulation.

Contribution of Functional Simulation

The functional simulation is used for obtaining the number of transferred bytes per process network channel as well as the sequence of used communication channels. Latter is employed for matching entries of an execution trace to a channel context.

Contribution of Instruction-Accurate Simulation

Processing times are obtained by instruction-accurate simulation. Here, an extended execution trace with additional information about memory

and register contents is used for determining process and channel contexts.

5.2.3 Calibration Mapping Generation

As the evaluation of different resource sharing strategies is not subject of the calibration process, a set of calibration mappings covers the whole design space if all options in the spatial domain have been addressed.

Neglecting special mapping constraints, this means that each software channel needs to be mapped to any read path and any write path at least once. As communication paths are specific to the used processor, this implies that each process will also be mapped to any processor at least once.

5.2.4 Simulation Runtime

In general, signal processing applications are designed to run for an unlimited time as long as the hardware platform is switched on. In this context, the comparatively short system initialization phase can be neglected.

Here, a simulation terminates after a defined number of data samples has been processed. This number and the data samples itself need to be reasonably chosen for each application such that critical execution paths are covered with a high probability.

5.2.5 Program Path Sectioning

Prior knowledge about the HdS API, the DOL API as well as the DOL programming model is used for splitting an executed program path into different sections that can be assigned to categories.

Without reasoning about the chosen call depth at this point, Tab. 3 gives an overview about all functions that are used for splitting and categorizing the program path of each process.

5.2.6 Sequential Call Trees

A so-called sequential call tree can be generated by post-processing an extended execution trace of an instruction-accurate simulation. This kind of tree contains all information that are necessary for extracting processing times in the requested detail level.

In detail, a sequential call tree contains a node for each single call of a characteristic function (see Tab. 3).

The real execution sequence is preserved within each layer of the

Tab. 3: List of characteristic functions

Function Name	Category
DOL Programming Model	
`*_fire`	Computation
* refers to the base name of any involved process implementation	
DOL API	
`DOL_read`	Communication
`DOL_write`	Communication
HdS API	
`generic_swfifo_read`	Communication
`generic_swfifo_write`	Communication
`d940_extfifo_read`	Communication
`d940_extfifo_write`	Communication

hierarchical graph, it can be read from left to right. The hierarchy as well as edges between nodes describe caller and callee relationships.

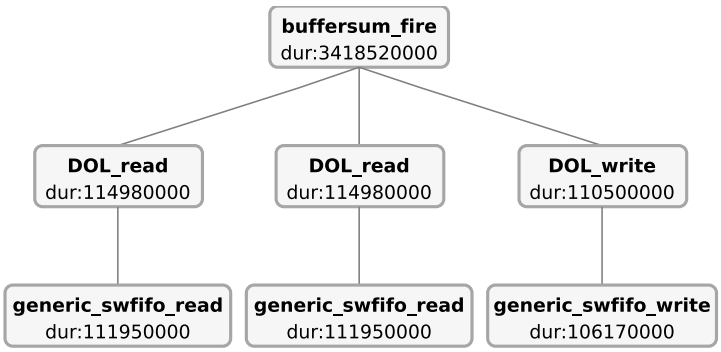


Fig. 17: Sub-tree of one firing. Two messages have been sequentially read, the result was written afterwards. The annotated numbers denote processing times in picoseconds. The processing time of any related node on a lower level is included in the processing time of a node on a higher level.

Fig. 17 shows a sub-tree of a sequential call tree.

The corresponding full tree contains a sub-tree of this kind for any firing of each process.

In the final result extraction, all sub-trees that correspond to one firing of a certain process instance are evaluated separately for finding the best-case and worst-case processing times of each process.

5.2.7 Tree Visualization

The shown call tree visualization is based on a tool that was developed by Sandro Blum within his semester thesis [4].

5.2.8 Workflow

In detail, the whole calibration workflow starts with the generation of calibration mappings (see 5.3).

Then, each calibration mapping is evaluated separately. This evaluation starts with the generation of the corresponding software stacks (see 4.5). Instruction-accurate simulation is employed for extracting execution traces (see 5.4). Based on the resulting traces, call trees are generated (see 5.5). Finally, the information of the created call trees as well as the output of the functional simulation are combined in the result annotation where the calibration results are added to the application specification (see 5.6).

The whole calibration procedure finishes after all calibration mappings have been evaluated.

5.3 Calibration Mapping Generation

Two patterns are sufficient for covering all currently available mapping options in the spatial domain.

5.3.1 Mapping all Processes to the ARM9 Core

Since the mAgic core is limited to only one running task, using generic software FIFO buffers is not reasonable here. So, only one calibration mapping is necessary for binding all software channels to the single read and write paths that target a generic software FIFO buffer.

All processes are implicitly bound to the ARM9 core in this calibration mapping. Obviously, this strategy is only feasible if the memory demand of the corresponding software stack is not exceeding the resources of the ARM9 core. Concerning that the ARM platform implements a Von Neumann architecture, the memory demand of the software stack refers to the memory requirements for storing both instructions and data.

Applications with a slightly exceeding memory demand could be analyzed by using multiple calibration mappings that swap one task to the mAgic core. However, the current limitations concerning the memory demand are rather caused by the availability of only one SHAPES tile instead of the available memory for the ARM9 core.

5.3.2 Mapping One Process to the mAgic Core

As both ARM9 and mAgic cores are able to access channel buffers that are implemented as a D940 external FIFO device, there exists a read path and a write path for each processor.

All needed bindings can be achieved by creating one dedicated calibration mapping for each task. In this mapping, the chosen task is bound to the mAgic core, while all other tasks are bound to the ARM9 core. All software channels are bound to communication paths that employ a D940 external FIFO buffer.

Again, restrictions on the memory demand of the generated software stacks apply.

5.4 Instruction-Accurate Simulation

The described calibration procedure requests a separate processing time information for each call of a characteristic function during a simulation. Before a sequential call graph with that information can be generated, instruction-accurate simulation has to be employed for generating execution traces.

In particular, the Virtual Platform Analyzer (see 1.2.4) is used for instruction-accurately simulating the Virtual Shapes Platform architecture.

5.4.1 Interfacing the VPA

A purpose-built Tcl script is used for interfacing the Virtual Platform Analyzer during the fully automated calibration procedure.

For running a simulation based on the Tcl script, a configuration file with certain details about the analyzed software stacks has to be supplied. The result of a simulation is given in the form of an execution trace. Here, tagged trace entries of all simulated processors are combined in one single trace.

Terminology and Scope

In general, both available cores of the VSP platform are simulated concurrently, but also independently. For simplification, all concepts that are introduced in the following sections are always regarding an arbitrary single core, if not stated otherwise. Similar, the term program counter refers to the content of the program counter register of an arbitrary single core.

The following sections give an overview about the basic features of the Tcl script.

5.4.1.1 Simulation Control

For using a certain processor, an image file that corresponds to a software stack for the specific core has to be stated in the configuration file.

A simulated core gets marked for termination after its program counter hitted a given address for a predefined number of times. For instance, it can be specified that tracing the ARM9 core should end after the symbol `DOL_detach` was hit for five times. As the termination flag refers only to an internal state of the script, toggling the flag is not directly affecting the simulation.

But, the whole simulation is terminated when all used cores are marked for termination.

5.4.1.2 Breakpoints and Breakpoint Ranges

Firstly, a new entry is appended to the execution trace when the program counter hits a single address that corresponds to a breakpoint.

Secondly, an entry is generated when the program counter enters a memory region that corresponds to a breakpoint range.

Breakpoint Range Entrance Detection

The support for setting up breakpoints and breakpoint ranges, respectively, is a native feature of the VPA. But, the VPA is intentionally pausing the simulation each time when the program counter reaches an address that lies within a breakpoint range.

Instead of inefficiently tracing any single step within a breakpoint range, a breakpoint range gets temporarily disabled on entrance. It gets not enabled again before another defined memory region that refers to a breakpoint range was hit.

5.4.1.3 Memory Observations

The current content of an address that is defined in a memory observation is added to all trace entries. This setting is also specific to a core.

5.4.1.4 Execution Trace Format

Overall, an execution trace entry contains the following information:

- Identifier of the breakpoint or breakpoint range, respectively

- Core Index
- Content of core program counter register
- Core clock timestamp in picoseconds
- Platform clock timestamp in picoseconds
- Content of all defined memory observations specific to the observed core
- Content of the first four general purpose registers specific of the corresponding core

A trace entry can be regarded as a system state snapshot at the time when the corresponding event occurred. Fig. 18 shows an example entry of an execution trace that was generated by the described Tcl script.

Fig. 18: Example execution trace entry

```
[LOG] TYPE:context NAME:main CORE_IDX:0 CORE_PC:538969820
    CORE_CLOCK:1585020000 VPA_CLOCK:1585020000 CORE_MEM:0x20303ae8
    CORE_REG:1,540161008,536870915,1
```

5.4.2 Function Call Tracing

Due to preemptive scheduling, an execution path that corresponds to a certain function might not be executed from start to end at once. Additionally, jumps to other function contexts can occur during the execution of a specific function.

The following strategy is used for accurately tracing all slices that refer to a call of a traced function.

Function Entrance

By setting a breakpoint on the address of the first instruction of a traced function, a trace entry is generated on each function entrance. Concerning recursive functions, an entrance breakpoint must always stay enabled during a simulation.

Function Re-Entrance

Function re-entrances can basically occur at any address of a function context, but not at the address of the first instruction. Though, function re-entrance is covered by setting a breakpoint range on the memory region that includes all but the first instruction of a traced function. As already stated, breakpoint ranges get dynamically enabled and disabled by the simulation control.

Function Exit

There is no explicit notification mechanism for detecting when a certain function context was exited. The only possible way is to trace the function that initially called the traced function. Here, the execution of the called function ended one instruction before the context of the calling function was re-entered.

Unfortunately, either all functions need to be traced, or knowledge about the involved execution paths must be available for implementing this functionality.

5.4.3 Traced Functions

Theoretically, only tracing all calls of fire functions, `DOL_read` and `DOL_write` should be sufficient for sectioning the program path of an application process (see 5.2.5).

However, the list of traced functions needs to be extended for accurately tracing the initially interesting set of functions.

Application Main Functions

The two functions `scheduler_schedule` and `__APP_ENTRY_POINT` refer to the application main function of a software stack that was generated for the ARM9 core and the mAgic core, respectively.

As a fire function is always called from the application main function, calls of these two functions must be traced for implicitly determining when the function context of a fire function was exited.

Low-Level IO Functions

On the ARM9 core, the execution of an application process is preempted if the specific process has to wait for a communication resource or data, respectively, to get available.

This preemption is implemented in the low-level IO functions `generic_swfifo_read`, `generic_swfifo_write`, `d940_extfifo_read`

and `d940_extfifo_write`. As the processing time of calls of these functions is included in the processing time of a `DOL_read` or `DOL_write` call, re-entrances into the corresponding contexts of the stated four functions must be traced.

5.4.4 Scheduler Context Detection

On the ARM9 core, concurrently running application task instances are calling the same set of functions, such as `DOL_read` and `DOL_write`. For assigning a specific function call to a task instance, the current scheduler context is included in each trace entry that refers to a function call on the ARM9 core.

In detail, the used scheduler implementation is maintaining an internal structure that contains an object for each available task. The HdS source code has been modified so that a pointer to the object that corresponds to the currently running task is always available in a global variable with the name `active_task`. A memory observation is used for tracking the value of the mentioned pointer.

5.4.5 Simulation Setup

While the names of all traced functions are known a priori, the corresponding memory regions must be obtained for setting up a simulation. This information is stated in the symbol tables of the generated software stacks.

As the introduced variable `active_task` is defined globally, the corresponding address can also be extracted from the symbol table.

Obtaining Symbol Tables

For the ARM9 core, the symbol table itself is obtained by the tool "objdump" of the employed GNU ARM-ELF toolchain. A symbol table that refers to the software stack for the mAgic DSP core is automatically generated by the toolchain from TARGET [40] during the software stack generation.

5.5 Call Tree Extraction

Based on an execution trace, a separate sequential call tree for each employed processor can be generated. In turn, an optional so-called compact call tree can be assembled by concentrating a sequential call tree.

5.5.1 Node Data Model

Each node of a sequential call tree refers to a specific function call of a traced function. In general, a node of a sequential call tree corresponds to several entries of the related execution trace.

A node of a sequential call tree contains the following information:

- Name of corresponding function
- Core index
- Program counter of first contributing trace entry, also called start address
- Value of scheduler context pointer
- Value of fourth general purpose register of the first contributing trace entry
- Number of contributing trace entries
- Function entrance timestamp
- Function exit timestamp
- Overall processing time
- Processing time of the first contributing trace entry
- Processing time of the last contributing trace entry

Firstly, this data is used for finding matches during the node extraction. Secondly, it will be used later for implementing a filter that removes waiting times from a given processing time (see 5.6.4). Lastly, the value of the general purpose register plays a role in the channel context detection (see 5.6.3).

5.5.2 Node Extraction

For extracting the call tree nodes, the given execution trace is sequentially read. While creating the first node is trivial, Algorithm 1 describes how each trace entry is matched to already extracted nodes.

If a matching node is found, the timing information of the found node is updated. Otherwise, a new node is created and added to the list of already extracted nodes.

Algorithm 1: Trace entry matching

```
   Input: List of already extracted nodes
   Input: Execution trace entry
 1 n ← number of already extracted nodes;
 2 matchFound ← false;
 3 for i ← n to 1 do
 4 |   Select node i for comparison with trace entry;
 5 |   if ¬ Function name matches then
 6 |   |   continue;
 7 |   end
 8 |   if ¬ Core matches then
 9 |   |   continue;
10 |   end
   |   // There is only one process
   |   // that runs on the mAgic core
11 |   if Core ≠ mAgic then
   |   |   // The scheduler has no process context
12 |   |   if Function name ≠ scheduler_schedule then
13 |   |   |   if ¬ Process context matches then
14 |   |   |   |   continue;
15 |   |   |   end
16 |   |   end
17 |   end
   |   // The start address of a node refers
   |   // to the address of the first instruction
   |   // of the corresponding function.
18 |   if Program counter of trace entry > Start address of node i then
19 |   |   matchFound ← true;
20 |   end
21 |   break;
22 end
23 if matchFound then
24 |   Update node i;
25 else
26 |   Create new node;
27 end
```

5.5.3 Calculating Processing Times

Apparently, an execution trace entry contains only a timestamp that refers to the entrance into a function context. Eq. 5.1 describes how the processing time of a trace entry is calculated.

In addition, Eq. 5.2 describes how function exit timestamps are calculated. Here, it is assumed that the previous function ended exactly one instruction before the function context was switched.

$$t_{processing} = timestamp_{next\ entry} - timestamp_{current\ entry} \tag{5.1}$$

$$t_{exit} = timestamp_{next\ entry} - \frac{1}{CPU_{clock\ speed}[Hz]} \tag{5.2}$$

5.5.4 Sequential Call Tree Ordering

The described algorithm has to be carried out separately for each processor. Thus, it is assumed that the used set of extracted nodes has been filtered by the core index beforehand.

Initially, a set of extracted nodes is ordered by the timestamp of the first contributing trace entry.

The tree generation starts with selecting the node that refers to the application main function. While this node gets marked as the root node of the new tree, Algorithm 2 is used for sequentially adding all following nodes to the tree. Apparently, the described algorithm is based on the known order of the extracted nodes.

5.5.5 Compact Call Tree Generation

In a so-called compact call tree, multiple nodes of a sequential call tree that refer to function calls of a specific function within the same process context are merged. While this representation is better for visually analyzing the profiling results, compact call trees play no further role in the described implementation.

Fig. 19 shows the corresponding compact call sub-tree of the sequential call sub-tree that was shown in Fig. 17. In opposite to a sequential call tree, a compact call tree includes the number of merged firings as well as basic statistics about the totaled processing times.

5.6 Result Annotation

The goal of the result annotation step is to extract processing times in the granularity that is stated in 3.4.2.

Following the idea of the already existing functional simulation, the resulting processing times are also back-annotated to the SHAPES application specification of the investigated application. Fig. 20 shows

Algorithm 2: Sequential call tree ordering

```
   Input: Root node of the partially assembled sequential call tree
   Input: New node n, that needs to be added
 1 current ← root node;
 2 while current has children nodes do
 3     foreach Child node c do
 4         Select c for comparison with n;
           // There is only one process
           // that runs on the mAgic core
 5         if Core ≠ mAgic then
               // The scheduler has no process context
 6             if Function name of current ≠ scheduler_schedule then
 7                 if ¬ Process context matches then
 8                     continue;
 9                 end
10             end
11         end
12         if Timestamp range of c overlaps timestamp range of n then
13             current ← c;
14             Add processing time of n to current;
15             break;
16         end
17     end
18 end
19 Add n as a new child of current;
```

an example XML application specification with annotated results of a functional simulation as well as results of a model calibration.

5.6.1 Basic Approach

In general, the needed results are extracted by sequentially analyzing each sub-tree of a sequential call tree that refers to a fire function. The maximum and the minimum of all values that are found by analyzing several firings of the same process instance correspond with the worst-case and best-case processing time, respectively.

Known Tree Structure

Given by the structure of the analyzed software stack, nodes that correspond to fire functions can only be found on the second most top level of the whole tree. Similar, nodes that refer to calls of `DOL_read` and

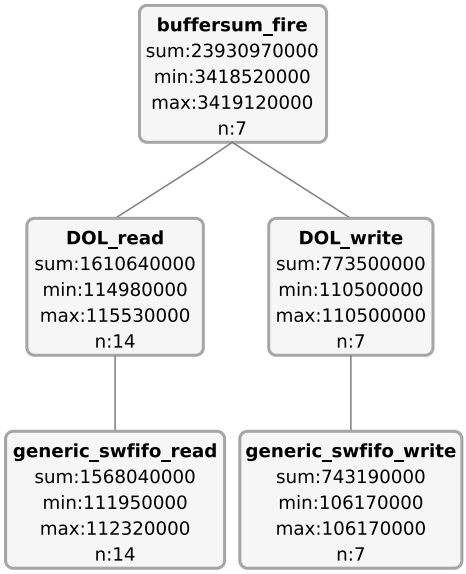


Fig. 19: Compact call tree of one process instance

Fig. 20: Example calibration result annotation

```
<process name="generator" basename="generator" range="">
  <port name="out" basename="out" range="" type="output"/>
  <source type="c" location="generator.c"/>
  <profiling name="NumOfFires" value="50"/>
  <profiling name="WCED arm" value="2832520000"/>
  <profiling name="BCED arm" value="2826370000"/>
  <profiling name="WCED arm out generic_swfifo" value="117630000"/>
  <profiling name="BCED arm out generic_swfifo" value="11490000"/>
  <profiling name="WCED arm out d940_extfifo" value="1091590000"/>
  <profiling name="BCED arm out d940_extfifo" value="758580000"/>
</process>
```

`DOL_write` are guaranteed to be located on the third most top level of the whole tree.

Processing Time without Communication

Tab. 3 shows that only fire functions refer to computation. Initially, the processing time of a call of a fire function also contains time that was spent for communication. This extra time can be removed by subtracting the contributions of all children nodes that refer to a call of `DOL_read` or `DOL_write`, respectively.

Processing Time for Communication

Each node that corresponds to a call of `DOL_read` or `DOL_write`, respectively, refers to a communication operation on a specific channel. Given by the properties of a sequential call tree, processing times that have been spent in function calls of low-level IO functions, such as `d940_extfifo_read`, are already included in the processing time of the parent node.

5.6.2 Process Instance Name Detection

As previously mentioned, nodes of a sequential call tree can be discriminated by the value of a pointer that refers to the scheduler context. While this is sufficient for generating a sequential call tree, the matching name of the underlying process instance is not known. But, this information is necessary for a correct back-annotation as well as the channel context detection that is described later.

While finding the matching name of a single task that runs on the mAgic core is trivial, a special strategy has to be used on the ARM9 core.

Function Call Parameters

Given by the specification of the used compiler, the first four general purpose registers of the ARM9 core are always filled with the first four function call parameters on function entrance.

By definition, the fourth parameter of `DOL_read` and `DOL_write`, respectively, is defined as a pointer to the local state data structure of the involved process instance (see Tab. 4.3). Thus, the memory location of the local state data structure of the corresponding process instance is given by the content of the fourth general purpose register of the ARM9 core on function entrance.

Given by the code generator that is used during the software stack generation, two properties can be named for the local state data structure. Firstly, the variable name matches with the name of the related process instance. Secondly, the variable is defined globally. Concluding, a mapping between the memory location of a local state data structure and the name of the corresponding process instance can be extracted from the symbol table of the generated software stack.

5.6.3 Channel Context Detection

Currently, it is not possible to assign call tree nodes that correspond to communication functions to abstract process network channels by only employing instruction-accurate simulation. Although the first parameter

of `DOL_read` and `DOL_write`, respectively, contains a reference to the used communication port variable, this information is not sufficient for the given purpose. Firstly, the referenced variables have only a local scope. Additionally, the variable names are not necessarily related to the names of the accessed process network channels.

Information of Functional Simulation Output

The output of the functional simulation is used for matching call tree nodes that refer to `DOL_read` and `DOL_write`, respectively, to a certain process network channel.

Given the name of a certain process instance, the channel usage sequence specific to the corresponding process instance can be extracted from the trace file of the functional simulation (see 4.3). Here, the names of the accessed process network channels are included.

Matching Both Extracted Sequences

The extracted sequence matches with the sequence of nodes in the corresponding sequential call tree if two conditions hold. Firstly, the sequence can only be matched specific to a process instance, but not on application-level. This is due to the totally different task activation behavior of the functional simulation and the scheduler that is simulated in the instruction-accurate simulation. Secondly, either the same input data has to be used for driving both kinds of simulations, or communication must not be conditional.

In the context of this work, conditional communication is already denied by the synchronous data flow constraint that applies to the overall model generation approach.

5.6.4 Filtering Waiting Times

As stated in Chapter 3, extracted processing times must not contain waiting times that occur due to resource sharing.

While filtering waiting times on the mAgic core is currently not possible due to missing knowledge about the involved DMA controller, there exists an approach for the ARM9 core.

Low-Level IO Function Analysis

Listing 3 outlines the basic algorithm that is used for writing data into a D940 FIFO buffer. The general concept of the shown algorithm is descriptive for both read and write operations of both available buffer implementations.

The multi-task operation system kernel that runs on the ARM9 core changes the process state from active to waiting in two cases. First, if a communication buffer is totally full or completely empty, respectively (Lst. 3, line 23). Secondly, if a communication operation cannot be finished because the used buffer or data, respectively, is only partially available (Lst. 3, line 18).

Intentionally, the processing time of a write operation should only contain the time that was spent for executing the parts of the code that are enclosed between label A and label C once (see Lst. 3).

But, if for instance the buffer was not available for writing in four sequential tries, the measured processing time includes the execution of the sequence A→B→E→B→E→B→E→B→E→B→C.

The proposed filter is now used for shortening this long sequence to A→B→E→B→C. Having in mind that the overall processing time of a function call may consist of several slices due to function context switches during execution, the desired sequence refers to the sum of the first and the last slice that was used to calculate the overall processing time.

Because this strategy would cut off parts that were really spent for exchanging data if partial data transfers (Lst. 3, line 13) are allowed, this case is no longer supported. Concretely, this case is prevented by the requirement that the size of a channel buffer must always be a multiple of the fixed amount of data that is transferred per operation specific to the considered channel.

More Optimistic Estimation

By only accounting the last contribution of the processing time that was spent in `d940_extfifo_write`, the measured sequence could again be shortened to E→B→C. But, this is not used because the measured time would be more optimistic than the measurement of the desired case without any iteration due to not available data. Currently, the validity of such an optimization has not been measured.

Additional Solution for Single Processor Mappings

The best results concerning waiting times can be achieved by using communication channel buffers that are sufficiently large for storing all messages that a certain task produces during a simulation. Also using a suited scheduling policy, each task can perform all communication operations at once during a single activation. Here, free buffer space and data are always immediately available.

But, this approach is only working if only the ARM9 core is used, it can no longer be controlled when two processors are concurrently

Algorithm 3: d940_extfifo_write

```
Input: Number of bytes that have to be written
Output: Number of written bytes
Label: A
1  nr ← number of bytes that have to be written;
2  Initialization of other variables;
3  while nr > 0 do
       Label: B
4      Acquire buffer;
5      available ← number of bytes that are currently available on the
       buffer for writing;
6      if available > 0 then
7          if nr ≤ available then
8              Write nr bytes;
9              nr ← 0;
10             Update buffer status;
11             Release buffer;
               Label: C
12         end
13         else if nr > available then
14             Write available bytes;
15             nr ← nr − available;
16             Update buffer status;
17             Release buffer;
               // Only on multi-tasking OS kernel
18             Suspend task, activate another task;
               Label: D
19         end
20     end
21     else
22         Release buffer;
           // Only on multi-tasking OS kernel
23         Suspend task, activate another task;
           Label: E
24     end
25 end
26 return nr
```

producing and consuming data. Additionally, it is limited to feed forward communication.

While the approach could still be used for simple configurations, it has been fully discarded for preserving the same buffer configuration for

the evaluation of both single and dual core mappings. Furthermore, this method is also limited by the size of the available memory resources that are rather too small.

5.7 Analysis Model Generation

After the one-time model calibration has been performed for a given application, an analysis model can be generated for arbitrary design points. In the desired tool flow, the configuration of a design point is given in the form of an application specification, a platform specification and a mapping specification.

While many parts of the SHAPES specifications and the abstract model are quite similar, a list of special details has to be considered for generating an instance of the abstract model that corresponds to a chosen design point.

5.7.1 Adding Event Sources and Sinks

The SHAPES application specification does not explicitly include event sources and sinks in the process network description. In the analysis model generation, an event source is added to any process that has no input channels. In turn, an event sink is added to any process that has no output channels.

It is assumed that all parameters that are needed for modeling the specific event sources are given, i. e. as annotated values in the application specification.

5.7.2 Modeled Resources

All resources for which the SHAPES mapping specification contains a schedule are also modeled in an instance of the abstract model. In general, this includes processors and communication bus resources.

Basically, the resource sharing configuration can be directly converted between both representations. As one exception, a static schedule of a mapping specification is modeled as a FIFO scheduler in an instance of the abstract model. This refers to the best matching replacement as static scheduling cannot be modeled explicitly in the abstract model.

5.7.3 Modeling Software Channels

In the application specification, a connection between two processes is implemented by a software channel. In the generated instance of the

abstract model, a software channel is modeled as a chain of tasks that represent the amount of transferred data per firing on a specific channel. The amount of transferred data per channel is extracted from the output of the functional simulation.

An extra task is added to the schedule of any communication bus resource that is referenced on the write path to which a software channel is bound. Similar, an extra task is added to the schedule of each communication bus resource that is referenced on the read path to which a specific software channel is bound. The list of involved communication bus resources is given by the platform specification.

5.7.4 Calculating Connection Scalings

For the validity of the method that is presented in this section, an analyzed process network must conform to synchronous data flow.

The activation scheme of a task in MPA-RTC is given by the corresponding incoming arrival curve. For instance, if a following task should only be activated half the amount of its predecessor task, the outgoing arrival curve of the predecessor task has to be scaled by the factor 0.5 before it is fed to the destination task.

Eq. 5.3 describes how this scaling factor is calculated for each process network connection during the generation of an instance of the abstract model. The needed information about the number of firings is obtained from the output of the functional simulation.

$$scaling = \frac{Number\ of\ firings_{destination}}{Number\ of\ firings_{source}} \tag{5.3}$$

Obviously, the number of executed read and write operations of a specific task is related to its number of activations. Thus, the calculated scaling factor is applied to the specific connection of the extended process network that refers to the interface between write path and read path of the modeled software channel. In other words, this ensures that the same amount of data that is written to a channel buffer gets also read in the generated analysis model.

5.8 Implementation Environment

In general, all described concepts have been implemented by using Java 1.5 and Apache Ant [1]. Apache Ant is especially used for controlling the overall tool flow.

6

Case Study

This chapter presents a case study in which the introduced concept is applied. Here, the focus lies on the performance of the presented approach as well as the performance of MPA-RTC.

The verification part of this chapter aims to proof that the final results that are obtained with MPA-RTC are not conflicting with the real behavior of the system.

6.1 Case Study Application

An eligible case study application has to provide mainly four attributes. Firstly, the application must be parallelizable. Next, the application should reasonably utilize the platform. Thirdly, the application should take an advantage of the heterogeneous architecture. Lastly, the application must conform to the restrictions of the workflow implementation, most notably the synchronous data flow constraint.

Here, a SHAPES application that performs floating-point operations on vectors is used. Eq. 6.1 describes how a set of four following vectors is summarized to one scalar value.

$$\vec{v_i} \in \mathbb{R}^n, i \in \mathbb{N}_{\geq 1}, l \in \mathbb{N}_{\geq 1} : r_l = \sum_{i=(4\cdot l)-3}^{4\cdot l} \sum_{k=1}^{n} v_{i,k}^2 \tag{6.1}$$

The data flow graph (see Fig. 21) as well as the process network (see Fig. 22) show how this task can be parallelized. An equal load

balancing between both paths is achieved by using two paths for the main part of the calculations.

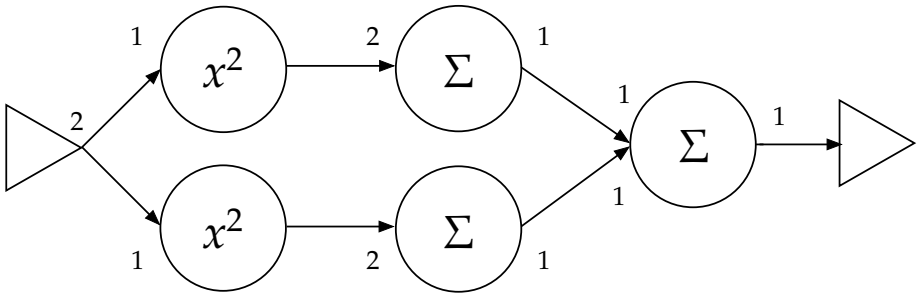


Fig. 21: Data flow graph of the case study application. Numbers adjacent to each input and output node indicate the number of tokens that are consumed or produced when the node fires.

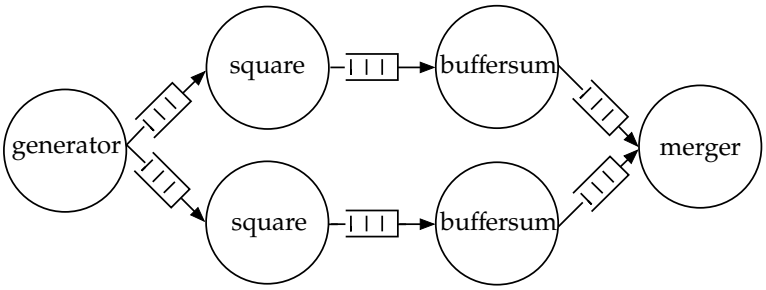


Fig. 22: Process network of the case study application. In the abstract view, each channel is equipped with an unbounded FIFO buffer. In the used implementation, each channel buffer is limited to the fixed size of one message.

6.1.1 Application Characteristics

All tasks are exchanging messages, or tokens, of the same length. Concretely, each message contains a `float` value for each row of the represented vector with a length of n. On a communication channel, a `float` always corresponds to 4 bytes. Tab. 4 describes the resulting communications characteristics.

The number of floating-point operations per firing of a certain task is shown in Tab. 5. All operations of the operating system are only working with integers.

Lastly, Tab. 6 shows the number of firings per task for processing i vectors that result in $i/4$ scalar values.

Tab. 4: Communication characteristics of one firing

Task	Read bytes	Written bytes
generator	0	$2 \cdot n \cdot 4$
square	$n \cdot 4$	$n \cdot 4$
buffersum	$2 \cdot n \cdot 4$	$n \cdot 4$
merger	$2 \cdot n \cdot 4$	0
n: Static vector length		

Tab. 5: Computation characteristics of one firing

Task	FP multiplications	FP additions
generator	$2 \cdot n$	0
square	n	0
buffersum	0	n
merger	0	$2 \cdot n$
n: Static vector length		

Tab. 6: Process firing scheme

Task	Firings per instance	Total firings
generator	$i/2$	$i/2$
square	$i/2$	i
buffersum	$i/4$	$i/2$
merger	$i/4$	$i/4$
i: Number of samples $i \mod 4 = 0$		

Tab. 7: Calibration mapping configuration

Task	Configuration						
	Generic SW FIFO	1	D940 Ext. FIFO	2	3	4	5	6
generator		ARM		mAgic	ARM	ARM	ARM	ARM
square_1		ARM		ARM	mAgic	ARM	ARM	ARM
square_2		ARM		ARM	ARM	mAgic	ARM	ARM
buffersum_1		ARM		ARM	ARM	ARM	mAgic	ARM
buffersum_2		ARM		ARM	ARM	ARM	ARM	mAgic
merger		ARM		ARM	ARM	ARM	ARM	ARM

6.1.2 Case Study Configuration

During the case study, the generator is used to produce $i = 100$ vectors with a length of $n = 1250$ values per vector.

While a smaller number of vectors would obviously be sufficient for evaluating this simple application without conditional paths, the given parameters were chosen for obtaining results that are more descriptive concerning the validity of the later presented results in the long run.

6.2 Calibration Mappings

Due to a confirmed bug in the current version of the HdS, it is not allowed to bind the final task of a process network to the mAgic core. Concretely, the operating system kernel that runs on the ARM9 core releases shared resources before the execution of the final task that is bound to the mAgic core has finished.

Corresponding to Section 5.3, Tab. 7 shows the resulting calibration mapping configuration for the case study application.

6.3 Simulation Performance

Referring to the VPA documentation, there exist two simulation modes. First, the so-called speed-optimized simulation mode offers a faster execution by reducing the result accuracy. In detail, memory accesses are done through so-called backdoors. Then, memory accesses by a core consume no simulation cycles.

In turn, all memory accesses go over the simulated bus in the so-called full simulation mode. As the desirable higher accuracy is not significantly decreasing the performance, the case study results are based on the full simulation mode.

Tab. 8: Simulation overhead due to profiling

Scenario	**Number of interruptions**	**Wall-clock duration**
Simulated processing time: 458.706.770 ns		
No breakpoints	0	0 min 32 sec
Calibration setup	1.731	5 min 2 sec
Single-Stepping	45.855.488	8 hrs 25 min
Simulation host: AMD Athlon XP 2800+, 2 GHz, 1 GB RAM, RHEL 5.1		

Beside of the chosen simulation mode, the simulation runtime is dependent on the number of interruptions due to the set breakpoints. Tab. 8 shows the impact of breakpoints on the simulation runtime. Obviously, it pays off to only trace functions that are really necessary for the desired purpose.

6.4 Calibration Toolflow Performance

Table 9 shows the performance of the calibration tool flow for the given case study configuration. Here, the given durations refer to the overall time that was spent for performing the given task during the analysis model calibration.

The overall performance of the software stack generation is mainly affected by the bad performance of the used compiler from TARGET for generating the software stack for the mAgic DSP. While the compilation of one software stack for the ARM9 core takes around two seconds, about one minute is necessary for generating one software stack for the mAgic DSP. In both cases, the operating system cores were already available as pre-compiled objects.

While the time for setting up the simulations is mainly based on the number of calibration mappings, the runtime of the simulations itself is apparently depending on the processing demand of the executed application as well as the number of processed samples.

The result extraction scales for an increasing size of an execution trace, the optional tree visualization should not be used in the context of simulations with many firings.

Tab. 9: Calibration toolflow performance

Task	Frequency	Totaled wall-clock duration
Functional simulation setup	1	0 min 18 sec
Functional simulation	1	0 min 1 sec
Calibration mapping generation	1	0 min 1 sec
Software stack generation	6	6 min 50 sec
Simulation setup	6	0 min 12 sec
Instruction-accurate simulation	6	66 min 29 sec
Result extraction	6	0 min 26 sec
Optional tree visualization	6	12 min 56 sec
Simulation host: AMD Athlon XP 2800+, 2 GHz, 1 GB RAM, RHEL 5.1		

6.5 Comparing Real Event Streams with Calculated Arrival Curves

One goal of the case study is to show that the real event streams lie within the calculated upper and lower bounds of the arrival curves in MPA-RTC.

After extracting a trace of a real event stream, a cumulative function that refers to the given event stream is calculated. Concretely, the calculated function $R(t)$ denotes how many events occurred in any time interval $[0; t]$. In this representation, an event stream can be directly plotted into the coordinate system of a calculated arrival curve.

6.5.1 Event Stream Traces

An event stream trace is specific to a process network channel, or software channel, respectively.

Fig. 23 shows an example event stream trace. Each line corresponds to one particular event. While the first number of each row denotes when the corresponding message was written into the FIFO buffer, the second number specifies when the same message was read out of the buffer. All numbers refer to platform clock timestamps in picoseconds.

Fig. 23: Example event stream trace

```
2936730000;3688070000
13552140000;14300550000
30937960000;31687600000
41539810000;42288200000
```

The extraction of an event stream trace is based on the analysis of the

sequential call trees (see 5.2.6) of all involved cores. Here, the sequence property of a sequential call tree is used. First, it is exploited for matching the tree with the channel sequence from a functional simulation for assigning channel contexts. Secondly, the sequence property is also used for matching corresponding read and write timestamps that might be part of two different sequential call trees.

6.5.2 Conjunction and Disjunction of Multiple Event Stream Traces

In MPA-RTC, multiple event streams can be merged by activation units that are either performing a conjunctive or disjunctive combination. For comparisons with the outgoing event stream of an activation unit in MPA-RTC, event stream traces can also be conjunctively and disjunctively combined.

Firstly, the cumulative function of all involved event streams must be given. Next, both cumulative functions must be aligned to the same origin in the discrete time domain. Then, the conjunctive and disjunctive combination of two event stream traces that are represented by $R(t)$ and $S(t)$ is given by pointwise calculating $min(R(t), S(t))$ and $R(t) + S(t)$, respectively, for all discrete values t.

6.5.3 Merging Event Stream Traces

For the completeness of the concept without further application, it is to mention that event stream traces can also be merged. This might be interesting in cases where a task is conditionally accessing different process network channels.

Concretely, the event stream trace that refers to the overall output or input event stream, respectively, can be computed by combining all relevant single event stream traces. Here, all entries must be merged and ordered by the first or second column, respectively. The relationship between the first and second column of each entry has to be retained.

6.6 Modeling Event Sources

In embedded systems design, it is normally asked if a certain configuration is able to cope with an event source that has certain properties. Thus, the properties of the event source are known beforehand from the specification.

Here, the event source is included in the application. The next sample is basically generated after the processing of the previous sample finished.

For the verification purpose of this case study, the parameters of the assumed PJD event source have to be reconstructed from the event stream traces.

6.6.1 Parameter Extraction

Concretely, an event is regarded as been generated at the time when the corresponding message was written by the `generator` task. So, the parameter extraction is based on the vector $\vec{t} \in \mathbb{R}^n$ of all n timestamps that correspond to a message that was written by the `generator` task.

For $n \in \mathbb{N}_{\geq 1}$, the equations 6.2, 6.3 and 6.4 describe how the three necessary parameters for modeling a PJD event source are computed.

$$p = \frac{t_n - t_1}{n - 1} \tag{6.2}$$

$$k \in \mathbb{N}_{1 \leq k \leq n} : j = \max |t_1 + \{p \cdot (k-1)\} - t_k| \tag{6.3}$$

$$i \in \mathbb{N}_{2 \leq i \leq n} : d = \min\{t_i - t_{i-1}\} \tag{6.4}$$

6.7 Model Verification

The overall verification procedure on which this case study is based starts with performing a one-time analysis model calibration.

For each investigated design point, the further procedure starts with generating a corresponding instance of the abstract model (see Chapter 2 and 5.7). Then, the related event traces are extracted and used for estimating the parameters of the corresponding PJD event source. Next, a Matlab script for MPA-RTC is written based on the given instance of the abstract model. Finally, Matlab is used for the verification of the generated analysis model.

6.7.1 Extending the Period of the Implicitly Modeled Event Source

While the case study application is running on the platform, all processing resources are fully utilized. As the sample generation is part of the application itself, the implicitly modeled event source refers to the most demanding event source that can be served with the given resources.

Although the number of activations is rather low in this case study, there is already a difference between the measured worst-case and best-case execution demands. While the average-case demand is already fully

utilizing the processing resources, a more pessimistic worst-case analysis leads to an unfeasible schedule.

To overcome this problem, the period of the implicitly modeled event source is extended. This is done by adding an idle function to the `generator` task. In this function, processing time is spent for performing a number of floating-point calculations. For the calibration result, the time that was spent in the idle function is removed.

Thus, by extending the period of the implicitly modeled event source, extra computation power for handling worst-case behavior is made available.

As generally any embedded system must be dimensioned for higher demands than the average-case, this strategy is not affecting any property of the presented approach. Even more, it is rather seldom that data is purely generated by an application without interacting with other components such as communication controllers.

6.7.2 Using Continuous Event Streams in MPA-RTC

Initially, event streams refer to discrete functions in MPA-RTC. The current implementation of the toolbox models an event stream as a list of curve segments. Basically, each curve segment has to be treated separately when calculations are performed.

While performing calculations, initially well-formed event streams get distorted very fast. The problem is that the number of curve segments grows tremendously fast when an event stream gets more complex. Not only leading to a high memory consumption, running mathematical operations on data sets of this size is not feasible.

To overcome this limitation of the current toolbox implementation, the results of this case study are based on continuous event streams. While the concrete impact on the quality of the results is currently not known, the calculated estimations are definitely more pessimistic compared to an estimation based on discrete event streams.

6.7.3 Results of Real Measurements and Estimation with MPA-RTC

For the model verification with MPA-RTC, mapping configuration 1 and mapping configuration 3 (see Tab. 7) have been chosen. While mapping configuration 1 refers to the most simple configuration of the given case study application, mapping configuration 3 has been chosen as the representative of all configurations that employ both processing cores.

6.7.3.1 Analysis Performance

The generation of the corresponding Matlab script from the SHAPES specifications of a certain mapping configuration takes less than one second on a machine that is equipped with an Intel Core Duo processor.

Analyzing mapping configuration 1 with MPA-RTC took 8.3 seconds on the same machine, 61 iterations were necessary until a fixed point was reached. Similar, 14.0 seconds were spent for analyzing mapping configuration 3, 79 iterations were necessary until a fixed point was reached.

6.7.3.2 Corresponding MPA-RTC Models

Fig. 24 and Fig. 25 show the corresponding MPA-RTC models for both analyzed configurations. FIFO schedulers are used for both modeling the static scheduler of the ARM9 core as well as the resource sharing of the communication bus that is employed in mapping configuration 3.

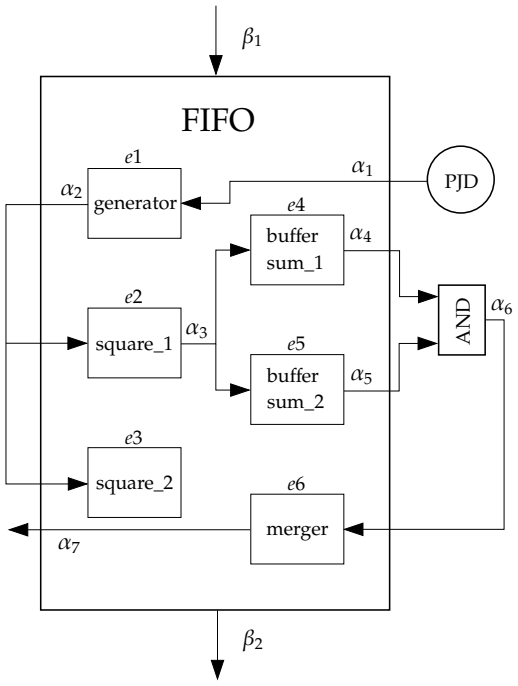


Fig. 24: MPA-RTC model for mapping configuration 1

6.7.3.3 Execution Demands from Calibration

Obtained execution demands per process and firing are stated in Tab. 10. Tab. 11 compares the execution demands without communication for both processing cores.

Tab. 10: Overall execution demands per firing for computation and communication

Task	Worst-case demand	Best-case demand
Mapping configuration 1		
generator	3.1 ms	3.0 ms
square_1	1.6 ms	1.5 ms
square_2	1.6 ms	1.6 ms
buffersum_1	3.4 ms	3.4 ms
buffersum_2	3.4 ms	3.4 ms
merger	2.8 ms	2.7 ms
Mapping configuration 3		
generator	5.0 ms	4.3 ms
square_1	24.9 ms	18.6 ms
square_2	3.6 ms	2.9 ms
buffersum_1	5.4 ms	5.3 ms
buffersum_2	5.7 ms	5.3 ms
merger	5.5 ms	4.0 ms

Tab. 11: Worst-case execution demands per processor and firing without communication

Task	ARM9	mAgic
generator	2.8 ms	0.3 ms
square_1	1.4 ms	0.3 ms
square_2	1.4 ms	0.3 ms
buffersum_1	3.0 ms	0.9 ms
buffersum_2	3.0 ms	0.9 ms
merger	2.6 ms	—

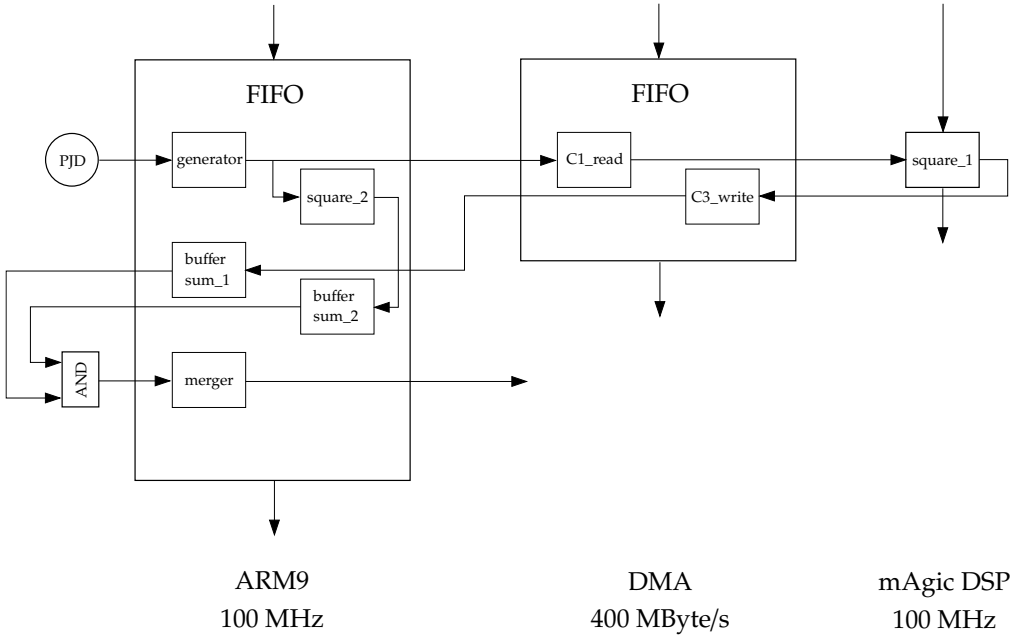


Fig. 25: MPA-RTC model for mapping configuration 3

Tab. 12: Extracted PJD event source parameters

Parameter	Mapping config. 1	Mapping config. 3
Period	18.1 ms	22.1 ms
Jitter	6.7 ms	13.8 ms
Minimum distance	12.8 ms	11.7 ms

6.7.3.4 Parameters of Extracted Event Sources

Tab. 12 shows the event source parameters that result from analyzing the event traces as described in 6.6.

6.7.3.5 Resulting Plots

Fig. 26 and Fig. 27 visualize the calculated upper and lower bounds in comparison with the real event traces. While only the plots of the first and the last task in terms of the data flow are shown here, the results of all process instances are displayed in A.1.

6.7.3.6 Further Calculated Characteristics

Tab. 13 shows calculated system-level characteristics that have been obtained from MPA-RTC. The measured values and the worst-case estimates for the end-to-end delay between `generator` and `merger` are shown in Tab. 14

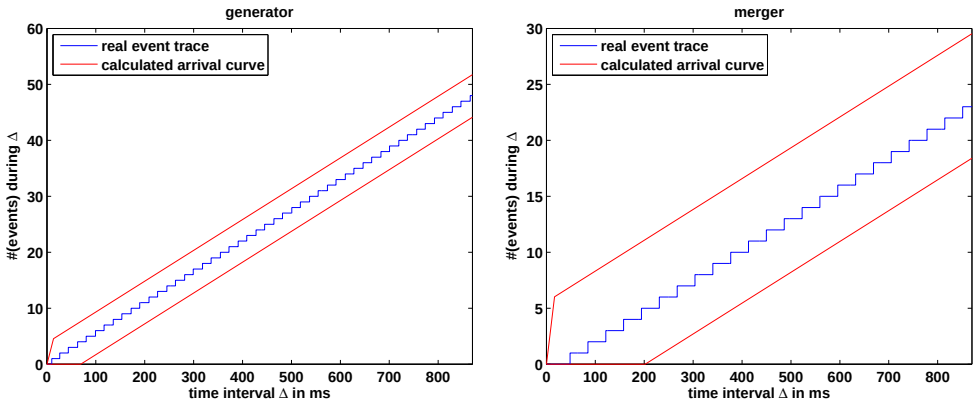


Fig. 26: Comparison of calculated upper and lower bounds with the real event traces. Mapping configuration 1.

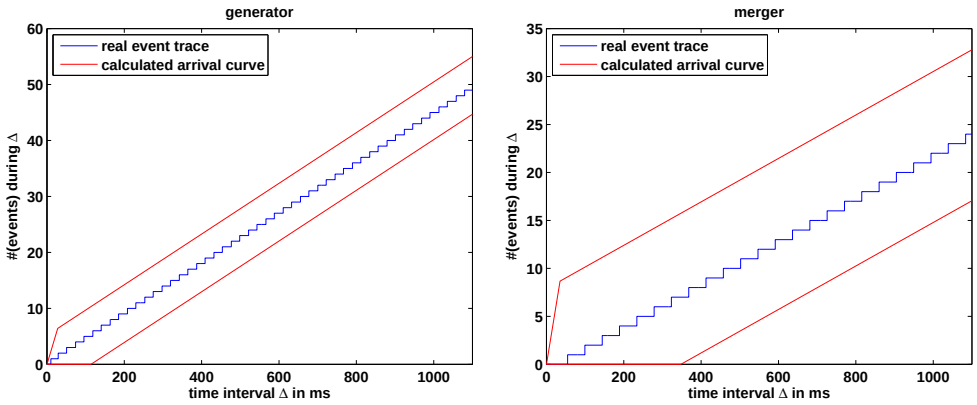


Fig. 27: Comparison of calculated upper and lower bounds with the real event traces. Mapping configuration 3.

Tab. 13: Calculated performance characteristics

Task	Worst-case buffer size	Worst-Case delay
Mapping configuration 1		
generator	3.8	44.3 ms
square_1	6.2	44.3 ms
square_2	6.2	44.3 ms
buffersum_1	4.3	44.3 ms
buffersum_2	4.3	44.3 ms
merger	5.6	44.3 ms
Mapping configuration 3		
generator	5.2	77.9 ms
square_1	4.9	87.7 ms
square_2	8.7	77.9 ms
buffersum_1	2.4	77.9 ms
buffersum_2	6.1	77.9 ms
merger	7.9	77.9 ms

Tab. 14: Calculated and real end-to-end delay between `generator` and `merger`

Source	Mapping config. 1	Mapping config. 3
Real measurement	55.2 ms	69.4 ms
Calculated worst-case delay	492.3 ms	782.5 ms
Additive model	22.2 ms	49.8 ms

6.7.4 Result Evaluation

At first, all extracted event traces from the simulation lie within the calculated upper and lower bounds. Even more, the slopes of real and estimated curves fit, which implies the validity of the results in the long run.

6.7.4.1 Quality of Estimated End-to-End Delays

Apparently, the calculated end-to-end delays (see Tab. 14) are very pessimistic. While the reasoning for this effect has not been totally discovered, several suspicions exist.

Firstly, the obtained results are based on a fixed point convergence. Currently, there exists no information about the quality of results that are calculated in this manner.

Secondly, the impact of using continuous event streams in MPA-RTC has also not been investigated, yet.

While the delays that are shown here correspond to the sum of task-specific standalone delays, better results could be obtained by using "pay bursts only once" [21]. This could especially help as the values that have been extracted to model the PJD event sources contain a large jitter.

Lastly, the highest contribution to the end-to-end delays is given by the worst-case delay of the conjunctive activation of the `merger` task. Namely, the worst-case delay of this activation contributes 315.3 ms and 461.2 ms, respectively, which corresponds to more than the half of the overall values. While these calculated delays are also caused by the modeled bursts, a detailed reasoning for this behavior is not available, yet.

6.7.4.2 Exploitation of the Second Processing Core

Although parallelization normally should lead to a speedup of the application, Tab. 14 shows that the opposite occurred. The same trend can already be observed in Tab. 10 by considering that all execution demands are higher for mapping configuration 3.

The reason for this behavior can be found in the two different buffer implementations. The generic software buffer implementation uses a local event notification system for the arbitration of communication buffers between concurrently running tasks. In turn, the implementation of the Diopsis 940 external FIFO buffer is based on a spinlock that is used for controlling the concurrent buffer access of multiple tasks that might run on different processing cores.

By definition, a spinlock is acquired by busy waiting until it gets available. As communication buffers are always located on the distributed external memory independent of the used buffer implementation,

the buffer access synchronization is to blame for the currently low performance.

6.7.4.3 Busy Waiting for DMA Channel Access

Intentionally, a request for reading or writing a channel buffer should only be sent after the needed data or free buffer space, respectively, are available. In practice, busy waiting for not produced data seems to take place in the form of busy waiting for a DMA channel to get available.

Fig. 28 shows the distribution of processing times that have been spent on waiting for the DMA controller while trying to read or write data, respectively. Here, both accesses for querying buffer status information as well as for transferring data are included. In a more detailed analysis, it can be observed that reading a buffer status is always reasonably fast, while reading data from a buffer is delayed up to 20 ms.

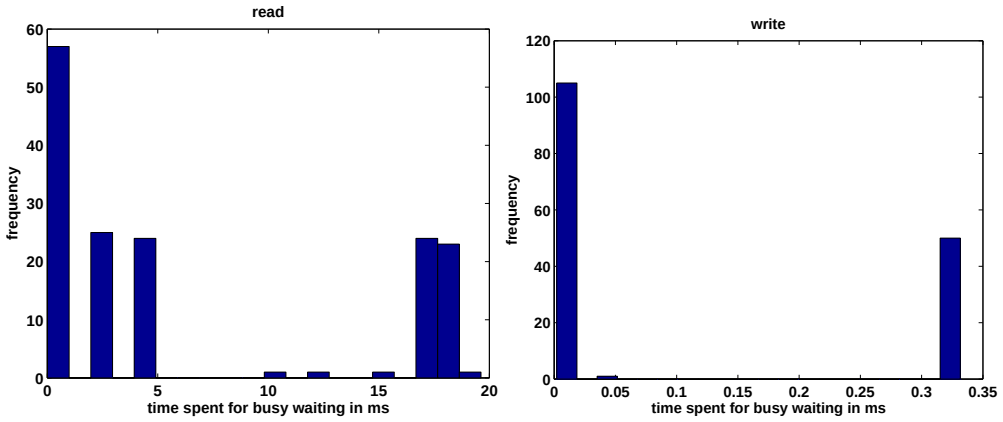


Fig. 28: Distribution of waiting times for DMA read and write access

The problem here is, that waiting times due to resource sharing can no longer be filtered which leads to distorted calibration results. For instance, the measured excution demand of `square_1` when bound to the mAgic core is much too high (see Tab. 10).

Here, one possible solution could be to replace these distorted processing times by static values that are obtained from a benchmark measurement.

For the currently shown results, the execution demand of `square_1` in mapping configuration 3 has been manually decreased to 18 ms in the MPA-RTC model. This value is pretty close to the boundary where the systems get no longer schedulable in a worst-case sense. Nevertheless, 18 ms is still a too high estimate for the processing time as it still contains waiting times that occur due to the mentioned problem.

6.7.4.4 Theoretical Usage of Heterogenous Processing Cores

As the presented case study application is performing a large number of floating-point operations, the mAgic DSP core is outperforming the ARM9 core only regarding the processing time spent for computation without communication (see Tab. 11). Here, the reason can be found in the missing native floating-point support of the ARM9 core.

While basically all parts of the case study application are tailored for the mAgic DSP core, Tab. 10 gives a hint on the advantage of having specialized processing resources for purposes such as massive floating-point calculations.

7

Conclusions and Outlook

In Chapter 2, a new abstract specification language for modeling complex embedded systems has been introduced. Here, the most important innovation can be found in the presented MPA-RTC writer that is capable to syntactically resolve all kinds of cyclic dependencies that can occur during the analysis of embedded systems.

Chapter 3 presented a new method for the calibration of accurate performance analysis models. Here, the accuracy of the results can be given in three dimensions. Firstly, the approach is based on instruction-accurate simulation. Next, the used performance data is extracted from the simulation of the real application. Lastly, the calibration procedure offers a high granularity on the level how different contributions to the processing time of a task are reported separately.

As seen in Chapter 5, the introduced approach can be executed fully automatically. Basically, all prerequisites for an integration of the presented concept into a full design space exploration tool flow are already satisfied.
Finally, a proof-of-concept was presented in the case study in Chapter 6.

Outlook

Regarding a short time interval, the applicability of the presented concept to more complex applications needs to be verified. Related to this topic, performing verification measurements on the real hardware is also thinkable as soon as the hardware gets available. Embedded in

the course of the SHAPES project itself, it is also desirable to detect and repair issues that are currently affecting the performance of the whole system.

Furthermore, it needs to be checked if the instruction-accurate simulation can be extended to cover the monitoring of communication bus activities. Initially, this is dependent on the availability of a software tool that is suited for this purpose.

Pointing to another direction, the full integration of the shown implementation into a design space exploration loop in connection with EXPO is considerable. Here, minor improvements in the tool flow such as a model for describing mapping constraints may help to additionally enhance the presented tool flow.

In the long run, a new challenge will come up when the number of hardware tiles is increased. Here, an interesting question will be if results that were obtained by analyzing small process networks can be transferred to more parallelized process networks that are based on the same task implementations.

A

Appendix

A.1 Model Verification Result Plots

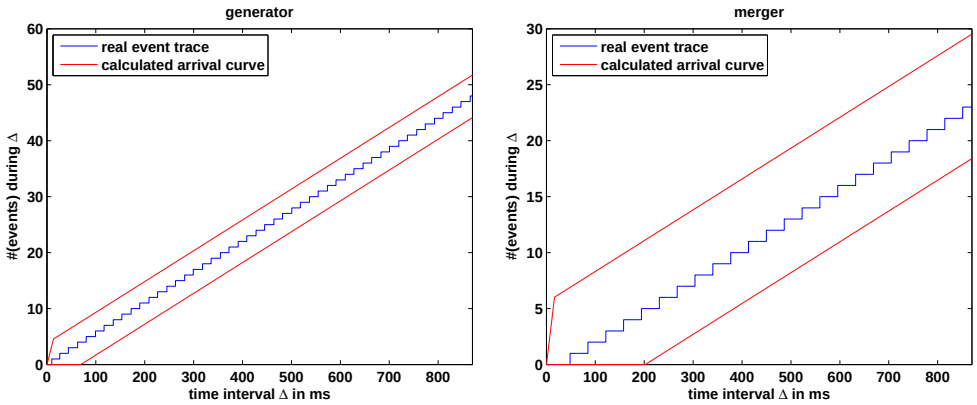


Fig. 29: Comparison of calculated upper and lower bounds with the real event traces. Mapping configuration 1.

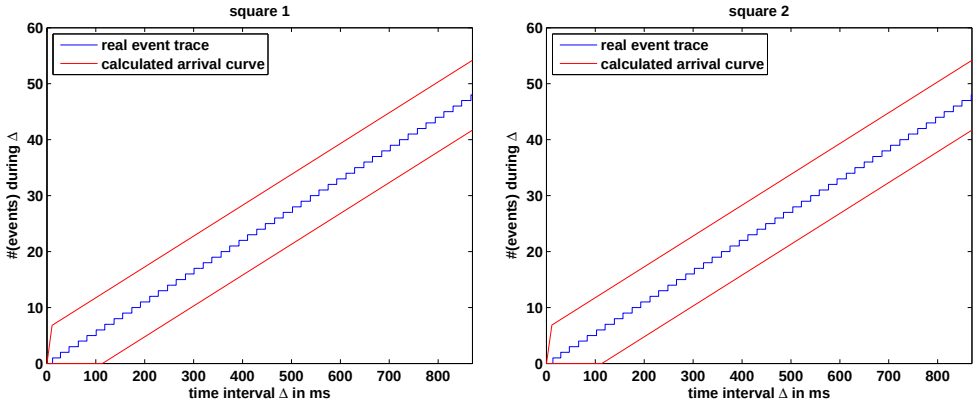


Fig. 30: Comparison of calculated upper and lower bounds with the real event traces. Mapping configuration 1.

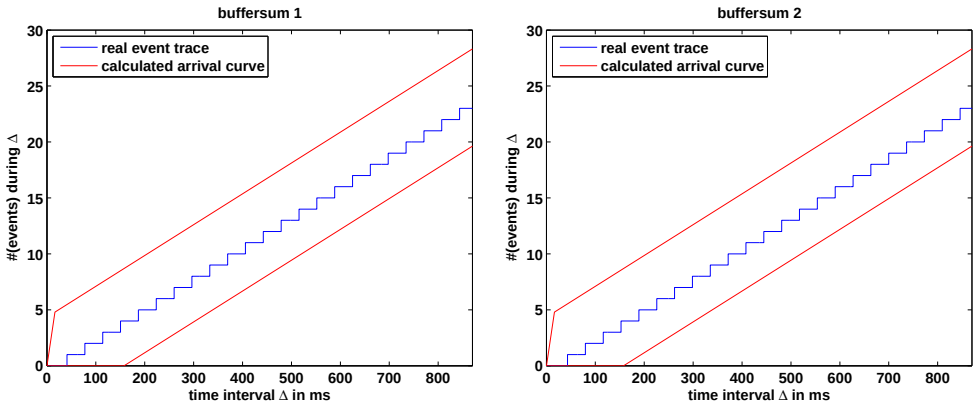


Fig. 31: Comparison of calculated upper and lower bounds with the real event traces. Mapping configuration 1.

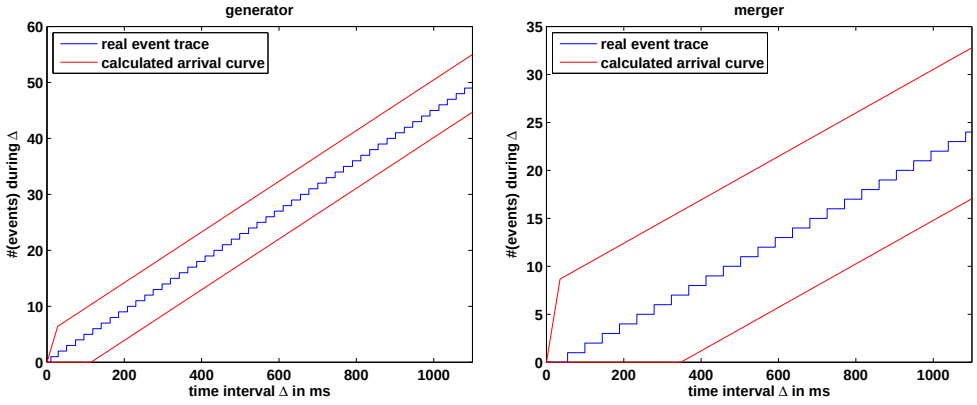


Fig. 32: Comparison of calculated upper and lower bounds with the real event traces. Mapping configuration 3.

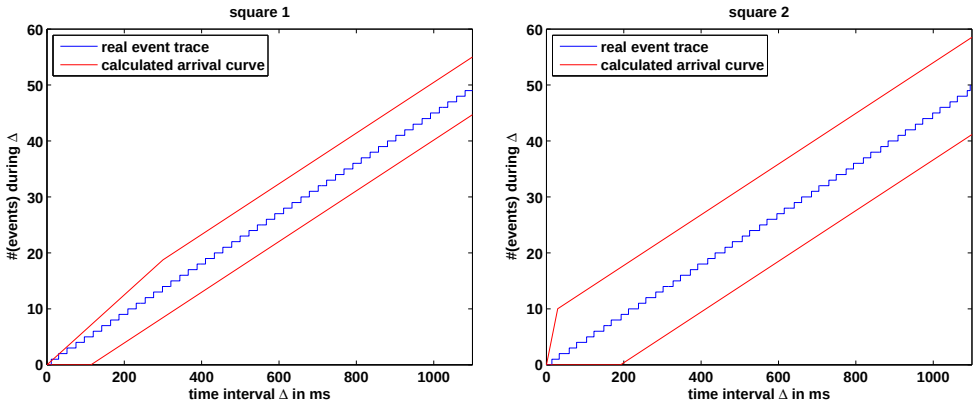


Fig. 33: Comparison of calculated upper and lower bounds with the real event traces. Mapping configuration 3.

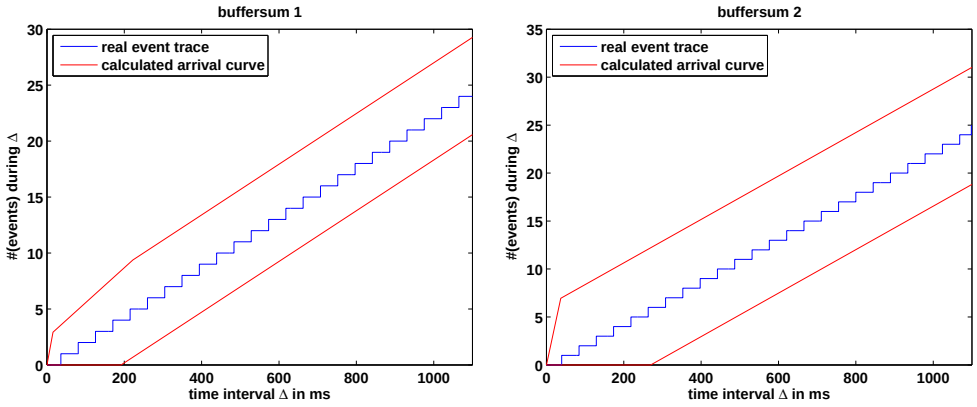


Fig. 34: Comparison of calculated upper and lower bounds with the real event traces. Mapping configuration 3.

A.2 HdS Software Stack Configuration

Unfortunately, the software stack configuration offers some pitfalls. As non-syntactical error are not checked, but result in a non-predicted behaviour of the software stack during execution, it is crucial to understand the configuration process.

There is a configuration file for each processor, the software stack for the ARM9 core is configured by altering the linker script.

A.2.1 Heap Size Setup

Depending on the number of tasks and their memory consumption, kernel and user heap might have to be increased. During execution, the software stack is switching to a safe state if a heap space is too small. In this state, an instruction that jumps to its own address is executed.

A.2.2 Stack Frame Configuration

The stack frame size has to be considered for dealing with large messages. While the stack frame size of the multi-tasking kernel can be set in the linker script, the stack frame size of the kernel that runs on the mAgic core needs to be set in the configuration file of the "bridge" tool.

A.2.3 Driver Configuration

Next, it has to be checked, if all necessary drivers modules are included in the configuration. This has to be checked against the communication paths that are referenced in the used mapping. In case that a driver is missing, the software stack will run, but no data will be transferred. In particular, an uninitialized function pointer will be called instead of a low-level IO function of the buffer driver.

A.2.4 Buffer Device Configuration

Lastly, the configuration of the needed buffer devices has to be verified. At first, the number of buffer devices must be equal or bigger than the number of mapped software channels. In case of a D940 external FIFO buffer, the configuration has to be equal for both processors. Additionally, it is important that the defined data type of a buffer device matches with the data type that is used in the application sources.

B

Presentation Slides

Generation of Accurate Performance Analysis Models for Embedded Systems

Master Thesis Presentation

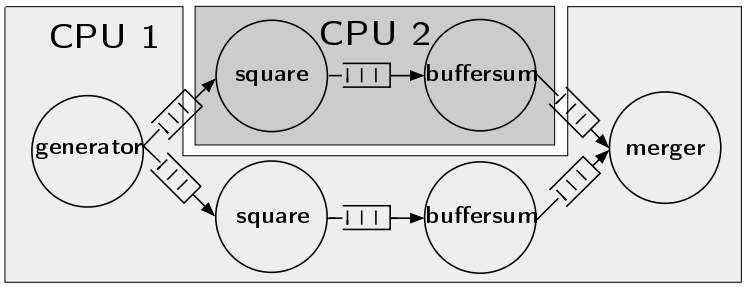
Matthias Keller

ETH Zurich, April 8, 2008

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

TIK Institut für Technische Informatik und Kommunikationsnetze

Background



Design space exploration of MPSoC systems

MPSoC systems can be characterized by their large design space. The goal is to find design points that match with the design requirements.

Background

System-level performance analysis

Estimation of system characteristics, such as end-to-end delays, throughput, or buffer requirements.

Design Space Exploration Based on Formal Analysis Methods

Formal based analysis methods offer a short evaluation time as well as corner-case coverage.

Problem and Task

Problem of current approaches

- Performance data must be extracted manually
- Analysis model must be generated by hand

Task

- Extract performance data by employing instruction-accurate simulation
- Generate accurate performance models for analysis with MPA-RTC
- Implement a proof of concept for SHAPES

Table of Contents

Related Work

Design space exploration based on formal analysis methods

- Henia et al. proposed a design space exploration tool flow based on SymTA/S
- Wandeler and Künzli used MPA-RTC during a design space exploration

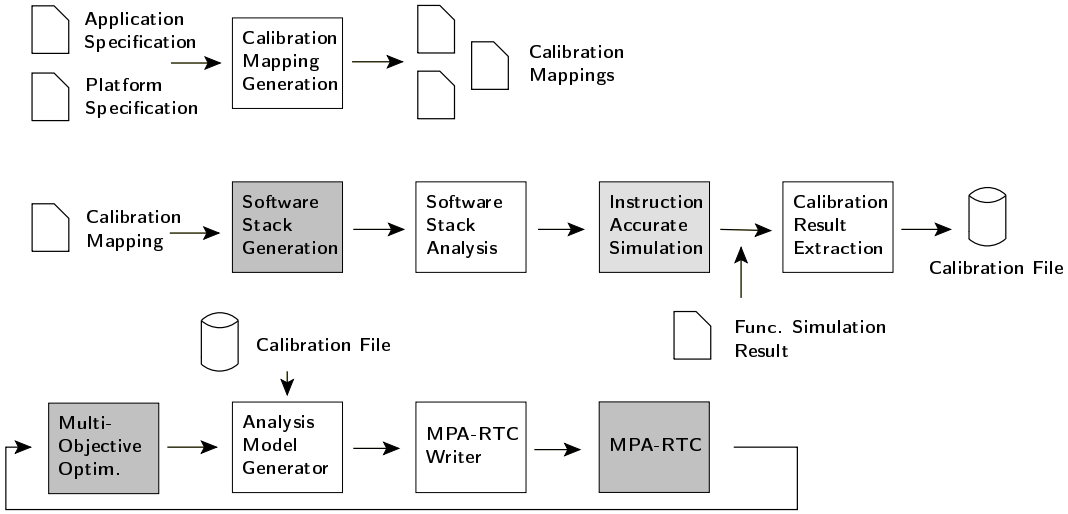
Combining simulation and formal methods for DSE

- Künzli, Poletti et al. showed the much faster evaluation speed of formal methods compared to simulation by using MPA-RTC for single components.

One-time calibration

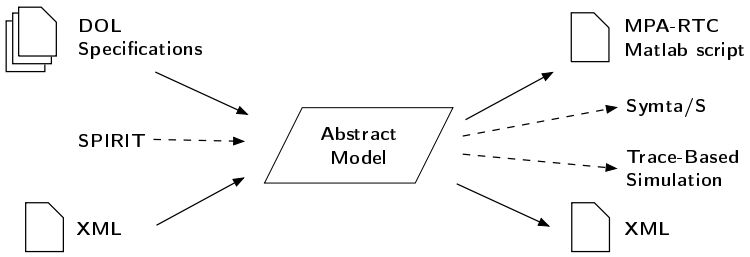
- Pimentel et al. performed a one-time model calibration for trace-based simulation.

Approach



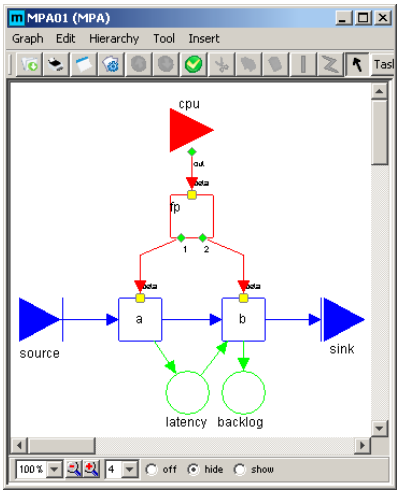
Motivation

Abstract description language for embedded systems



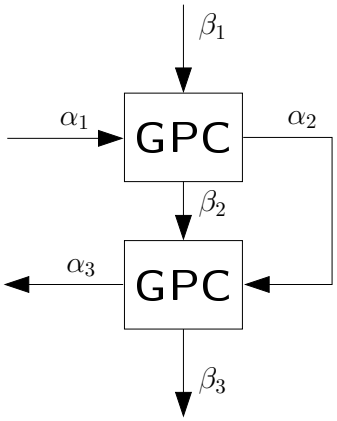
- Possibility to use the model for interfacing other frameworks
- Better software design compared to a single converter

Example System



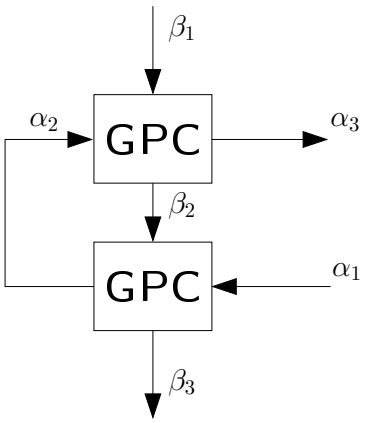

```
<?xml version="1.0" encoding="UTF-8"?>
<analysismodel>
  <pjd name="source" period="2"/>
  <sink name="sink"/>
  <task name="b" bced="1" wced="3"/>
  <task name="a" bced="1" wced="1"/>
  <connection from="source" to="a" sampling="1"/>
  <connection from="a" to="b" sampling="1"/>
  <connection from="b" to="sink" sampling="1"/>
  <resource name="cpu" bandwidth="4">
    <fp preemptive="true">
      <priority value="1">
        <binding task="a"/>
      </priority>
      <priority value="2">
        <binding task="b"/>
      </priority>
    </fp>
  </resource>
  <latency from="a" to="b"/>
  <backlog task="b"/>
</analysismodel>
```

Command Order Constraint



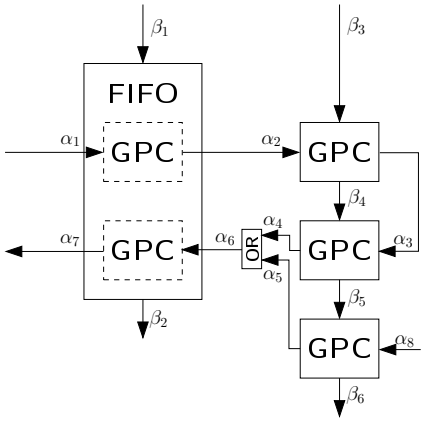

```
a1 = rtcpjd(1);
b1 = rtcfs(5);
[ a2 b2 ] = rtcgpc(a1, b1, [ 2 1 ]);
[ a3 b3 ] = rtcgpc(a2, b2, [ 2 1 ]);
```


Solving Dependencies in MPA-RTC



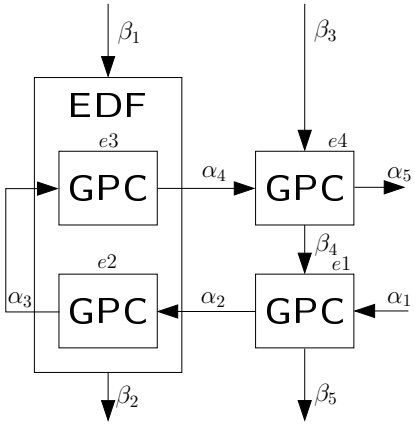

```
% a1, b1 are already defined
b2 = b1;
for i=1:20
  [ a2 b3 ] = rtcgpc(a1, b2, [ 2 1 ]);
  [ a3 b2 ] = rtcgpc(a2, b1, [ 2 1 ]);
  % Check fixed-point convergence
  % Code removed for illustration
end
```

Component Graph: Dependency Relation Graph for Sorting



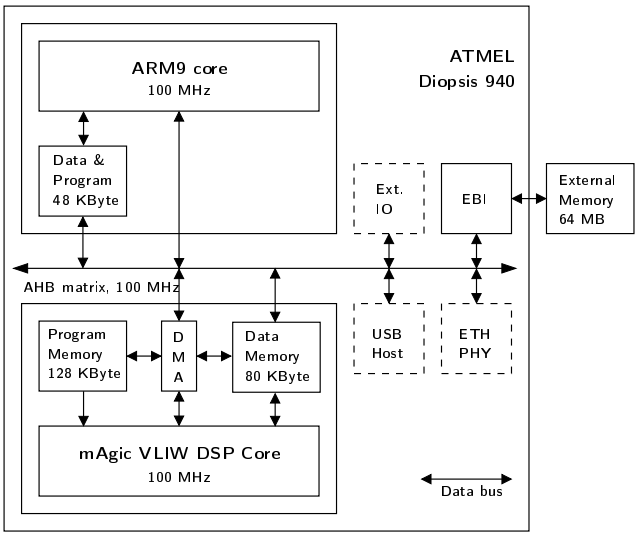
1. Service Dependency
2. Event Dependency
3. Strict Event Dependency
4. Inner-Component Dependency

Example




```
% b1, b3, a1, e1, e2, e3, e4
% are already defined
b4 = b3;
for it_service = [1:20]
    [ a2 b5 ] = rtcgpc( a1, b4, e1 );
    [ a3 b2 ] = rtcgpc( a2, b1, e2 );
    for it_event = [1:20]
        [ a3 a3_d a3_b a4 a4_d ...
            a4_b b5 ] = ...
            rtcfifo( a2, e2, a3, ...
            e3, b1 );
        % Check fixed-point convergence
        % Code removed for illustration
    end
    [ a5 b4 ] = rtcgpc( a4, b3, e4 );
    % Check fixed-point convergence
    % Code removed for illustration
end
```

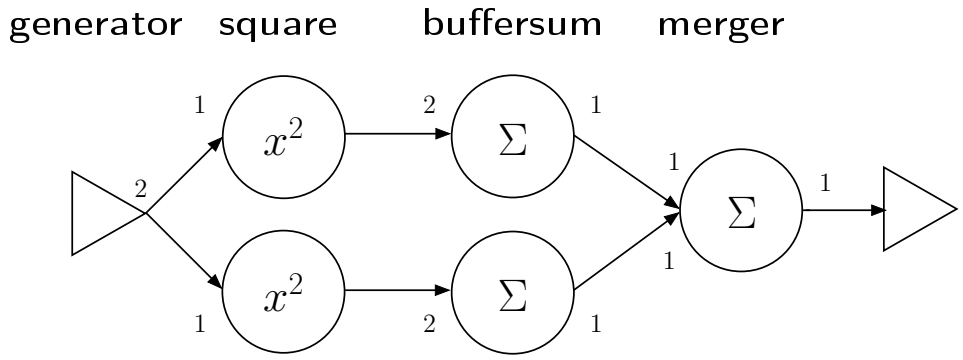
Atmel Diopsis 940 Architecture



Two Buffer Implementations

1. Generic SW FIFO
2. D940 External FIFO

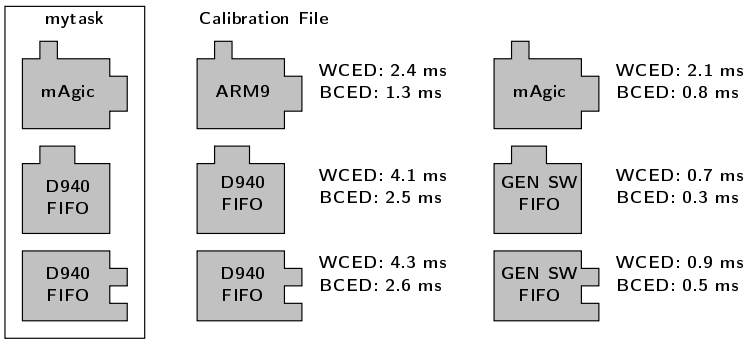
Data Flow Graph



One token refers to $1250 \cdot 4$ *Byte* $= 5000$ *Byte*.

Goal

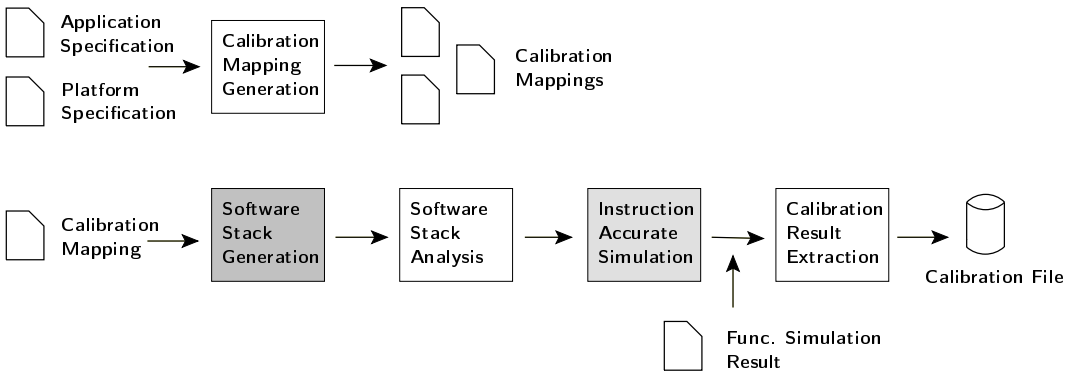
Fine-grained calibration results



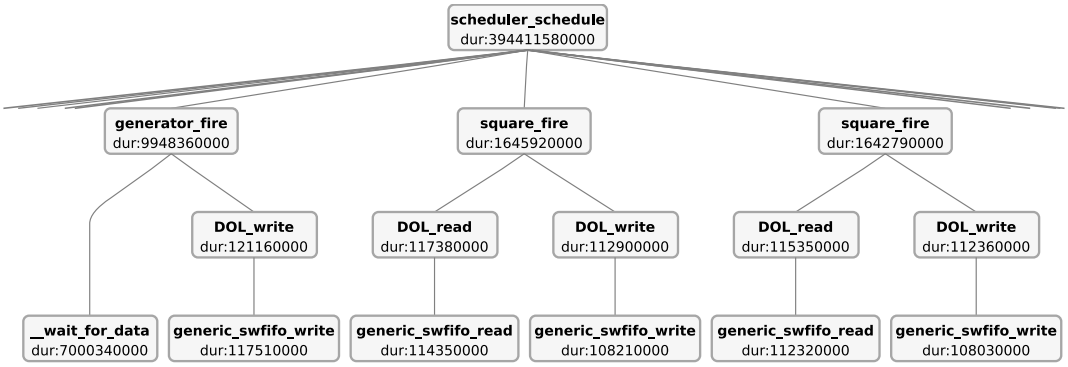
$$WCED_{mytask} = 2.1ms + 4.1ms + 4.3ms = 10.5ms$$

$$BCED_{mytask} = 0.8ms + 2.5ms + 2.6ms = 5.9ms$$

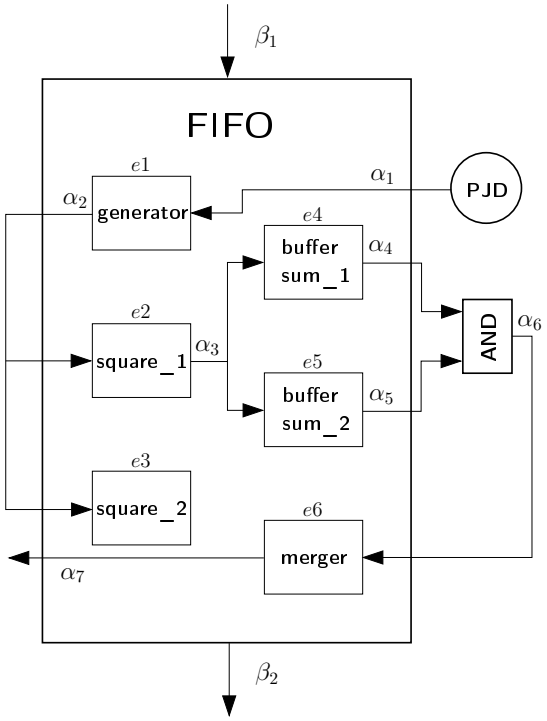
Approach



Sequential Call Tree Analysis



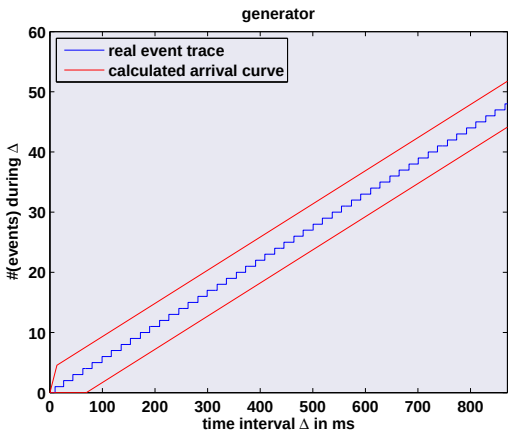
Case 1: MPA-RTC Model

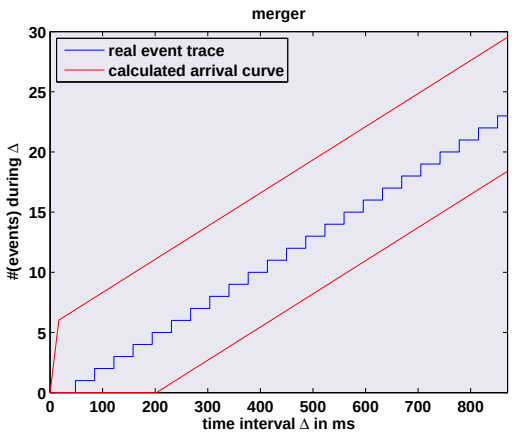


β_1 = 1 ms/ms
p = 18.1479 ms
j = 6.6985 ms
d = 12.7806 ms
e1 = [3.0662 ms, 2.9497 ms]
e2 = [1.6473 ms, 1.5374 ms]
e3 = [1.6455 ms, 1.6364 ms]
e4 = [3.4342 ms, 3.4209 ms]
e5 = [3.4240 ms, 3.4185 ms]
e6 = [2.8516 ms, 2.7721 ms]

Generation time: 1 sec

Case 1: Results



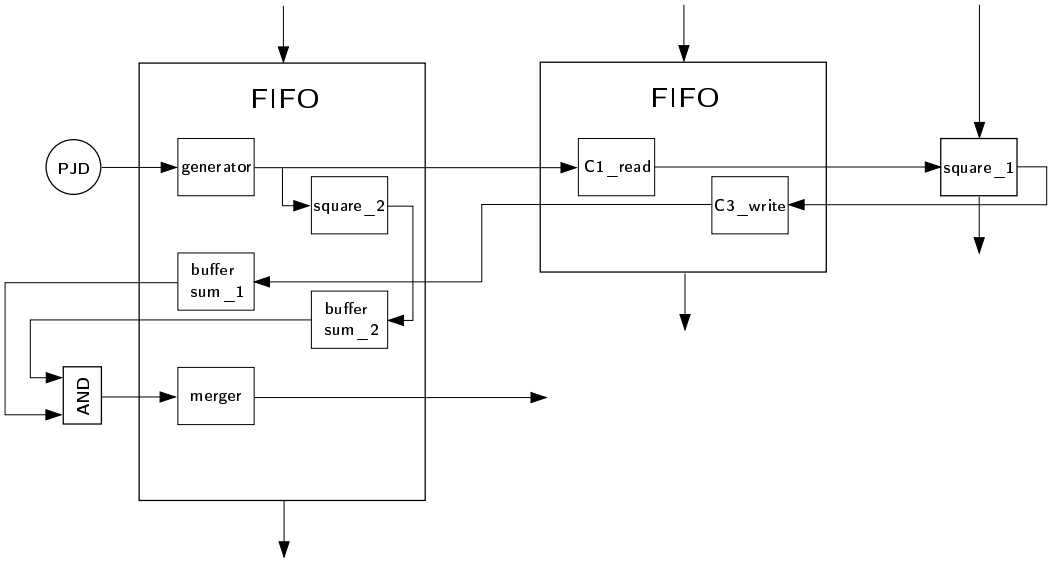


Measured end-to-end delay:
55.3 ms

Calculated end-to-end delay:
492.9 ms

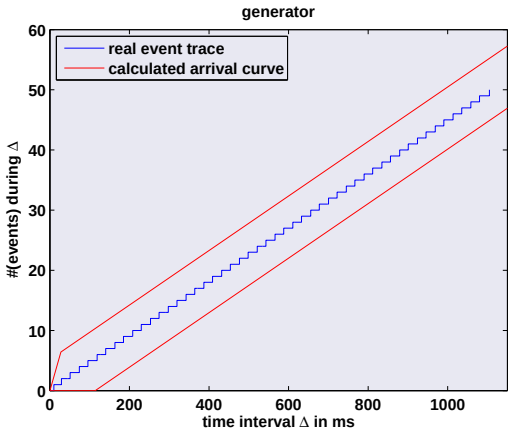
Evaluation time:
6.1 sec

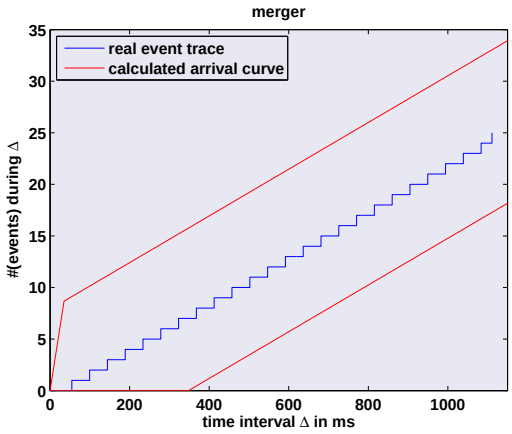
Case 2: MPA-RTC Model



ARM9	DMA	mAgic DSP
100 MHz	400 MByte/s	100 MHz

Case 2: Results





Measured end-to-end delay:
69.5 ms

Calculated end-to-end delay:
783.3 ms

Evaluation time:
17.0 sec

Conclusions

Abstract Model

Comprehensive model for a broad field of applications, especially in the context of MPA-RTC.

Model Generation

Fine-grained, fully automatically generated model. Only minimal manual effort during setup, good performance.

SHAPES

Implementation of a fully automated tool flow as a proof of the introduced concept.

Model Verification

Promising results concerning an application during a design space application loop.

List of Tables

List of Figures

Abbreviations

API	Application Programming Interface
CMOS	Complementary Metal–Oxide–Semiconductor
CPU	Core Processing Unit
DOL	Distributed Operation Layer
DSE	Design Space Exploration
DSP	Digital Signal Processor
EDF	Earliest Deadline First
FIFO	First-In First-Out
FPGA	Field-Programmable Gate Array
GPC	Greedy Processing Component
HAL	Hardware Abstraction Layer
HdS	Hardware Dependent Software
IP	Intellectual Property
ISA	Instruction Set Architecture
MPA	Modular Performance Analysis
MPA-RTC	Modular Performance Analysis with Real-Time Calculus
MPSoC	Multi-Processor System-on-Chip
PJD	Periodic Event Source with Jitter
RAM	Random Access Memory
RTC	Real-Time Calculus
SHAPES	Scalable Software Hardware Architecture Platform for Embedded Systems
SoC	System-on-Chip
SymTA/S	Symbolic Timing Analysis for Systems
TDMA	Time Division Multiple Access
VLIW	Very Long Instruction Word
VPA	Virtual Platform Analyzer
VSP	Virtual SHAPES Platform
WFS	Wave Field Synthesis
XML	Extensible Markup Language

Bibliography

[1] Apache Ant.
http://ant.apache.org/.

[2] Baccelli, F.L. and Olsder, G.J. and Quadrat, J.P. and Cohen, G. *Synchronization and Linearity: An Algebra for Discrete Event Systems*. Wiley Series on Probability and Mathematical Statistics: Probability and Mathematical Statistics, 1992.

[3] Balarin, F. *Hardware-Software Co-Design of Embedded Systems: The Polis Approach*. Springer, 1997.

[4] Blum, S. Darstellung von Abhängigkeitsgraphen. Semester Thesis, ETH Zürich, Computer Engineering and Networks Laboratory, 2007.

[5] Buck, J. and Ha, S. and Lee, E.A. and Messerschmitt, D.G. Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems. *International Journal of Computer Simulation*, 4(2):155–182, 1994.

[6] Chakraborty, S. and Künzli, S. and Thiele, L. A General Framework for Analysing System Properties in Platform-based Embedded System Designs. *Proc. 6th Design, Automation and Test in Europe (DATE)*, pages 190–195, 2003.

[7] CoWare Virtual Platform.
http://www.coware.com/products/virtualplatform.php.

[8] Distributed Operation Layer Web Site.
http://www.tik.ee.ethz.ch/~shapes/dol.html.

[9] EXPO – A Tool for Design Space Exploration of Network Processor Architectures.
http://www.tik.ee.ethz.ch/expo/.

[10] Gresser, K. An Event Model for Deadline Verification of Hard Real-Time Systems. *Real-Time Systems, 1993. Proc. 5th Euromicro Workshop on Real-Time Systems*, pages 118–123, 1993.

[11] Gries, M. Methods for Evaluating and Covering the Design Space during early Design Development. *Integration, the VLSI Journal*, 38(2):131–183, 2004.

[12] Guerin, X. and Popovici, K. and Youssef, W. and Rousseau, F. and Jerraya, A. Flexible Application Software Generation for Heterogeneous Multi-Processor System-on-Chip. *Computer Software and Applications Conference, 2007. COMPSAC 2007-Vol. 1. 31st Annual International*, 1, 2007.

[13] Haid, W. and Thiele, L. Complex Task Activation Schemes in System Level Performance Analysis. *Proc. of the 5th IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis*, pages 173–178, 2007.

[14] Henia, R. and Hamann, A. and Jersak, M. and Racu, R. and Richter, K. and Ernst, R. System-Level Performance Analysis - The SymTA/S Approach. *Computers and Digital Techniques, IEEE Proc.*, 152(2):148–166, 2005.

[15] Ho, R. and Mai, KW and Horowitz, MA. The Future of Wires. *Proc. of the IEEE*, 89(4):490–504, 2001.

[16] Kahn, G. The Semantics of a Simple Language for Parallel Programming. *Information Processing*, 74:471–475, 1974.

[17] Kraemer, S. Virtual SHAPES Simulation Platform. Workshop Poster at Automation and Test in Europe (DATE), 2007.

[18] Künzli, S. and Poletti, F. and Benini, L. and Thiele, L. Combining Simulation and Formal Methods for System-Level Performance Analysis. In *Design Automation and Test in Europe (DATE)*, pages 236–241. IEEE Computer Society, 2006.

[19] Künzli, S. and Thiele, L. Generating Event Traces Based on Arrival Curves. In *13th GI/ITG Conference on Measurement, Modeling, and Evaluation of Computer and Communication Systems (MMB)*, pages 81–98. VDE Verlag, March 2006.

[20] Lahiri, K. and Raghunathan, A. and Dey, S. System-Level Performance Analysis for Designing On-chip Communication Architectures. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 20(6):768–783, 2001.

[21] Le Boudec, J.Y. and Thiran, P. *Network Calculus: A Theory of Deterministic Queuing Systems for the Internet*. Springer, 2001.

[22] Lee, EA and Messerschmitt, DG. Synchronous Data Flow. *Proc. of the IEEE*, 75(9):1235–1245, 1987.

[23] Lieverse, P. and Stefanov, T. and van der Wolf, P. and Deprettere, E. System-Level Design with Spade: An M-JPEG Case Study. *Proc. Int. Conference on Computer Aided Design (ICCAD 01)*, pages 31–38, 2001.

[24] The Moses Project – Modeling, Simulation and Evaluation of Systems.
`http://www.tik.ee.ethz.ch/~moses/`.

[25] Modular Performance Analysis with Real-Time Calculus.
`http://www.mpa.ethz.ch`.

[26] Paolucci, P.S. SHAPES Project Web Site.
`http://www.shapes-p.org`.

[27] Paolucci, P.S. and Jerraya, A.A. and Leupers, R. and Thiele, L. and Vicini, P. SHAPES: A Tiled Scalable Software Hardware Architecture Platform for Embedded Systems. *Proc. of the 4th International Conference on Hardware/software Codesign and System Synthesis*, pages 167–172, 2006.

[28] Paolucci, P.S. and Kajfasz, P. and Bonnot, P. and Candaele, B. and Maufroid, D. and Pastorelli, E. and Ricciardi, A. and Fusella, Y. and Guarino, E. mAgic-FPU and MADE: A Customizable VLIW Core and the Modular VLIW Processor Architecture Description Environment. *Computer Physics Communications*, 139(1):132–143, 2001.

[29] Performance Simulation of Distributed Embedded Systems.
`http://www.mpa.ethz.ch/PESIMDES/Overview`.

[30] Pier Stanislao Paolucci. Janus: A Gigaflop VLIW+RISC SoC Tile. Hot Chips 15 IEEE Stanford Conference, 2003.

[31] Pimentel, A.D. and Thompson, M. and Polstra, S. and Erbas, C. On the Calibration of Abstract Performance Models for System-level Design Space Exploration. *Embedded Computer Systems: Architectures, Modeling and Simulation, 2006. IC-SAMOS 2006.*, pages 71–77, July 2006.

[32] Polstra, S. A Systematic Approach to Exploring Embedded System Architectures at Multiple Abstraction Levels. *IEEE Transactions on Computers*, 55(2):99–112, 2006.

[33] Richter, K. and Jersak, M. and Ernst, R. A Formal Approach to MPSoC Performance Verification. *Computer*, 36(4):60–67, 2003.

[34] Rüegg, C. A Tool-Independent System Description for Real-Time Embedded Systems. Semester Thesis, ETH Zürich, Computer Engineering and Networks Laboratory, 2006.

[35] Simon Künzli. *Efficient Design Space Exploration for Embedded Systems*. PhD thesis, Swiss Federal Institute of Technology Zürich, April 2006.

[36] SimpleScalar LLC.
`http://www.simplescalar.com/`.

[37] The SPIRIT Consortium.
`http://www.spiritconsortium.org/`.

[38] Sporer, T. and Beckinger, M. and Franck, A. and Bacivarov, I. and Haid, W. and Huang, K. and Thiele, L. and Paolucci, P.S. and Bazzana, P. and Vicini, P. and others. SHAPES – A Scalable Parallel HW/SW Architecture Applied to Wave Field Synthesis. 2007.

[39] The Open SystemC Initiative (OSCI).
`http://www.systemc.org`.

[40] TARGET Compiler Technologies.
`http://www.retarget.com/`.

[41] Tcl – Tool Command Language.
`http://www.tcl.tk/`.

[42] Thiele, L. and Bacivarov, I. and Haid, W. and Huang, K. Mapping Applications to Tiled Multiprocessor Embedded Systems. In *Proc. 7th Intl Conference on Application of Concurrency to System Design (ACSD 2007)*, pages 29–40, Bratislava, Slovak Republic, July 2007. IEEE Computer Society.

[43] Thiele, L. and Chakraborty, S. and Gries, M. and Künzli, S. Design Space Exploration of Network Processor Architectures. In *First Workshop on Network Processors at the 8th International Symposium on High-Performance Computer Architecture (HPCA8)*, pages 30–41, Cambridge MA, USA, February 2002.

[44] Thompson, M. and Nikolov, H. and Stefanov, T. and Pimentel, A.D. and Erbas, C. and Polstra, S. and Deprettere, E.F. A Framework for Rapid System-Level Exploration, Synthesis, and Programming of Multimedia MPSoCs. *Proc. of the 5th IEEE/ACM international conference on Hardware/software codesign and system synthesis*, pages 9–14, 2007.

[45] Computer Engineering and Networks Laboratory (TIK). http://www.tik.ee.ethz.ch.

[46] Wandeler, E. *Modular Performance Analysis and Interface-Based Design for Embedded Real-Time Systems*. PhD thesis, Swiss Federal Institute of Technology Zürich, 2006.

[47] Wandeler, E. and Thiele, L. and Verhoef, M. and Lieverse, P. System Architecture Evaluation Using Modular Performance Analysis: A Case Study. *International Journal on Software Tools for Technology Transfer (STTT)*, 8(6):649–667, 2006.

[48] Zitzler, E. and Thiele, L. Multiobjective Evolutionary Algorithms: A Comparative Case Study and the Strength Pareto Approach. *Evolutionary Computation, IEEE Transactions on*, 3(4):257–271, 1999.