Modular Performance Analysis and Interface-Based Design for Embedded Real-Time Systems

A dissertation submitted to the

SWISS FEDERAL INSTITUTE OF TECHNOLOGY
ZURICH

for the degree of

Doctor of Sciences

presented by

ERNESTO WANDELER
Dipl. El.-Ing. ETH Zurich

born 01.12.1978
citizen of Gansingen, AG

Prof. Dr. Lothar Thiele, examiner
Prof. Dr. Petru Eles, co-examiner

2006

Institut für Technische Informatik und Kommunikationsnetze
Computer Engineering and Networks Laboratory

TIK-SCHRIFTENREIHE NR. 82

Ernesto Wandeler

Modular Performance Analysis and Interface-Based Design for Embedded Real-Time Systems

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

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

Prof. Dr. Lothar Thiele, examiner
Prof. Dr. Petru Eles, co-examiner
Examination date: 14. September, 2006

Abstract

System level performance analysis methods play an important role in the design process of complex embedded systems. They allow to analyze essential performance characteristics of a system design in an early design phase and consequently support the choice of important design decisions before much time and resources are invested in detailed implementations.

While formal analysis based methods for system level performance analysis lead to hard bounded analysis results and can thus be employed in the design of hard real-time systems, these methods are often restricted in their modeling and analysis capabilities, and the obtained results are often overly pessimistic due to a lack of details such analytical methods can incorporate in their system analysis.

In this thesis we identify challenges for system level performance analysis of embedded systems, and based on these challenges we further develop and extend a framework for formal modular performance analysis and design of complex distributed embedded real-time systems. The main contributions of this work are listed in the following.

- Novel models and methods are presented for performance analysis of complex embedded systems that combine components with various different processing semantics within one system design.
- Novel models and methods are introduced to address the challenges of complex inputs, variable execution demands, and workload correlations in performance analysis of complex embedded systems.
- The theory of Real-Time Interfaces that connects the principles of Real-Time Calculus and interface-based design is introduced, together with a component system that enables interface-based embedded real-time system design.
- Novel models and methods are introduced that facilitate the modeling, analysis, and design of hierarchical scheduling policies in complex embedded systems.
- A new MATLAB Toolbox is introduced that supports efficient system level performance analysis and interface-based design of distributed embedded real-time systems.

Zusammenfassung

Methoden zur Performanzanalyse auf Systemebene spielen eine wichtige Rolle im Designprozess von komplexen Eingebetteten Systemen. Sie erlauben die Analyse von essentiellen Performanzcharakteristiken eines Systemdesigns in einer frühen Designphase und helfen dabei wichtige Designentscheidungen zu fällen bevor viel Zeit und viele Ressourcen in Detailimplementationen investiert werden.

Während formale Performanzanalysemethoden es erlauben hart begrenzte Analyseresultate zu finden, und dadurch auch im Designprozess von harten Echtzeitsystemen angewandt werden können, sind die Modellierungs- uns Analysefähigkeiten dieser Methoden oft eingeschränkt. Und aufgrund dem Mangel and Details welche diese Methoden in die Analyse einbinden können, sind die erzielten Resultate oft auch übermässig pessimistisch.

In der vorliegenden Arbeit identifizieren wir Herausforderungen für die Performanzanalyse von Eingebetteten Systemen, basierend auf welchen wir ein Framework zur formalen, modularen Performanzanalyse, und zum Design von komplexen, verteilten Eingebetteten Echtzeitsystemen weiterentwickeln. Nachfolgend die wichtigsten Resultate.

- Es werden neue Modelle und Methoden präsentiert zur Performanzanalyse von Eingebetteten Systemen, welche Komponenten mit verschiedenen Ablaufsemantiken in einem System kombinieren.
- Neue Modelle und Methoden werden eingeführt zur Analyse von Eingebettete Systeme mit komplexen Einspeisungen, variablen Abarbeitungsanforderungen, und Arbeitsbelastungskorrelationen.
- Die Theorie der Real-Time Interfaces wird eingeführt, zusammen mit einem Komponentensystem welches Interface-basiertes Systemdesign von Eingebetteten Echtzeitsystemen ermöglicht.
- Neue Modelle und Methoden werden eingeführt, welche die Modellierung, die Analyse, sowie das Design von hierarchischen Ablaufplanungsmethoden in Eingebetteten Systemen unterstützen.
- Eine neue MATLAB Toolbox wird präsentiert, welche effiziente Performanzanalyse sowie Interface-basiertes Design von verteilten Eingebetteten Echtzeitsystemen ermöglicht.

Contents

Abstract i

Zusammenfassung iii

1 Introduction 1
- 1.1 System-Level Performance Analysis . . . 2
  - 1.1.1 The Role in the Design Process . . . 2
  - 1.1.2 Challenges of Performance Analysis . . . 3
  - 1.1.3 Formal Analysis vs. Simulation . . . 4
  - 1.1.4 Requirements . . . 6
- 1.2 Aim of this Thesis . . . 7
- 1.3 Thesis Outline and Contributions . . . 8

I Modular Performance Analysis 11

2 Modular Performance Analysis with Real-Time Calculus 13
- 2.1 Overview . . . 14
- 2.2 Model of Environment . . . 15
  - 2.2.1 Arrival Curves: A General Event Stream Model . . . 15
  - 2.2.2 Obtaining Arrival Curves . . . 16
  - 2.2.3 Determining the Resource Demand . . . 17
- 2.3 Model of Resources . . . 19
  - 2.3.1 Service Curves: A General Resource Model . . . 19
  - 2.3.2 Obtaining Service Curves . . . 20
- 2.4 Model of Tasks and Components . . . 20
- 2.5 System Performance Model . . . 23
  - 2.5.1 Flow of Data . . . 24
  - 2.5.2 Scheduling and Arbitration . . . 24
- 2.6 Analysis . . . 25
  - 2.6.1 Performance Analysis . . . 26
  - 2.6.2 Sensitivity Analysis . . . 27
- 2.7 Case Study . . . 27

2.7.1 A Distributed In-Car Navigation System 28
2.7.2 Constructing the System Performance Models . . . 33
2.7.3 System Analysis 34
2.8 Related Work 39
2.9 Discussion 43

3 Abstract Components 45
3.1 Greedy Shapers 46
3.1.1 Embedding Greedy Shapers 47
3.1.2 Concrete Greedy Shapers 48
3.1.3 Abstract Greedy Shapers 50
3.1.4 Applications and Experimental Results 52
3.2 Components with Multiple Inputs 59
3.2.1 Embedding Components with Multiple Inputs . . . 60
3.2.2 Abstract OR-Connector 62
3.2.3 Abstract AND-Connector 62
3.2.4 Experimental Results 65
3.3 Discussion 69

4 Workload Variability and Correlations 71
4.1 Workload Transformations 72
4.1.1 Events and Workloads 73
4.1.2 Workload Curves 74
4.1.3 Embedding Workload Transformations 75
4.2 Event-Based Workload Variability 76
4.2.1 Type Rate Curves 77
4.2.2 Computing Workload Curves 79
4.3 Functional Workload Variability 80
4.3.1 Event Sequence Automata 80
4.3.2 Workload Variability Automata 81
4.3.3 Computing Workload Curves 82
4.3.4 Experimental Results 84
4.4 Workload Correlations 88
4.4.1 An Application Scenario 88
4.4.2 Event-Based and Resource-Based Analysis 89
4.4.3 Workload Correlations Curves 92
4.4.4 Computing Workload Correlation Curves 95
4.4.5 Experimental Results 97
4.5 Solving the Maximum-Weight Path Problem 102
4.6 Related Work 103
4.7 Discussion 105

II Interface-Based Design 107

5 Interface-Based Design with Real-Time Interfaces 109
5.1 Interface-Based Design and A/G Interfaces 110
5.2 Real-Time Interfaces 113
5.2.1 Abstract Real-Time Components 113
5.2.2 Interface Variables and Predicates 114
5.3 A Component System with Interfaces 115
5.3.1 Event Stream Component 116
5.3.2 Resource Component 116
5.3.3 Processing Component for FP Scheduling 116
5.3.4 Processing Component for RM Scheduling 119
5.3.5 Processing Component for EDF Scheduling 119
5.3.6 Processing Component for FIFO Scheduling 121
5.4 Applications and Experimental Results 122
5.4.1 Application Scenario 122
5.4.2 Interface-Based Schedulability Analysis 123
5.4.3 Interface-Based System Design 125
5.4.4 Interface-Based System Adaption 126
5.4.5 Interface-Based Admission Tests 128
5.5 Related Work 129
5.6 Discussion 131

6 Design & Analysis of Systems with Hierarchical Scheduling 133
6.1 Time Division Multiple Access 134
6.1.1 Performance Analysis 135
6.1.2 Parameter Selection 137
6.1.3 Experimental Results 140
6.1.4 Related Work 145
6.2 Polling Servers 146
6.2.1 Performance Analysis 146
6.2.2 Parameter Selection 149
6.2.3 Experimental Results 150
6.2.4 Related Work 153
6.3 Discussion 156

III Tool Support 157

7 Efficient Computation of Real-Time Calculus 159
7.1 Classification of VCCs 161
7.1.1 Variability Characterization Curves 161
7.1.2 Classification Scheme 161

7.1.3 Practically Relevant Classes of VCCs 162
7.2 A Compact Representation for VCCs 165
7.2.1 Curve Segment Sequences 165
7.2.2 Compact VCCs 166
7.3 Operations on Compact VCCs 167
7.3.1 Method 168
7.3.2 Unary Operators 168
7.3.3 Binary Operators 169
7.4 Discussion 172

8 The RTC Toolbox 173
8.1 Software Architecture 173
8.2 Case Study 175
8.3 Discussion 176

Conclusions 177

Outlook 180

Bibliography 182

A Real-Time Calculus 197
A.1 Convolutions and Deconvolutions 197
A.2 Sub-Additivity and Sub-Additive Closure 198
A.3 Selected Properties 198
A.4 Greedy Processing Component 199
A.4.1 Basic Function Processing 200
A.4.2 Remaining Service 201
A.4.3 Processed Events 203

List of Publications 205

Curriculum Vitae 209

1

Introduction

Embedded systems are special-purpose information processing systems that are closely integrated into their environment. An embedded system is typically dedicated to a specific application domain, and knowledge about this domain and the system's environment are used to develop customized and optimized system designs.

The embedding into a technical environment and the constraints imposed by a particular application domain often require a distributed implementation of embedded systems, where a number of hardware components communicate via some interconnection network. The hardware components in such systems are often specialized and aligned to their local environment and their functionality. And also the interconnection networks are often not homogeneous, but may instead be composed of several interconnected sub-networks, each with its own communication protocol and topology. And in more recent embedded systems, the architectural concepts of heterogeneity, distributivity and parallelism can even be observed on single hardware components themselves, as they are often implemented as so-called systems-on-chip (SoC). In these components, a collection of digital or analogue interfaces, busses, memory, and heterogeneous computing resources such as FPGAs, CPUs, controllers and digital signal processors are implemented on a single device, and communicate using networks-on-chip (NoC) that can be regarded as dedicated interconnection networks involving adapted protocols, bridges or gateways.

Embedded systems are typically reactive systems that are in continuous interaction with their physical environment to which they are connected through sensors an actuators. Consequently they must execute at a pace determined by their environment. This has as result than many

embedded systems must meet real-time constraints, i. e. they must react to stimuli within a time interval dictated by the environment. Such a real-time constraint is called hard, if not meeting it could result in a impermissible failure of the system, and it is called soft otherwise.

It becomes apparent that heterogeneous and distributed embedded real-time systems as described above are inherently difficult to design and to analyze. Particularly, as often not only the availability, the safety, and the correctness of the computations of the whole embedded system are of major concern, but also the timeliness of the computed results.

1.1 System-Level Performance Analysis

During the system level design process of an embedded system, a designer is typically faced with questions such as whether the timing properties of a certain system design will meet the design requirements, what architectural element will act as a bottleneck, or what the on-chip memory requirements will be. Consequently it becomes one of the major challenges in the design process to analyze specific characteristics of a system design, such as end-to-end delays, buffer requirements, or throughput in an early design stage, to support making important design decisions before much time is invested in detailed implementations. This analysis is generally referred to as system level performance analysis.

1.1.1 The Role in the Design Process

Because of the many alternatives for partitioning, allocation, and binding in the system design, a designer of a complex embedded system is confronted with a large design space. To cope with this large design space, automated or semi-automated design space exploration is often employed, where a large number of implementation choices are investigated in order to determine design trade-offs between various possibly conflicting design objectives, see e. g. [TKZ04]. Within the design space exploration cycle, performance analysis plays a crucial role to evaluate the quality of different system designs, as can be seen in Figure 1.

But even if design space exploration is not a part of the chosen design methodology, performance analysis is often employed in the development process of embedded systems, as not only the functional, but also the non-functional constraints of a system must typically be validated after each major design step.

Finally, performance analysis is often also employed after completion of the system design phase, to validate and possibly certify the real-time properties of an embedded system. Particularly for hard real-time

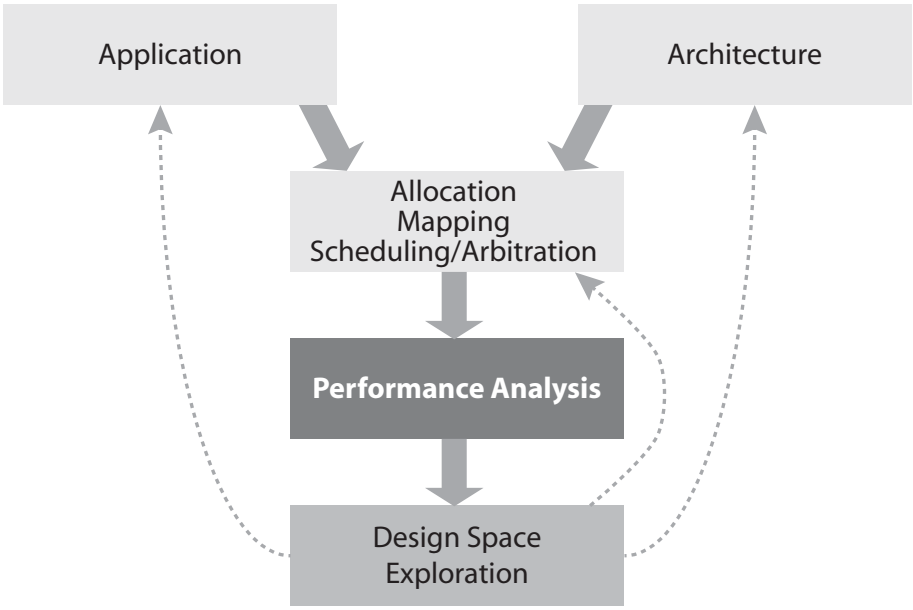


Fig. 1: Performance analysis within the design space exploration cycle.

systems validation plays a crucial role and guarantees that the system will never lead to an impermissible failure.

1.1.2 Challenges of Performance Analysis

System level performance analysis of complex distributed embedded systems faces many challenges that make the analysis difficult, and that potentially cause inaccurate analysis results. In order to better understand these challenges and to identify potential areas of research, we categorize some of them in the following.

- *Resource Sharing*: Complex embedded systems typically host multiple applications, with tasks executing concurrently on the various distributed hardware components, and communicating via the interconnection network. To control resource access of these tasks on the various computation and communication components, a large range of scheduling and arbitration policies are available. A single system often employs various different policies on its respective components, and with hierarchical scheduling emerging, different policies may even be employed on a single component. However, the choice of the scheduling and arbitration policy for a component and its parametrization largely influence the performance of the complete system.

- *Interferences*: Depending on the resource sharing policies that are employed on a system, different applications may interfere with each other. The performance of a single application then depends on the processing load that other applications put on the system. For the analysis of interferences, special care must be taken with so-called scheduling anomalies that are often observed on complex embedded systems. On these systems, the worst-case behavior of one application sometimes occurs only when other applications execute with their best-case behavior.

- *Correlations*: Various complex timing and workload correlations are often observed when applications are executed on a distributed embedded system. These correlations often foreclose the existence of some worst-case situations, and their consideration can improve the accuracy of the analysis results.

- *Complex Inputs*: The events on the input event streams of an embedded system often do not arrive strictly periodically, but follow instead possibly complex arrival patterns, and exhibit jittery or bursty behavior. As a consequence, some events of a burst for example may get buffered as the system's resources are occupied processing the preceding events of the same event stream, thus increasing buffer requirements and end-to-end delays of the system. Moreover, the events on a single event stream are often not homogeneous, but carry instead different payloads or follow different execution paths in an application's task graph.

- *Variable Execution Demands*: The execution demand of a single application task is often variable and depends for example on the payload or the type of the event that is processed. And even if all processed events are homogeneous, the execution demand may vary due to caching effects.

- *Processing Semantics*: A distributed embedded system is typically built up from a mixture of various components with different processing semantics that each require specific analysis. This not only necessitates to develop a specific analysis for every processing semantics, but it also complicates system level performance analysis, as the various specific analyses must be integrated with each other to analyze a complete system.

1.1.3 Formal Analysis vs. Simulation

Most methods for system-level performance analysis can broadly be divided into the two main classes of simulation-based and formal-analysis-

based methods. In the area of embedded real-time systems design, one of the major differentiation criteria between these two classes of methods is the quality of results that are obtained with the respective methods.

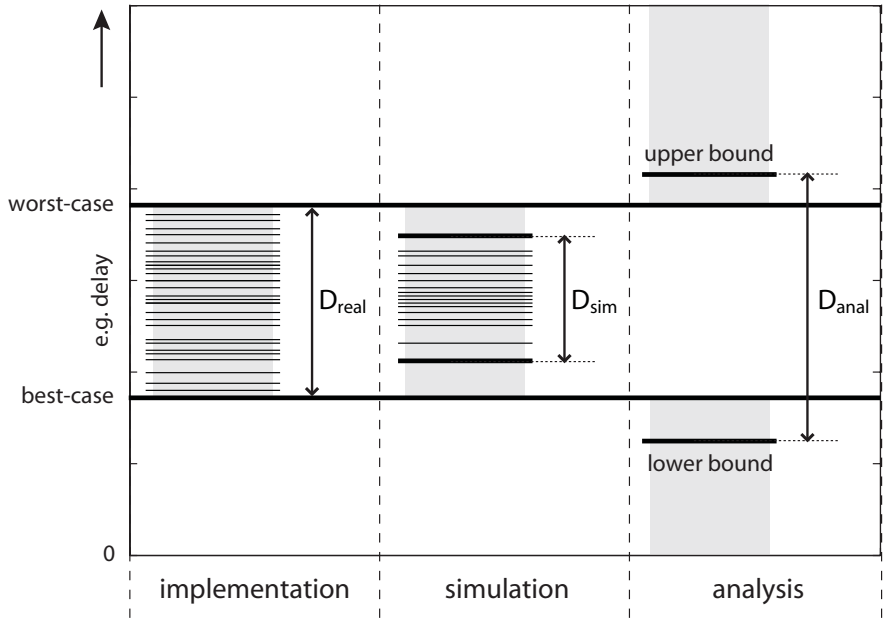


Fig. 2: The range of end-to-end delays in a real-time system obtained from system simulation and form formal analysis, compared to the range of end-to-end delays observed on the real system implementation.

An implementation of an embedded real-time system must meet a number of performance requirements related for example to end-to-end delays, buffer requirements, or throughput. When we measure these quantities on the final system implementation, we normally observe major variations over time, as for example end-to-end delays may vary largely due to different input data or interference between concurrent system activities. However, there typically exists a worst-case and a best-case result for every quantity, such that we know for example that every observed end-to-end delay is larger or equal the best-case delay d_{BC} and smaller or equal the worst-case delay d_{WC}. Or mathematically expressed, for all observed delays d, we know $d_{BC} \leq d \leq d_{WC}$. We denote the range of possible delays with $D_{real} = [d_{BC}, d_{WC}]$, as depicted in Figure 2. It is important to note however that even though d_{BC} and d_{WC} exist, their values are usually not know for sufficiently complex systems.

When we simulate the same system and measure the same end-to-end delay on the simulated system, we typically also observe major variations

over time. And as in the final implementation, the observed delays are within the range D_{real}. However, as every simulation run is of finite length, and as thus only a finite set of initial states, environment behaviors and execution traces can be considered, the observed delays will typically not reveal the worst-case and best-case results. This problem is well-know as the problem of insufficient corner-case coverage in simulations. Consequently, a system simulation can only reveal that there exist delays in the range $D_{sim} \subset D_{real}$, but it cannot foreclose the existence of delays outside D_{sim}. And even if a simulation run exposes the worst-case or best-case delay, we have typically no means to detect this, as d_{BC} and d_{WC} are often not known.

In difference to simulation based methods, formal analysis based methods are normally concerned with finding upper and lower bounds to the worst-case and best-case results, respectively. With a correct analysis based method, any resulting upper bound d_{upper} to a delay will always be larger or equal the worst-case delay d_{WC}, and any resulting lower bound d_{lower} will always be smaller or equal the best-case delay d_{BC}. Mathematically expressed, we know $d_{lower} \leq d_{BC}$ and $d_{WC} \leq d_{upper}$, and since $d_{BC} \leq d \leq d_{WC}$, we can conclude that $d_{lowre} \leq d \leq d_{upper}$. Thus, a formal analysis based method can guarantee that all observed delays in the real system implementation are in the range $D_{anal} \supset D_{real}$.

From the above assessment it becomes clear that only formal analysis based methods allow to guarantee that a hard real-time system strictly adheres to its design time performance requirements.

1.1.4 Requirements

Based on the above discussion, we list some of the requirements that a formal analysis based methodology for performance analysis of distributed embedded systems should ideally satisfy.

- *Correctness*: The results of the performance analysis should be correct, i. e. there must exist no reachable system states and feasible reactions of the system environment such that the calculated bounds are violated.

- *Accuracy*: The lower and upper bounds determined by the performance analysis should be close to the actual worst case and best case properties.

- *Embedding into the Design Process*: The underlying performance model should be sufficiently general to allow the representation of the application, of the environment, of the hardware platform, and of the mapping including the resource sharing policies. Ideally,

the method should seamlessly integrate into the functional specification and design methodology. Moreover, the method should be able to cope with incomplete design information, as typically the lower layers are not designed or implemented yet in the early design phases of a system.

- *Modularity*: As distributed systems are heterogeneous in terms of the underlying execution platform, the diverse concurrently running applications, and the different used scheduling and arbitration policies, modularity is a key requirement. In particular, a methodology should support process composition, scheduling composition, resource composition, as well as the building of components. Process composition supports the analysis of task chains, as events often need to be processed by several consecutive application tasks. Scheduling composition is required, as within one implementation, different scheduling policies are often combined, sometimes even hierarchically within one component. Resource composition ensures that systems consisting of different heterogeneous computing and communication resources can be analyzed. And finally, it should be possible to combine combinations of processes, associated scheduling methods, and architectural elements into components. This way, a designer can associate a performance component to a combined hardware/OS/software module of the implementation that exposes the performance requirements but hides internal implementation details.

- *Short Analysis Time*: If the performance analysis is part of a design space exploration, a short analysis time is important. In addition, the underlying model should allow for easy reconfigurability in terms of application, hardware platform, and allocation, mapping and scheduling.

1.2 Aim of this Thesis

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

"It is possible to formally analyze complex distributed embedded real-time systems with a modular and extensible framework for system level performance analysis that enables the efficient computation of correct and accurate performance analysis results, and that can seamlessly be embedded into an embedded systems' design process. It is further possible to extend the same framework to actively support system design through interface-based design methodologies."

1.3 Thesis Outline and Contributions

This thesis is subdivided into three major parts. In Part I, we extend the modeling and analysis capacities of the Modular Performance Analysis framework, in Part II we extend the applicability of the Modular Performance Analysis framework into the area of interface-based design for embedded real-time systems, and in Part III, we present methods and tools to efficiently compute analysis results within the Modular Performance Analysis framework. In the following we summarize the contents and the main contributions of these three parts.

Part I: Modular Performance Analysis

The first part of this thesis focuses on extending the framework for Modular Performance Analysis by addresses some of the challenges for system level performance analysis described in Section 1.1.2.

Chapter 2: Modular Performance Analysis with Real-Time Calculus

This chapter introduces the framework of Modular Performance Analysis with Real-Time Calculus, which we will call in short the MPA framework. Besides this introduction, the following main contribution can be identified in this chapter.

- We present an extensive case study of a distributed in-car navigation system, where the MPA framework is used to answer design questions that typically arise in the early design phase of such a system. This case study demonstrates how performance analysis with the MPA framework can be seamlessly embedded into a UML-based design process, and it presents the first application of sensitivity analysis within the MPA framework.

Chapter 3: Abstract Components

This chapter addresses the challenge of different processing semantics within a complex embedded system, as described in Section 1.1.2. In particular, we identify the following main contributions of this chapter.

- We introduce a component that enables the modeling and analysis of greedy shapers within the MPA framework, and we present a range of applications of greedy shapers within embedded real-time systems.
- We introduce components that enable the modeling and analysis of tasks with multiple inputs, where the task activation is determined as a boolean function of the event availability on the various inputs.

Chapter 4: Workload Variability and Correlations

This chapter addresses the challenges of complex inputs, variable execution demands, and workload correlations in a complex embedded system, as described in Section 1.1.2. In particular, we identify the following main contributions of this chapter.

- We introduce Type Rate Curves that enable the modeling of event type correlations on event streams, and that allow analysis of embedded systems with event-based workload variability.
- We introduce Event Sequence Automata and Workload Variability Automata that enable the modeling of event type correlations on event streams and functional workload dependencies in components, respectively, and that allow analysis of embedded systems with event-based, as well as functional workload variability.
- We introduce Workload Correlation Curves that enable the modeling of workload correlations between components, and that allow analysis of embedded systems with workload correlations.

Part II: Interface-Based Design

The second part of this thesis focuses on extending the MPA framework to enable interface-based design for complex embedded real-time systems.

Chapter 5: Interface-Based Design with Real-Time Interfaces

This chapter introduces the necessary concepts, models, and methods to enable interface-based design of embedded real-time systems within the MPA framework. In particular, we identify the following main contributions of this chapter.

- We introduce the theory of Real-Time Interfaces that connects the principles of Real-Time Calculus and interface-based design, and that enables interface-based embedded real-time system design within the MPA framework.
- We introduce a component system with Real-Time Interfaces, for interface-based design within the MPA framework.
- We present a range of applications that demonstrate how Real-Time Interfaces simplify and accelerate the design process of embedded real-time systems, and how they could enable interesting on-line applications.

Chapter 6: Design & Analysis of Systems with Hierarchical Scheduling

This chapter addresses the challenge of modeling, analyzing, and designing hierarchical scheduling policies in a complex embedded system, as described in Section 1.1.2. In particular, we identify the following main contributions of this chapter.

- We introduce a component and its interface model that enable modeling, analysis and design of hierarchical schedulers with a top-level TDMA scheduler within the MPA framework. We further present methods for optimal parameter selection of such a TDMA scheduler.
- We introduce a component and its interface model that enable modeling, analysis and design of hierarchical schedulers with a top-level polling server within the MPA framework. We further present methods that support parameter selection of such a polling server.

Part III: Tool Support

The third part of this thesis focuses on efficient computation and tool support for the MPA framework.

Chapter 7: Efficient Computation of Real-Time Calculus

This chapter introduces models and methods to efficiently conduct system level performance analysis and interface-based design within the MPA framework. In particular, we identify the following main contributions of this chapter.

- We introduce a compact representation for a special class of variability characterization curves.
- We introduce methods to efficiently compute various Real-Time Calculus curve operation on these compact variability characterization curves.

Chapter 8: The RTC Toolbox

This chapter concentrates on tool support for the MPA framework. We identify the following main contribution.

- We introduce the Real-Time Calculus (RTC) Toolbox for MATLAB. The RTC Toolbox was developed as part of this thesis to support system level performance analysis and interface-based design of embedded real-time systems with the MPA framework. Moreover, the toolbox was also used to validate the various models and methods presented in this thesis, and to prove their applicability.

Part I

Modular Performance Analysis

2

Modular Performance Analysis with Real-Time Calculus

In the domain of communication networks, powerful abstractions have been developed to model flow of data through a network. In particular Network Calculus [LT01] provides the means to deterministically reason about timing properties of data flows in queuing networks, and can be viewed as a deterministic queuing theory. Real-Time Calculus (RTC) [TCN00] extends the concepts of Network Calculus to the domain of real-time embedded systems, and in [CKT03] a unifying approach to Modular Performance Analysis with Real-Time Calculus has been proposed. It is based on general event, resource and task models, allows for hierarchical scheduling and arbitration, and can take computation as well as communication resources into account.

With Real-Time Calculus, hard upper and lower bounds can be computed to various performance criteria in a real-time system, such as end-to-end delays of event streams, or buffer requirements. The framework of Modular Performance Analysis with Real-Time Calculus hence qualifies to analyze hard real-time systems, and thus clearly distinguishes itself from probabilistic performance estimation methods, or from performance estimation through simulation.

This chapter introduces the framework of Modular Performance Analysis with Real-Time Calculus, which we will call in short the MPA framework in the following. The next section provides an overview on the fundamental elements of the MPA framework and on their relation within the framework. The individual elements are introduced in more detail in Sections 2.2–2.6. An extensive case study of a distributed in-car navigation

system is then presented in Section 2.7, where the MPA framework is used to answer design questions that typically arise in the early design phase of such systems. Besides demonstrating the typical application of the MPA framework for performance analysis, the case study also presents the first application of sensitivity analysis within the MPA framework. The chapter concludes with an overview on related methods for performance analysis in Section 2.8, and a discussion in Section 2.9.

2.1 Overview

The central idea of Modular Performance Analysis (MPA) is, to first build a so-called performance model of the concrete system that bundles all information needed for performance analysis with Real-Time Calculus. The abstract performance model thereby unifies essential information about the environment, about the available computation and communication resources, about the application tasks and dedicated HW/SW components, as well as about the system architecture itself.

Within the system performance model, environment models describe how a system is being used by the environment: how often will events (or function calls) arrive and how much data is provided as input to the system, and how many events and how much data is generated in return by the system and is fed back to the environment. Resource models provide information about the properties of the computing and communication resources that are available within a system, such as processor speed and communication bus bandwidth. And finally, application task models or dedicated HW/SW component models provide information about the processing semantics that is used to execute the various application tasks or to run the dedicated HW/SW components. The system model that is typically obtained following the well-known *Y-Chart* scheme as proposed in [KDVW97] specifies how the different models are put together to build the system performance model. The system model thereby captures information about the applications and the available hardware architecture of the system, and it also defines the mapping of tasks to computation or communication resources and specifies the scheduling and arbitration schemes used on these resources.

Figure 3 presents an overview on the fundamental elements of Modular Performance Analysis and the relations between them. Following, we will introduce the model of the environment in Section 2.2, the model of computation and communication resources in Section 2.3, and the model of application tasks and dedicated HW/SW components in Section 2.4. The construction of the system performance model is presented in Section 2.5, and its analysis is explored in Section 2.6.

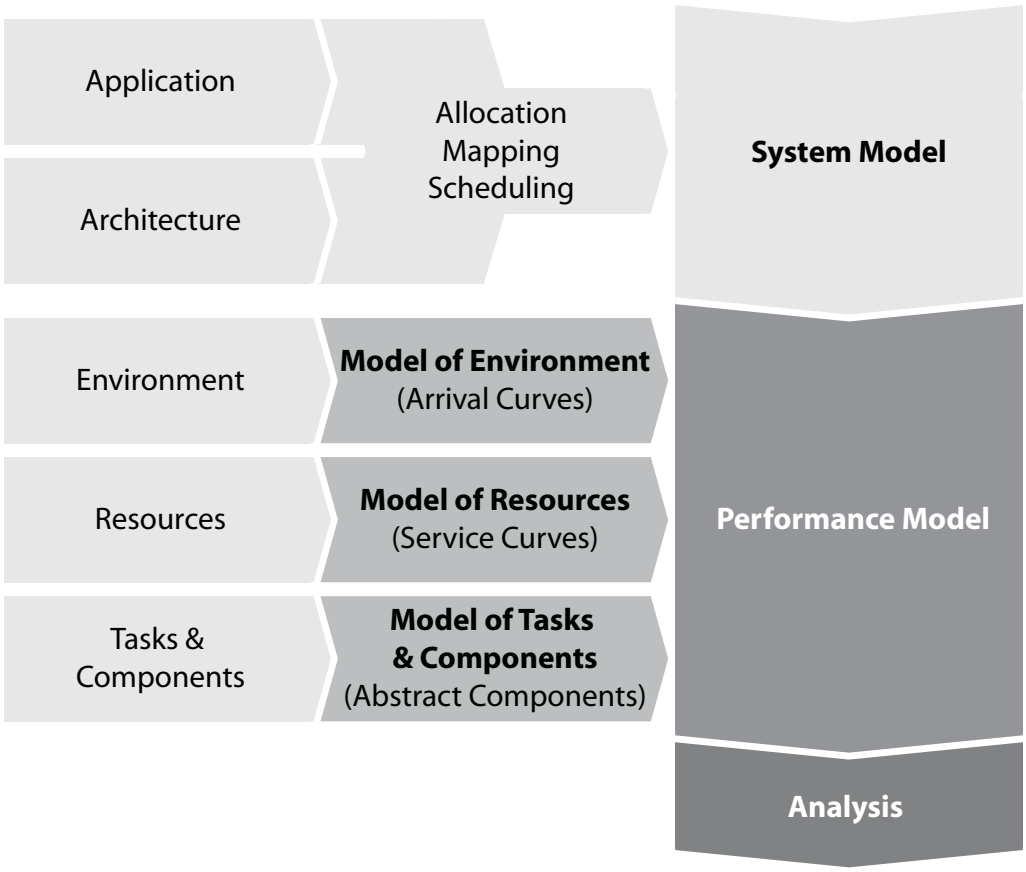


Fig. 3: Elements of Modular Performance Analysis.

2.2 Model of Environment

The environment models describe how a system is being used by the environment: how often will system functions be called, how much data is provided as input to the system, and how much data is generated by the system back to its environment. In the framework of Modular Performance Analysis the concept of arrival curves that was first introduced by Cruz in [Cru91] is used to model event streams of the environment.

2.2.1 Arrival Curves: A General Event Stream Model

A trace of an event stream can conveniently be described by means of a differential arrival function $R[s,t)$ that denotes the sum of events that arrive in the time interval $s \leq \tau < t$, with $R[s,s) = 0$, and with $s,t \in \mathbb{R}$. Sometimes, we will also use the cumulative arrival function $R(\tau)$ that is defined as $R(\tau) = R[0,\tau)$ for all $\tau \geq 0$. While any arrival function R always describes *one* concrete trace of an event stream, a tuple $\alpha(\Delta) =$

$[\alpha^u(\Delta), \alpha^l(\Delta)]$ of upper and lower *arrival curves* provides an event stream model, representing *all* possible traces of an event stream.

For this, the upper arrival curve $\alpha^u(\Delta)$ provides an upper bound on the number of events that are seen on the event stream in *any* time interval of length Δ, and analogously, the lower arrival curve $\alpha^l(\Delta)$ provides a lower bound on the number of events in a time interval Δ. In other words, in any time interval of length Δ there will always arrive at least $\alpha^l(\Delta)$ and at most $\alpha^u(\Delta)$ events on an event stream that is modeled by $\alpha(\Delta)$.

Def. 1: **(Arrival Curves)** *Let $R[s,t)$ denote the number of events that arrive on an event stream in the time interval $s \leq \tau < t$. Then, R, α^u and α^l are related to each other by the following inequality*

$$\alpha^l(t-s) \leq R[s,t) \leq \alpha^u(t-s), \forall t \geq s \geq 0 \tag{2.1}$$

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

Note that in contrary to the conventional definitions of arrival curves by Cruz [Cru91], and Le Boudec and Thiran [LT01], we use differential arrival functions $R[s,t)$ that extend to the whole time domain, i. e. $s, t \in \mathbb{R}$. Cruz, and Le Boudec and Thiran use only cumulative arrival functions $R(\tau)$ that are defined for positive time instances only, i. e. $\tau \geq 0$. With our extended definition, we will be able to obtain tighter performance analysis results, as we will see later in Section 2.4.

2.2.2 Obtaining Arrival Curves

In order to be applicable in the analysis of hard real-time systems, arrival curves must represent guaranteed bounds on the possible occurrence of events on the modeled event streams. In consequence of this requirement, they are typically obtained from a formal specification or description. Arrival curves thereby substantially expand the modeling power of classical deterministic standard event arrival patterns such as sporadic, periodic, periodic with jitter, or others.

Ex. 1: *A common set of standard event arrival patterns that is used in literature can be specified by the parameter triple (p, j, d), where p denotes the period, j the jitter, and d the minimum inter-arrival distance of events in the modeled stream [RZJE02], [Ric05]. Event streams that are specified using these parameters can directly be modeled by the following arrival curves:*

$$\alpha^l(\Delta) = \left\lfloor \frac{\Delta - j}{p} \right\rfloor \tag{2.2}$$

$$\alpha^u(\Delta) = \min\left\{ \left\lceil \frac{\Delta + j}{p} \right\rceil, \left\lceil \frac{\Delta}{d} \right\rceil \right\} \tag{2.3}$$

In Figure 4, the relation between these parameters and the corresponding arrival curves is graphically depicted. Note that in this particular example the jitter is larger than the period which is typical for a so-called event streams with bursts.

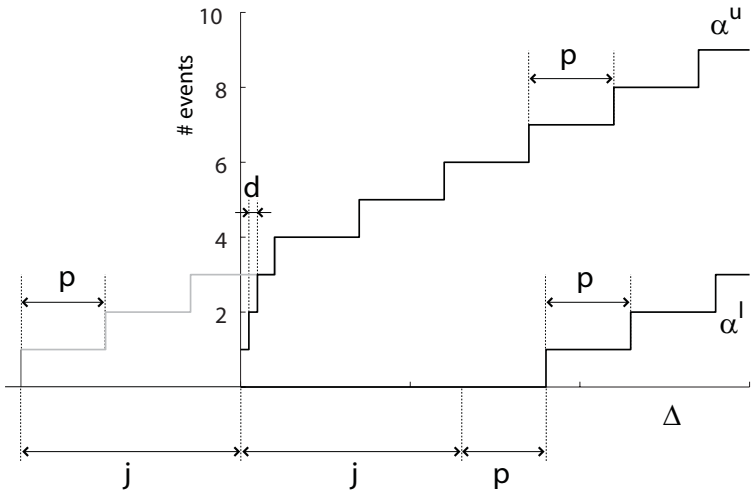


Fig. 4: Relation between the parameter triple (p,j,d) and corresponding arrival curves.

Relations as shown in the previous example can typically also be obtained between any other deterministic event arrival specification and arrival curves. Figure 5 shows some more examples of arrival curves that are obtained from deterministic event arrival specifications.

Sometimes it may also be useful to determine the arrival curves corresponding to a set of finite length event stream traces, obtained for example from observation or simulation. For this, a sliding window approach can be used. Even though the so obtained curves must not be used for analysis of hard real-time systems, they can be useful to analyze systems with soft real-time constraints that are common for example in the area of multimedia applications. For more details see [MLCO04] or [Max05].

2.2.3 Determining the Resource Demand

As defined above, the arrival curves α^u and α^l denote the number of events that arrive on an event stream in any given time interval. For performance analysis we are however not so much interested in the number of events that arrive, but rather in the resource demand that these arriving events produce on a processor or on a HW/SW component. We must therefore introduce *resource based arrival curves*. While event based arrival curves represent the number of arriving events per unit of time interval, the

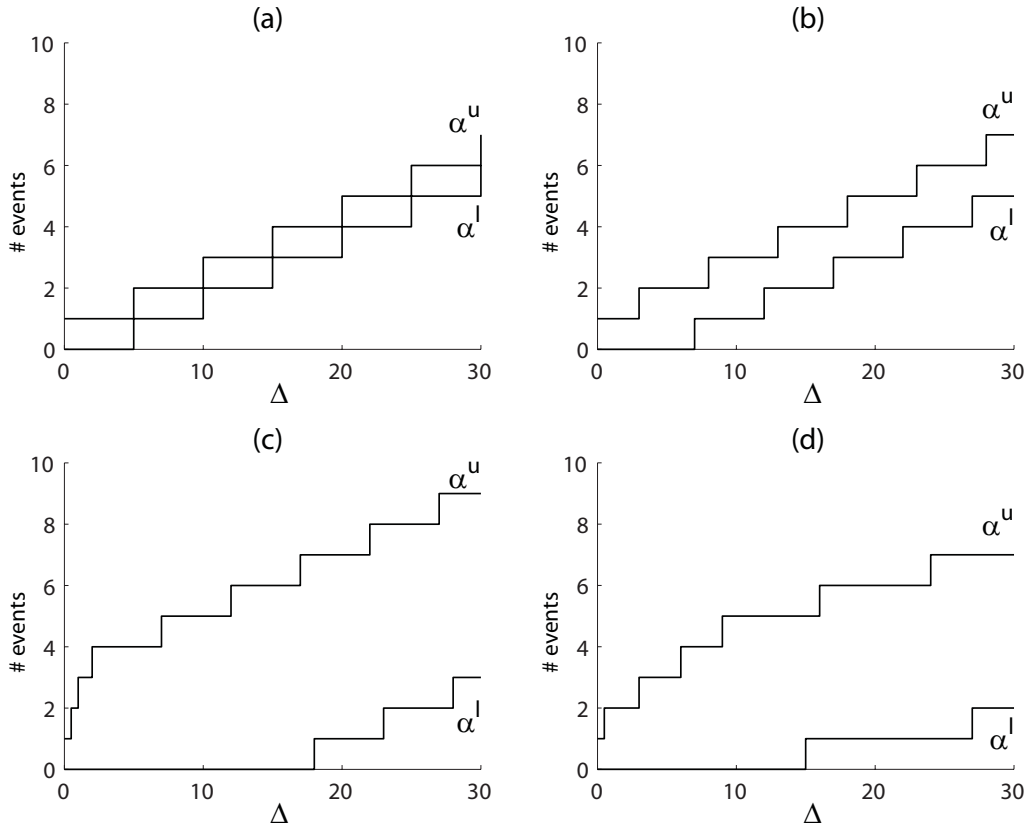


Fig. 5: Examples of arrival curves that are obtained from deterministic event arrival specifications. (a) Models a periodic event stream. (b) Models a periodic event stream with jitter. (c) Models a periodic event stream with bursts. (d) Models an event stream with more complex deterministic timing behavior: the modeled event stream may exhibit short steep bursts, longer lasting less steep bursts, and the maximum long-term period does not equal the minimum long-term period.

resource based arrival curves represent the generated resource demand per unit of time interval.

In the most basic scenario, every arriving event generates the same resource demand on a HW/SW component, i. e. the worst-case execution demand equals the best-case execution demand. Resource based arrival curves can then be obtained directly by multiplying the event based arrival curves with a constant that represents the resource demand of a single event. And analogously, event based arrival curves are obtained by dividing resource based arrival curves by the same constant.

In more complex systems, the events arriving on an event stream may be of one of several different event types, each having a different resource demand, or it may be known that not all events lead to the worst-case

execution demand. In such systems, automata may be used to represent possible arrival patterns of the different event types, and the information captured in these automata may then be used to transform event based to resource based arrival curves. A similar approach may also be used to model system state dependent resource demands, as introduced for example by caches. Chapter 4 will elaborate on these topics.

In this thesis we will typically assume that workload transformations are implicitly performed wherever required. For easier readability we will then use α to denote both event based as well as resource based arrival curves. In Chapter 4 where we want to explicitly distinguish between event based and resource based curves, we will decorate the former with a bar ($\overline{\alpha}$).

2.3 Model of Resources

The resource models provide information about the properties of the computing and communication resources that are available within a system, such as processor speed and communication bus bandwidth. In the framework of Modular Performance Analysis the concept of service curves is used to model resources.

2.3.1 Service Curves: A General Resource Model

Analogously to the differential arrival function $R[s,t)$ that is used to describe a concrete trace of an event stream, the concrete availability of a computation or communication resource can be described by a differential service function $C[s,t)$ with $s,t \in \mathbb{R}$, where $C[s,t)$ denotes the sum of available resource units, e. g. processor cycles or transmittable bits on a bus, in the time interval $s \leq \tau < t$, with $C[s,s) = 0$. Sometimes, we will also use the cumulative service function $C(\tau)$ that is defined as $C(\tau) = C[0,\tau)$ for all $\tau \geq 0$.

And analogously to arrival curves that provide an event stream model, a tuple $\beta(\Delta) = [\beta^u(\Delta), \beta^l(\Delta)]$ of upper and lower *service curves* provides a resource model. The upper service curve $\beta^u(\Delta)$ provides an upper bound on the available resources in *any* time interval of length Δ, and the lower service curve $\beta^l(\Delta)$ provides a lower bound on the available resources in a time interval Δ. And in other words again, in any time interval of length Δ there will always be at least $\beta^l(\Delta)$ and at most $\beta^u(\Delta)$ resource capacity available on a resource that is modeled by $\beta(\Delta)$.

Def. 2: **(Service Curves)** *Let $C[s,t)$ denote the number of processing or communication cycles available from a resource over the time interval $s \leq \tau < t$. Then C, β^u and*

β^l are related by the following inequality

$$\beta^l(t-s) \leq C[s,t) \leq \beta^u(t-s), \forall t \geq s \geq 0 \tag{2.4}$$

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

Note, that the above definition of lower service curves corresponds to the definition of strict service curves in Network Calculus [LT01], while the definition of upper service curves as provided above is not used in [LT01]. Moreover, the above definition also differs from the definition by Le Boudec and Thiran in [LT01] by the fact that the differential service function $C[s,t)$ that extends to the whole time domain, i. e. $s, t \in \mathbb{R}$ is used, while Le Boudec and Thiran use cumulative service functions $C(\tau)$ that are defined for positive time instances only, i. e. $\tau \geq 0$.

2.3.2 Obtaining Service Curves

Again in order to be applicable in the analysis of hard real-time systems, service curves must also represent guaranteed bounds on the availability of the modeled resources. In consequence of this requirement, they are typically obtained from a formal specification or description, using data sheets or analytically derived properties.

Ex. 2: *In the simplest case of an unloaded processor, whose capacity we measure in available processing cycles per time unit, both the upper and the lower resource curves are equal and are represented by straight lines* $\beta^u(\Delta) = \beta^l(\Delta) = f \cdot \Delta$, *where* f *equals the processor speed, i. e. the number of available processing cycles per time unit.*

Figure 6 shows some examples of service curves that model the resource availability on processors or communication channels.

For the analysis of soft real-time systems it may sometimes also be useful to use a sliding window approach to determine the service curves corresponding to a measured resource availability.

2.4 Model of Tasks and Components

The application task or dedicated HW/SW component models provide information about the processing semantics that is used to execute the various application tasks or to run the dedicated HW/SW components.

In an embedded system, an incoming event stream is typically processed by a sequence of tasks and components, that we all will interpret

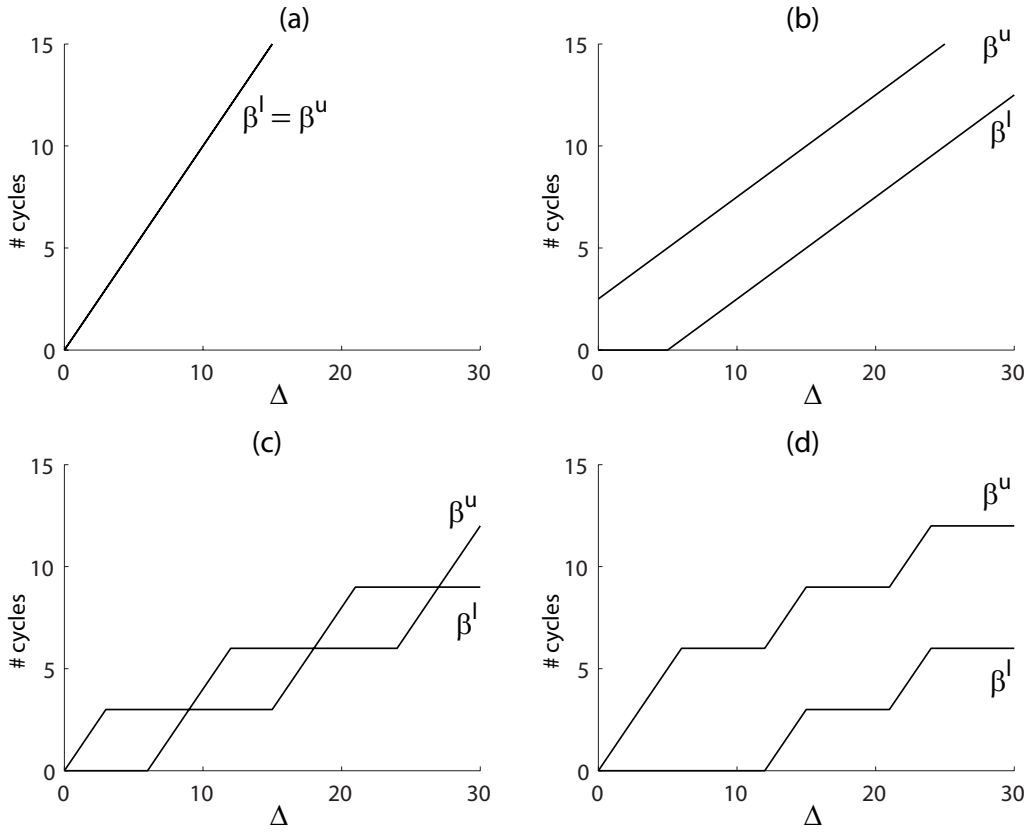


Fig. 6: Examples of service curves. (a) Models a resource with full availability. (b) Models a bounded delay resource as defined in [MFC01]. (c) Models the resource availability of one slot on a time division multiple access (TDMA) resource. (d) Models a periodic resource as defined in [SL03].

as tasks on a task chain that are executed on possibly different hardware resources.

Figure 7(a) shows such a component. A trace of an event stream, described by R, enters the component and is processed using a resource whose availability is described by C. After being processed, the events are emitted on the output of the component, resulting in an outgoing event stream trace, described by R', and the remaining resources that were not consumed to process the event trace R are made available to other components and are described by an outgoing resource availability C'.

The relations between R, C, R' and C' depend on the processing semantics of the component. The outgoing event stream R' will typically not equal the incoming event stream R, as it may, for example, exhibit more (or less) jitter. Analogously, C' will differ from C.

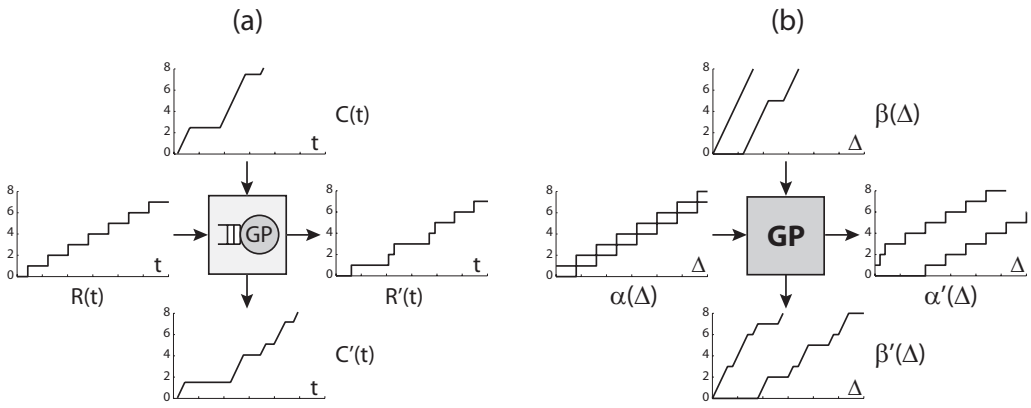


Fig. 7: (a) A concrete component, processing an event stream on a resource. (b) An abstract component, processing an abstract event stream on an abstract resource.

In the MPA framework, we model such a component as an abstract component as shown in Figure 7(b). Here, an abstract event stream $\alpha(\Delta)$ enters the abstract component and is processed using an abstract resource $\beta(\Delta)$. The output is then again an abstract event stream $\alpha'(\Delta)$, and the remaining resources are expressed again as an abstract resource $\beta'(\Delta)$.

Internally, such an abstract component is specified by a set of functions, that relate the incoming arrival and service curves to the outgoing arrival and service curves:

$$\alpha' = f_\alpha(\alpha, \beta) \tag{2.5}$$

$$\beta' = f_\beta(\alpha, \beta) \tag{2.6}$$

For a given abstract component, these relations f_α and f_β depend on the processing semantics of the modeled concrete component, and must be determined such that $\alpha'(\Delta)$ correctly models the event stream with event trace $R'(t)$ and that $\beta'(\Delta)$ correctly models the resource availability $C'(t)$.

Ex. 3: *As an example of an abstract component, consider a concrete component that is triggered by the events of an incoming event stream. A fully preemptable task is instantiated at every event arrival to process the incoming event, and active tasks are processed in a greedy fashion in FIFO order, while being restricted by the availability of resources. Such a component can be modeled as an abstract component with following internal relations*[1] *[CKT03] that are proved in*

[1]See Appendix A.1 for a definition of $\otimes$, $\oslash$, $\overline{\otimes}$ and $\overline{\oslash}$

Appendix A.4:

$$\alpha'^{u}_{GP} = \min\{(\alpha^u \otimes \beta^u) \oslash \beta^l, \beta^u\} \quad (2.7)$$

$$\alpha'^{l}_{GP} = \min\{(\alpha^l \oslash \beta^u) \otimes \beta^l, \beta^l\} \quad (2.8)$$

$$\beta'^{u}_{GP} = (\beta^u - \alpha^l) \overline{\oslash} 0 \quad (2.9)$$

$$\beta'^{l}_{GP} = (\beta^l - \alpha^u) \overline{\otimes} 0 \quad (2.10)$$

Note that these relations are only valid with arrival and service curves as defined in this thesis, that are based on differential arrival and service functions $R[s,t)$ and $C[S,T)$ that extend to the whole time domain. With the arrival and service curves defined by Cruz [Cru91] and by Le Boudec and Thiran [LT01], that are based on cumulative arrival and service functions $R(\tau)$ and $C(\tau)$ that are only defined for the positive time domain, the above tight relations are not valid.

Components with the processing semantics described in the above example are very common in the area of real-time embedded systems, and we will refer to them as a *Greedy Processing (GP)* components.

To model a component with different processing semantics, we have to determine the appropriate internal relations f_α and f_β to obtain a corresponding abstract component. In Chapter 3 we will present a HW component with different processing semantics and we will establish the appropriate internal relations.

2.5 System Performance Model

At this point, we know how to model event streams, computation and communication resources, as well as single application tasks and HW/SW components. But in order to analyze performance criteria of a system, we need to build a *system performance model*. The system performance model thereby captures information about the applications and the available hardware architecture. Moreover it also reflects the mapping of tasks to computation or communication resources and specifies the scheduling and arbitration schemes used on these resources, as defined by the system model.

To obtain the performance model of a system, we first need to model all event streams that trigger the system, all computation and communication resources that are available to the system, as well as all tasks and components in the system, using the corresponding abstract representations as described in the preceding sections. Then, by correctly interconnecting the arrival and service inputs and outputs of all these models, we obtain the system performance model. An example is depicted in Figure 15.

2.5.1 Flow of Data

The arrival inputs and outputs in the system performance model are interconnected to reflect the flow of data in the system. An interconnection between two components thereby signifies that the events created on the output of the first component trigger the connected subsequent component.

The flow of data is thereby not limited to one-to-one connections. The events created on the output of a single application task or HW/SW component may serve as input to multiple subsequent tasks or components. And analogously, a single application task or HW/SW component may be triggered by the output of multiple preceding tasks or components. While the one-to-many connections are straightforward to model, the many-to-one connections are more involved and Chapter 3 will elaborate on these.

2.5.2 Scheduling and Arbitration

The service inputs and outputs in the system performance model are interconnected to reflect the resource sharing policies in the system. To elaborate on these service interconnections, suppose that several tasks of a system are allocated to the same resource. In the concrete system, these tasks share this resource according to a scheduling or arbitration policy. In the performance model, this scheduling or arbitration policy on a resource can then be modeled by the way the abstract resources β are distributed among the different abstract tasks.

Consider for example preemptive fixed priority scheduling: a task *A* with the highest priority may use all available resources of a CPU, whereas a task *B* with the second highest priority only gets the resources that were not consumed by *A*. This resource sharing policy is modeled in the performance model by using the service curves β'_A that exit the abstract GP component *A* as input to the abstract GP component *B*.

For some other scheduling or arbitration policies, such as generalized processor sharing (GPS) [PG93, PG94] or time division multiple access (TDMA), the available resources must be distributed differently, while for some policies, such as earliest deadline first (EDF) or non-preemptive scheduling, different abstract components, with tailored internal relations (2.5) and (2.6), must be established. Some examples of how to model different scheduling policies are depicted in Figure 8.

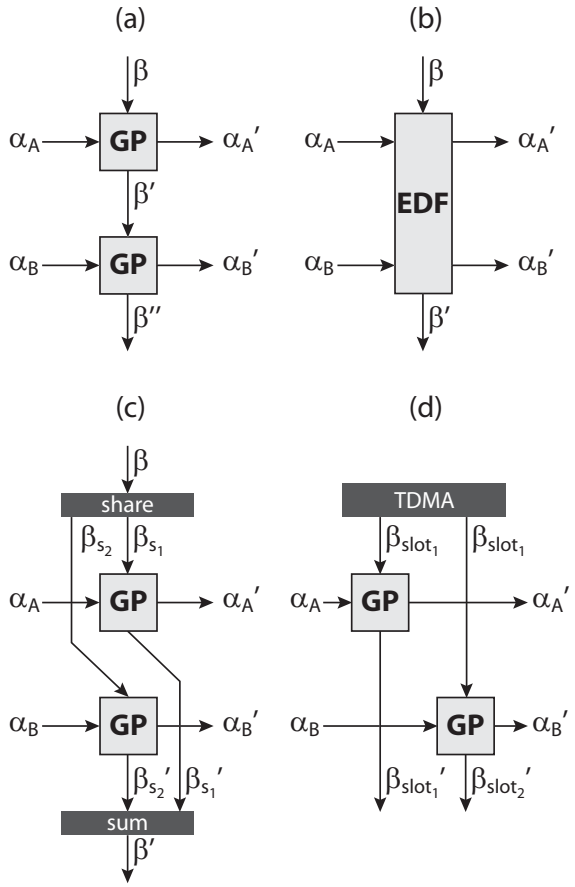


Fig. 8: Modeling of various scheduling and arbitration policies in the system performance model. (a) Tasks with preemptive fixed priority (FP) scheduling. (b) Tasks with earliest deadline first (EDF) scheduling. (c) Tasks with generalized processor sharing (GPS) scheduling. (d) Tasks with time division multiple access (TDMA) scheduling.

2.6 Analysis

After interconnecting all models of a system to the system performance model as described in the previous section, this performance model captures all the information that builds the basis for performance analysis. Various performance criteria, such as end-to-end delay guarantees or buffer requirements can be computed analytically in the performance model. The exact analysis methods may thereby slightly vary for different abstract components but remains deterministic at all times. Following, we present the performance analysis methods for GP components. The analysis methods for other abstract components are mostly very similar or even equal to these.

2.6.1 Performance Analysis

When an event stream with arrival curves α is processed by a GP component on a resource with service curve β, then the *maximum delay* d_{max} experienced by any event on the event stream is bounded by [LT01]:

$$d_{max} \leq \sup_{\lambda \geq 0} \left\{ \inf\{\tau \geq 0 : \alpha^u(\lambda) \leq \beta^l(\lambda + \tau)\} \right\} \stackrel{def}{=} Del(\alpha^u, \beta^l) \tag{2.11}$$

On the other hand, the maximum buffer space b_{max} that is required to buffer an event stream with arrival curve α in the input queue of a GP component on a resource with service curve β is bounded by [LT01] :

$$b_{max} \leq \sup_{\lambda \geq 0} \{\alpha^u(\lambda) - \beta^l(\lambda)\} \stackrel{def}{=} Buf(\alpha^u, \beta^l) \tag{2.12}$$

In Figure 9, the relations between α, β, d_{max} and b_{max} are depicted graphically. From this figure, we see that d_{max} and b_{max} are bounded by the maximum horizontal and maximum vertical distance between the upper arrival curve and the lower service curve respectively. This corresponds to the intuition, that d_{max} and b_{max} occur when the maximum load arrives at the time of minimum resource availability.

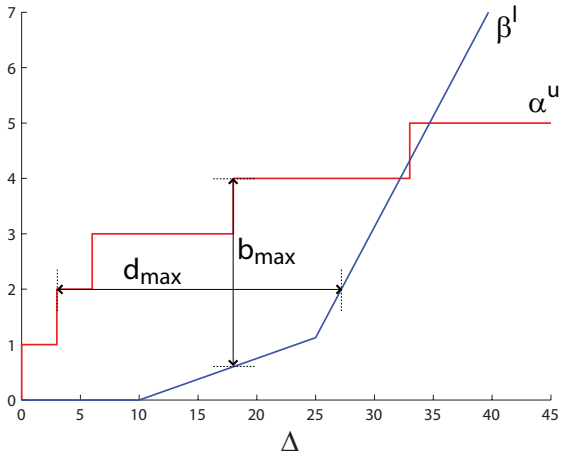


Fig. 9: Delay and backlog obtained from arrival and service curves

Using (2.11) and (2.12), the total end-to-end delay experienced by an event at the system or the total buffer requirement at a resource can be computed as the sum of the single delays and the single buffer requirements at the various abstract components, respectively.

Besides this strictly modular analysis approach the MPA framework of also enables a partly holistic analysis of certain subsystems. In particular,

when an event stream is processed by a sequence of tasks or components, we can exploit the phenomenon known as "Pay Bursts Only Once" [LT01], and the end-to-end delay guarantee can be tightened to:

$$d_{max} \leq Del(\alpha^u, \beta_1^l \otimes \beta_2^l \otimes \ldots \otimes \beta_n^l) \tag{2.13}$$

and analogously when the buffers of several consecutive tasks or components use the same shared memory, the total required buffer space can be tightened to:

$$b_{max} \leq Buf(\alpha^u, \beta_1^l \otimes \beta_2^l \otimes \ldots \otimes \beta_n^l) \tag{2.14}$$

Besides enabling the computation of the various end-to-end delay guarantees and buffer requirements in a system, the system performance model may also provide other interesting insights to a system, that may for example be obtained by analyzing the characteristics of the outgoing service curves. This analysis may among other things expose the utilization of the various computation or communication resources in the system.

2.6.2 Sensitivity Analysis

Besides only computing the various performance metrics of a system at a specific design point, it is often also useful to perform a sensitivity analysis for the various metrics. The results of such an analysis will allow to identify the bottlenecks of a system design and will consequently help to come up with more robust designs.

In [RJE05] Racu et al. present a method for sensitivity analysis that aims at determining the minimum or maximum permissible value for various system parameters. The presented method is based on a binary search algorithm and is also directly applicable to the MPA framework.

An alternative approach to sensitivity analysis aims at determining the partial derivatives of performance metrics towards changes of system parameters at the specific design point. Since it is typically not possible to determine these partial derivatives analytically, we can instead determine them by conducting a set of successive system performance analyses with infinitesimal parameter variations around the design point.

2.7 Case Study

The case study presented in this section is inspired by a system architecture definition study for a distributed in-car radio navigation system. Such a system typically executes a number of concurrent applications that share a common platform. Nevertheless, each application might

have hard individual performance requirements that need to be met by the platform. During the system definition phase, several candidate platform architectures might be proposed by the engineers and the system architect needs to evaluate each one. Typical questions that need to be answered are: (1) does this platform meet the performance requirements of all applications (2) how robust is the platform with respect to changes in application or architecture parameters and (3) can components be replaced in the architecture by cheaper (and less powerful) components to save cost but still ensuring to meet the performance criteria of all applications? We present the applications and the architecture candidates in Section 2.7.1. In Section 2.7.2 we briefly show how these are modeled in the MPA framework. And finally, in Section 2.7.3 we will analyze typical design questions, as the ones mentioned above.

2.7.1 A Distributed In-Car Navigation System

An overview of the system under consideration is presented in Figure 10, it is composed of three main clusters of functionality:

Man-Machine Interface The man-machine interface (MMI) takes care of all interaction with the user, such as handling key inputs and graphical display output.

Navigation Functionality The navigation functionality (NAV) is responsible for destination entry, route planning and turn-by-turn route guidance giving the driver both audible and visual advices. The navigation functionality relies on the availability of a map database, typically stored on a CD or DVD, and positioning information, e. g. , speed and GPS. The latter is not shown here.

Radio Functionality The radio functionality (RAD) is responsible for basic tuner and volume control as well as handling of traffic information services such as RDS TMC (Radio Data System / Traffic Message Channel). RDS TMC is broadcast along with the audio signal of radio channels.

The key question that is investigated in this case study is how to distribute the functionality over the available resources, such that we meet the global timing requirements. To achieve this goal, the following steps are taken:

1. Identify key usage scenarios and system functions.
2. Quantify event rates, message sizes and execution times.
3. Identify resources and their communication structure.

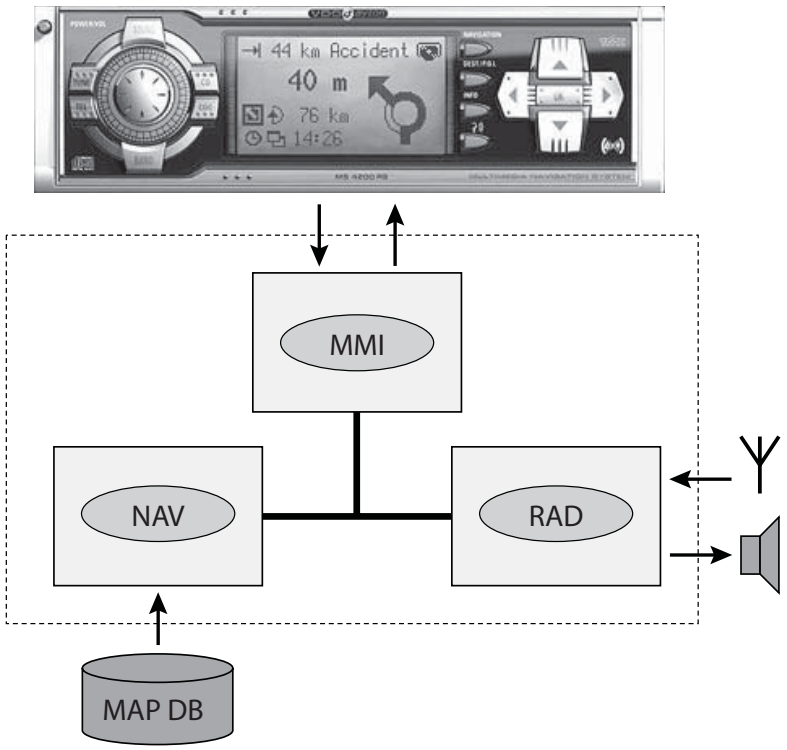


Fig. 10: High-level overview of a distributed radio navigation system

4. Quantify resource and communication capacities.

5. Compose the MPA performance models, calculate and evaluate.

A general description of a new product is typically made during the initial phase of an industrial product creation process. For example, an *Operational Concept Description* from the IEEE 12207 system life cycle standard [IEE98] may be produced. Such a document does not only list functional and non-functional requirements, boundary conditions and other restrictions for the design, it should also contain high-level use-cases. These use-cases are the starting point for the design of the system architecture. During the steps 1 and 2 of the recipe described above, the use-cases and associated sequence diagrams are first analyzed and annotated for MPA analysis. Although there is no principle limit to the amount of scenarios that can be analyzed, it is not uncommon to first concentrate on those scenarios that are expected to have the highest impact on the set of requirements to be met. It is the system architect who makes this decision, often based on previous experience. In our case study, we have selected three distinctive scenarios:

Change Volume The user turns the rotary button and expects instanta-

neous audible feedback from the system. Furthermore, the visual feedback (the volume setting on the screen) should be timely and synchronized with the audible feedback. This seemingly trivial use-case is actually quite complex because many components are affected. Changing volume might involve commanding a digital signal processor (DSP) and an amplifier in such a way that the quality of the audio signal is maintained while changing the volume. This scenario is shown in detail in Figure 11. Note that three operations are identified, *HandleKeyPress*, *AdjustVolume* and *UpdateScreen*. Execution times, event rates and message sizes are estimated and annotated in the sequence diagram together with the principle timing requirements applicable to this scenario.

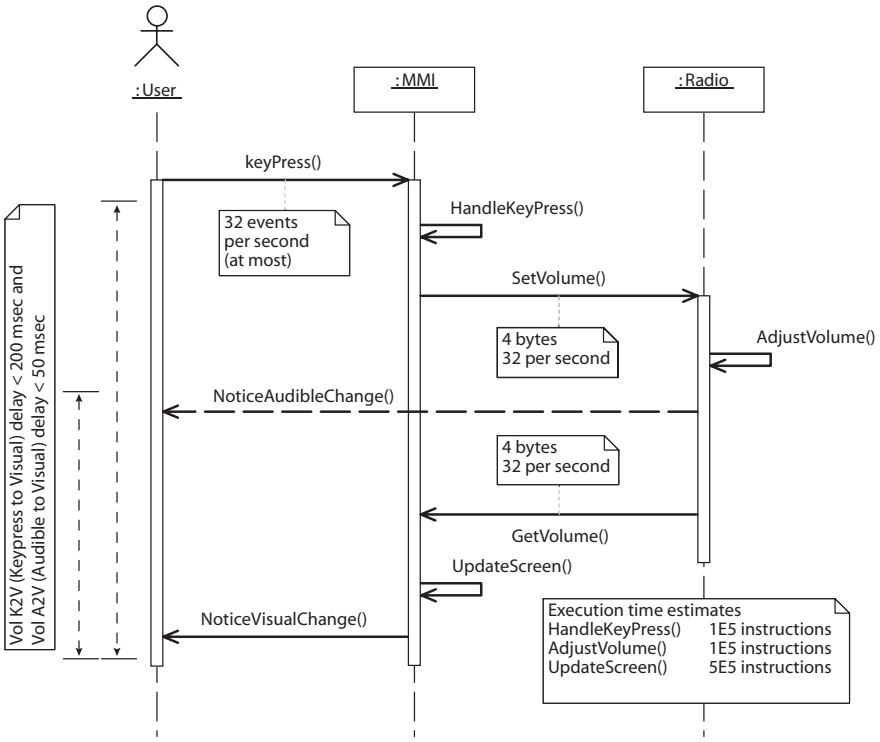


Fig. 11: Annotated sequence diagram for the *Change Volume* scenario.

Address Look-up Destination entry is supported by a smart interface. By turning a knob the user can move from letter to letter; by pressing it the user will select the currently highlighted letter. The map database is searched for each letter that is selected and only those letters in the on-screen alphabet are enabled that are potential next letters in the list. This scenario is shown in detail in Figure 12. Note that the *DatabaseLookup* operation is expensive compared to

the other operations and that the size of the output value of the operation is 16 times larger than the input message.

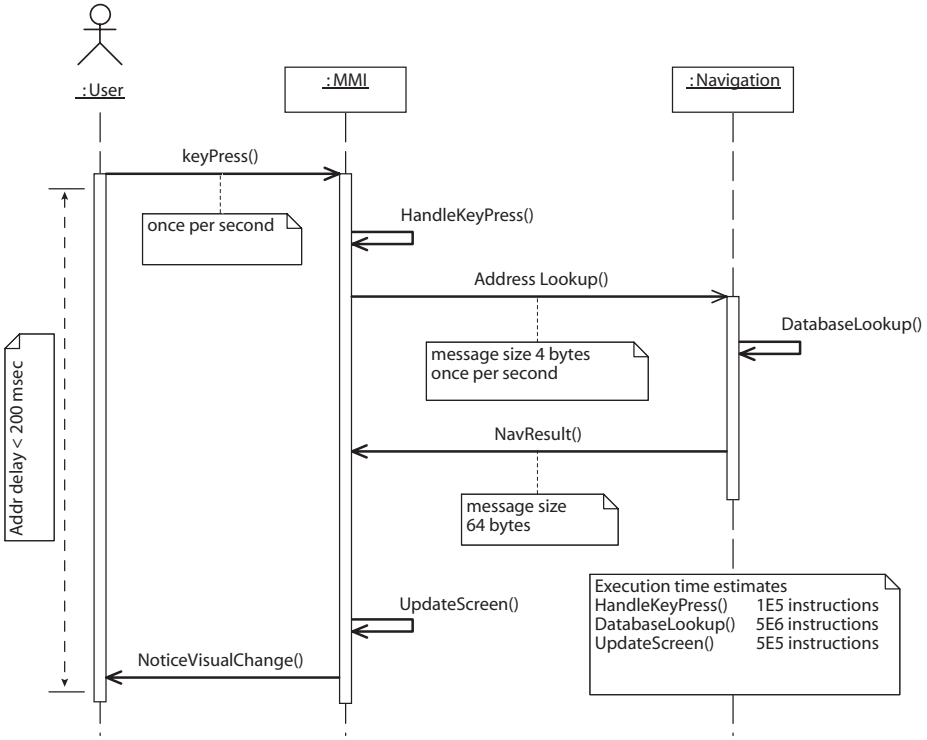


Fig. 12: Annotated sequence diagram for the *Address Look-up* scenario.

TMC Message Handling Digital traffic information is very important for in-car radio navigation systems. It enables features such as automatic re-planning of the planned route in case a traffic jam occurs ahead. It is also increasingly important to enhance road safety by warning the driver, for example when a ghost driver is spotted on the planned route. RDS TMC is such a digital traffic information service. TMC messages are broadcast by radio stations together with stereo audio sound. RDS TMC messages are encoded: only problem location identifiers and message types are transmitted. The map database is accessed to translate these identifiers and to construct human readable text. The TMC message handling scenario is shown in Figure 13.

The scenarios sketched above have an interesting property: they can occur in parallel. RDS TMC messages must be processed while the user changes the volume or enters a destination. However *Change Volume* and *Address Look-up* can not occur at the same time because they share a common resource: the rotary button is used for both. The architecture shown

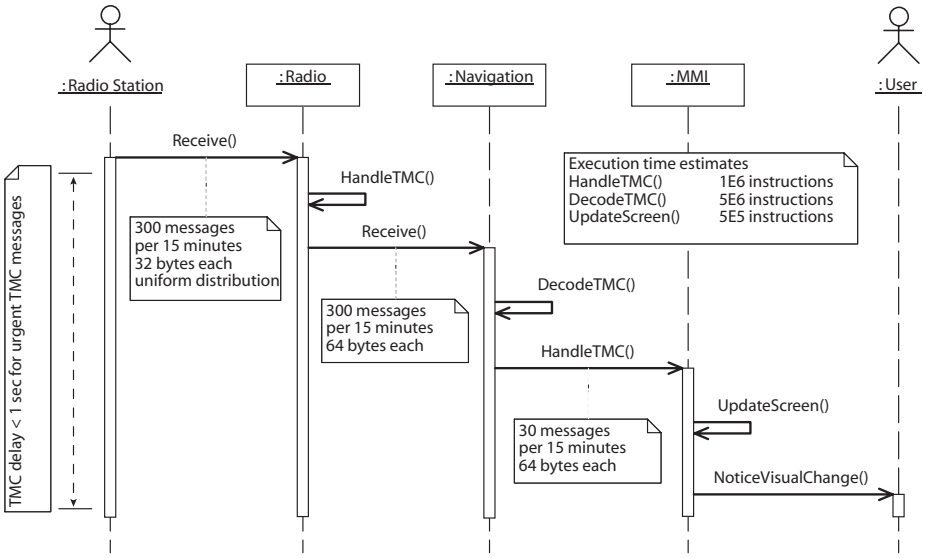


Fig. 13: Annotated sequence diagram for the *TMC Message Handling* scenario.

in Figure 10 suggests to assign the three clusters of functionality each to its own processing unit. The computation resources are interconnected by a single communication bus. Does this architecture meet our requirements and is it the best architecture for our applications?

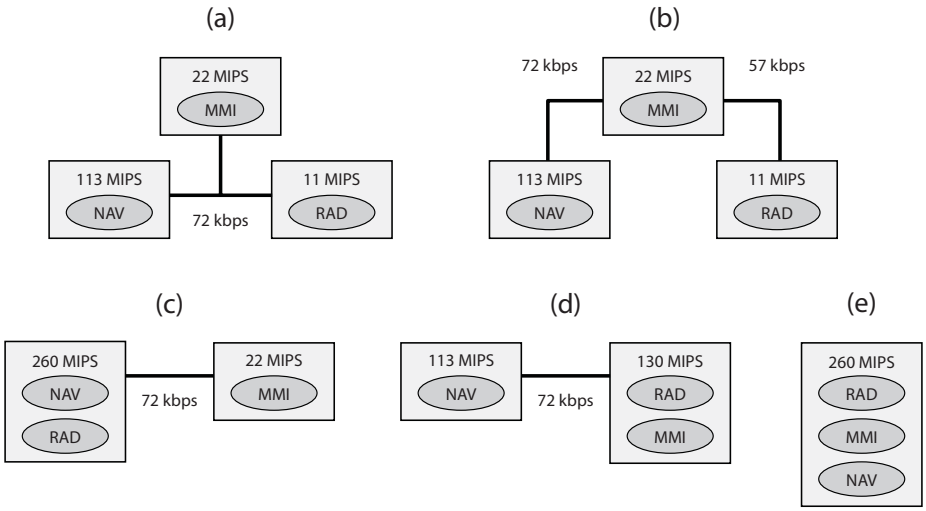


Fig. 14: Alternative system architecture proposals to explore.

Figure 14 shows that there are many more potential architectures that might be applicable. Note that the capacity of the resource units and communication infrastructure is quantified, completing step 3 and 4 of

our recipe. Observe that architecture (b) can only be evaluated if we introduce an additional task on the MMI resource that transfers the data from one communication link to the other, in the case that NAV wants to communicate to RAD or vice versa.

2.7.2 Constructing the System Performance Models

Sufficient information is now available to construct the MPA performance models for each architecture. The model for architecture (a) is shown in Figure 15.

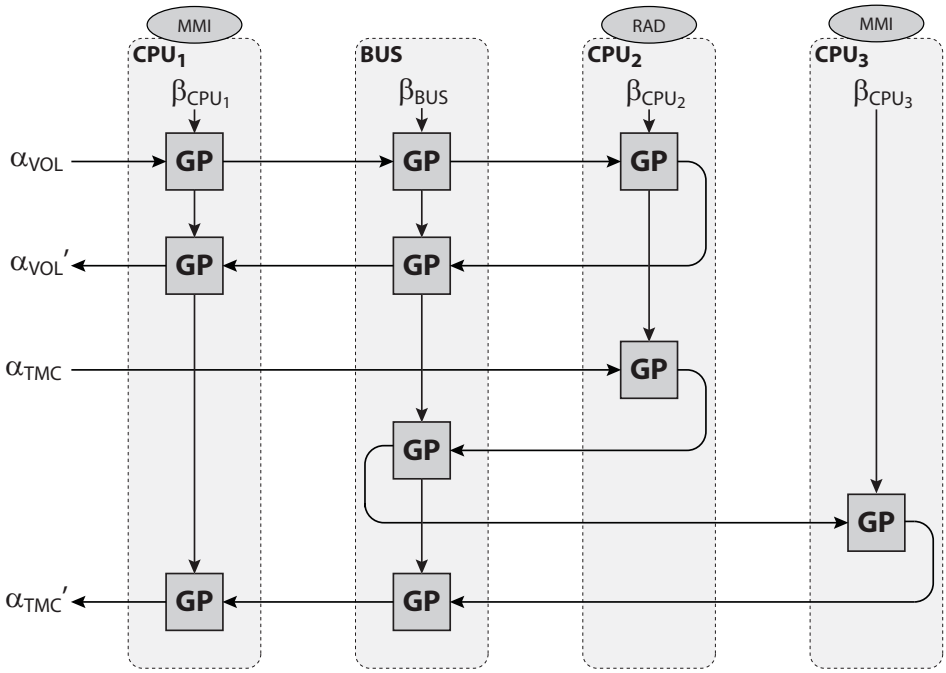


Fig. 15: Performance model for system architecture (a) of Figure 14

The data flow of the sequence diagrams can be followed in the MPA model. For example the *Change Volume* scenario from Figure 11. Events arrive at the MMI where *HandleKeyPress* is executed and the result is forwarded, via the communication bus, to RAD. *AdjustVolume* is executed and the result is sent back to MMI via the same communication bus. Finally, *UpdateScreen* is executed and the scenario is completed. The load scenario data α is extracted from the annotations in the sequence diagram. The resource model data β on the other hand is extracted from the informal deployment diagrams shown in Figure 14.

As described in Section 2.5, the order in which the service inputs and output of the components are connected in the MPA performance model

determine their priority. In this case, the *Change Volume* scenario is assigned a higher priority than *Handle TMC*. The system architect decides the initial priority setting again based on experience. This does not hinder the evaluation in any way, since the priorities are easily changed by rearranging the vertical order of the scenarios. A MPA performance model must be constructed for each proposed architecture. This is normally a simple tasks because it merely involves reconnecting event flows and service flows between abstract components.

2.7.3 System Analysis

In this section, we will look at some typical design problems that occur during the early phases of a system design cycle.

For a correct interpretation of the results, we need to remember that in order to be applicable for the analysis of hard real-time systems, the MPA framework is designed to compute hard upper and lower bounds for all system characteristics. While these upper and lower bounds are always hard, they are in general not tight (exact). So, the analysis performed is conservative and the computed maximum delays are therefore hard upper bounds to the real maximum delays in the real system.

Due to this conservative approach, it may be that we reject a system architecture that would fulfill all system requirements in reality, but for which the analysis cannot guarantee the fulfillment of all system requirements. The other way around, we can guarantee that any system architecture accepted by the analysis fulfills all system requirements in reality.

To compute the end-to-end delay of an event stream in a given system architecture, we first construct the performance model of the given system architecture as described in the previous section. By applying the transformations given in formulas 2.7 - 2.10 at every abstract component of the performance model, we eventually end up with the output event streams and the remaining system resources. We then use formula 2.11 to compute the upper bound of the maximum delay of an event stream at every performance component it passes in the performance model. Finally, we sum up all these delays or use formula 2.13 to obtain a hard upper bound on the maximum end-to-end delay of the event stream in the given system architecture.

We will now present three typical design problems and show how they are analyzed:

Prob. 1: *In Figure 14, five different system architectures are proposed for the in-car radio navigation system. How do these system architectures compare with respect to the different end-to-end delay requirements of the three use-cases?*

We build the performance model for the *Change Volume* & *TMC Message Handling* situation (depicted in Figure 15), as well as the performance model for the *Address Lookup* & *TMC Message Handling* situation. For both models, we compute the upper bounds to the end-to-end delay of every event stream as described in the last section, and we take the end-to-end delays obtained from the two analysis runs (for the TMC delay, we take the maximum value of the two runs). From the results presented in Figure 16, we see that all proposed system architectures fulfill the requirements on the different maximum end-to-end delays that are specified in the sequence diagrams in Figures 11, 12 and 13. The results also suggest that architectures (D) and (E) process the input data to the system particularly fast. This may be explained partly by the reduced communication overhead in these architectures, but most probably, these architectures are also over-dimensioned.

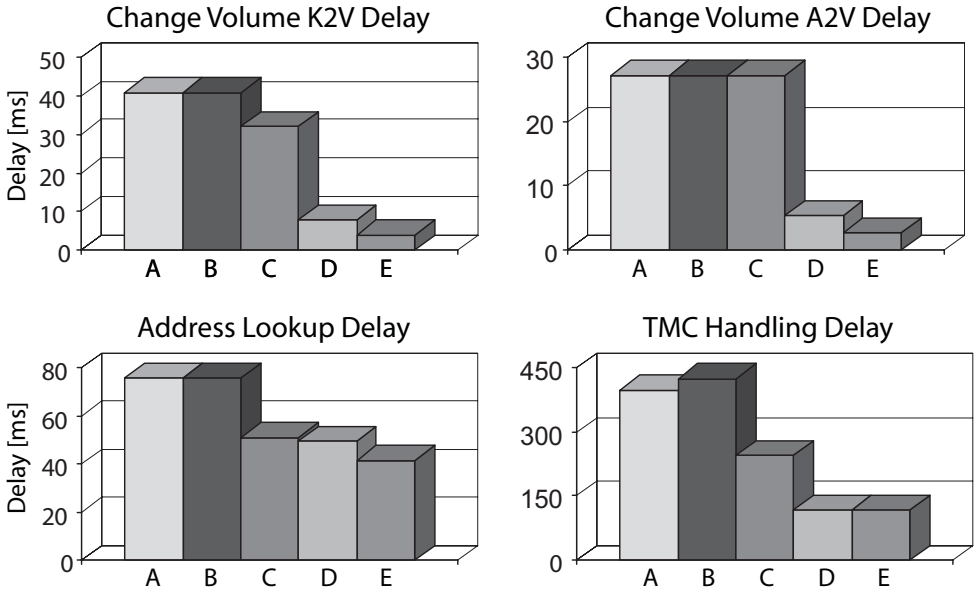


Fig. 16: Maximum end-to-end delays for the five system architectures in Figure 14.

Prob. 2: *Suppose that the in-car radio navigation system is implemented using architecture (A). How robust is this architecture? Where is the bottleneck of this architecture?*

To investigate the robustness of architecture (A), we first compute its sensitivity towards changes in the input data rates. These sensitivity results are shown in Figure 17. The height of the columns in this figure depict the increase of end-to-end delays relative to the respective specified maximum end-to-end delays, in dependence to increasing input data

rates. For example, the tallest column in Figure 17 shows us that if we increase the data rate of the *Change Volume* scenario slightly (i. e. by 3 %, to 33.3 *events*/*s*), the end-to-end delay of the TMC message handling increases by 1.14 % of its specified maximum end-to-end delay (i. e. 1.14 % of 1000 *ms* or 11.4 *ms*).

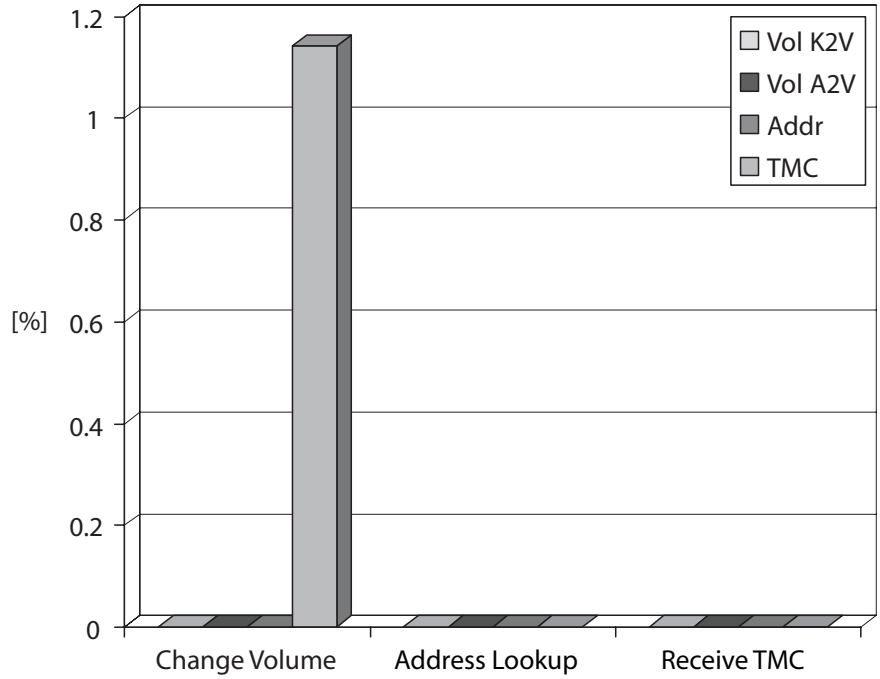


Fig. 17: Sensitivity of the end-to-end delays in architecture (A) towards changes in the input data rates.

From the results shown in Figure 17, we see that architecture (A) is very sensitive towards increasing the input data rate of the *Change Volume* scenario, while increasing the input data rate of the *Address Look-up* and the *TMC Message Handling* scenarios do not affect the response times at the specific design point. And in fact, further analysis reveals that in order to guarantee all system requirements, we must not increase the input data rate of the *Change Volume* scenario by more than 7 %, while we could increase the input data rate of the other two scenarios by a factor of more than 20.

After investigating the system sensitivity towards changes in the input data rates, we investigate the system sensitivity towards changes in the resource capacities. These sensitivity results are shown in Figure 18. The height of the columns in this figure depicts the increase of end-to-end delays relative to the respective specified maximum end-to-end delays, in dependence to decreasing resource capacities. For example, from the

tallest column in Figure 18 we know that if we decrease capacity of the MMI processor by 1 % (i. e. to 21.78 MIPS), the end-to-end delay of the TMC message handling increases by 3.22 % of its specified maximum end-to-end delay (i. e. 3.22 % of 1000 *ms* or 32.2 *ms*).

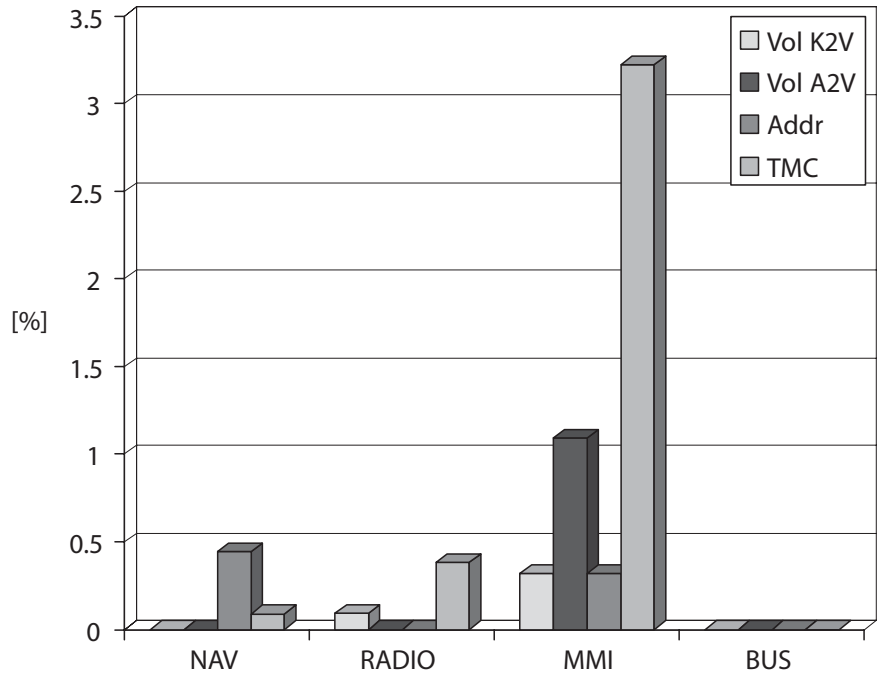


Fig. 18: Sensitivity of the end-to-end delays in architecture (A) towards changes in the resource capacities

From the results shown in Figure 18, we see that architecture (A) is most sensitive towards the capacity of the MMI processor. This suggests that the MMI processor is a potential bottleneck of architecture (A). To investigate this further, we compute the end-to-end delay of the *TMC Message Handling* for different MMI processor capacities. The results of these computations are shown in Figure 19.

From Figure 19, we see that indeed at its given operation point, the end-to-end delay of the *TMC Message Handling* in architecture (A) is very sensitive towards changes of the MMI processor capacity. And the analysis reveals that with a decrease of the MMI processor capacity to 89 % of its initial capacity, we cannot guarantee finite response times anymore.

To sum up, the above analysis results suggest that increasing the capacity of the MMI processor would make architecture (A) more robust. To support this statement, we individually increase the capacity of each resource by 20 %, and we then analyze how much we can increase the input data rate of the *Change Volume* scenario while still fulfilling the require-

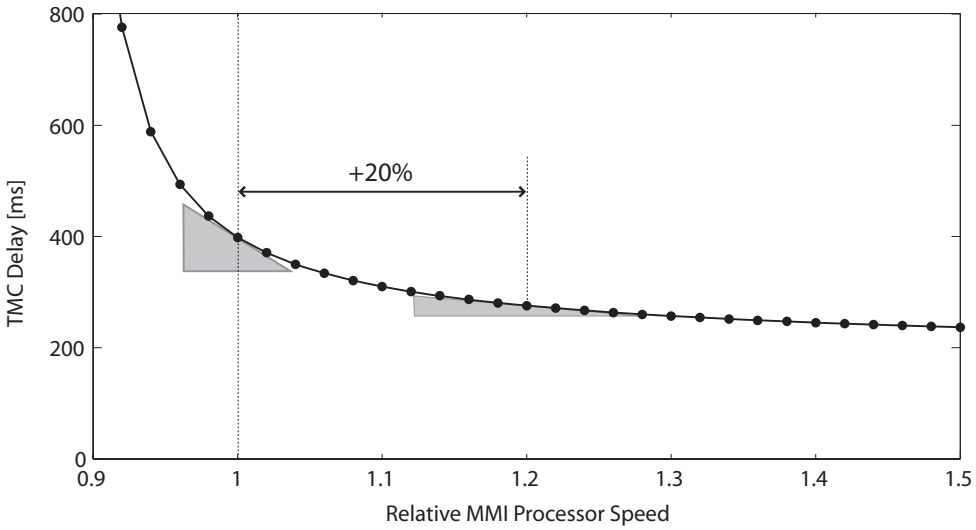


Fig. 19: End-to-end delay of the *TMC Message Handling* as a function of the relative MMI processor speed in architecture (A).

ments. Remember, with the initial resource capacities, we can increase the data rate of the *Change Volume* scenario by 7 % and the data rate of the other two scenarios by a factor of more than 20 while still guaranteeing all requirements. From this analysis, we learn that increasing the resource capacities of the RAD processor, the NAV processor and the BUS does not allow to increase the input date rate of the *Change Volume* scenario more than with the initial capacities, while increasing the MMI processor capacity allows us to increase the data rate of the *Change Volume* scenario by 60 %.

Prob. 3: *Suppose system architecture (D) is chosen for further investigation. The results of Problem 1 indicate, that architecture (D) is probably over-dimensioned. How should the two processors in this system architecture be dimensioned, to obtain an economic system that still fulfills the end-to-end delay requirements of all scenarios?*

We compute the upper bound to the end-to-end delay of every event stream in architecture (D) for different processor capacities. The results are shown in Figure 20.

In the plots in Figure 20, the NAV processor capacity is varied in steps of 5 % from 100 % down to 10 % of its initial capacity. At the same time, the MMI/RAD processor capacity is varied in steps of 5 % from 100 % down to 20 % of its initial capacity.

As we see from the plots, the delays of the *Change Volume* scenario are not much affected by changes of the NAV processor capacity and

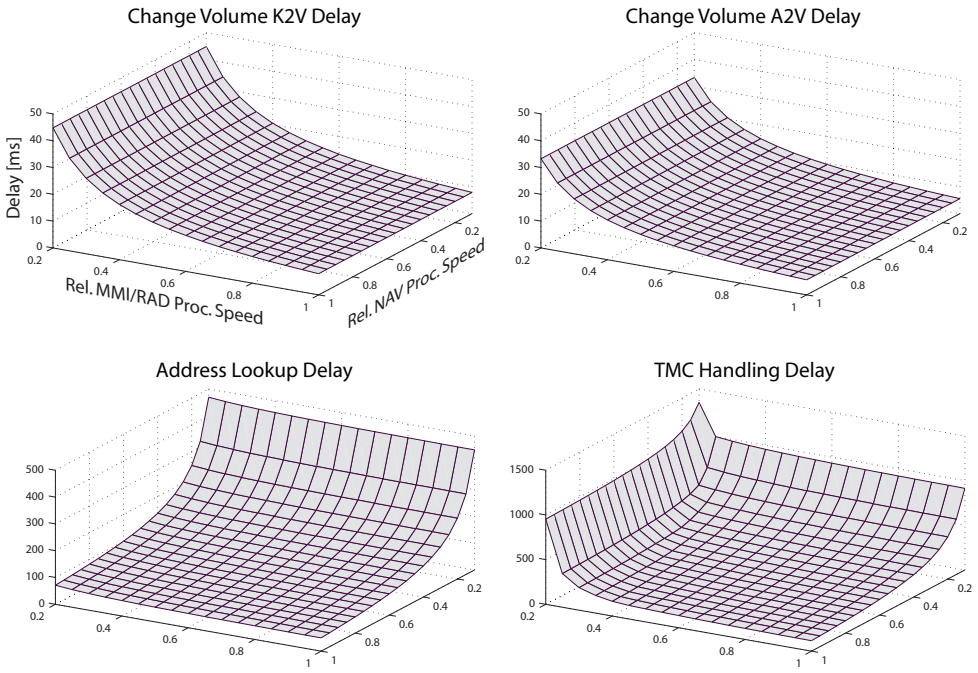


Fig. 20: Delays of the various event streams as a function of the processor speeds in architecture (D).

the delay of the *Address Look-up* scenario on the other hand is not much affected by changes of the MMI/RAD processor capacity. On the other hand, the delay of the *TMC Message Handling Scenario* is affected by the changes of both processor capacities. From the results, we learn that we could decrease both the NAV processor capacity as well as the MMI/RAD processor capacity down to 25 % of their initial capacity (i. e. 29 MIPS and 33 MIPS, respectively) while still guaranteeing the fulfillment of all system requirements.

2.8 Related Work

There is a large body of work devoted to system-level performance analysis of embedded system architectures. For an extensive overview see for example [GVNG94], [Gri04] or [TW05] as well as the references therein.

Most methods for system-level performance analysis of embedded system architectures can broadly be divided into the two main classes of simulation based methods and formal analysis based methods. Additionally there exist also stochastic analysis methods, see e. g. [FM03], [Leh96] or [Lin98], which we however will not discuss further in this context.

Currently, the use of simulation based methods for performance estimation is the state of the art in industry, and system designers mostly rely on commercial tool suites such as Cadence's VCC [VCC], Mentor Graphics' Seamless [Sea], ARM's MaxSim [Max] or Synopsis' SystemStudio [Sysb]. Besides these commercial tool suites there exist also open frameworks for simulation based performance estimation such as SystemC [Sysa, GLMS02] or SimpleScalar [Sim].

The main advantage of using simulation based methods is that many dynamic and complex interaction in a system architecture can be taken into account. However, in terms of *guaranteeing correctness*, simulation based approaches typically suffer from insufficient corner case coverage, because any concrete simulation-run can in general not guarantee to cover all corner cases. Hence simulation based methods do not reveal worst-case bounds on essential system properties like end-to-end delay of events, throughput, and memory requirement. In addition, they often suffer from long running times (depending on the accuracy aimed for) and from high set-up effort for each new architecture and mapping to be analyzed.

To overcome the latter two disadvantages of strictly simulation based methods, approaches were proposed that combine simulation and analysis for performance estimation. In [LRD01] Lahiri et al. combine simulation and analysis by a hybrid trace-based simulation methodology, and in [KPBT06] Künzli et al. propose a method to combine SystemC based simulation [BBB+03] with formal analysis based on Real-Time Calculus. While these mixed methodologies help to shorten simulation run-times, the problem of insufficient corner case coverage still remains.

To guarantee correctness of performance analysis results, methods based on formal analysis must be adopted. These methods do not only guarantee to deliver worst-case (and best-case) results for various system properties, but they often also exhibit fast analysis run-times. The main disadvantage of formal analysis based methods is typically their lack to incorporate complex interactions and state-dependent behavior. Analysis results are therefore often pessimistic, but this pessimism does not threaten their correctness.

Analytical performance models for DSP systems and embedded processors were proposed in [Aga92] and [FW03]. Here, the computation, communication, and memory resources of a processor are all described using simple algebraic equations that do not take into account the dynamics of the applications such as variations in resource loads and shared resources. These methods are therefore lacking in accuracy and the analysis results typically show large deviations from the properties of the final system implementation.

On the other hand there exists a large body of literature on scheduling

of tasks on shared computing resources, see e. g. [But97] and the references therein. In particular in the real-time systems domain many results are available on schedulability analysis and worst-case response time analysis of individual tasks on single processor systems with various scheduling policies. Examples are analysis methods for fixed-priority, rate-monotonic [LL73], deadline monotonic [LW82], or earliest deadline first scheduling [LL73, BRH90], or for time triggered policies like TDMA or round-robin.

Several proposals have been made to extend the concepts of classical scheduling theory to distributed systems. Such extensions must in particular consider the delays caused by the use of possibly shared communication resources that can typically not be neglected. The analytic integration of processor and communication infrastructure scheduling is often referred to as holistic scheduling analysis. But rather than denoting a specific performance analysis method, holistic scheduling analysis comprises a collection of techniques for scheduling analysis of distributed embedded systems.

The seminal work of Tindell and Clark [TC94] was the first approach towards holistic scheduling analysis. Tindell and Clark combine fixed priority preemptive scheduling on processing resources of a distributed system with TDMA scheduling on the interconnecting communication bus. This work was improved in accuracy by Yen et al. [YW95] by taking into account data dependencies, and by Pop et al. [PEP00] by considering control dependencies. Later many holistic scheduling analysis techniques for various other combinations of scheduling and arbitration policies have been investigated, see e. g. [TBW95], [PEP02], [PEP03] or [GH03] and the references therein.

In the collection of holistic scheduling analysis techniques, every technique is tailored towards a particular combination of input event model, resource sharing policy and communication arbitration. While this permits detailed analysis of the temporal behavior of a specific distributed system, it has the drawback that a new analysis method must be developed for every new input event model, communication protocol, resource sharing policy and combinations thereof. This circumstance not only restricts the applicability of holistic scheduling analysis, but the consequently large heterogeneous collection of different techniques also makes it difficult to use holistic scheduling analysis in practice. This latter problem was however largely relaxed by González Harbour et al. with the release of the Modeling and Analysis Suite for Real-Time Application (MAST) [GGPD01] that implements and aggregates several holistic scheduling analysis techniques within an open source tool suite.

A more general approach to extend the concepts of classical scheduling theory to heterogeneous distributed systems was presented by Richter

et al. in [RE02], [RZJE02] and [RJE03]. In contrast to the holistic scheduling analysis that attempts to extend classical scheduling analysis to special classes of distributed systems, Richter et al. propose a compositional performance analysis methodology with the main goal to directly exploit the successful results of classical scheduling theory, in particular for sharing a single processor or a single communication link.

In this compositional approach, every single processor or communication link of a distributed system is analyzed locally. To interconnect the various components, the method relies on a set of standard event arrival patterns. Based on the arrival patterns of the incoming event streams and on the scheduling policy of the component, the appropriate classical analysis technique is chosen individually for every single processor or communication link to compute the worst-case and best-case response time of every event stream at the component as well as to compute the arrival patterns of the outgoing event streams that will trigger succeeding components. The local analysis results are then combined to obtain global end-to-end delays and buffer requirements.

The approach is however only feasible if the arrival patterns of the incoming event streams at a component fit the basic models for which results on computing bounds on the response times are available. To overcome this limitation, Richter et al. define two types of interfaces that may be placed between components. Event Model Interfaces (EMIF) perform a type conversion between certain arrival patterns, i. e. they change the mathematical representation of event streams. Event Adaption Functions (EAF) on the other hand must be used whenever there exists on EMIF. In this case, the HW implementation of the analyzed system must be changed in order to make the system analyzable, e. g. by adding play-out buffers between components.

Many extensions have been worked out for the compositional approach described above, leading to a powerful framework for system-level performance analysis that was eventually implemented in the SymTA/S tool suite [JRE04, HHJ+05, Sym]. Nevertheless, the approach also has some inherent drawbacks. First of all, the compositional approach is bound to a limited set of classical arrival patterns that is often not sufficient to represent event streams with complex timing behaviors. These must be represented in one of the supported arrival patterns, usually with loss in accuracy. Furthermore, arrival patterns often need to be adapted between components, either again with loss in accuracy (EMIF), or even with enforcing a change in the system HW implementation (EAF). Furthermore, the approach is not compositional in terms of the resources, as their service is not modeled explicitly. For example, if several scheduling policies need to be combined in one resource (hierarchical scheduling), then for each new combination an appropriate analysis method must be

developed. In this way, the approach suffers from similar problems as the holistic methods described earlier.

The results obtained with the formal analysis based methods described so far, including the MPA framework, are in general hard upper bounds to the worst-case result and hard lower bounds to the best-case result of the various analyzed properties of a system. In contrast to these methods, Timed Automata based performance analysis is capable to compute the exact worst-case and best-case of various properties of a system. Timed Automata were first proposed by Alur et al. in [AD94], and in [EWY99] Ericsson et al. showed that timed automata can be used as task models for event-driven systems and that the schedulability problem in such a model can be transformed to a reachability problem for timed automata and is thus decidable. Timed Automata based schedulability analysis is implemented in the TIMES tool [AFM+02, Tim], that is however limited to the schedulability analysis of single processor systems.

But it is also possible to model complex component behaviors in distributed systems in any level of detail using Timed Automata, these models can then be analyzed for example using the UPPAAL tool environment [BDL, Upp]. However, the limitation to synchronous communication in Timed Automata, but also the existence of different event types in the modeled system requires in general to explicitly model all buffers in a stream based multi-processor application with asynchronous communication. But since a single buffer of size b that may hold events of e different types, already requires $e^{(b+1)} - 1$ states in an Timed Automata model, analysis of a distributed system with explicitly modeled buffers will quickly lead to state-space explosion, turning the analysis effort to be prohibitive.

This problem was addressed by Hendriks and Verhoef in [HV06], where they propose a Timed Automata based approach to performance analysis where buffers between components are modeled with global variables if they do not hold events of different event types. But, while this technique defers the problem of state-space explosion, it does not dispose of it. And even though the size of the Timed Automata model of a distributed system is often drastically reduced with this approach, model checking times are typically still by several orders of magnitude longer than with any of the previously described formal analysis based methods.

2.9 Discussion

The framework of Modular Performance Analysis is tailored towards performance analysis of distributed real-time systems, where independent applications share a common execution platform to process event streams.

In such systems, the framework can be used to compute hard upper and lower bounds on maximum end-to-end delays and buffer requirements, but also other performance criteria such as individual resource utilizations may be analyzed. The obtained analysis results are deterministic and provide hard upper and lower bounds for any analyzed quantity. This enables the framework to be used for the analysis of hard-real time systems. However, as a consequence it is not possible to obtain average case results for any performance criteria in a system.

In contrast to most formal analysis based methods that were discussed in the last section the MPA framework follows a completely different approach to performance analysis, which relies neither on any standard event arrival patterns nor on the results of classical scheduling analysis. The followed approach leads to a high degree of generality and modularity. The key enabling factor for this generality and modularity within the MPA framework, and for the easy analyzability of system performance models, is the consequent representation of all time-varying quantities (event streams and resources) in the time interval domain. This abstraction from the time domain to the time interval domain does however also not come for free. In particular, it is for example difficult to accurately exploit implicit timing correlations between event arrivals on different event streams within the MPA framework.

3

Abstract Components

The central building blocks of a system performance model are the abstract components, that model the application tasks and dedicated HW/SW components of an embedded system. In a concrete system, application tasks and dedicated HW/SW components have ingoing event streams as input and generate outgoing event streams on their output. The characteristics of the outgoing event streams are thereby determined by the characteristics of the ingoing event streams and the processing semantics of the component, and depend sometimes also on the computation or communication services that are available to the component. In the system performance model on the other hand, event streams are modeled as arrival curves, and service availabilities as service curves, and an abstract component therefore relates ingoing arrival and service curves on its input to outgoing arrival and service curves on its output. The internal relations of the abstract component are thereby determined by the processing semantics of the modeled concrete component, such that the outgoing arrival curves correctly model the outgoing event streams in the concrete system, and that possible outgoing service curves correctly model the remaining computation or communication services in the concrete system.

The previous chapter introduced an abstract component with its internal relations, that models what we call a greedy processing component, and that is used for example to model tasks for preemptive fixed priority scheduling. Greedy processing components are very common in the area of real-time embedded systems, and large parts of a system can often be modeled with these components. However, more complex embedded

system may also contain components with different processing semantics. To model these components for performance analysis within the MPA framework, new abstract components with corresponding internal relations must be defined.

This chapter introduces three new components with different processing semantics, and establishes the corresponding abstract components for analysis within the MPA framework, together with the internal relations, and with methods to determine performance metrics such as experienced delays and buffer requirements. The next section first introduces an abstract component to model a greedy shaper. Greedy shapers are a special instance of traffic shapers, and their use within multiprocessor systems often allows to drastically reduce the total buffer requirements of a system. In Section 3.2, two new abstract components are then introduced that allow to model tasks with multiple inputs, where the task activation is determined as a boolean function on the events arriving on the various inputs.

3.1 Greedy Shapers

In the area of broad-band networking, traffic shaping is a well-known and well-studied technique to regulate connections and to avoid buffer overflow in network nodes, see e. g. [GSE+98] or [RBGW97]. A traffic shaper in a network node buffers the data packets of an incoming traffic stream and delays them such that the output stream conforms to a given traffic specification. A shaper may ensure for example that the output stream has limited burstiness, or that packets on the output stream have a specified minimum inter-arrival time. A greedy shaper is a special instance of a traffic shaper, that not only ensures an output stream stream that conforms to a given traffic specification, but that also guarantees that no packets get delayed any longer than necessary.

By limiting the burstiness of the output stream of a network node, shapers typically drastically reduce the buffer requirements on subsequent network nodes. And in particular if some sort of priority scheduling is used on a network node to share bandwidth among several incoming streams, then a limited burstiness of high-priority streams leads to better responsiveness of low-priority streams. In addition, under some circumstances, shaping comes for free from a performance point of view.

Due to these favorable properties, shapers also play an increasingly important role in the design of real-time embedded systems. Particularly, since modern embedded systems are often implemented as multiprocessor systems with a considerable amount of on-chip traffic. But despite their growing importance in this area, no methods exist to incor-

porate shapers into a system-level performance analysis. Hence it is until now not possible to determine the effect of shapers to end-to-end delay guarantees of buffer requirements in such systems.

Within the framework of Network Calculus [LT01], Le Boudec and Thiran present methods to analyze the effects of traffic shapers in communication networks. However, in the area of performance analysis of embedded systems, it is only Richter et al. [RJE03] that introduces a restricted kind of traffic shaping through the event adaption functions (EAF). But EAF's play a crucial role in the fundamental ability of Richter's compositional method to analyze systems, and a designer has therefore only a very limited freedom to place or leave away, or even to parameterize EAF's.

In this section, we will extend the MPA framework to enable performance analysis of embedded systems with greedy traffic shapers. It has to be noted here, that in [LT01], Le Boudec and Thiran challenge the ability of Real-Time Calculus [TCN00] to analyze traffic shapers, and in [SJNL05], Schiøler et al. even claim that it is not possible to analyze traffic shapers within the MPA framework.

3.1.1 Embedding Greedy Shapers

In Figure 21(a) a greedy shaper component is depicted that shapes an ingoing event stream $R(t)$ with a shaping curve σ. After being shaped, the events are emitted on the component's output, resulting in a shaped outgoing event stream $R'(t)$.

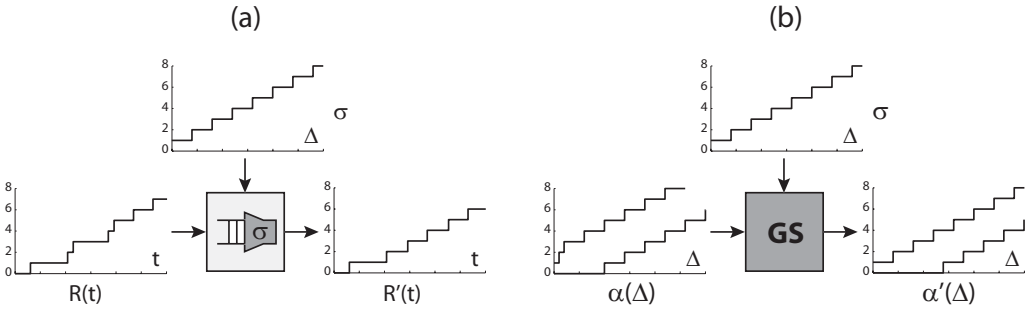


Fig. 21: (a) A concrete greedy shaper, shaping a concrete event stream. (b) An abstract greedy shaper, shaping an abstract event stream.

To enable analysis of systems with greedy shapers within the MPA framework, we need to introduce a new abstract component that models a greedy shaper, as depicted in Figure 21(b). Here, an abstract event stream $\alpha(\Delta)$ enters the abstract greedy shaper component to be shaped with the shaping curve σ. The shaped output is then again an abstract

event stream $\alpha'(\Delta)$. According to (2.5) and analogously to (2.7) and (2.8) we consequently need to find relations that relate the incoming arrival curves to the outgoing arrival curves in order to specify such an abstract greedy shaper component:

$$\alpha' = f_{GS}(\alpha, \sigma) \tag{3.1}$$

Following, we will first explain the behavior and the implementation of concrete greedy shapers, and we will then introduce the internal relations that define an abstract greedy shaper component within the MPA framework.

3.1.2 Concrete Greedy Shapers

A *greedy shaper* with a shaping curve σ delays events of an input event stream, such that the output event stream has σ as an upper arrival curve. Additionally, a greedy shaper ensures that no events get delayed any longer than necessary.

Greedy shapers can therefore be used to ensure that an event stream is upper bounded by an upper arrival curve α^u. For this, the event stream R is input to a greedy shaper with shaping curve $\sigma \leq \alpha^u$. The output event stream R' of the greedy shaper is then upper bounded by α^u.

To analyze the behavior of a greedy shaper, consider a greedy shaper with shaping curve σ, which is sub-additive and with $\sigma(0) = 0$, and assume that the shaper buffer is empty at time 0, and that it is large enough so that there is no event loss. In [LT01], Le Boudec and Thiran prove that for an input event trace R to such a greedy shaper, the output event trace R' can be computed as:

$$R' = R \otimes \sigma \tag{3.2}$$

In practice, a greedy shaper with a shaping curve

$$\sigma(\Delta) = \left\lfloor \min_{\forall i} \{(b_i) + r_i \Delta\} \right\rfloor \tag{3.3}$$

with $\sigma(0) = 0$ can be implemented using a cascade of so-called leaky bucket greedy shapers. Every leaky bucket greedy shaper is a greedy shaper with a shaping curve $\sigma_i(\Delta) = \lfloor (b_i) + r_i \Delta \rfloor$, and can be implemented using some sort of leaky bucket with bucket size b_i that is filled (instead of emptied) at a constant rate of r_i. If an event arrives at such a leaky bucket greedy shaper, it can pass the shaper immediately if the fill level of the bucket is larger or equal 1. Otherwise the event gets delayed until the bucket got filled enough again. Finally, every event that is sent on the output of the shaper reduces the bucket fill level by 1.

At such a leaky bucket stage, the first $\lfloor b_i \rfloor$ events of a burst can pass without any delay. But further events of the burst will be delayed and

can only pass at a rate of r_i. If no events arrive for some time, the bucket will eventually be full again, allowing another burst of $\lfloor b_i \rfloor$ events to pass.

Algorithm 1 shows the pseudo code to implement a leaky bucket greedy shaper. Initially, the bucket is full and the clock is reset to $c = 0$. Then the shaper does a blocking read on its FIFO input buffer. If an event is available on the FIFO input buffer, it is immediately sent to the shaper output. But before sending the event, the new bucket fill level f is computed as the sum of the last computed bucket fill level and the amount $c \cdot r$ by which the bucket got filled since the last clock reset and hence the last update of the fill level. The fill level is thereby limited by the bucket size b. Immediately after sending the event, the bucket fill level is reduced by 1 and the clock is reset to $c = 0$. If necessary, the shaper then waits until the bucket fill level is larger or equal 1 again before doing the next blocking read on its FIFO input buffer.

Algorithm 1 Leaky bucket greedy shaper with bucket size b and filling rate r.

Given a clock $c \in \mathbb{R}_{\geq 0}$ that is continuously running and that can be reset to $c = 0$.

init
reset c
$f = b$ // bucket fill level

while true **do**
  blocking read of event e on FIFO input buffer
  $f = \min\{f + c \cdot r, b\}$
  send event e
  $f = f - 1$
  reset c
  wait $\max\{0, \frac{1-f}{r}\}$
end while

In Figure 22, a shaping curve is depicted that can be implemented by a cascade of two leaky buckets. The first leaky bucket has a bucket size of $b_1 = 1$ and a leaking rate of $r_1 = 1/4$, while the second leaky bucket has a bucket size of $b_2 = 2.8$ and a leaking rate of $r_2 = 1/15$.

It is also possible to implement greedy shapers with more complex shaping curves than (3.3). For this, the greedy shaper needs to consider the history of sent events and compare it against the shaping curve to determine the next point in time when an event can pass the shaper. The implementation of such a greedy shaper would typically be more complex than a simple cascade of leaky bucket greedy shapers, and in

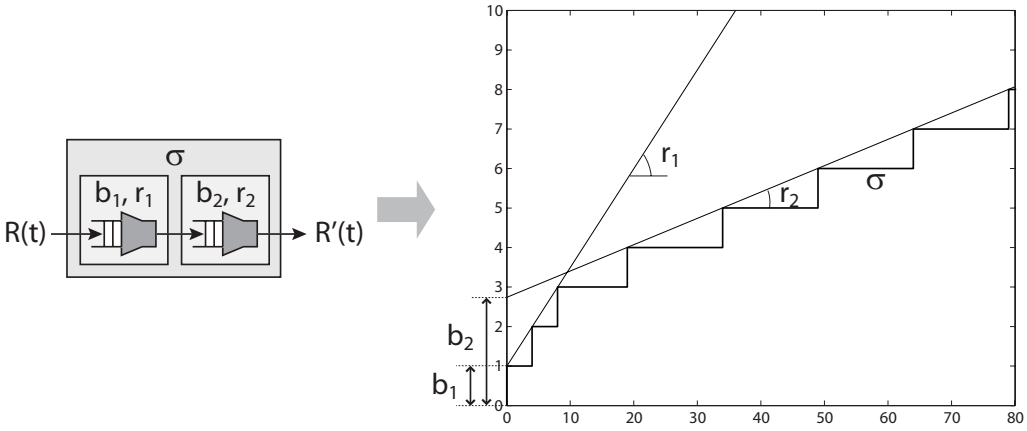


Fig. 22: A greedy shaper that is implemented by a cascade of two leaky bucket stages and the resulting total shaping curve.

practice one would therefore often approximate a complex shaping curve by a shaping curve as defined in (3.3).

In practice, one of the simplest examples of a shaper is the play-out buffer that is read at a constant rate. Play-out buffers are widely used in multimedia processing, and can be implemented by a single leaky bucket greedy shaper.

An interesting property of greedy shapers is that re-shaping does not increase delay or buffer requirements. That is to say, if an event stream $R(t)$ that is constrained by the upper arrival curve α^u is the input to a processing component as depicted in Figure 23, and if a greedy shaper with shaping curve $\sigma \geq \alpha^u$ is used to re-shape the output event stream, then the maximum delay experienced by any event is not increased by adding the greedy shaper, i. e. $d_{max,tot} = d_{max,GP}$. Moreover, if the greedy shaper with shaping curve $\sigma \geq \alpha^u$ and the input buffer to the processing component access the same shared memory, then the total buffer requirement is also not increased by adding the greedy shaper, i. e. $b_{max,tot} = b_{max,GP}$. These properties are sometimes referred to as "greedy shapers come for free", and Le Boudec and Thiran prove them in [LT01].

3.1.3 Abstract Greedy Shapers

Thm. 1: (Abstract Greedy Shapers) *Assume an event stream that is modeled as an abstract event stream with arrival curves* $[\alpha^u, \alpha^l]$ *serves as input to a greedy shaper with a sub-additive shaping curve* σ *with* $\sigma(0) = 0$. *Then, the output of the greedy shaper is an event stream that can be modeled as an abstract event*

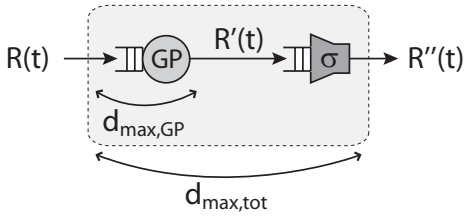


Fig. 23: Cascade of a processing component and a greedy shaper that re-shapes the output event stream.

stream with arrival curves

$$\alpha_{GS}^{u'} = \alpha^u \otimes \sigma \tag{3.4}$$
$$\alpha_{GS}^{l'} = \alpha^l \otimes (\sigma \overline{\oslash} \sigma) \tag{3.5}$$

Further, the maximum delay and the maximum backlog at the greedy shaper are bounded by

$$d_{max,GS} = Del(\alpha^u, \sigma) \tag{3.6}$$
$$b_{max,GS} = Buf(\alpha^u, \sigma) \tag{3.7}$$

Proof. To prove (3.4) we use the fact that $R \oslash R$ is the minimum upper arrival curve of a cumulative function R, and we use the properties

$$(f \oslash g) \oslash h = f \oslash (g \otimes h)$$
$$(f \otimes g) \oslash g \leq f \otimes (g \oslash g)$$

that are proven in [LT01]. We can then compute

$$\begin{aligned} R' \oslash R' &= (R \otimes \sigma) \oslash (R \otimes \sigma) \\ &= ((R \otimes \sigma) \oslash R) \oslash \sigma \\ &= ((\sigma \otimes R) \oslash R) \oslash \sigma \\ &\leq (\sigma \otimes (R \oslash R)) \oslash \sigma \\ &\leq (\sigma \otimes \alpha^u) \oslash \sigma \\ &= (\alpha^u \otimes \sigma) \oslash \sigma \\ &= \alpha^u \otimes \sigma \end{aligned}$$

To prove (3.5) we use the fact that $R \overline{\oslash} R$ is the maximum lower arrival curve of a cumulative function R, and we use the property

$$(f \otimes g) \overline{\oslash} (h \otimes j) \geq (f \overline{\oslash} h) \otimes (g \overline{\oslash} j)$$

that we prove in Appendix A.3. We can then compute

$$\begin{aligned} R'\overline{\oslash}R' &= (R \otimes \sigma)\overline{\oslash}(R \otimes \sigma) \\ &\geq (R\overline{\oslash}R) \otimes (\sigma\overline{\oslash}\sigma) \\ &\geq \alpha^l \otimes (\sigma\overline{\oslash}\sigma) \end{aligned}$$

To prove (3.6), we compute

$$\begin{aligned} d(t) &= \inf\{\tau \geq 0 : R(t) \leq R'(t+\tau)\} \\ &= \inf\{\tau \geq 0 : 0 \leq \inf_{0\leq u\leq t+\tau}\{\sigma(t+\tau-u) + R(u) - R(t)\}\} \\ &\leq \inf\{\tau \geq 0 : 0 \leq \inf_{0\leq u\leq t}\{\sigma(t-u+\tau) - \alpha^u(t-u)\}\} \\ &\leq \inf\{\tau \geq 0 : 0 \leq \inf_{0\leq v}\{\sigma(v+\tau) - \alpha^u(v)\}\} \\ &= \inf\{\tau \geq 0 : \sup_{0\leq v}\{\alpha^u(v) - \sigma(v+\tau)\} \leq 0\} \\ &= \sup_{0\leq\Delta}\{\inf\{\tau \geq 0 : \alpha^u(\Delta) \leq \sigma(\Delta+\tau)\}\} \end{aligned}$$

To prove (3.7), we compute

$$\begin{aligned} b(t) &= R(t) - R'(t) = R(t) - (\sigma \otimes R)(t) \\ &= R(t) - \inf_{0\leq u\leq t}\{\sigma(t-u) + R(u)\} \\ &= R(t) + \sup_{0\leq u\leq t}\{-\sigma(t-u) - R(u)\} \\ &= \sup_{0\leq u\leq t}\{R(t) - R(u) - \sigma(t-u)\} \\ &\leq \sup_{0\leq u\leq t}\{\alpha^u(t-u) - \sigma(t-u)\} \\ &= \sup_{0\leq v\leq t}\{\alpha^u(v) - \sigma(v)\} \leq \sup_{0\leq\Delta}\{\alpha^u(\Delta) - \sigma(\Delta)\} \end{aligned}$$

□

Relations (3.4) and (3.5) can now be used as internal relations of an abstract greedy shaper, and (3.6) and (3.7) can be used to analyze delay guarantees and buffer requirements of greedy shapers in a performance model.

3.1.4 Applications and Experimental Results

In the domain of embedded systems, we may identify two main application areas for traffic shaping. First, shapers may be used system-internally, to re-shape internal traffic streams to reduce global buffer requirements and end-to-end delays, and secondly, shapers may be added at the boundaries of a system, to ensure conformant input streams and to thereby

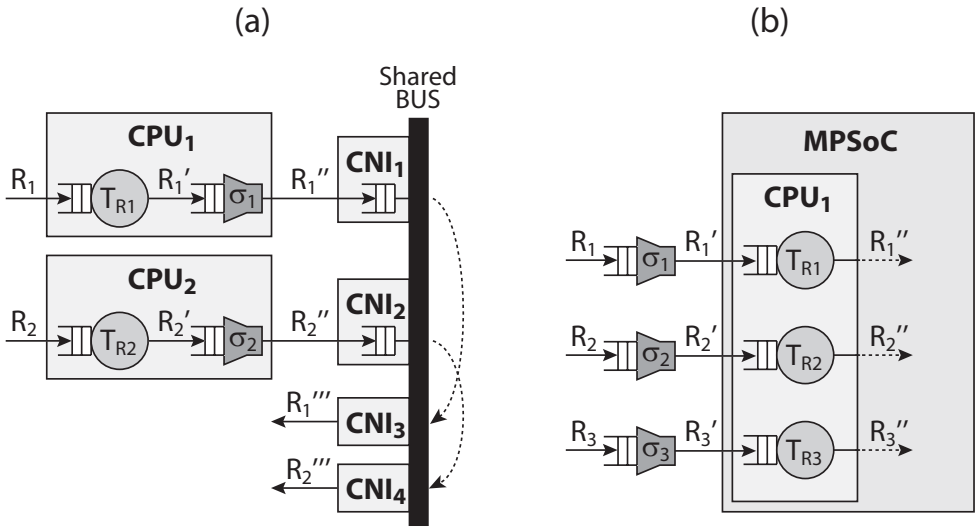


Fig. 24: (a) A system architecture with internal re-shaping to reduce global buffer requirements. (b) A system architecture with external input-shaping to prevent internal buffer overflows.

prevent internal buffer overflows caused by malicious input. Figure 24 shows two example systems from both of these two application areas.

Following, we analyze these two systems, and we discuss an additional application of greedy shapers. The analysis results will clearly reveal the positive influence of greedy shapers to a system's performance and buffer requirements when applied internally, or to a system's robustness when applied externally.

3.1.4.1 Internal Shaping for System Improvement

Consider a distributed real-time system with 2 CPU's that communicate via a shared bus, as depicted in Figure 24 (a). CPU_1 and CPU_2 both process an incoming event stream R_1 and R_2, and send the resulting event streams R_1'' and R_2'' via the shared bus to other components. The shared bus implements a fixed-priority protocol, where sending the events from CPU_1 has priority over sending the events from CPU_2. Events that are ready to be sent get buffered in the communication network interfaces CNI_1 and CNI_2 that connect CPU_1 and CPU_2 with the shared bus.

In this system, R_1'' may differ considerably from R_1. For example R_1'' may be bursty even when R_1 is a strictly periodic event stream. This may happen for example, if besides T_{R1}, other tasks are executed on CPU_1 using a TDMA scheduling policy. Or also if FP scheduling is used and T_{R1} has a low priority. In both cases, the processor may not be available to T_{R1} during some time interval in which all arriving events of R_1 get buffered,

and it may be fully available to T_{R1} during a later time interval in which all the buffered events will be processed and emitted, leading to a burst on R_1''. Now suppose that event stream R_1'' is bursty. Whenever a burst of events arrive on R_1'', the shared bus gets fully occupied until all buffered events of R_1'' are sent. During this period, event stream R_2'' will receive no service, and R_2'' will experience a delay caused by the burstiness of R_1''. Moreover, also the buffer demand in CNI_2 will increase with increasing burstiness of R_1''.

In this system, it may be an interesting option to place a greedy shaper at the output of CPU_1, that shapes event stream R_1'. This greedy shaper will limit the burstiness of R_1'', and will therefore reduce the influence of CPU_1 and R_1 to the delay of R_2' and the buffer requirements of CNI_2.

To investigate the effect of adding greedy shapers to the system with internal re-shaping in Figure 24(a), we analyze it with the MPA framework, using the abstract greedy shaper component that we introduced in the last section.

We assume that R_1 and R_2 are both strictly periodic with a period $p = 1ms$. Further we model both CPU's as bounded delay resources: the CPU may not be available to process the tasks T_{Si} for up to $5ms$. But after this period of at most $5ms$, the processor is fully available and can process 5 events per ms ($\beta^u_{CPU1} = \beta^u_{CPU2} = 5\Delta[e/ms]$, $\beta^l_{CPU1} = \beta^l_{CPU2} = \max\{0, \Delta - 5\}[e/ms]$). The bus can send 2.5 events per ms ($\beta^u_{BUS} = \beta^l_{BUS} = 2.5\Delta[e/ms]$).

With this specification, we analyze the four different system designs that are depicted in Figure 25. First, we analyze the system without greedy shapers (Figure 25 (a)), secondly, we place a greedy shaper only at the output of CPU_1 to shape R_1' (Figure 25 (b)), then, we place a greedy shaper only at the output of CPU_2 to shape R_2' (Figure 25 (c)), and finally we will add two greedy shapers to shape both R_1' as well as R_2' (Figure 25 (d)). We use the upper arrival curves α^u_{R1} and α^u_{R2} as shaping curves σ_1 and σ_2, respectively, and we assume that the buffers of the greedy shapers and the corresponding processing tasks access the same memory. Using the four depicted performance models, we analyze the maximum required buffer spaces of the different buffers, as well as the end-to-end delays of both event streams R_1 and R_2.

From the results that are shown in Table 1, we learn that placing greedy shapers helps to reduce the total buffer requirements from 25 down to 14 events that need to be buffered at most. Moreover, the greedy buffers also reduce the end-to-end delay of both event streams, namely by 7.4% for R_1, and by a total of 40% for R_2. We also recognize the well-known property of greedy shapers that re-shaping is for free [LT01]. Since we use $\sigma_1 = \alpha^u_{S1}$ and $\sigma_2 = \alpha^u_{S2}$, the greedy shapers effectively only re-shape R_1^1 and R_2^2, and therefore the buffer requirements of CPU_1 and CPU_2 are not affected by adding the greedy shapers.

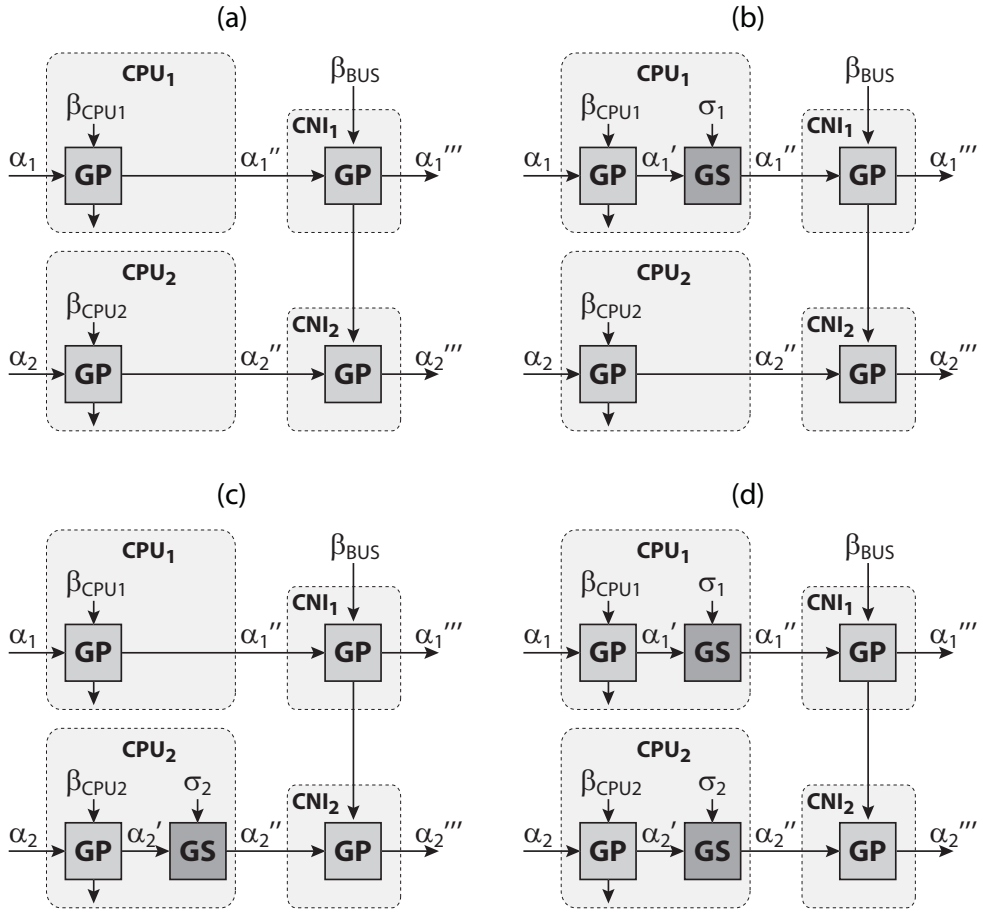


Fig. 25: Performance models of the four system architecture scenarios with internal reshaping.

scenario	buffer					delay	
	CPU_1	CPU_2	CNI_1	CNI_2	Tot	S_1	S_2
(a)	6	6	4	9	25	5.4	9
(b)	6	6	1	6	19	5	5.8
Δ%	-	-	−75%	−33%	−24%	−7.4%	−36%
(c)	6	6	4	4	20	5.4	8.6
Δ%	-	-	-	−56%	−20%	-	−4.4%
(d)	6	6	1	1	14	5	5.4
Δ%	-	-	−75%	−89%	−44%	−7.4%	−40%

Tab. 1: Total buffer requirements and end-to-end delays of the four system architecture scenarios depicted in Figure 25.

3.1.4.2 Input-Shaping for Separation of Concerns

Typical large embedded systems often process several event streams in parallel. To achieve separation of concerns in such systems, they are often implemented using time-triggered scheduling policies, or servers. While these scheduling policies help to decouple the influence of the various event streams to each other, they often do not use the available resources efficiently. On the other hand, powerful methods were developed to analyze systems with event-triggered scheduling policies, such as RM or EDF. In these systems, resources are used efficiently, but on the downside, the various event streams may heavily influence each other. Slight changes in the timing behavior of a high-priority stream may increase the total delay of a lower-priority stream considerably, possibly leading to a missed deadline, or to buffer overflows somewhere in the system.

To overcome this problem, greedy shapers may be placed at the input to such systems. Every incoming event stream R_i gets shaped with an individual shaping curve σ_i that corresponds to its design-time timing specification. The system can then be analyzed using the design-time timing specifications, and at run-time, non-adherence of R_i to its timing specification will have no influence to the delay of any other event streams, but will at most increase the total delay of R_i itself. And moreover, no buffers will overflow inside the system. Instead, only the buffers of the greedy shapers themselves may overflow. But since these buffers are clearly localized at the boundary of the system, individual handling policies can easily be implemented.

Lets assume a real-time system as shown in Figure 24 (b). Here, a single CPU processes three event streams with a fixed-priority scheduling policy. The high-priority stream R_1 is strictly periodic with $p_1 = 5ms$, the medium-priority stream R_2 is strictly periodic with $p_2 = 10ms$, and the low-priority stream R_3 is strictly periodic with $p_3 = 20ms$. The CPU processes 0.35 events per *ms*. To illustrate the influence of greedy shapers at the input of such a system, we add a jitter of $j_1 = 0.1ms$ to stream R_1, and we then analyze the effect of this to the end-to-end delays of the three event streams, both without (Fig 26 (a)) and with (Fig 26 (b)) greedy shapers.

	Without Shaping (a)			With Shaping (b)		
	d_1	d_2	d_3	d_1	d_2	d_3
$j_1 = 0$	2.86	8.57	20	2.86	8.57	20
$j_1 = 0.1$	2.86	8.57	28.57	2.96	8.57	20
Δ%	0	0	+43%	+3.5%	0	0

Tab. 2: End-to-end delays of the two system architecture scenarios depicted in Figure 26.

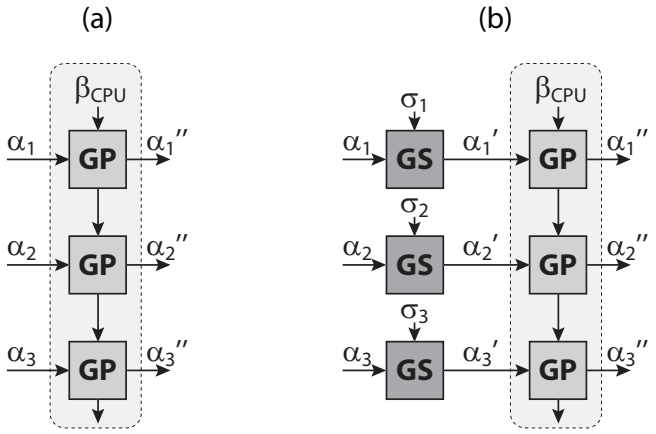


Fig. 26: Performance models of the two system architecture scenarios without (a) and with (b) external input shaping.

Looking at the results in Table 2, we clearly see the big influence of the little non-adherence of R_1 to the maximum delay of the completely independent stream R_3, if no input shaping is applied. On the other hand, we observe that input shaping effectively isolates the influence of the malicious input stream R_1 to the other present event streams. Now, only R_1 is affected from its own malbehavior.

3.1.4.3 Input Shaping for NRT Event Streams in Real-Time Systems

Inspired from the application of greedy shapers for input shaping for separation of concerns, one could also use greedy shapers to shape non-real-time event streams as depicted in Figure 27. Here, the non-real-time event streams are processed with the highest priority, guaranteeing a good reactivity and typically short delays, and the greedy shaper guarantees that the load created by the non-real-time events is limited such that the real-time event-streams remain schedulable.

In such an application, the greedy shaper is an alternative to typical server implementations such as the periodic [SLR86] or the deferrable server [SLS95], and in a certain respect the greedy shaper also behaves like a server. The advantage of the greedy shaper however is its flexibility through the parameterizable shaping curve σ_{NRT}. With an appropriate shaping curve, a shaper can for example guarantee a reactive periodic service such as the deferrable server, but additionally it may also allow a burst of service from time to time. Moreover, a greedy shaper is typically easy to implement.

The main question that then arises in an application as depicted Figure 27 is how to dimension the greedy shaper. That is, how to choose

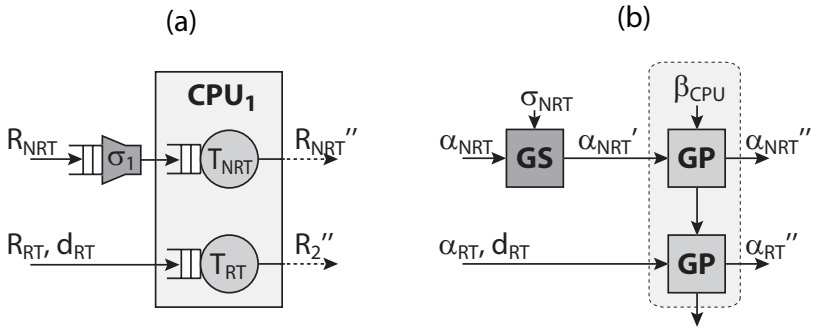


Fig. 27: System model (a) and performance model (b) with input shaping for NRT-traffic.

σ_{NRT} such as to provide the maximum possible service to the non-real-time events without jeopardizing the schedulability of the real-time event streams. In Chapter 5 we will introduce the theory of Real-Time Interfaces that provides methods to find the maximum allowable shaping curve σ_{NRT}. With Real-Time Interfaces, the maximum allowable non-real-time input load to the system in Figure 27 can be computed as

$$\alpha^u_{NRT,max} = RT^{-\alpha}(\alpha^u_{RT}(\Delta - d_{RT}), \beta^l_{CPU}) \tag{3.8}$$

with

$$RT^{-\alpha}(\beta', \beta)(\Delta) = \beta(\Delta + \lambda) - \beta'(\Delta + \lambda)$$
$$\text{for} \quad \lambda = \sup\{\tau \ : \ \beta'(\Delta + \tau) = \beta'(\Delta)\} \tag{3.9}$$

where α^u_{RT} models the maximum real-time input load, d_{RT} denotes the relative deadline of the real-time events, and β^l_{CPU} models the minimum available service from the CPU. The greedy shaper must then only enforce that the load on its output is limited by $\alpha^u_{NRT,max}$ which is easily achieved by setting the shaping curve equal to this upper bound. And without any loss, the shaping curve can even be ensured to be sub-additive by setting it equal to the sub-additive closure of this upper bound:

$$\sigma_{NRT} = \overline{\alpha}^u_{NRT,max} \tag{3.10}$$

The methods presented in Chapter 5 also allow to compute σ_{NRT} for systems with more than one real-time event stream, and even for more complex systems with mixed and hierarchical scheduling.

As an example consider a system similar to the one depicted in Figure 27, but with three real-time event streams with decreasing priorities: $R_{RT,1}$ has a period of 100*ms*, a jitter of 40*ms* and an execution demand of 25*ms*, $R_{RT,2}$ has a period of 200*ms*, a jitter of 150*ms* and also an execution demand of 25*ms* and $R_{RT,3}$ has a period of 500*ms* and an execution demand

of 100*ms*. The relative deadlines of all real-time event streams equal their periods. To compute the shaping curve σ_{NRT} for the non-real-time event streams in this system, we use the methods presented in Chapter 5 and (3.10). From the result depicted in Figure 28, we learn that within this system, any non-real-time event streams that are upper bounded by σ_{NRT} will be served immediately with the highest priority, i. e. they will not be delayed by the input shaper, while events that contravene to this upper bound will be delayed by the input shaper to ensure schedulability of the real-time streams.

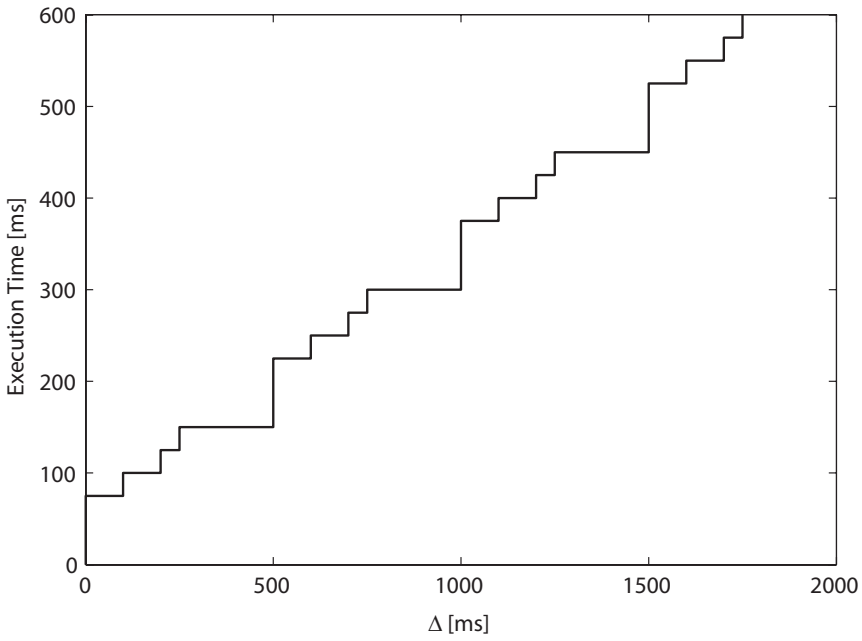


Fig. 28: Input shaping curve σ_{NRT} for the NRT-traffic in the example system.

3.2 Components with Multiple Inputs

In realistic embedded systems, the flow of data between components is typically not limited to one-to-one connections, but instead a component activation often depends on events arriving from multiple other components, and reciprocally a single component often sends its output events to multiple connected components.

The case of a one-to-many connection is typically straight-forward to model, as the event stream at the output of a component only has to be copied to the inputs of all the connected components. Within the MPA

framework this is achieved directly by using the same output arrival curve of a single abstract component as input to all connected abstract components.

Many-to-one connections on the other hand are in general more complex to implement. In particular, since the activation of a component with a many-to-one input connection is in general determined as a boolean function of the events on the various input event streams. Thus, analysis of many-to-one connections is typically more involved, and in this section we will extend the MPA framework to enable performance analysis of embedded systems with many-to-one connections. We will however restrict ourselves to boolean functions without negation, that is acceptable boolean operators are *OR* and *AND*, as well as combinations of these.

In [JE03], Jersak et al. present first results to analyze tasks with OR- and AND-activation within their compositional framework [RJE03]. The results in [JE03] are however partially incorrect, and corrected results are presented and discussed in detail in [Jer05]. Jersak et al. propose to embed many-to-one connections within their compositional framework by first determining the activation pattern that results from the boolean function on the various input event streams of a component with multiple inputs. This activation pattern is then used as the sole input to the component. We will follow the same strategy to embed analysis of many-to-one connections into the MPA framework. And while the results for OR-activations presented in this section equal the results presented in [Jer05], we will present a tighter analysis for AND-activations. Moreover, within the compositional framework of Jersak et al. [RJE03], the input activation pattern resulting from the boolean function must again be represented by the limited set of classical arrival patterns. Therefore, analysis of many-to-one connections within the MPA framework leads typically not only to tighter results for AND-activations, but also for OR-activations.

3.2.1 Embedding Components with Multiple Inputs

To flexibly embed analysis of many-to-one connections into the MPA framework, we will separate the analysis of components with multiple inputs into two steps. In a first step, the combined event stream that results from the boolean function on the various input event streams is determined. In a second step, this event stream serves then as single input to the component that is ordinarily analyzed. This modular strategy allows to add multiple inputs to components with any processing semantics for which analysis methods exist. To determine the combined event stream that results from the boolean function on multiple input event streams, we will introduce boolean connector components that can be combined to any boolean function without negation.

In Figure 29(a) an OR-connector component is depicted that combines two ingoing event streams $R_1(t)$ and $R_2(t)$ into a single combined event stream $R_{OR}(t)$, using a boolean OR operator on the incoming events. Within the MPA framework, we model such an OR-connector as a new abstract component as depicted in Figure 29(b). Here, two abstract event streams $\alpha_1(\Delta)$ and $\alpha_2(\Delta)$ enter the abstract OR-connector component, and the combined output is again an abstract event stream $\alpha_{OR}(\Delta)$.

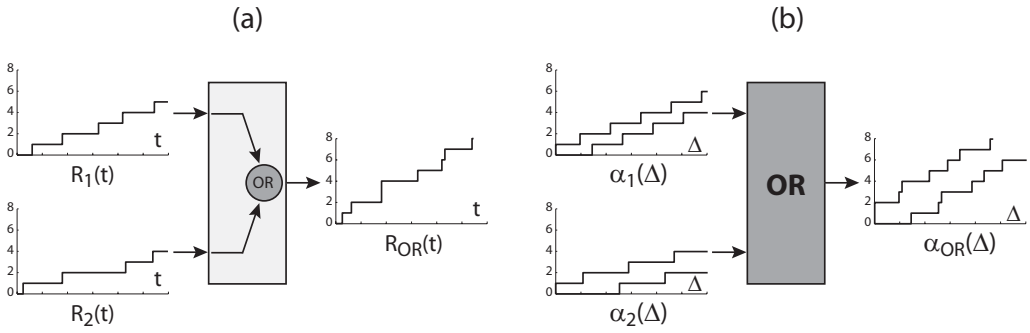


Fig. 29: (a) A concrete OR-connector, combining two concrete event streams. (b) An abstract OR-connector, combining two abstract event streams.

In Figure 30(a) on the other hand, an AND-connector component is depicted that combines two ingoing event streams $R_1(t)$ and $R_2(t)$ into a single combined event stream $R_{AND}(t)$, using a boolean AND operator on the incoming events. Compared to the OR-connector depicted in Figure 29(a), the implementation of an AND-connector requires internal buffers, since events arriving on one input event stream must be buffered until partner events arrive on the other input event stream. Partnering events immediately pass the AND-connector, and consequently either of the internal buffers is empty at any point of time. Within the MPA framework we model such an AND-connector again as a new abstract component as depicted in Figure 30(b). Here, two abstract event streams $\alpha_1(\Delta)$ and $\alpha_2(\Delta)$ enter the abstract AND-connector component, and the combined output is again an abstract event stream $\alpha_{AND}(\Delta)$.

According to (2.5) and analogously to (2.7) and (2.8) we again need to find relations that relate the incoming arrival curves to the outgoing arrival curves in order to specify abstract OR- and AND-connector components:

$$\alpha_{OR} = f_{OR}(\alpha_1, \alpha_2) \tag{3.11}$$

$$\alpha_{AND} = f_{AND}(\alpha_1, \alpha_2) \tag{3.12}$$

Following, we will introduce the internal relations that define ab-

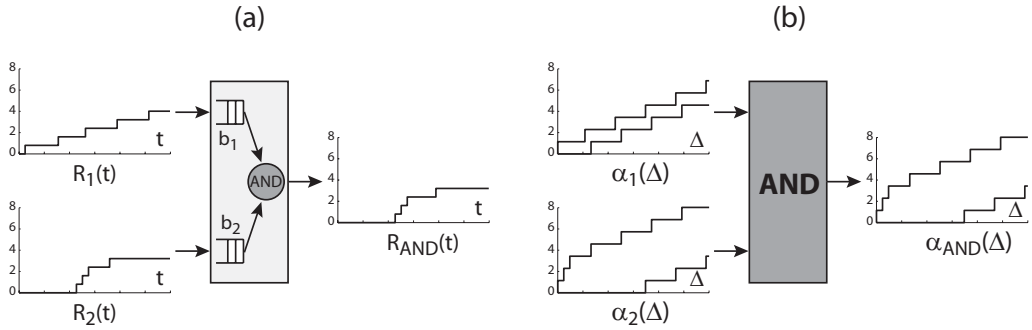


Fig. 30: (a) A concrete AND-connector, combining two concrete event streams. (b) An abstract AND-connector, combining two abstract event streams.

stract OR-connector components and abstract AND-connector components within the MPA framework.

3.2.2 Abstract OR-Connector

Thm. 2: **(Abstract OR-Activation)** *Assume a component that is triggered by OR-activation of two event streams that are modeled as abstract event streams with arrival curves* $[\alpha_1^u, \alpha_1^l]$ *and* $[\alpha_2^u, \alpha_2^l]$. *Then, the component activation can be modeled with the arrival curves*

$$\alpha_{OR}^u = \alpha_1^u + \alpha_2^u \tag{3.13}$$

$$\alpha_{OR}^l = \alpha_1^l + \alpha_2^l \tag{3.14}$$

Proof. The differential flow after the OR-activation can be computed as

$$R_{OR}[s,t) = R_1[s,t) + R_2[s,t) \tag{3.15}$$

To obtain (3.13) and (3.14) we compute the upper bound and the lower bound to (3.15), respectively, using (2.1).

□

3.2.3 Abstract AND-Connector

Lem. 1: *The difference between two event streams that are described as cumulative flows* $R_1(\tau)$ *and* $R_2(\tau)$ *with* $R_1(0) = R_2(0)$ *can be bounded by the following inequality*

$$\eta_{12}^l(t-s) \leq R_1(t) - R_2(s) \leq \eta_{12}^u(t-s), \forall s < t \tag{3.16}$$

with

$$\eta_{12}^u(\Delta) = \sup_{\lambda \geq 0} \left\{ \alpha_1^u(\lambda + \Delta) - \alpha_2^l(\lambda) \right\} = (\alpha_1^u \oslash \alpha_2^l)(\Delta) \tag{3.17}$$

$$\eta_{12}^l(\Delta) = \inf_{\lambda \geq 0} \left\{ \alpha_1^l(\lambda + \Delta) - \alpha_2^u(\lambda) \right\} = (\alpha_1^l \overline{\oslash} \alpha_2^u)(\Delta) \tag{3.18}$$

Proof. To prove (3.17), we use (2.1) and we compute

$$\begin{aligned} R_1(t) - R_2(s) &= (R_1(t) - R_1(0)) - (R_2(s) - R_2(0)) \\ &\leq \alpha_1^u(\lambda + \Delta) - \alpha_2^l(\lambda) \\ &\leq \sup_{\lambda \geq 0} \left\{ \alpha_1^u(\lambda + \Delta) - \alpha_2^l(\lambda) \right\} \end{aligned}$$

and we analogously prove (3.18), again using (2.1).

□

Note, that the constraint $R_1(0) = R_2(0)$ can be demanded without loss of generality. If $R_1(0) > R_2(0)$, this difference can be expressed by an initial buffer fill level $B_1^0 = R_1(0) - R_2(0)$ and we can then set $R_1(0) = R_2(0)$.

Thm. 3: (Abstract AND-Activation) *Assume a component that is triggered by AND-activation of two event streams that are modeled as abstract event streams with arrival curves* $[\alpha_1^u, \alpha_1^l]$ *and* $[\alpha_2^u, \alpha_2^l]$. *Then, the component activation can be modeled with the arrival curves*

$$\begin{aligned} \alpha_{AND}^u &= \max \left\{ \min \left\{ \alpha_1^u \oslash \alpha_2^l + B_1^0 - B_2^0, \alpha_2^u \right\}, \right. \\ &\qquad \left. \min \left\{ \alpha_2^u \oslash \alpha_1^l + B_2^0 - B_1^0, \alpha_1^u \right\} \right\} \qquad (3.19) \\ \alpha_{AND}^l &= \max \left\{ \min \left\{ \alpha_1^l \overline{\oslash} \alpha_2^u + B_1^0 - B_2^0, \alpha_2^l \right\}, \right. \\ &\qquad \left. \min \left\{ \alpha_2^l \overline{\oslash} \alpha_1^u + B_2^0 - B_1^0, \alpha_1^l \right\} \right\} \qquad (3.20) \end{aligned}$$

, where B_1^0 *and* B_2^0 *denote the initial buffer fill levels of the two input ports, with the constraint that* $\min\{B_1^0, B_2^0\} = 0$, *i. e. one of the buffers must initially be empty.*

Further, the maximum delays due to the AND-activation at the two input buffers are bounded by

$$\begin{aligned} d_{max,1} &\leq Del(\alpha_1^u + B_1^0, \alpha_2^l + B_2^0) \qquad (3.21) \\ d_{max,2} &\leq Del(\alpha_2^u + B_2^0, \alpha_1^l + B_1^0) \qquad (3.22) \end{aligned}$$

and the maximum backlogs due to the AND-activation at the two input buffers are bounded by

$$\begin{aligned} b_{max,1} &\leq \max \left\{ Buf(\alpha_1^u + B_1^0, \alpha_2^l + B_2^0), 0 \right\} \qquad (3.23) \\ b_{max,2} &\leq \max \left\{ Buf(\alpha_2^u + B_2^0, \alpha_1^l + B_1^0), 0 \right\} \qquad (3.24) \end{aligned}$$

using the functions Del and Buf as defined by (2.11) and (2.12), respectively.

Proof. Let us first note that the events that are in either of the two input buffers at time t experience a maximum delay that can be computed as

$$d_1(t) = \inf\left\{\tau \geq 0 : R_1(t) + B_1^0 \leq R_2(t+\tau) + B_2^0\right\} \quad (3.25)$$

$$d_2(t) = \inf\left\{\tau \geq 0 : R_2(t) + B_2^0 \leq R_1(t+\tau) + B_1^0\right\} \quad (3.26)$$

and that the buffer fill levels of the two input buffers at time t can be computed as

$$b_1(t) = \max\left\{R_1(t) - R_2(t) + B_1^0 - B_2^0, 0\right\} \quad (3.27)$$

$$b_2(t) = \max\left\{R_2(t) - R_1(t) + B_2^0 - B_1^0, 0\right\} \quad (3.28)$$

To prove (3.19) and (3.20), we first compute the differential flow after the AND-activation

$$R_{AND}[s,t) = \min\left\{R_1[s,t) + b_1(s), R_2[s,t) + b_2(s)\right\}$$

And since we know that $min\{b_1(t), b_2(t)\} = 0, \forall t$ we can also write

$$\begin{aligned} R_{AND}[s,t) = \max\{&\min\left\{R_1[s,t) + b_1(s), R_2[s,t)\right\}, \\ &\min\left\{R_1[s,t), R_2[s,t) + b_2(s)\right\}\} \end{aligned}$$

and using (3.27) and (3.28) we can rewrite this as

$$\begin{aligned} R_{AND}[s,t) = \max\{&\min\left\{R_1(t) - R_2(s) + B_1^0 - B_2^0, R_2[s,t)\right\}, \\ &\min\left\{R_1[s,t), R_2(t) - R_1(s) + B_2^0 - B_1^0\right\}\} \end{aligned} \quad (3.29)$$

To prove (3.19), we now compute the upper bound to (3.29) using (3.16), (3.17) and (2.1). And analogously, to obtain (3.20) we compute the lower bound to (3.29) using (3.16), (3.18) and (2.1).

To prove (3.21), we use (3.25) and we compute

$$\begin{aligned} d_1(t) &= \inf\left\{\tau \geq 0 : R_1(t) + B_1^0 \leq R_2(t+\tau) + B_2^0\right\} \\ &= \inf\left\{\tau \geq 0 : 0 \leq R_2(t+\tau) - R_1(t) + B_2^0 - B_1^0\right\} \\ &\leq \inf\left\{\tau \geq 0 : 0 \leq \eta_{21}^l(\tau) + B_2^0 - B_1^0\right\} \\ &= \inf\left\{\tau \geq 0 : 0 \leq \inf_{\lambda \geq 0}\left\{\alpha_2^l(\tau+\lambda) - \alpha_1^u(\lambda)\right\} + B_2^0 - B_1^0\right\} \\ &= \inf\left\{\tau \geq 0 : \sup_{\lambda \geq 0}\left\{\alpha_1^u(\lambda) - \alpha_2^l(\tau+\lambda) + B_1^0 - B_2^0\right\} \leq 0\right\} \\ &= \sup_{\Delta \geq 0}\left\{\inf\left\{\tau \geq 0 : \alpha_1^u(\Delta) + B_1^0 \leq \alpha_2^l(\tau+\Delta) + B_2^0\right\}\right\} \end{aligned}$$

and we analogously prove (3.22) using (3.26).

To prove (3.23), we use (3.27) and we compute

$$\begin{aligned} b_1(t) &= \max\left\{R_1(t) - R_2(t) + B_1^0 - B_2^0, 0\right\} \\ &\leq \max\left\{\eta_{12}^u(0) + B_1^0 - B_2^0, 0\right\} \\ &= \max\left\{\sup_{\lambda\geq 0}\{\alpha_1^u(\lambda) - \alpha_2^l(\lambda)\} + B_1^0 - B_2^0, 0\right\} \\ &= \max\left\{\sup_{\lambda\geq 0}\{\alpha_1^u(\lambda) + B_1^0 - \alpha_2^l(\lambda) - B_2^0\}, 0\right\} \end{aligned}$$

and we analogously prove (3.24) using (3.28).

□

3.2.4 Experimental Results

Following, we analyze two greedy processing components with multiple inputs within the MPA framework, one with two OR-connected inputs and the other with two AND-connected inputs. As a reference, we also compute the exact results using a timed automata based approach [HV06] and UPPAAL v3.5.9 [Upp], and we also declare the results that were obtained using SymTA/S v0.8 beta EVAL [Sym] that implements the methods presented by Jersak [Jer05].

3.2.4.1 Greedy Processing Component with OR-Activation

We consider a system architecture with a single OR-activated task T_1 as depicted in Figure 31(a). We assume that both input streams R_1 and R_2 are periodic streams with jitter, with $p_1 = 100ms$ and $j_1 = 20ms$, and with $p_2 = 150ms$ and $j_2 = 60ms$, respectively. For this system we want to analyze the total buffer requirement at the input of task T_1, as well as the total end-to-end delay experienced by any of the events of the two input event streams when processed by task T_1. In order to examine the influence of the OR-activation in more depth, we compute the various analysis results as a function of the execution time of T_1, that we vary in steps of $5ms$ from $5ms$ to $60ms$.

To analyze the system architecture depicted in Figure 31(a) with MPA, we first construct the corresponding performance model as depicted in Figure 31(b). In this performance model we determine α_{OR} using (3.13) and (3.14), and we then compute the total buffer requirement and the total end-to-end delay at task T_1 using (2.12) and (2.11), respectively.

From the results that are depicted in Figures 32 and 33, we learn that the MPA framework is able to determine the exact worst-case results for

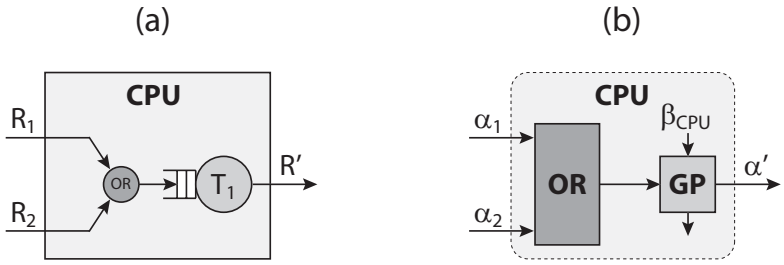


Fig. 31: (a) A system architecture with an OR-activated task. (b) Performance model of the same system architecture.

the total buffer requirement and the total end-to-end delay at the OR-activated task T_1 depicted in Figure 31(a).

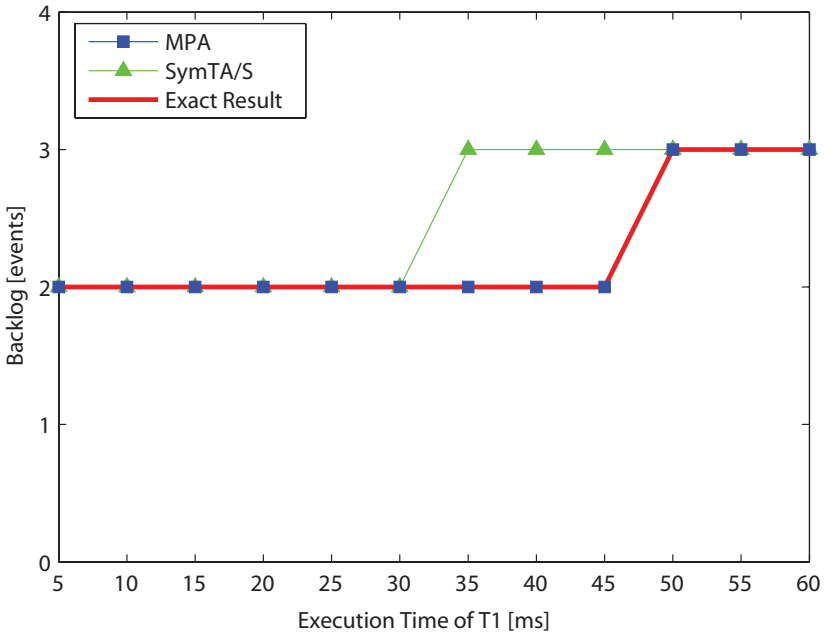


Fig. 32: Total buffer requirements at the OR-activated task T_1 in Figure 31(a) as a function of the execution time of task T_1.

3.2.4.2 Greedy Processing Component with AND-Activation

To experiment with AND-activation, we consider again a system architecture with a single task T_1 with two inputs, but this time with AND-activation, as depicted in Figure 34(a). We assume that input stream R_1 is periodic with jitter, with $p_1 = 100ms$ and $j_1 = 10ms$, while input stream

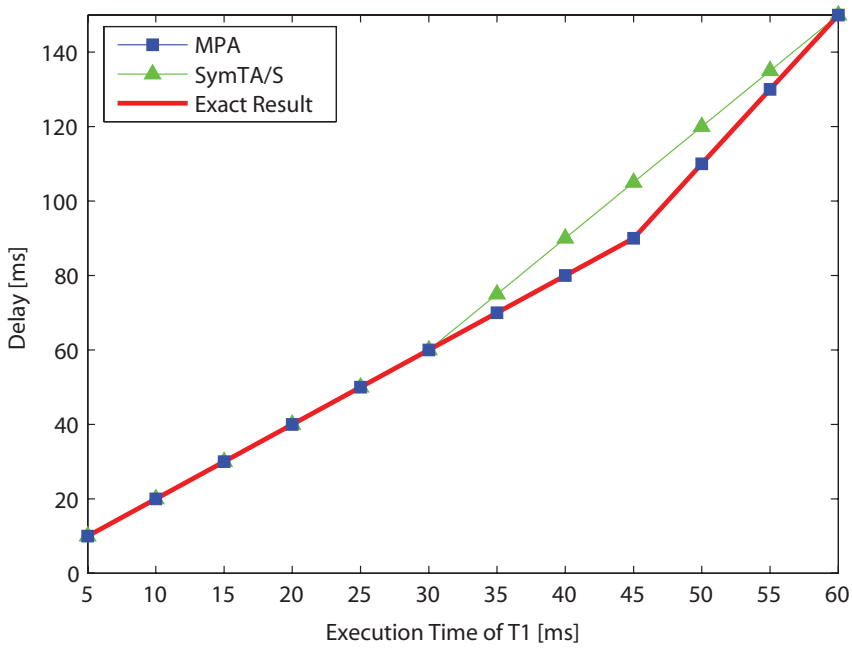


Fig. 33: Total end-to-end delay experienced by any event at the OR-activated task T_1 in Figure 31(a) as a function of the execution time of task T_1.

R_2 is periodic with bursts, with $p_2 = 100ms$, $j_2 = 190ms$ and $d_2 = 20ms$. For this system we want again to analyze the total buffer requirement at the input of task T_1, as well as the total end-to-end delay experienced by any of the events of the two input event streams when processed by task T_1. In order to examine the influence of the AND-activation in more depth, we again compute the various analysis results as a function of the execution time of T_1, that we vary in steps of $5ms$ from $5ms$ to $60ms$.

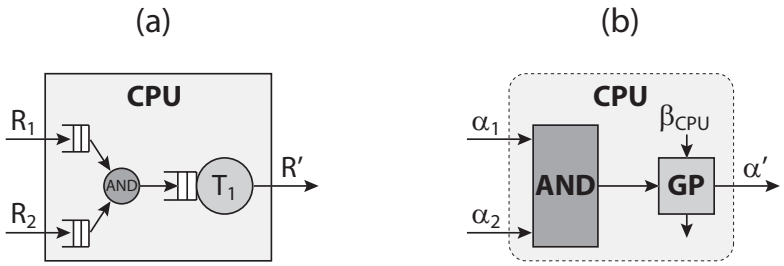


Fig. 34: (a) A system architecture with an AND-activated task. (b) Performance model of the same system architecture.

To analyze the system architecture depicted in Figure 34(a) with MPA,

we again first construct the corresponding performance model as depicted in Figure 34(b). In this performance model we determine α_{AND} using (3.19) and (3.20). To determine the buffer requirement at the input of T_1, we must consider that events may be buffered either at the AND-connection, waiting for a partner event, or directly at T_1, waiting to be processed. The respective individual buffer requirements can be computed using (3.23), (3.24) and (2.12). To determine the total buffer requirement at the input of T_1 however, we can exploit the fact that either of the two buffers at the AND-connection is empty at every point of time, and further we must consider that for every task activation of T_1 waiting to be processed we must hold two partnering events in the input queue to T_1. The total buffer requirement at the input of T_1 can therefore be bounded by

$$b_{max,T_1} \leq max\left\{Buf(\alpha_1^u, \alpha_2^l), Buf(\alpha_2^u, \alpha_1^l)\right\} + 2Buf(\alpha_{AND}^u, \beta_{CPU}^l) \qquad (3.30)$$

From the results that are depicted in Figure 35, we learn that the total buffer requirement analysis within the MPA framework is considerably improved compared to the analysis using SymTA/S. However, we also learn that the MPA framework is not able to determine the exact total buffer requirement at an AND-activated task in general.

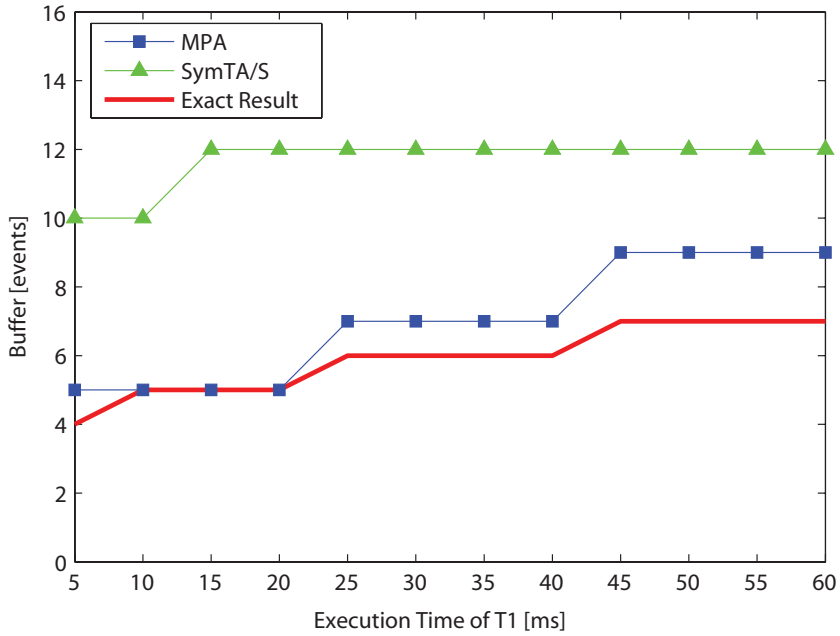


Fig. 35: Total buffer requirements at the AND-activated task T_1 in Figure 34(a) as a function of the execution time of task T_1.

To determine the the maximum end-to-end delay experienced by any

of the events of the two input event streams when processed by task T_1, we must again consider that events may experience a delay either at the AND-connection, or directly at T_1. The respective individual delays can be computed using (3.21), (3.22) and (2.11), and the total end-to-end delays for the two event streams can then be computed by adding the corresponding individual delays. However, these results can be further improved by exploiting the "Pay Bursts Only Once" phenomenon described in Section 2.6.1. This is, because from the point of view of the abstract event stream α_1, the AND-connector behaves similar to a greedy processing component with a service curve α_2^l, but in contrast to a greedy processing component the AND-connector can "buffer" its "service", and therefore the maximum delay experienced by any event of the abstract event stream α_1 at an AND-connector is always smaller than the maximum delay experienced at a greedy processing component with service curve α_2^l. The total end-to-end delay experienced by a cascade of an AND-connector and a greedy processing component can therefore also be upper bounded by the end-to-end delay experienced by a cascade of two greedy processing components where the first has a service of α_2^l. Using (2.13), the maximum total end-to-end delay experienced at T_1 can therefore be bounded by

$$d_{max,R_1} \leq Del(\alpha_1^u, \alpha_2^l \otimes \beta_{CPU}^l) \tag{3.31}$$

$$d_{max,R_2} \leq Del(\alpha_2^u, \alpha_1^l \otimes \beta_{CPU}^l) \tag{3.32}$$

From the results that are depicted in Figure 36, we learn that the total end-to-end analysis within the MPA framework is again improved compared to the analysis using SymTA/S. The MPA framework is however in general again not able to determine the exact total end-to-end delays at an AND-activated task. In particular it has to be noted that both analysis frameworks, SymTA/S as well as MPA, compute the same total end-to-end delay for both event streams in this example, even though events on R_2 experience in reality a considerably smaller delay than events on R_1.

3.3 Discussion

The three new abstract components that we introduced in this chapter extend the analysis capabilities of the MPA framework to the domain of distributed embedded real-time systems that contain greedy shapers and tasks with multiple inputs. All three components discussed in this chapter are triggered by event streams on their inputs and generate event streams on their outputs. The abstract components that model these components thus contain arrival curve inputs and arrival curve outputs,

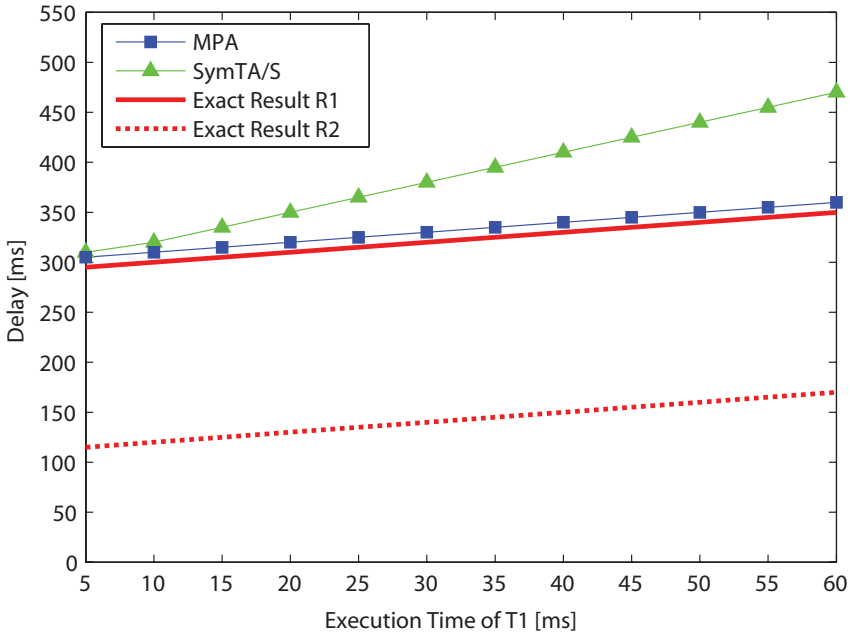


Fig. 36: Total end-to-end delay experienced by any event at the AND-activated task T_1 in Figure 34(a) as a function of the execution time of task T_1.

and by interconnecting the abstract components via these arrival curves, they can seamlessly be embedded into a MPA performance model with other abstract components.

Similarly as shown in this chapter, we could also develop abstract components for various other components with different processing semantics, thus constantly enlarging the application domain of the MPA framework. The embedding of such new abstract components into MPA performance models would thereby typically also be enabled via and interconnection of arrival curves or service curves.

When we take the idea of embedding abstract components with various processing semantics into the MPA framework one step further, we could even think of embedding abstract components that are not specified by a set of mathematical relations, but that instead use other specification and analysis methods. Such a component could for example internally be specified and analyzed by a timed automata or by a SymTA/S model. To interconnect such an abstract component, we would only need to develop methods to convert arrival curves into the input event models that are used by the respective methods, and to convert the output event models back into arrival curves.

4

Workload Variability and Correlations

In complex embedded systems the total execution demand of a sequence of consecutive events is often smaller than the sum of the individual worst-case execution demands. This phenomenon is most notably observed in embedded systems for multimedia applications [HKA+01], but it is also common in many other embedded systems with complex applications.

When we investigate systems that exhibit this phenomenon, we discover that events arriving on their input streams can often be classified into different classes. Some streams thereby explicitly contain events of different types, such as for example an MPEG-2 stream, while on others, events can be classified for example based on their payload size. Events of different classes then typically impose different workload to the system, and in particular they usually have different worst-case and best-case execution demands. And sometimes, even the execution demand of a single event class is not fixed, but depends instead on the internal state of a system, such as for example on its cache state.

Confronted with the problem of performance analysis of such complex embedded systems, we are interested in finding performance bounds based on the workload imposed by the whole event stream rather than by individual worst-case events within the stream. However, many performance analysis methods have no means to analyze the workload imposed by a complex event stream on a possibly complex system. To obtain hard analysis bounds, these methods assume every event to have the largest possible execution demand any event could have on the system, see e. g. [LL73]. Effectively, these methods analyze the worst-case situation

where every event is an element of the event class with the largest execution demand and on the same time every event leads to a cache miss. While this situation is theoretically possible in some systems, it can often be excluded in other systems because of known event correlations and dependencies on the input streams. For the latter systems, classical worst-case performance analysis leads to overly pessimistic results, and thus to expensive system designs with considerable performance reserves.

An additional phenomenon is often observed when complex embedded systems are implemented on multiple processors, be it as a distributed system or as a multi-processor system on chip. On these systems, events of different classes typically create different workloads on most or all components, and these workloads are often correlated. For example in data processing systems, the size of an event's payload data will typically determine its execution demand on most or all system components, leading to highly correlated workloads. Exploiting the knowledge of these workload correlations often allows to obtain again less pessimistic performance analysis results.

This chapter introduces models and methods to capture the knowledge of existing event correlations and workload correlations and to exploit this information for performance analysis with the MPA framework. The next section provides an introduction and overview to the problem of workload modeling within the MPA framework. In Section 4.2, a flexible method to capture event type correlations on event streams is introduced that allows to improve performance analysis results on systems with event-based workload variability. In Section 4.3 on the other hand, a powerful but more restricted method is introduced to capture both event type correlation on event streams, as well as functional workload dependencies within a component. This allows to improve performance analysis results on systems with both, event-based as well as functional workload variability. In Section 4.4, a method to capture and exploit workload correlations is presented. And in Section 4.5, an efficient algorithm to solve the maximum-weight path problem in a weighted graph is presented, a problem whose solution plays a crucial role in most methods presented in this chapter. The chapter concludes with an overview on related work in Section 4.6, and a discussion in Section 4.7

4.1 Workload Transformations

In Section 2.2.3 we first asserted the need for workload transformations; because as defined in Chapter 2, arrival curves denote the number of events that arrive on an event stream in any given time interval, while service curves denote the resource supply, typically expressed in resource

units such as number of processor cycles, that is available within a given time interval. It is apparent that in order to apply any of the mathematical relations within the MPA framework, such as for example (2.5) or (2.6), both, arrival curves and service curves, must be expressed in the same base unit.

4.1.1 Events and Workloads

To express both, arrival curves and service curves, in the same base unit, we can either express both in units of events, or we can express both in units of resources. We therefore distinguish between event-based arrival and service curves and resource-based arrival and service curves:

Def. 3: **(Event-Based Arrival Curves)** *An event-based arrival curve* $\bar{\alpha}(\Delta) = [\bar{\alpha}^u(\Delta), \bar{\alpha}^l(\Delta)]$ *models an event stream, where* $\bar{\alpha}^u(\Delta)$ *and* $\bar{\alpha}^l(\Delta)$ *provide an upper and a lower bound on the number of events that arrive in any time interval* Δ*, respectively.*

Def. 4: **(Event-Based Service Curves)** *An event-based service curve* $\bar{\beta}(\Delta) = [\bar{\beta}^u(\Delta), \bar{\beta}^l(\Delta)]$ *models a resource, where* $\bar{\beta}^u(\Delta)$ *and* $\bar{\beta}^l(\Delta)$ *provide an upper and a lower bound on the number of events that can be processed in any time interval* Δ*, respectively.*

Def. 5: **(Resource-Based Arrival Curves)** *A resource-based arrival curve* $\alpha(\Delta) = [\alpha^u(\Delta), \alpha^l(\Delta)]$ *models an event stream, where* $\alpha^u(\Delta)$ *and* $\alpha^l(\Delta)$ *provide an upper and a lower bound on the resource demand imposed by the event stream in any time interval* Δ*, respectively.*

Def. 6: **(Resource-Based Service Curves)** *A resource-based service curve* $\beta(\Delta) = [\beta^u(\Delta), \beta^l(\Delta)]$ *models a resource, where* $\beta^u(\Delta)$ *and* $\beta^l(\Delta)$ *provide an upper and a lower bound on the available resource supply in any time interval* Δ*, respectively.*

Since performance analysis problem specifications typically do not express arrival curves and service curves in the same base units, we need workload transformations that relate event-based curves to resource-based curves and vice versa. In order to retain hard bounded performance analysis results within the MPA framework, it is thereby important that these workload transformations are conservative. In the most simple case, where all events create a constant resource demand d on a component (i. e. the worst-case demand equals the best-case demand), event-based curves can simply be multiplied with the resource demand of a single event, and resource-based curves can be divided by the same number. In

the latter case floor and ceil operators can be applied additionally:

$$\alpha^u(\Delta) = \overline{\alpha}^u \cdot d \qquad \alpha^l(\Delta) = \overline{\alpha}^l \cdot d \tag{4.1}$$

$$\overline{\alpha}^u(\Delta) = \lceil \alpha^u / d \rceil \qquad \overline{\alpha}^l(\Delta) = \lfloor \alpha^l / d \rfloor \tag{4.2}$$

$$\beta^u(\Delta) = \overline{\beta}^u \cdot d \qquad \beta^l(\Delta) = \overline{\beta}^l \cdot d \tag{4.3}$$

$$\overline{\beta}^u(\Delta) = \lceil \beta^u / d \rceil \qquad \overline{\beta}^l(\Delta) = \lfloor \beta^l / d \rfloor \tag{4.4}$$

In a more general case where various events may create different resource demands, the workload transformations are not as simple anymore and we need more powerful methods to capture the relation between the number of events and the created resource demand in a component. Workload curves that were first introduced by Maxiaguine et al. [MKT04] provide the required expressiveness and are presented in the following section.

4.1.2 Workload Curves

Workload curves are a powerful model to characterize workload variability on a component, and they can be used to transform event-based curves into resource-based curves.

Def. 7: **(Workload Curves)** *Let $W(u)$ denote the total resource demand created on a component by u consecutive events of an incoming event stream. For every event sequence on the incoming event stream, the lower workload curve γ^l and the upper workload curve γ^u satisfy the relation:*

$$\gamma^l(v-u) \leq W(v) - W(u) \leq \gamma^u(v-u) \quad , \forall u < v \tag{4.5}$$

To transform resource-based curves into event-based curves on the other hand, we need to introduce the notion of pseudo-inverse workload curves.

Def. 8: **(Pseudo-Inverse of Workload Curves)** *The pseudo-inverse of an upper workload curve γ^u is defined by the function*

$$\gamma^{u^{-1}}(r) = \sup\{e : \gamma^u(e) \leq r\} \tag{4.6}$$

, and the pseudo-inverse of a lower workload curve γ^l is defined by the function

$$\gamma^{l^{-1}}(r) = \inf\left\{e : \gamma^l(e) \geq r\right\} \tag{4.7}$$

From these definitions, it follows that any sequence of e consecutive events, will create a total resource demand of at least $\gamma^l(e)$ and at most $\gamma^u(e)$ on a component. And analogously, any resource supply of r consecutive units of resources allows to process at most $\gamma^{l-1}(r)$ and at least $\gamma^{r-1}(r)$ events on a component. Using workload curves and their pseudo-inverse, arrival and service curves can therefore be transformed from event-based to resource-based quantities and vice versa:

$$\alpha^u(\Delta) = \gamma^u(\overline{\alpha}^u(\Delta)) \qquad \alpha^l(\Delta) = \gamma^l(\overline{\alpha}^l(\Delta)) \tag{4.8}$$

$$\overline{\alpha}^u(\Delta) = \gamma^{l-1}(\alpha^u(\Delta)) \qquad \overline{\alpha}^l(\Delta) = \gamma^{u-1}(\alpha^l(\Delta)) \tag{4.9}$$

$$\beta^u(\Delta) = \gamma^u(\overline{\beta}^u(\Delta)) \qquad \beta^l(\Delta) = \gamma^l(\overline{\beta}^l(\Delta)) \tag{4.10}$$

$$\overline{\beta}^u(\Delta) = \gamma^{l-1}(\beta^u(\Delta)) \qquad \overline{\beta}^l(\Delta) = \gamma^{u-1}(\beta^l(\Delta)) \tag{4.11}$$

4.1.3 Embedding Workload Transformations

As already mentioned above, workload transformations must be conservative in order to guarantee to retain hard bounded analysis results within the MPA framework. Workload curves fulfill this requirement by definition, and therefore the following relations (and the analogue relations for resource-based arrival curves and event- as well as resource-based service curves) hold for all workload curves and their pseudo-inverse:

$$\overline{\alpha}^u(\Delta) \leq \gamma^{l-1}(\gamma^u(\overline{\alpha}^u(\Delta))) \tag{4.12}$$

$$\overline{\alpha}^l(\Delta) \geq \gamma^{u-1}(\gamma^l(\overline{\alpha}^l(\Delta))) \tag{4.13}$$

From the definition of workload curves and their pseudo-inverse, it becomes apparent that the farther apart $\gamma^u(e)$ and $\gamma^l(e)$ are, that is the more workload variability there exists on a component, the larger will be the difference between the left and the right sides of (4.12) and (4.13). And in general, this difference may be substantial, as we will see in Section 4.4.5. Hence, a double workload transformation (i. e. transforming event based to resource-based curves and back again to event-based curves, or the other way around) typically leads to valid but overly pessimistic bounds, even if all input curves to the double transformation are tight. In contrast to this, a single workload transformation (i. e. transforming event-based to resource-based curves or the other way around) leads again to a tight curve provided that the input curves are tight. Consequently, we should omit the use of double workload transformations during performance analysis, whenever possible. The optimal method to embed workload transformations in the analysis of an abstract component, while completely omitting double workload transformations is depicted in Figure 37. Any other arrangement of transformations and computations

within an abstract component would require a double workload transformation. For a more thorough discussion on this, see [Max05].

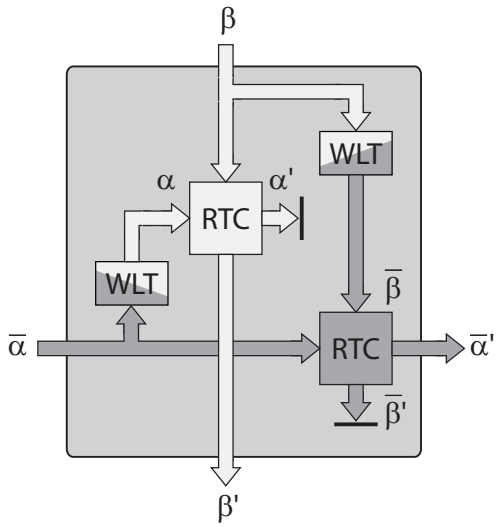


Fig. 37: Embedding of workload transformations in the analysis of an abstract component, without double workload transformation. The bright data flows denote resource-based curves, while the dark data flows denote event-based curves. In the two blocks labeled *WLT*, the workload transformations are applied according to (4.8)–(4.11), and in the two blocks labeled *RTC*, the abstract component relations are used to compute outgoing arrival and service curves in the respective base unit, according to (2.5) and (2.6).

4.2 Event-Based Workload Variability

A large fraction of the workload variability within a system can often directly be connected to the correlations of different event types on the incoming event stream. If these event-correlations are known, they can be used, together with the worst-case and best-case execution demand of every event type, to determine the workload curves for a system. In this section, we introduce type rate curves, a model to capture arbitrary event type correlations on event streams, and we present a method to compute workload curves, based on the information on event correlations that is captured with type rate curves.

4.2.1 Type Rate Curves

Type rate curves are a powerful and compact model to characterize the event-occurrence variability within an event stream.

Def. 9: **(Type Rate Curve)** *Consider an event stream on which events of n different types t_i are present, and let $E_i(u)$ denote the total number of events of type t_i on a sequence of u consecutive events. For every event sequence on the incoming event stream, and for any event type t_i, the lower type rate curve σ_i^l and the upper type rate curve σ_i^u satisfy the relation:*

$$\sigma_i^l(v-u) \leq E_i(v) - E_i(u) \leq \sigma_i^u(v-u) \quad \forall 0 \leq u \leq v$$

Using type rate curves, the event correlations on an event stream can be characterized by the set of all type rate curves $\{\sigma_i^u, \sigma_i^l\}$ of all event types occurring in it.

Ex. 4: *Consider an event stream with events of three different event types A, B and C respectively, that always occur strictly following one of two patterns, either ABCBCA, or AACB. The order in which the patterns themselves occur on the stream is assumed to be completely random. In Figure 38, the complete set of type rate curves for the stream is shown. From these type rate curves, we can directly obtain information on possible type compositions in event sequences of the stream. For example, we can conclude that in any event sequence of length* 12*, at least* 4 *and at most* 7 *events are of type A, and at least* 2 *and at most* 5 *events are of type B, whereas the number of events of type C is between* 3 *and* 4*.*

Note, that in case of an event with completely uncorrelated, i. e. statistically independent, occurrences, the corresponding type rate curve has the form $\sigma^u(e) = e$ and $\sigma^l(e) = 0$, respectively. The existence of any event-occurrence correlations on the other hand becomes immediately apparent, since the corresponding type rate curve has the form $\sigma^u(e) < e$ or $\sigma^l(e) > 0$ for some values of e.

Type rate curves can be derived from virtually any formal event specification method. For example, suppose a finite state machine representation of possible event-occurrence patterns; every edge in this finite state machine is labeled with an event type, and any path within the finite state machine corresponds to a valid event type sequence. To compute the type rate curves of an event type t_i from such an automaton, we first annotate every edge that is labeled with t_i with a weight 1, and we annotate all other edges with a weight 0. In the resulting weighted graph, the weight of the maximum-weight path with length e equals the value of the upper type rate curve $\sigma_i^u(e)$, while the weight of the minimum-weight path with length e equals the value of the lower type rate curve $\sigma_i^l(e)$.

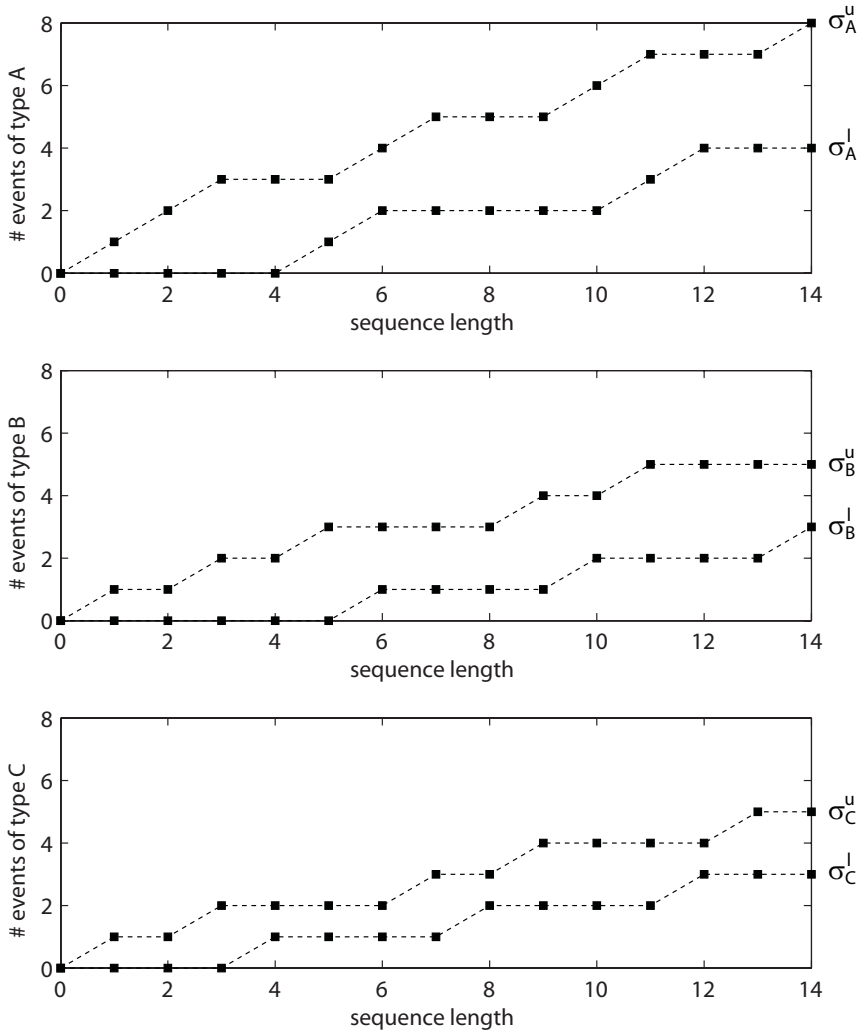


Fig. 38: The type rate curves σ_A, σ_B, and σ_C, of the event stream of Example 4.

The maximum- and minimum-weight paths can be computed efficiently using Algorithm 2 presented in Section 4.5.

In practice it may often also be useful to determine the type rate curves corresponding to a set of finite length event stream traces, obtained for example from observation or simulation. Even though the so obtained curves must not be used for the analysis of hard real-time systems, they can be very useful to analyze systems with soft real-time constraints.

4.2.2 Computing Workload Curves

To compute the workload curves that model the workload imposed by an event stream on a system, we first need to determine the worst-case (WCED) and best-case (BCED) execution demands of every event type of the incoming event stream on the corresponding system. Together with the respective type rate curves, we collect this data in a data set $data_i = \{\sigma_i^u, \sigma_i^l, WCED_i, BCED_i\}$ for each of the n event types. To compute the upper workload curve, the indices of these data sets must then be reordered, such that $data_1$ contains the data of the event type with the largest WCED, while $data_2$ represents the event type with the second largest WCED, and so on. The upper workload curve $\gamma^u(e)$ can then be computed as:

$$\gamma^u(e) = \sum_{i=1}^{n} \min\left\{ \max\left\{ e - \left(\sum_{k=1}^{i-1} \sigma_k^u(e) + \sum_{k=i+1}^{n} \sigma_k^l(e) \right), \sigma_i^l(e) \right\}, \sigma_i^u(e) \right\} \cdot WCED_i \tag{4.14}$$

The above formula computes the worst-case workload, that an event sequence of length $e \in \mathbb{Z}_{\geq 0}$ could possibly impose to the system. In this worst-case event sequence, at least $\sigma_i^l(e)$ events of every event type t_i must be present. After fulfilling all minimum requirements, the remainder of the sequence is filled up using events with largest possible WCED, while still considering the upper bound $\sigma_i^u(e)$ on the possible occurrence for every event type.

The lower workload curve $\gamma^l(e)$ can be computed similarly. For this, the indices of the data sets must first be reordered, such that $data_1$ contains the data of the event type with the smallest BCET, and then (4.14) can be used, whereas we need to replace $WCED_i$ with $BCED_i$:

$$\gamma^l(e) = \sum_{i=1}^{n} \min\left\{ \max\left\{ e - \left(\sum_{k=1}^{i-1} \sigma_k^u(e) + \sum_{k=i+1}^{n} \sigma_k^l(e) \right), \sigma_i^l(e) \right\}, \sigma_i^u(e) \right\} \cdot BCED_i \tag{4.15}$$

Ex. 5: *Suppose an abstract component that is triggered by the event stream described in Example 4. The worst-case and best-case execution demands for the three different event types of the event stream are given as* $BCED_A = 3$, $WCED_A = 5$, $BCED_B = 2$, $WCED_B = 9$, $BCED_C = 1$ *and* $WCED_C = 2$. *With these execution demand bounds, and with the type rate curves depicted Figure 38, we can then use (4.14) and (4.15) to compute the workload curve* γ_{TRC}, *depicted in Figure 39. In the same figure, the workload curves* γ_{WC} *are depicted, that are computed without considering event correlations, thus assuming that the worst-case execution demand of every event is* $WCED_B = 9$, *while its best-case execution demand is* $BCED_C = 1$.

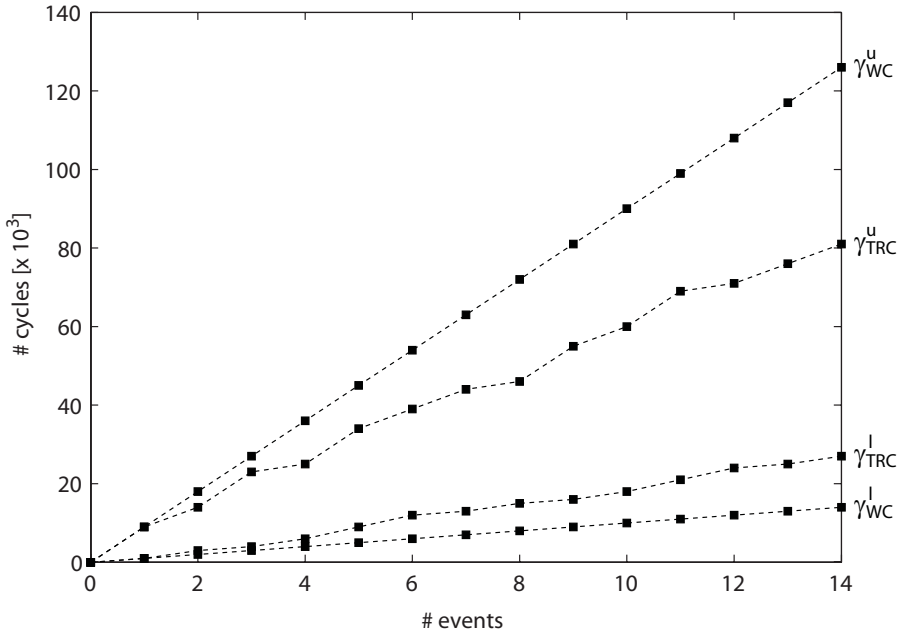


Fig. 39: The workload curves of the system described in Example 5, when considering event correlations using type rate curves (γ_{TRC}), and without considering event correlations (γ_{WC}).

4.3 Functional Workload Variability

In the previous section, we related every event type of an incoming event stream to a fixed worst-case and best-case execution demand within a system. In many complex systems however, the worst-case and best-case execution demand of a single event type is not fixed, but depends instead on the internal state of the system. In this section, we introduce event automata and workload variability automata that allow to capture complex state dependent workload variabilities within a system, and we present methods to compute workload curves, based on the workload variability information that is captured with these automata models.

4.3.1 Event Sequence Automata

Event sequence automata capture the functional information of an event stream, that is the information on admissible event type sequences that may arrive on the event stream. This information is modeled in a state-transition graph whose transitions are labeled with symbols corresponding to the different event types occurring on the event stream.

Def. 10: **(Event Sequence Automaton)** *An event sequence automaton F_σ is a tuple (S, S^0, Σ, T), where S is a set of states, $S^0 \subseteq S$ is a set of initial states, Σ is a set of event types, and $T \subseteq S \times \Sigma \times S$ is a set of transitions. The system starts in an initial state, and if $s \xrightarrow{\sigma} s'$, then an event σ may occur on the event stream, leading to a state change from s to s'.*

When we combine the functional information of an event stream that is captured in its event sequence automaton F_σ, with its timing information that is captured in its arrival curve $\bar{\alpha} = [\bar{\alpha}^u, \bar{\alpha}^l]$, we know that in any time interval of length Δ, at least $\bar{\alpha}^u(\Delta)$ and at most $\bar{\alpha}^l(\Delta)$ number of events arrive on the stream, and that the possible sequences of arriving events is limited to valid runs of given length in the event automaton F_σ.

Ex. 6: *Figure 40 shows an example of an event sequence automaton for an event stream with three different types of events, A, B, and C. The event automaton restricts the admissible event sequences on the event streams to valid runs within the automaton.*

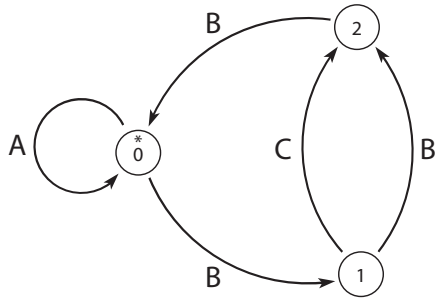


Fig. 40: An event sequence automaton that specifies the admissible event sequences on an event stream with three different event types, *A*, *B*, and *C*. The initial state is marked with an asterisk.

4.3.2 Workload Variability Automata

Workload variability automata capture the functional workload variability information of an abstract component that is triggered by an event stream with different event types, whose admissible event sequences are specified by an event sequence automaton. A workload variability automaton thereby specifies the worst-base and best-case execution demand of an incoming event, depending on its event type, as well as on the internal state of the abstract component. Besides this, a workload variability automaton also captures the information required to generate an event sequence automaton for a possibly existing outgoing event stream of the

abstract component. All this information is modeled in a state-transition graph.

Def. 11: (Workload Variability Automaton) *A workload variability automaton F_γ is a tuple $(S, S^0, \Sigma_I, \Sigma_O, D, T)$, where S is a set of states, $S^0 \subseteq S$ is a set of initial states, Σ_I is the non-empty set of accepted incoming event types and Σ_O is the set of generated outgoing event types. Further, D is a demand function $D : S \times \Sigma_I \times \Sigma_O \times S \rightarrow [\mathbb{Z}^+, \mathbb{Z}^+]$. This function determines the upper and lower bound of the resources required by an abstract component to process an incoming event, generate an outgoing event and change its internal state. Finally, $T \subseteq S \times \Sigma_I \times \mathbb{Z}^+ \times \mathbb{Z}^+ \times \Sigma_O \times S$ is a set of transitions.*

A workload variability automaton starts in an initial state $s \in S^0$, and if $s \xrightarrow{\sigma_I/[d_l,d_u]/\sigma_O} s'$, and if the system is triggered by an incoming event of type σ_I, the abstract component has a resource demand of at least d_l and at most d_u resource units to emit an event of type σ_O and change its state from s to s'.

Workload variability automata allow to flexibly choose the level of detail for every system component, such that single, performance critical system components may be modeled with more details than others. The simplest workload variability automaton would thereby consist of only a single state with a self-loop for every accepted event type.

Ex. 7: *Figure 41 shows a workload variability automaton that models the functional workload variability behavior of a simple component with a LRU cache that accepts an event stream with the event sequence automaton depicted in Figure 40. The LRU cache of the modeled component has one cache block that can hold the program code to process an event of either type* A, *or* B. *Initially the cache is empty, and whenever an event of type* A *or* B *arrives, and the correct program code is not available within the cache, the code is loaded into the cache and the event is processed. This generates a resource demand of* 10*E*6 *and* 15*E*6 *cycles for an event of type* A, *or* B, *respectively. If the program code is already available within the cache, events of both types,* A *and* B, *generate a resource demand of only* 5*E*6 *cycles. The program code to process events of type* C *cannot be loaded into the cache, and an arriving event of type* C *always generates a state-independent resource demand of at least* 3*E*6 *and at most* 20*E*6 *cycles. Additionally, the output generated after processing an event of type* C *depends on the internal state of the component; the component generates an event of type* F *if the LRU cache is empty, and an event of type* G *otherwise.*

4.3.3 Computing Workload Curves

To compute the workload curves for an abstract component with a workload variability automaton F_γ, that is triggered by an event stream with an

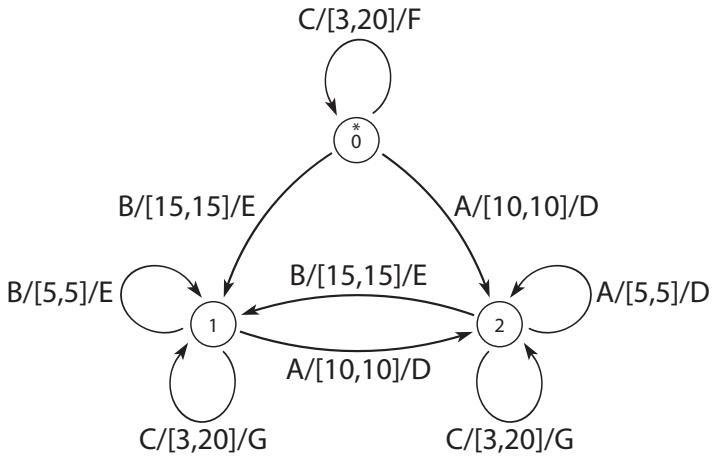


Fig. 41: The workload variability automaton automaton specified in Example 7. The worst-case and best-case resource demands are given in units of 10^6 cycles, and the initial state is marked with an asterisk.

event sequence automaton F_σ, we first need to build the product automaton $F_{Prod} = F_\sigma \times F_\gamma$. The states of the product automaton are determined as the product of the their individual states $S_{Prod} = S_\sigma \times S_\gamma$, with $S^0_{Prod} = S^0_\sigma \times S^0_\gamma$ as initial states, and in the product automaton a transition between two states exists, if either corresponding transitions with equal event types σ_I existed in both, the event sequence automaton and the workload variability automaton (we say that these two automata synchronize on such transitions):

$$\begin{aligned} T_{Prod} \quad = \quad & \{((u,v),\sigma_I,[d_l,d_u],\sigma_O,(u',v'))| \\ & (u,\sigma_I,u') \in T_\sigma \wedge (v,\sigma_I,[d_l,d_u],\sigma_O,v') \in T_\gamma\}\} \end{aligned}$$

And finally, we remove all states and state trajectories within the product automaton that are not reachable anymore from any initial state.

Ex. 8: *Figure 42 shows the product automaton F_{Prod} of the event sequence automaton F_σ from Example 6, and the workload variability automaton F_γ from Example 7. Note, the product automaton consists of only 5 states, since 4 states of the initial product are not reachable.*

The upper workload curve $\gamma^u(e)$ of an abstract component bounds the maximum resource demand that any event stream sequence of length e may create in the component. To compute this upper workload curve γ^u from the product automaton, we interpret the upper resource demand d_u on every transition as the weight of the transition. The weight $w^u(e)$ of the maximum-weight path with length e in this weighted product automaton

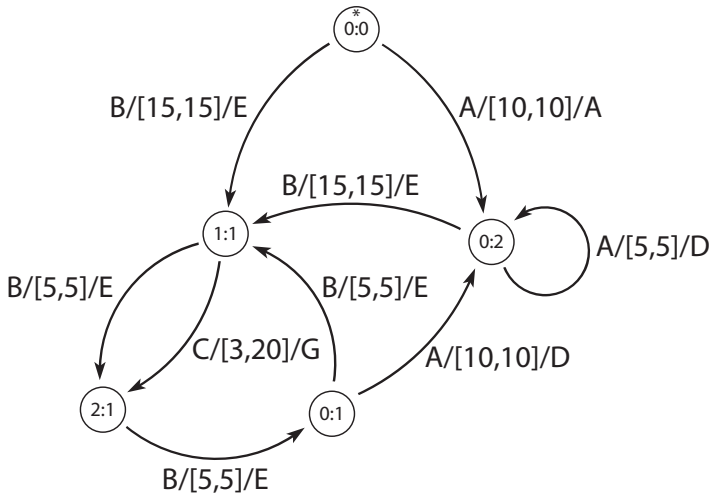


Fig. 42: The product automaton F_{Prod} of the event sequence automaton F_σ from Example 6 and the workload variability automaton F_γ from Example 7. The worst-case and best-case resource demands are given in units of 10^6 cycles, and the initial state is marked with an asterisk.

then equals the value of the upper workload curve $\gamma^u(e)$. For the lower workload curve γ^l, we follow the same procedure, but instead of the upper resource demand d_u, we interpret the lower resource demand d_l as the weight of a transition. The weight $w^l(e)$ of the minimum-weight path with length e in this weighted product automaton then equals the value of the lower workload curve $\gamma^l_I(e)$. Note that both, the maximum- and minimum-weight paths of a weighted graph can be computed efficiently using Algorithm 2 presented in Section 4.5.

4.3.4 Experimental Results

Following, we analyze a simple component with an LRU-cache within the MPA framework, using the presented functional automata models. As a reference, we analyze the same system first by employing traditional worst-case analysis, and secondly with by using type rate curves to model the event correlations on the input event stream.

4.3.4.1 Application Scenario

Consider the abstract component with LRU cache described in Example 7, with the workload variability automaton shown in Figure 41. Suppose this component runs on an unloaded processor with a fixed processing capacity of 20*E*6 cycles per second, and it must process an event stream with

period $p = 1s$ and jitter $j = 400ms$, with three different event types that arrive according to the event sequence automaton presented in Example 6, and depicted in Figure 40.

4.3.4.2 The Resource Demand Imposed by the Event Stream

Before analyzing the application scenario with the MPA framework, we must compute the workload curves γ^u and γ^l for the abstract component.

In traditional worst-case/best-case analysis, we have no means to exploit event correlations on the event stream and workload variability information of the component, and we must assume each arriving event to impose the maximum possible or minimum possible resource demand any event could impose on the processing unit. In our application scenario, event C has both the maximum and the minimum resource demand. The resulting workload curves γ^u_{WC} and γ^l_{WC} are shown in Figure 43.

Type rate curves, presented in Section 4.2, model event correlations on event streams by bounding the number of occurrences of every event type in an event stream sequence of given length. This model allows to capture correlations on an event stream to some extent, but with type rate curves it is not possible exploit any information about state-dependent workload variability within an abstract component. Instead every event type is assumed to lead to a fixed worst-case and best-case execution demand, and we can therefore not take into account the caching effects of the component. The workload curves γ^u_{TRC} and γ^l_{TRC} computed using a type rate curve model of the event stream are also shown in Figure 43.

In difference to the above analysis methods, the automata models presented in this section allow to capture detailed information of both the event stream as well as the abstract component. This allows to exploit both, the event correlations on the incoming event stream, as well as the caching effects on the abstract component. The resulting workload curves γ^u_{AM} and γ^l_{AM} are depicted in Figure 43.

4.3.4.3 Performance Analysis Problems and Results

Prob. 4: *The maximum delay experienced by an event when processed by the system must not be longer than $d_{max} = 1s$. What is the minimum processor frequency f_{min}, required to guarantee the processing of the event stream under this condition? If we build the system with a processor with this minimum processor frequency, share c_{rem} of the processing cycles are guaranteed to remain to process lower priority tasks?*

We first compute the resource-based arrival curve α of the event stream that enters the system, by performing the workload transformation (4.8) on the event-based arrival curve, using the workload curves depicted in

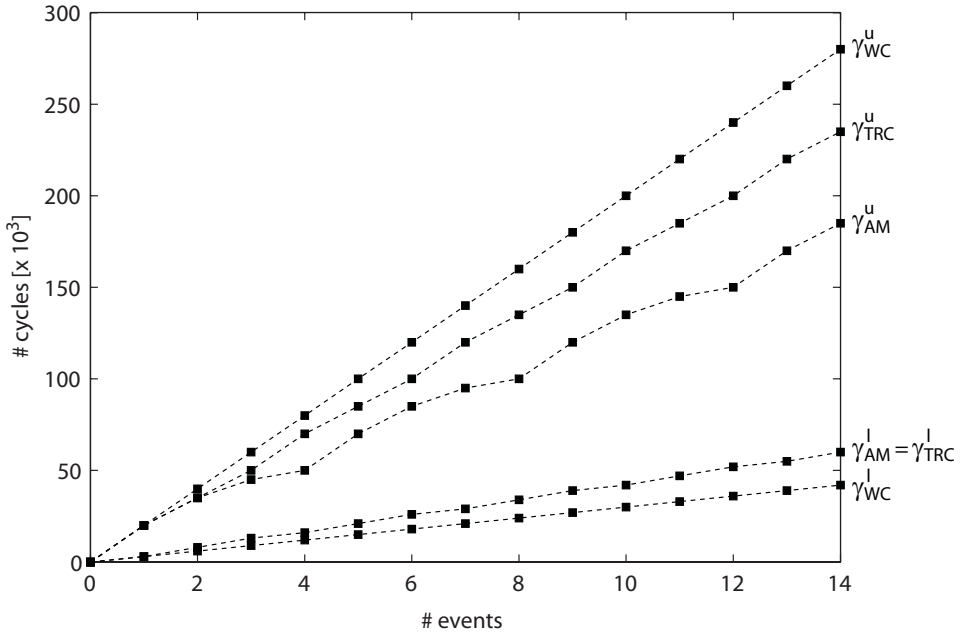


Fig. 43: The workload curves of the abstract component of the application scenario. γ^u_{WC} and γ^l_{WC} are computed using worst-case/best-case analysis, γ^u_{TRC} and γ^l_{TRC} are computed using type rate curves, and γ^u_{AM} and γ^l_{AM} are computed using event sequence automata and workload variability automata.

Figure 43. Using (2.11), we can then compute the minimum required processor frequency as

$$f_{min} = \inf_{\Delta \geq 0} \left\{ \frac{\alpha(\Delta - d_{max})}{\Delta} \right\} \tag{4.16}$$

And with the resource-based remaining service curve $\beta^{l'}$ that we get from (2.10), we can compute the guaranteed remaining processing capacity as

$$c_{rem} = \frac{\lim_{\Delta \to \infty} \left(\frac{\beta^{l'}(\Delta)}{\Delta} \right) - f_{min}}{f_{min}} \cdot 100\% \tag{4.17}$$

From the results in Table 3, we learn that the use of automata models to model event correlations and workload variability information, allows to obtain the tightest analysis results for both, the minimum required processor frequency, as well as for the guaranteed remaining processing capacity. The use of type rate curves on the other hand still allows to obtain tighter results than we would get from traditional worst-case/best-case analysis, but since the caching effects in the abstract component

cannot be considered, the results are inferior to the results we get by using automata models.

Analysis Method	f_{min}	c_{rem}	
		f_{min} = 219 MHz	f_{min} = 250 MHz
Worst-Case/Best-Case	250 MHz	—	20 %
Type Rate Curves	219 MHz	23.7 %	33.3 %
Automata Models	219 MHz	42.9 %	50.0 %

Tab. 3: Performance analysis results for Problem 4.

Prob. 5: *What is the minimum processor frequency f_min, required to guarantee the processing of the arriving event stream, if a finite size buffer is available, and if no requirements on the maximum delay exist? What is then the upper bound d_{max} to the maximum delay experienced by an event on the event stream?*

According to (2.12), the backlog at an abstract component is bounded if the long-term slope of α^u is less or equal than the long-term slope of β^l. We can therefore compute the minimum required processor frequency as

$$f_{min} = \lim_{\Delta \to \infty} \left(\frac{\alpha^u(\Delta)}{\Delta} \right) \tag{4.18}$$

The upper bound d_{max} to the maximum delay experienced by an event can then be computed with (2.11).

From the analysis results shown in Table 4, we learn that if we allow slightly longer delays, we can choose a processor with a considerably lower processing speed that is still guaranteed to process the incoming event stream with finite buffers and with a bounded maximum delay.

Analysis Method	f_{min}	d_{max}
Worst-Case/Best-Case	200 MHz	1.4 s
Type Rate Curves	167 MHz	1.6 s
Automata Models	125 MHz	2.2 s

Tab. 4: Performance analysis results for Problem 5.

4.4 Workload Correlations

When complex embedded systems with workload variability are implemented on multiple processors, we can often observe that different event types create different workload on most or all components, and that theses workloads are often correlated. In this section, we introduce workload correlation curves, a model to capture information on workload correlations between components, and we present a method to compute workload correlations curves, based on the event sequence automata and the workload variability automata of a distributed system.

4.4.1 An Application Scenario

Let us first consider an application scenario of a multi-processor system on chip (MPSoC) with workload correlations, that will serve as a running example throughout this section, and that will help to visualize the effects of the presented methods. In Section 4.4.5, the performance analysis results of this application scenario are provided discussed. An overview of the MPSoC under consideration is depicted in Figure 44. A two-processor system on chip serves two event streams, both entering and leaving independently through the I/O-interfaces of the chip, and both event streams have hard real-time processing requirements.

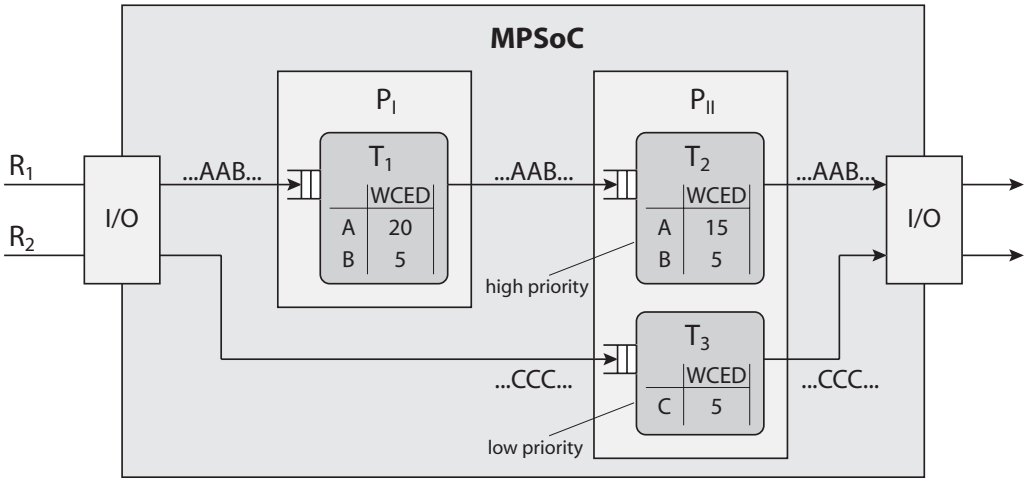


Fig. 44: A multi-processor system on chip with correlated execution demands. The execution demands are given in units of 10^3 cycles.

The event stream R_1 is processed consecutively by the tasks T_1 and T_2, that are running on the processors P_I and P_{II}, respectively. The events on this stream arrive with a period $p_{R_1} = 4ms$, a jitter $j_{R_1} = 15ms$ and a minimum inter-arrival time of $d_{R_1} = 1ms$. Further, the events may be

differentiated into two event types, with different execution demands. Events of type *A* have an execution demand of 20*E*3 cycles in T_1, and a demand of 15*E*3 cycles in T_2. Events of type *B* on the other hand are less resource demanding and demand only 5*E*3 cycles, both in T_1 as well as in T_2. Finally, no correlations are known on the arrival of the different events. The event sequence automaton of R_1, as well as the workload variability automata of T_1 and T_2 are depicted in Figure 45.

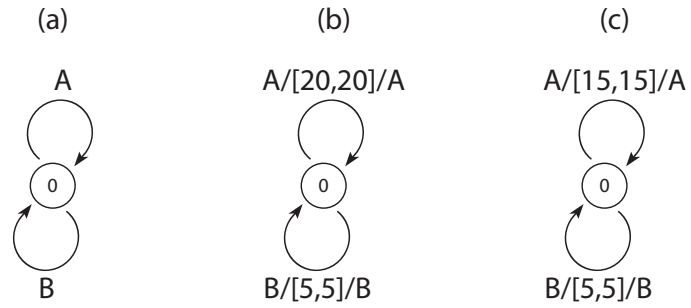


Fig. 45: (a) The event sequence automaton of the event stream R_1; (b) the workload variability automaton of the task T_1; and (c) the workload variability automaton of the task T_2.

The event stream R_2 is processed by the task T_3 that is running on processor P_{II}. Events on this stream arrive with a period $p_{R_2} = 6ms$, and a jitter of $j_{R_2} = 1ms$. All events on this stream are of type *C* and have an execution demand of 5*E*3 cycles in T_3.

On processor P_{II}, the tasks T_2 and T_3 are scheduled using a preemptive fixed priority scheduler, where, according to the rate monotonic scheduling scheme [LL73], T_2, processing the stream with period $p_{R_1} = 4ms$, has the higher priority than T_3 that is processing the stream with period $p_{R_2} = 6ms$). Since T_3 has a lower priority than T_2, the events of the stream R_2 may experience a processing delay due to the interference of T_2.

In this MPSoC, we nicely see the correlations of the workloads that the stream R_1 is creating on the two processors: an event of type *A*, that creates a high demand of 20*E*3 cycles on P_I again creates a high demand of 15*E*3 cycles on P_{II}. On the other hand, an event of type *B* that creates a demand of only 5*E*3 cycles on P_I, creates also a low demand of 5*E*3 cycles on P_{II}.

4.4.2 Event-Based and Resource-Based Analysis

In the analysis schema shown in Figure 37, (2.7)–(2.10) are applied to the resource-based curves α and β in the bright block labeled *RTC*, as well as to their event-based counterparts $\overline{\alpha}$ and $\overline{\beta}$, in the dark block labeled

RTC. Both sets of curves, α and β as well as $\overline{\alpha}$ and $\overline{\beta}$, model the same properties of a system, but both use a different method of abstraction, resource units vs. events, and both may therefore capture different aspects of the same system properties, as we will see below. However, in the analysis schema of an abstract component as shown in Figure 37, the outgoing resource-based arrival curve α' as well as the outgoing event-based service curve $\overline{\beta}'$ are discarded after the analysis, together with the different aspects of system properties modeled by them. It would of course be possible to obtain an event-based outgoing arrival curve $\overline{\alpha}'$ from a workload transformation (4.9) of the outgoing resource-based arrival curve α', and similarly we could obtain a resource based outgoing service curve β' from the event-based outgoing service curve $\overline{\beta}'$, using (4.10). But due to the double workload transformation step that would be introduced by this, these curves will normally be overly pessimistic.

Figures 46 and 47 show the event-based, and the resource-based outgoing arrival and service curves of task T_1 in the MPSoC in Figure 44, respectively. The event-based arrival curve $\overline{\alpha}_{T_1} = \overline{\alpha}_{R_1}$ and the resource-based service curves $\beta_{T_1} = \beta_{P_I}$ serve as initial input for the computation. With this input, all event-based and resource-based outgoing arrival and service curves are computed according to the analysis schema depicted in Figure 37. Additionally, the workload back-transformations of α'_{T_1} and $\overline{\beta}'_{T_1}$ are computed by applying (4.9) and (4.10), respectively: $\beta'^{*}_{T_1} = \gamma_{T_1}(\overline{\beta}'_{T_1})$ and $\overline{\alpha}'^{*}_{T_1} = \gamma^{-1}_{T_1}(\alpha'_{T_1})$.

Let us first discuss the event-based upper outgoing arrival curve $\overline{\alpha}^{u'}_{T_1}$ in Figure 46. Intuitively, we can say that the maximum possible event stream output (restricted by $\overline{\alpha}^{u'}$) will occur when the maximum possible event stream input (restricted by $\overline{\alpha}^{u}$) occurs together with the maximum possible resource availability (restricted by β^{u}). According to (2.7), the minimum possible resource availability (restricted by β^{l}) also plays a role, it restricts the maximum initial buffer fill level, but we will not consider it for the discussion here. From $\beta^{u}_{T_1}$ we know that in the best case there are 6*E*3 cycles available per millisecond to process incoming events on processor P_I. And since the less resource demanding event *B* on stream R_1 only requires 5*E*3 cycles to be processed by task T_1, we know that we could in the best case process 1.2 events per millisecond. This is expressed in $\overline{\beta}^{u}_{T_1}$. Because of this high event-based processing capacity (based on the case where all arriving events are of the less resource demanding type *B*), the maximum possible event stream output ($\overline{\alpha}^{u'}_{T_1}$) is only restricted by the availability of triggering input events ($\overline{\alpha}^{u}_{T_1}$), as we see in Figure 46.

Let us now concentrate on the resource-based upper outgoing arrival curve $\alpha^{u'}_{T_1}$ in Figure 47. The worst possible resource demand is created by

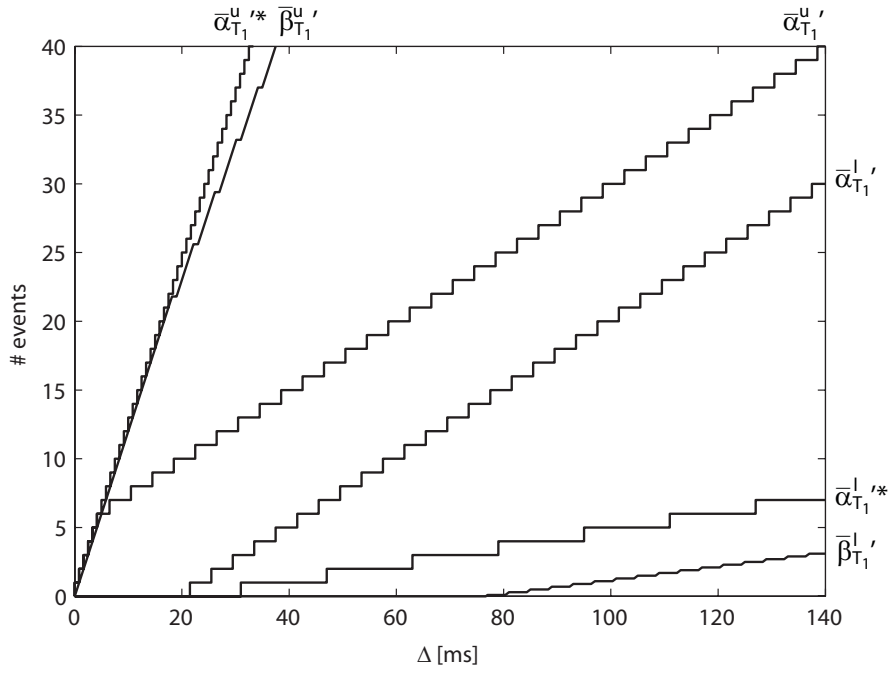


Fig. 46: The event-based output of the abstract component that models task T_1 in the MPSoC in Figure 44.

the input event stream when all arriving events are of the more resource demanding type *A*. In this case, a total execution demand of 6·20*E*3 cycles = 120*E*3 cycles is created during a burst of only little more than 5*ms*. But in the same time, a maximum of only 6·6*E*3 cycles = 36*E*3 cycles are available for processing, and therefore most of the arriving events will be buffered in the input buffer to task T_1. In this phase, the output event stream rate is restricted by the availability of resources. However, after a burst, the events on the input event stream will again arrive periodically, creating a maximum resource demand of 20*E*3 cycles every 4*ms*. And since in the same period a maximum of 24*E*3 cycles are available for processing, the input buffer of task T_1 will eventually be emptied again after a burst, leading into a phase where the maximum output event stream rate is restricted only by the availability of triggering input events. From the change of slopes of $\alpha^{u'}_{T_1}$ (and $\beta^{l'}_{T_1}$) at $\Delta \approx 77ms$ in Figure 47, we know that the length of the first phase, where the output event stream is restricted by the availability of resources, is upper bounded by 77*ms*.

When we use (4.9) to transform $\alpha^{u'}_{T_1}$ back to $\overline{\alpha}'^{*}_{T_1}$, we must assume that with every investment of 5*E*3 cycles we could in the best case generate an event of type *B* on the output event stream. But by taking this assumption, we neglect the fact that for the computation of $\alpha^{u'}_{T_1}$, we assumed that all

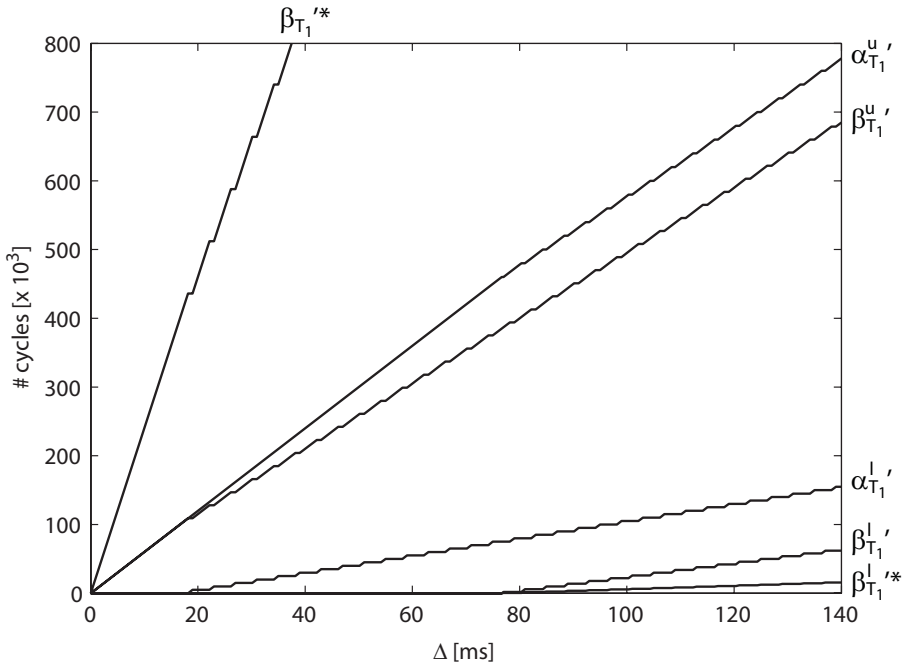


Fig. 47: The resource-based output of the abstract component that models task T_1 in the MPSoC in Figure 44.

events are of type *A*. This decoupling of worst-case and best-case events leads to the very pessimistic upper bound $\overline{\alpha}'^*_{T_1}$. Similar thoughts explain the pessimism of all other back-transformed curves in Figures 46 and 47.

From the results shown and discussed above, we clearly see that the resource-based outgoing arrival curve α' of an abstract component may carry valuable information, as for example the presence of a phase where the maximum available resources restrict the output event stream rate, that may not be present in the event based outgoing arrival curve $\overline{\alpha}'$, or in the back-transformed curve $\overline{\alpha}'^*$. But we also see that back-transforming the resource-based outgoing arrival curve α' into an event-based curve $\overline{\alpha}'^*$ is overly pessimistic, such that $\overline{\alpha}'^*$ does not contain much or any information of interest anymore. In the analysis of an abstract component, α' is therefore typically discarded, as depicted in Figure 37. In the following section, we present a new method, that allows to consider the information contained in α' in the analysis of succeeding performance components.

4.4.3 Workload Correlations Curves

We have seen in Section 2.5 that we can interconnect abstract components to the performance model of a complete system. But we may only use

event-based arrival curves $\overline{\alpha}$ and resource-based service curves β to interconnect abstract components. The reason for this restriction is that outgoing resource-based arrival curves of one component are not directly related to ingoing resource-based arrival curves of another component, since the same event causes in general a different resource demand in two components, when it is processed by two different tasks. Similar thoughts can be made for event-based service curves. As a direct consequence of this restriction, follows that the information captured by the outgoing resource based arrival curve α' of one component cannot be incorporated in the analysis of a succeeding component. To overcome this problem, we introduce workload correlation curves (WCC), that allow to directly transform the outgoing resource-based arrival curve α' of one abstract component into an ingoing resource-based arrival curve of another abstract component, by exploiting the knowledge of workload correlations existing between these components. No double workload transformation is needed for this, and therefore much of the information captured in α' will be preserved.

Figure 48 shows the performance model of the MPSoC of Figure 44, where the outgoing resource-based arrival curve of task T_1 is directly connected to the ingoing resource-based arrival curve of task T_2. The central part of this interconnection path is the box labeled *WCC*, where the resource-based arrival curves are transformed, using workload correlation curves. Workload correlation curves allow to analytically characterize and capture existing workload correlations between two components:

Def. 12: (Workload Correlation Curves) *Suppose r_{T_1} resource units are invested in a task T_1 to process incoming events and generate outgoing events. Further, let $W_{T_1 \to T_2}(r_{T_1})$ denote the total execution demand that is created in a task T_2 by the arrival of the events that were generated on the output of the preceding task T_1 by the investment of r_{T_1} resource units. Then, for any number of invested resources r_{T_1} in T_1, the lower workload correlation curve $\delta^l_{T_1 \to T_2}(r)$ and the upper workload correlation curve $\delta^u_{T_1 \to T_2}(r)$ satisfy the relation:*

$$\delta^l_{T_1 \to T_2}(v-u) \leq W_{T_1 \to T_2}(v) - W_{T_1 \to T_2}(u) \leq \delta^u_{T_1 \to T_2}(v-u), \forall u < v \qquad (4.19)$$

Therefore, whenever r_{T_1} resources are invested in a task T_1, the outgoing events that are generated by this resource investment will create a total resource demand of at least $\delta^l_{T_1 \to T_2}(r_{T_1})$ and at most $\delta^u_{T_1 \to T_2}(r_{T_1})$ resource units in a succeeding task T_2.

Using workload correlation curves, outgoing resource-based arrival curves of a task T_1 can be transformed directly into ingoing resource-based arrival curves of a task T_2:

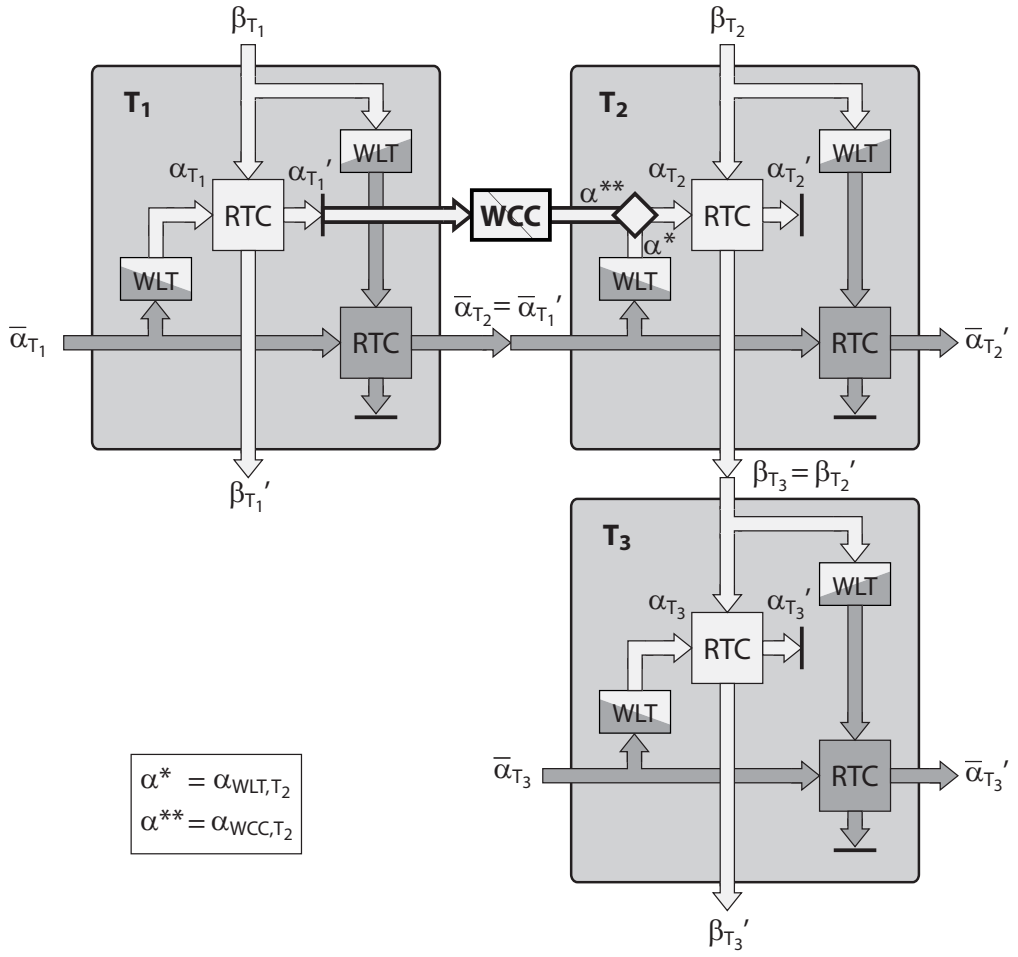


Fig. 48: The performance model of the MpSoC of Figure 44, with the new component interconnection method highlighted.

$$\alpha^{l}_{WCC,T_2}(\Delta) = \delta^{l}_{T_1 \to T_2}(\alpha^{l}_{T_1}{}'(\Delta)) \quad (4.20)$$

$$\alpha^{u}_{WCC,T_2}(\Delta) = \delta^{u}_{T_1 \to T_2}(\alpha^{u}_{T_1}{}'(\Delta)) \quad (4.21)$$

In parallel, as we see in Figure 48, we can still obtain the ingoing resource-based arrival curves α_{WLT,T_2} from a workload transformation of the ingoing event-based arrival curves, by applying (4.8) to $\overline{\alpha}_{T_2}$. And since all ingoing resource-based arrival curves, α_{WLT,T_2} as well as α_{WCC,T_2}, are hard bounds, we can combine the curves by taking the pairwise maximum and minimum of the upper and lower arrival curves, respectively:

$$\alpha^l_{T_2}(\Delta) = \max(\alpha^l_{WLT,T_2}(\Delta), \alpha^l_{WCC,T_2}(\Delta)) \quad (4.22)$$
$$\alpha^u_{T_2}(\Delta) = \min(\alpha^u_{WLT,T_2}(\Delta), \alpha^u_{WCC,T_2}(\Delta)) \quad (4.23)$$

4.4.4 Computing Workload Correlation Curves

To compute the workload correlation curves for two consecutive abstract components with workload variability automata F_{γ,T_1} and F_{γ,T_2}, respectively, that are triggered by an event stream with an event sequence automaton F_σ, we need to build a so-called workload correlation automaton F_δ. But to reduce the size of the different intermediate automata that are needed to compute the workload correlation curves, we first divide all resource demands on the transitions of the two workload variability automata, F_{γ,T_1} and F_{γ,T_2}, by their greatest common divisor (gcd). The workload correlation automaton is then built from the product automaton of these two workload variability automata, where transition between two states exists if and only if two transitions existed in the initial workload variability automata, where the output event type on the transition of first automaton F_{γ,T_1} equals the input event type on the transition of the second automaton F_{γ,T_2}:

$$\begin{aligned}
S_\gamma &= S_{T_1} \times S_{T_2} \\
S^0_\gamma &= S^0_{T_1} \times S^0_{T_2} \\
T_\gamma &= \{((u,v),[d_{l,T_1},d_{u,T_1}],[d_{l,T_2},d_{u,T_2}],(u',v'))| \\
&\quad (u,\sigma_{I,T_1},[d_{l,T_1},d_{u,T_1}],\sigma_{O,T_1},u') \in T_{T_1} \\
&\quad \wedge (v,\sigma_{I,T_2},[d_{l,T_2},d_{u,T_2}],\sigma_{O,T_2},v') \in T_{T_2} \\
&\quad \wedge \sigma_{O,T_1} = \sigma_{I,T_2}\}
\end{aligned}$$

Figure 49 depicts the two workload variability automata, F_{γ,T_1} and F_{γ,T_2}, of the two tasks T_1 and T_2 of the MPSoC of Figure 44, after dividing the resource demands by their gcd, as well as the workload correlation automaton F_δ, obtained from F_{γ,T_1} and F_{γ,T_2}.

From the workload correlation automaton F_δ, we can then compute the upper workload correlation curve $\delta^u_{T_1 \to T_2}$ following a four-step procedure:

Step 1: On the transitions of the workload correlation automaton F_δ, we retain the lower resource demand d_{l,T_1} from the first workload variability automaton F_{γ,T_1}, and the upper resource demand d_{u,T_2} of the second workload variability automaton F_{γ,T_2}, and discard the other information:

$$s \xrightarrow{[d_{l,T_1},d_{u,T_1}]/[d_{l,T_2},d_{u,T_2}]} s' \Rightarrow s \xrightarrow{d_{l,T_1}/d_{u,T_2}} s'$$

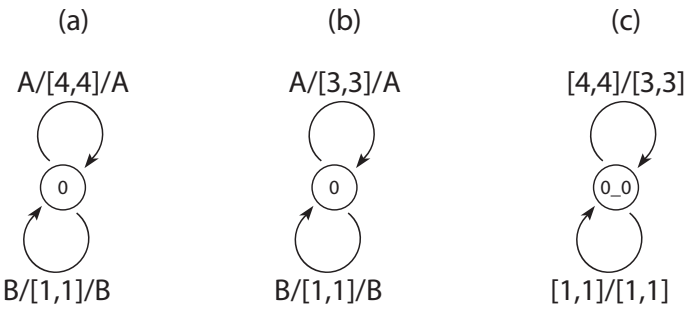


Fig. 49: (a) The workload variability automaton F_{γ,T_1} of the task T_1 of the MPSoC of Figure 44, and (b) the workload variability automaton F_{γ,T_2} of the task T_2 of the same MPSoC. Compared to the workload variability automata depicted in Figures 45(b) and 45(c), the resource demands on both automata are divided by their greatest common divisor $\gcd(20E3, 15E3, 5E3) = 5E3$. (c) The workload correlation automaton F_δ, obtained from F_{γ,T_1} and F_{γ,T_2}.

Step 2: We then transform the so obtained automaton into an automaton where every transition corresponds to a created demand of one resource unit on the first task T_1 and where the weights on the transition correspond to the created demand on the second task T_2. For this, we first replace every transition $s \xrightarrow{d_{l,T_1}/d_{u,T_2}} s'$ with $d_{l,T_1} > 0$, with a state trajectory according the following rule:

$$s \xrightarrow{d_{l,T_1}/d_{u,T_2}} s' \Rightarrow s \underbrace{\xrightarrow{0} s_{(1)} \xrightarrow{0} \ldots \xrightarrow{0} s_{(d_{l,T_1}-1)}}_{d_{l,T_1} \text{ transitions}} \xrightarrow{d_{u,T_2}} s'$$

After this, we replace all transitions $s' \xrightarrow{d_{l,T_1}/d_{u,T_2}} s''$ with $d_{l,T_1} = 0$ as follows:

$$s \xrightarrow{w} s' \xrightarrow{0/d_{u,T_2}} s'' \Rightarrow s \xrightarrow{w+d_{u,T_2}} s''$$

Step 3: In this automaton, we interpret the execution demand d as the weight of a transition. The weight $w^u(e)$ of the maximum-weight path with length e then equals the value of the upper workload correlation curve $\delta^u_{T_1 \to T_2}(e)$.

Step 4: We finally define the workload correlation curve $\delta^u_{T_1 \to T_2}(x)$ for all values $x \in \mathbb{R}_{\geq 0}$ as:

$$\delta^u_{T_1 \to T_2}(x) \quad = \quad \min_{e \geq x,\, e \in \mathbb{N}^+} \left\{ \delta^u_{T_1 \to T_2}(e) \right\} \tag{4.24}$$

, and we multiply all values on both, the x- and the y-axis by the gcd of the initial workload demands, by which we initially divided the demands of the workload variability automata.

To compute the lower workload correlation curve $\delta^l_{T_1 \to T_2}(e)$, we must follow the same four-step procedure as described above, but in *Step 1* we need to retain the upper resource demand d_{u,T_1} of the first workload variability automaton F_{γ,T_1}, and the lower resource demand d_{l,T_2} of the second workload variability automaton F_{γ,T_2}. In *Step 3*, the value of the lower transfer curve $\delta^l_{T_1 \to T_2}(e)$ then equals the weight $w^l(e)$ of the minimum-weight path with length e in the automaton obtained in *Step 2*, and in *Step 4*, the lower workload correlation curve $\delta^l_{T_1 \to T_2}(x)$ is defined for all values $x \in \mathbb{R}_{\geq 0}$ as:

$$\delta^l_{T_1 \to T_2}(x) \quad = \quad \max_{e \leq x,\, e \in \mathbb{N}^+} \left\{ \delta^l_{T_1 \to T_2}(e) \right\} \tag{4.25}$$

Figure 50 depicts the above-defined four-step procedure to compute the workload correlation curves between the two tasks T_1 and T_2 of the MPSoC in Figure 44. From the resulting curves, we see that with the availability of a single cycle (the $20,000^{th}$), task T_1 can generate an event of type A on its output event stream, leading to an execution demand of $15E3$ cycles in task T_2. But after this initial event of type A, the constant processing of events of type B (assuming that the input buffer of task T_1 never underflows) leads to the maximum workload on task T_2. This is because with events of type B, every cycle invested in T_1 generates in average a demand of 1 cycle in T_2, while with events of type A, every invested cycle in T_1 only generates an average demand of $15E3/20E3 = 0.75$ cycles in T_2. On the other hand, the constant processing of events of type A leads to the minimum workload on task T_2.

4.4.5 Experimental Results

In this section, we analyze the MpSoC presented in Section 4.4.1. In particular, we are interested in the upper bounds to the maximum delay that events of stream R_2 experience during processing on the MPSoC, as a function of the processor frequency $f_{P_{II}}$ of processor P_{II}. The processor frequency f_{P_I} of processor P_I is thereby assumed to be fixed to $6MHz$. Note, that in the system analysis presented in this section, we neglect the delay that events experience by the I/O-interfaces and by the on-chip communication network.

For the system performance analysis, we use the performance model of the MPSoC, depicted in Figure 48. To model the processing capacity of the initially unloaded processors P_I and P_{II}, we use $\beta_{T_1} \equiv \beta_{P_I} = f_{P_I} \cdot \Delta$, and $\beta_{T_2} \equiv \beta_{P_{II}} = f_{P_{II}} \cdot \Delta$, respectively. Further, the event-based arrival curves

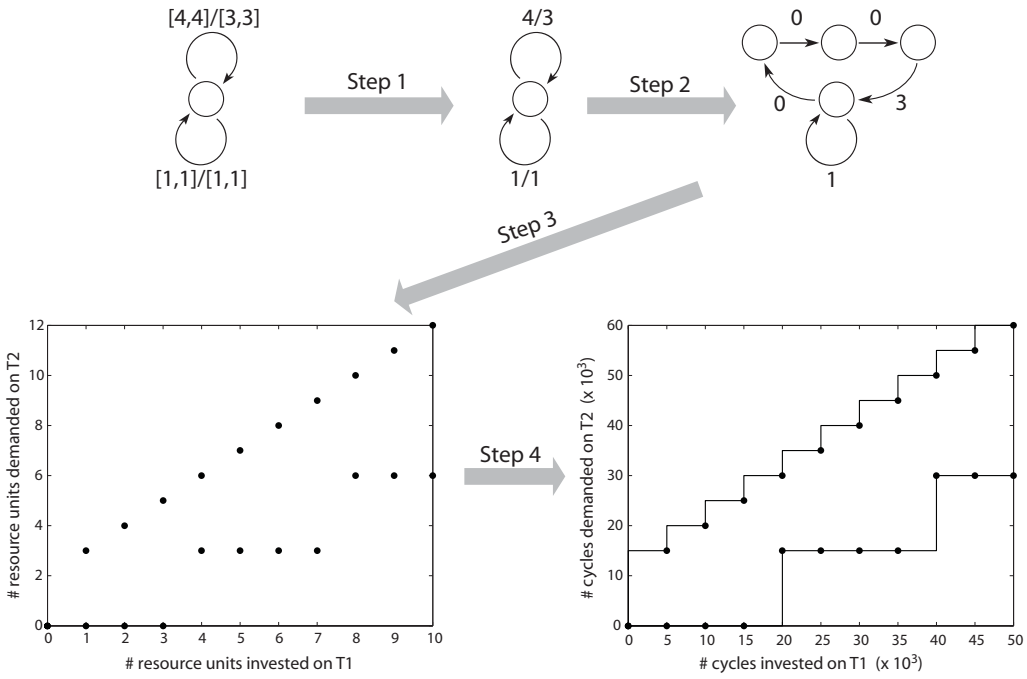


Fig. 50: The four-step procedure to compute the workload correlation curves between the two tasks T_1 and T_2 of the MPSoC in Figure 44, starting from the workload correlation automaton in Figure 49(c).

$\overline{\alpha}_{T_1} \equiv \overline{\alpha}_{R_1}$, and $\overline{\alpha}_{T_3} \equiv \overline{\alpha}_{R_2}$, are computed from p_{R_1}, j_{R_1}, and d_{R_1}, and from p_{R_2} and j_{R_2}, respectively, using (2.2) and (2.3).

4.4.5.1 Analytic Bound on the Maximum Delay

According to (2.11), we need $\alpha^u_{T_3}$ and $\beta^l_{T_3}$ to compute an upper bound to the maximum delay that an event experiences during processing by task T_3 on processor P_{II} of the MPSoC.

We can directly compute $\alpha^u_{T_3}$ by applying (4.8) to $\overline{\alpha}^u_{T_3}$. Obtaining $\beta^l_{T_3}$, on the other hand, is more involved. We need in a first step to compute the outgoing arrival curves $\alpha^{u'}_{T_1}$ and $\overline{\alpha}^{u'}_{T_1}$ of task T_1. We compute these curves according to the analysis schema shown in Figure 37, using (2.7)–(2.10) and (4.8)–(4.11). The resulting outgoing curves are shown in Figures 46 and 47, and are discussed in Section 4.4.2.

According to the performance model shown in Figure 48, the event-based upper outgoing arrival curve of task T_1 can directly be interconnected to the event-based upper ingoing arrival curve of task T_2: $\overline{\alpha}^u_{T_2} = \overline{\alpha}^{u'}_{T_1}$. To compute the resource-based upper ingoing arrival curve of task T_2, on the other hand, we first conventionally apply (4.8) to $\overline{\alpha}^u_{T_2}$, leading to α^u_{WLT,T_2}.

Applying (4.21) to $\alpha^{u'}_{T_1}$ will in turn lead to α^{u}_{WCC,T_2}, and computing the minimum of α^{u}_{WLT,T_2} and α^{u}_{WCC,T_2} will, according to (4.23), finally lead to $\alpha^{u}_{T_2}$. Similarly, the resource-based lower ingoing arrival curve of task T_2 can according to (4.22) be computed as the maximum of α^{l}_{WLT,T_2} and α^{l}_{WCC,T_2}, where α^{l}_{WLT,T_2} and α^{l}_{WCC,T_2} are obtained by applying (4.8) to $\overline{\alpha}^{l}_{T_2}$ and (4.20) to $\alpha^{l'}_{T_1}$, respectively. All these variants of resource-based ingoing arrival curves of task T_2 are depicted in Figure 51.

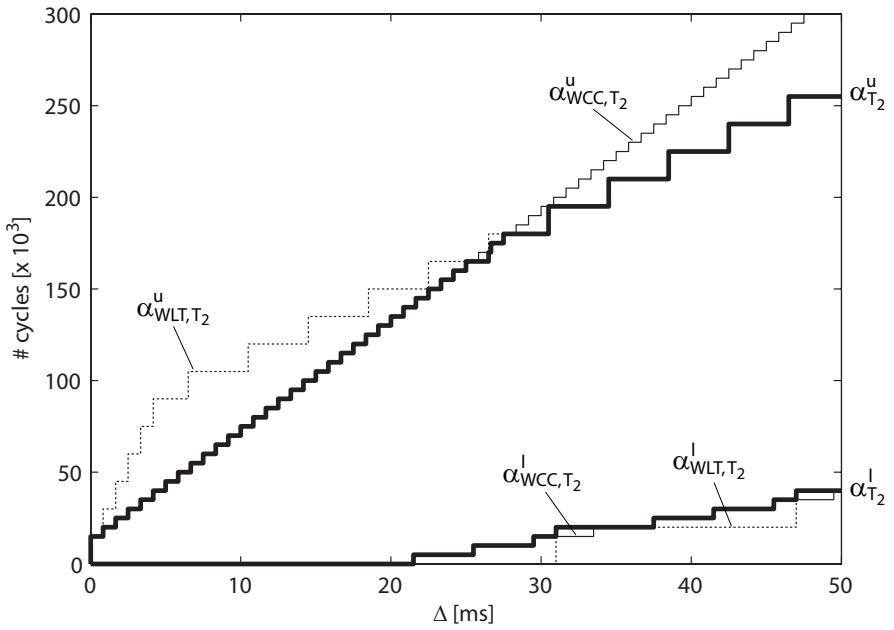


Fig. 51: Variants of resource-based input arrival curves to the abstract component of task T_2 in the MPSoC in Figure 44. α^{u}_{WLT,T_2} and α^{l}_{WLT,T_2} are computed by applying a workload transformation to $\overline{\alpha}^{u'}_{T_1}$ and $\overline{\alpha}^{l'}_{T_1}$. α^{u}_{WCC,T_2} and α^{l}_{WCC,T_2} are computed by applying a workload correlation transformation to $\alpha^{u'}_{T_1}$ and $\alpha^{l'}_{T_1}$. $\alpha^{u}_{T_2}$ and $\alpha^{l}_{T_2}$ are computed as the minimum and the maximum of the other curves, respectively.

Then, to compute the resource-based lower outgoing service curve of task T_2, we apply (2.10) to $\alpha^{u}_{T_2}$ and $\beta^{l}_{T_2}$, leading to $\beta^{l'}_{T_2}$, that can be interconnected to $\beta^{l}_{T_3}$. This finally allows to use (2.11) to compute an upper bound to the maximum delay that an event experiences during processing by task T_3 on processor P_{II}:

$$d_{T_3} \leq Del(\alpha^{u}_{T_3}, \beta^{l}_{T_3})$$

To analyze the impact of incorporating the workload correlation information to the obtained analysis results, we do the same performance

analysis using conventional analysis methods. There, we apply (2.10) directly to α^u_{WLT,T_2} and $\beta^l_{T_2}$, leading to $\beta^{l'}_{WLT,T_2} \equiv \beta^l_{WLT,T_3}$. We then use again (2.11) to compute an upper bound to the maximum delay that an event experiences during processing by task T_3 on processor P_{II}:

$$d_{WLT,T_3} \leq Del(\alpha^u_{T_3}, \beta^l_{WLT,T_3})$$

In Figure 52, the resource-based upper ingoing arrival curve $\alpha^u_{T_3}$ is shown, together with the resource-based lower ingoing arrival curve β^l_{WLT,T_3} that is obtained using conventional analysis methods, and the resource-based lower ingoing arrival curve $\beta^l_{T_3}$ that is obtained using workload correlation curves to incorporate existing workload correlations into the performance analysis.

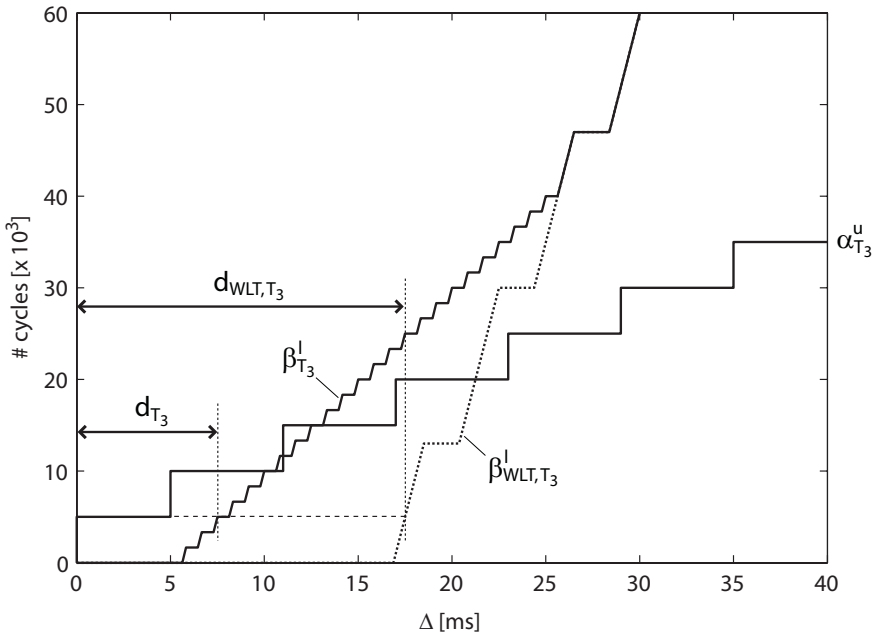


Fig. 52: The resource-based upper ingoing arrival curve $\alpha^u_{T_3}$ to the abstract component of task T_3, together with two variants of the resource-based lower ingoing service curve. $\beta^l_{T_3}$ results from a computation with WCC's, while $\beta^l_{T_3}$ results from a computation without WCC's. The curves are computed for $f_{P_{II}} = 8MHz$.

4.4.5.2 Performance Analysis Results

We use the above described analysis to compute the upper bound to the maximum delay that an event will experience during processing by task T_3 on processor P_{II}, for processor frequencies $f_{P_{II}}$ ranging from $6MHz$ to

25MHz, in steps of 1MHz. As a reference, we compute the exact worst-case delays, using a timed automata based approach [HV06] and UPPAAL v3.5.9 [Upp], and we also declare the results that are obtained using SymTA/S v0.8 beta EVAL [Sym] that implements a different approach to system-level performance analysis.

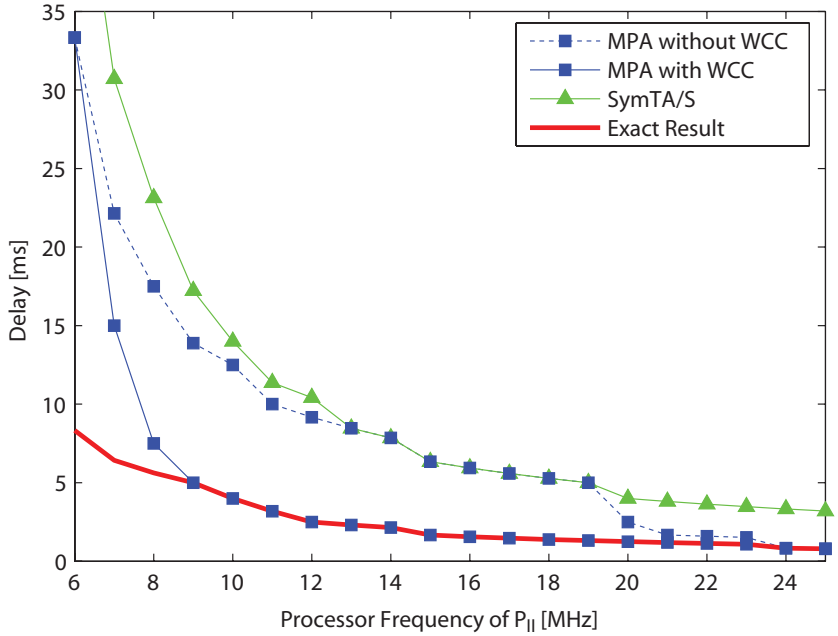


Fig. 53: The maximum delay experienced by an event during processing by task T_3 in the MPSoc in Figure 44 as a function of the frequency $f_{P_{II}}$ of processor P_{II}.

From the results shown in Figure 53, we learn that the use of workload correlation curves leads to considerably tighter analytic bounds for the investigated performance analysis problem. The obtained results clearly point out the performance reserves that often exist in complex systems due to workload correlations, and that are usually not considered by existing performance analysis methods. For example, while we need to run processor P_{II} at a processor speed of 14MHz to guarantee an upper bound to the maximum delay of 8ms within the conventional MPA framework, we can guarantee the same bound already at a processor speed of 8MHz when using WCC's. This corresponds to a processor speed reduction of more than 42%. Similarly, when we look at the delay guarantee for a processor speed of 8MHz, we see that we can guarantee a maximum delay of 17.05ms within the conventional MPA framework, while we can guarantee a maximum delay of only 7.5ms when using WCC's. This corresponds to a delay guarantee improvement of 56%.

Looking at the results in Figure 53, we may wonder why the analysis with WCC's does not lead to improved results for low processor speeds (e. g. $6MHz$). When we look at Figure 51, we see that without the use of WCC's, α^u_{WLT,T_2} limits the input to task T_2, consisting of a steep burst at the beginning and a less steep long-term load afterwards. With WCC's however, α^u_{WCC,T_2} reduces the burst load on task T_2 within short time intervals. In the long term however ($\Delta \geq 30ms$) the load on task T_2 is still limited by the long-term slope of α^u_{WLT,T_2}. When we reduce the speed of processor P_{II}, it eventually gets so slow that the available cycles per time interval are less than the arriving resource demand of the reduced burst that is specified by α^u_{WCC,T_2}, that is the processor gets too slow to even process the reduced burst without buffering. In this case, remaining resources to process task T_3 only exist in time intervals that are longer than the interval length where α^u_{WCC,T_2} and α^u_{WLT,T_2} cross for the last time in Figure 51, i. e. for $\Delta \geq 30ms$. Then however the remaining resources only depend on the long term slope of α^u_{WLT,T_2}, and hence WCC's do not have any influence on the remaining resources anymore.

4.5 Solving the Maximum-Weight Path Problem

Algorithm 2 presents an efficient method to compute the weights $w^u(e)$ and $w^l(e)$ of the maximum- and minimum-weight paths of a directed weighted graph $G(V, E, W)$ with vertices V, edges E and weights W. The computational complexity of Algorithm 2 to compute the values of $w^u(e)$ and $w^l(e)$ for all path lengths $e \in [1, n]$ is $O(n|V||E|)$, while the memory requirement has a complexity of $O(|V|)$.

The values $w^u(e)$ represent the maximum-weight paths of G for $e \leq n$. We define $w^u_*(e)$ as the periodic continuation of $w^u(e)$:

$$w^u_*(e) = \left\lfloor \frac{e}{n} \right\rfloor \cdot w^u(n) + w^u(e - \left\lfloor \frac{e}{n} \right\rfloor n) \tag{4.26}$$

Since for the weights of maximum weight paths in a weighted graph the relation $w^u(e_1 + e_2) \leq w^u(e_1) + w^u(e_2)$ holds true for $\forall e_1, e_2 \in \mathbb{Z}_{\geq 0}$ *(sub-additivity)*, it can be shown that the periodic continuation $w^u_*(e)$ of $w^u(e)$ provides a hard upper bound on the weight of the maximum-weight path in G of any length $e \in \mathbb{Z}_{\geq 0}$. Similarly, it can be shown that the periodic continuation of $w^l(e)$ is a hard lower bound on the weight of the minimum-weight path in G.

Using the presented method, tight (i. e. exact) values for weights of the maximum and minimum weight paths in a graph G can be computed up to an arbitrary path length n, while hard upper and lower bounds can be obtained for any longer paths. For strongly connected graphs and when

Algorithm 2 Computing the weights of the maximum-weight and minimum-weight paths in a weighted graph G(V,E,W)

Given a function $pred(v)$, which returns the set of all predecessor vertices of vertex v.
Given a function $weight^u(u,v)$ and a function $weight^l(u,v)$, which return the maximum weight and the minimum weight respectively, of all edges starting at vertex u and ending at vertex v.

$w_v^u(0) = 0, \forall v \in V$
$w_v^l(0) = 0, \forall v \in V$
for $i = 1$ to n **do**
  for $\forall v \in V$ **do**
    if $|pred(v)| > 0$ **then**
      $w_v^u(i) = \max_{p \in pred(v)}\{w_p^u(i-1) + weight^u(p,v)\}$
      $w_v^l(i) = \min_{p \in pred(v)}\{w_p^l(i-1) + weight^l(p,v)\}$
    else
      $w_v^u(i) = -\infty$
      $w_v^l(i) = +\infty$
    end if
  end for
  $w^u(i) = \max_{v \in V}\{w_v^u(i)\}$
  $w^l(i) = \min_{v \in V}\{w_v^l(i)\}$
end for

spending some more effort, it is also possible to obtain the tight values for $w^l(e)$ and $w^l(e)$ for all $e \in \mathbb{Z}_{\geq 0}$. The according techniques are described in [CDQV85] and [Kar78].

4.6 Related Work

Many results in the area of classical real-time scheduling theory are based on the task model introduced by Liu and Layland in [LL73]. In this model, tasks are characterized by tuples (C_i, T_i), where C_i is the execution time of a task τ_i, and T_i is the period with which τ_i is activated on the system. To provide hard real-time guarantees, methods used in the area of classical real-time scheduling theory typically set C_i to the worst-case execution time (WCET) of the task, thus assuming that every task instance requires WCET to complete. While this assumption is safe, it is too pessimistic for a large class of applications, that are characterized by high execution time variability. For these applications, the classical results lead to poor processor utilization, and consequently to system designs

with unreasonably high cost, or power consumption, or both. While this problem is typically not prevalent in simulation-based methods, several approaches have been proposed in literature to address it for stochastic, as well as for analysis-based methods. For an overview, see for example [SAÅ+04].

In the area of stochastic performance estimation, several proposals have been made of periodic task models that specify execution demands using probabilistic distributions, see for example [AB98, KM98, MEP04, TDS+95]. Another method was presented by Lehoczky in [Leh96], and is based on queueing theoretic methods for performance analysis. This real-time queueing theory uses stochastic methods not only to model execution demands, but also to model inter-arrival times, as well as deadlines. The performance estimation bounds that are obtained from these and other stochastic methods are however again of probabilistic nature, and hence their application area is limited to soft real-time systems.

For the analysis of hard real-time systems, we must instead rely on formal performance analysis. To address the problem of workload variability, several proposals have been made in this area, to develop more expressive deterministic task models that aim at reducing the pessimism of the classical real-time task models. Mok and Chen [MC97] proposes the multiframe task model that extends the classical periodic task model of Liu and Layland [LL73], by permitting periodic tasks whose WCETs may vary from one instance to another. In the multiframe task model, the WCETs of consecutive task instances are determined following a fixed cyclic pattern. The model was further extended in [BCGM99], allowing not only to determine the WCET of a task instance, but also the time separation between two task instances following a cyclic pattern.

In [Bar98, Bar03], Baruah presents a further generalization of the multiframe models, the recurring real-time task model. Here, a task is modeled by a set of subtasks arranged in a directed acyclic graph that represents the conditional, non-deterministic behavior of the task. Each subtask is characterized by a separate WCET, a relative deadline, and a minimum time separation from its direct predecessor. The whole task graph is triggered sporadically with a specified minimum time separation between the last subtask in the graph and the next task instance. A similar task model that also uses conditional directed acyclic graphs was proposed by Pop et al. in [PEP00]. The recurring real-time task model provides much flexibility in modeling workload variability and irregular event inter-arrival times. However, accurately modeling complex workload variability and bursty event arrival patterns often requires very large task graphs, and since analysis time increases exponentially with the problem size [Bar03], a designer must trade off the accuracy of the model for tolerable analysis times. To partly overcome this problem, Chakraborty and Thiele [CT05]

recently proposed a new task model that combines the concept of arrival curves with the recurring real-time task model. In this model, the time separation between two subtasks is decoupled from the task graph, and is instead modeled using an arrival curve.

In [JHE04], Jersak et al. introduce the concept of intra-stream contexts, to extend the compositional performance analysis framework presented by Richter et al. in [RE02], [RZJE02] and [RJE03]. Intra-stream contexts allow to specify a cyclic pattern of different events that arrive on an event stream. The timing properties of the event stream are thereby decoupled from the cyclic event pattern, and are specified using a set of classical arrival patterns. On such an event stream with intra-stream context, the WCET of every event, when triggering a computation resource, is then determined from its event type.

In [MKT04], Maxiaguine et al. first introduce workload curves to analyze workload variability within the MPA framework. However, [MKT04] does not provide models and methods to deterministically obtain workload curves from a system with workload variability. In [MKT04], workload curves are obtained using an ad-hoc method that is restricted to the problem under consideration, and in subsequent works, see e. g. [MLCO04, LMCO04], workload curves are obtained from simulation traces.

4.7 Discussion

Type rate curves, and event and workload variability automata, are two models that allow to capture system information on workload variability that is required to analytically derive the workload curves of an abstract component. Type rate curves are a very general model to capture information on event stream correlations and they can be derived from virtually any formal event stream specification. And if no formal specification is available, type rate curves of an event stream can also be derived from simulation traces. This is of interest, since type rate curves allow to strictly decouple the information on event correlations from the information on imposed workload on a specific component. Thus, the workload curves for any number of different components can be derived from the same set of type rate curves that are obtained from a single simulation trace of the triggering event stream.

The automata models on the other hand are more restrictive, and are only applicable if detailed specifications are available on the event stream, as well as on the component under consideration. The level of detail of these specifications can however be chosen flexibly, such that a designer can choose to only model the most performance critical components with

detailed workload variability automata. When we compare the automata models with the multiframe task model and with the recurring real-time task model, we observe primarily two differences. Firstly, the two task models, and the automata models rely on different task graphs. The multiframe task model relies on fixed cyclic patterns, while the recurring real-time task model relies on directed acyclic graphs to model possible task sequences. The automata models on the other hand allow to capture possible sequences in directed graphs, which often allows a more compact representation. Moreover, the differentiation between event automata and workload variability automata further simplifies modeling, since it allows a designer to separate the information on event stream correlations from the information on functional workload variability within a component. The second main difference is that the multiframe and the recurring real-time task model explicitly specify the timing between task occurrences, while the timing information of event streams is completely decoupled from the automata models. This decoupling drastically reduces the complexity of the model and allows to analyze systems that would otherwise be prohibitive to analyze. However, the downside of the decoupling is a loss of information that may lead to less accurate analysis results. The strictly decoupling of timing information from the task sequence structure is also present in the concept of intra-stream contexts. Compared to this concept, the automata models distinguish themselves by a considerably higher expressiveness. Intra stream contexts allow only to model the information that could be captured by a single event automaton with a strictly cyclic structure. More complex event stream sequences, as well as workload variability information of the component can however not be captured with intra-stream contexts.

Workload correlation curves are to our best knowledge the first approach to characterize the workload correlations that are often present in distributed embedded systems. It has to be noted, that workload correlation curves do not require that event automata and workload variability automata are used throughout a complete system. Instead, from the experimental results we have seen that workload correlation curves may also lead to considerable analysis improvements when the input event automaton is fully non-deterministic. Such a non-deterministic event automaton can be created at any point within the system, and designer has therefore the freedom to apply workload correlation curves isolated on two succeeding components whenever these have considerably correlated workloads.

Part II

Interface-Based Design

5

Interface-Based Design with Real-Time Interfaces

In the first part of this thesis, we focused on system-level performance analysis methods for real-time embedded systems. Common to all of these performance analysis methods is that they are applied to analyze a component-based real-time system design a posteriori. That is, while a real-time system gets designed and dimensioned in a first step, it is only after completion of this first step that performance analysis is applied on the system design in a second step. The analysis result will then give an answer to the binary question whether the system design that was developed in the first step will meet all real-time requirements, or not. A designer must then iterate on these two steps until an appropriate system design is found.

In contrast to this two-step approach is the idea of interface-based design [dAH01b, dAH05] that proposes a holistic one-step approach to design and analysis of systems, sometimes also referred to as correct-by-construction. In interface-based design, components are described by component interfaces, and in contrast to an abstract component that models what a component does, a component interface models how a component can be used. Through *input assumptions*, a component interface models the expectations that a component has about the other components in a system, and through *output guarantees*, a component interface tells other components in a system what they can expect from this component. The major goal of a good component interface is then to provide enough information to decide whether two or more components can work together properly, where in the case of component interfaces

for real-time system performance analysis, the term 'properly' refers to questions like: Does the composed system satisfy all requested real-time properties such as delay and throughput constraints?

Consequently, introducing a component system with interfaces for interface-based design of real-time systems advances the real-time system design process towfold. First, a system designer then only needs to understand a component's interface and not the details of how the functionality offered by the component is implemented. And secondly, such a component system with interfaces enables correct-by-construction design, since a set of components can only be integrated to a system, if and only if their interfaces are compatible.

This chapter introduces Real-Time Interfaces (RTI). By connecting the principles of Real-Time Calculus and interface-based design, Real-Time Interfaces enable interface-based real-time system design within the MPA framework. The next section first provides an introduction to interface-based design, with a special focus on assume/guarantee interfaces. In Section 5.2, we then define Real-Time Interfaces as an extension of assume/guarantee interfaces, and in Section 5.3, we introduce a component system with Real-Time Interfaces, for interface-based design within the MPA framework. In Section 5.4, we then provide an overview to the wide area of applications for Real-Time Interfaces. The chapter concludes with an overview on related work in Section 5.5, and a discussion in Section 5.6.

5.1 Interface-Based Design and A/G Interfaces

The definition of Real-Time Interfaces follows the principles of interface-based design, as described by de Alfaro and Henzinger in [dAH01b] and [dAH05]. Whereas most previous results relate to stateful interface languages, such as Interface Automata [dAH01a], Real-Time Interfaces are based on stateless assume/guarantee (A/G) interfaces. Following, we introduce the underlying principles of A/G interfaces and interface-based design. Note, that interfaces and all interface-related terms in this section are formally defined in [dAH01b] and [dAH05].

An A/G interface consists of two disjoint sets of input and output variables denoted by X^I, and X^O, respectively. The interface makes certain *assumptions* on X^I, which are specified using a predicate ϕ^I, and provided this predicate is satisfied, the interface *guarantees* that the output variables satisfy a predicate ϕ^O. Hence, ϕ^O is the guarantee that the component provides to its environment assuming the precondition ϕ^I, or in other words, $\phi^I \Rightarrow \phi^O$ is true. Clearly, the predicate ϕ^I needs not to be valid, since there typically exist environments where a component can not be used or can not provide the output guarantee. In what follows, we do not

distinguish between components and their interfaces. Hence, an interface implicitly refers to the component it belongs to.

Two interfaces F and G are composed by connecting the output variables of one interface to the input variables of the other interface, and the composed interface is typically denoted as $F||G$. The input variables of this composed interface are all the unconnected (or free) input variables of F and G, and its output variables are the unconnected outputs of F and G. We will use X_F^I, X_F^O, ϕ_F^I and ϕ_F^O to denote the variables and predicates of F, and X_G^I, X_G^O, ϕ_G^I and ϕ_G^O to denote the variables and predicates of G.

Two interfaces are syntactically composable if their output variables are disjoint, i. e. if $X_F^O \cap X_G^O = \emptyset$, and provided syntactical composability, they are semantically compatible, if whenever F provides inputs to G, then the output guarantee of F implies the input assumption of G, i. e. $\phi_F^O \Rightarrow \phi_G^I$. Similarly, whenever G provides inputs to F, then $\phi_G^O \Rightarrow \phi_F^I$ must hold. If F and G form a closed system, i. e. if all outputs of F are connected to the inputs of G, and vice-versa, then F and G are compatible, if the following closed formula is true:

$$(\forall X_F^O \cup X_G^O)\ (\phi_F^O \wedge \phi_G^O\ \Rightarrow\ \phi_F^I \wedge \phi_G^I) \tag{5.1}$$

If $F||G$ form an open system on the other hand, i. e. if some inputs of F and G are left free, then F and G are compatible, if (5.1) is satisfiable. In other words, there must exist *some* environment with which $F||G$ can be compatible. Formula (5.1) is then the input assumption $\phi_{F||G}^I$ of the composite interface $F||G$, as it encodes the weakest condition on the environment of $F||G$ to make F and G work together properly. An interface H that provides inputs to $F||G$ is thus compatible with $F||G$, if $\phi_H^O \Rightarrow \phi_{F||G}^I$ is satisfiable.

This way, a system can be designed by adding one component after the other and by verifying if the newly added component is compatible with the existing partially designed system. Component interfaces can then be composed one-by-one into subsystems in any order, and if any of the subsystems cannot be composed successfully, this already forecloses that the complete system cannot be composed successfully, and can therefore not work properly. We say that interface-based design supports *incremental design* of systems.

But the Real-Time Interfaces that we introduce in this chapter not only expose enough information to decide on composability and compatibility with other component interfaces, but in addition they also change their assumptions and guarantees, following principles of constraint propagation. Consequently, one can determine the combined assumptions of a complete subsystem at every single input of the subsystem, thereby enabling interactive design optimization. To enable this *dynamic adaptability*, an interface must define internal interface relations, that bring the

different input assumptions and output guarantees of all interconnected interfaces into relation.

Ex. 9: *To illustrate the above concepts on a simple example, imagine a component as depicted in Figure 54(a). The component has two input variables a and b, and one output variable c, and a component description, expressed by $c = b - a$. To put this component in a context, lets suppose it is the abstraction of a concrete component, and it is used to analyze real-time properties of the concrete component, very similar to the abstract components introduced in Chapter 2. We could then interpret this component as follows: input variable a describes the resource demand of an arriving event stream, while b describes the resources available to process the arriving event stream. Output variable c would then describe the resources that remain unused after processing the resource demand a.*

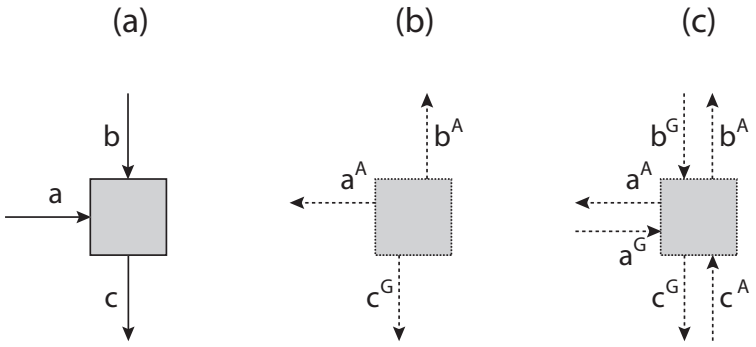


Fig. 54: (a) A simple component, (b) its interface with assumptions on the inputs and guarantees on the output, and (c) its interface including the input guarantees and output assumptions of connected interfaces.

Figure 54(b) depicts an A/G interface for the component in Figure 54(a). Although not depicted explicitly, this interface also has the two input variables a and b and the output variable c. Through its predicates, the interface puts constraints on the environment by assuming that $a \leq a^A$ and $b \geq b^A$. In return, it guarantees that $c \geq c^G$. In other words, these predicates express that the interface assumes that at least b^A resources are available to process an arriving resource demand of at most a^A, and it guarantees that at least c^G resources will remain after processing the arriving event stream.

When this interface gets connected to other interfaces, these will provide output guarantees on their output variables, e. g. $a \leq a^G$ and $b \geq b^G$, and input assumptions on their input variables, e. g. $c \geq c^A$, as depicted in Figure 54(c). Considering the definition of compatibility, we then see that our component is compatible with a component that provides an input to a, if the output guarantee $a \leq a^G$ of the connected interface complies with the input assumption $a \leq a^A$ of our interface, i. e. if $a^G \leq a^A$. Analogously our interface can be composed with an

interface that provides an input to b if $b^G \geq b^A$, and the output c of our interface can serve as input to another interface if $c^A \leq c^G$. Or in other words, the arrival input of our interface can be connected to the arrival output of an interface that guarantees less resource demand arrival than our interface assumes, the service input of our interface can be connected to the service output of an interface that guarantees to provide more resources than assumed by our interface, and the service output of our interface can be connected to the service input of an interface that assumes less resources than guaranteed by our interface.

To support dynamic adaptability, we must then find the correct set of interface relations, that bring the different input assumptions and output guarantees of all interconnected interfaces into relation. For our interface, these interface relations can be established as $c^G = b^G - a^G$, $a^A = b^G - c^A$, and $b^A = a^G + c^A$.

5.2 Real-Time Interfaces

Real-time interfaces can be considered as a special instance of A/G interfaces, tailored towards assumptions and guarantees on the throughput and delay of events, and the availability of resources. Based on Example 9 in the previous section, we identify three steps that eventually yield to a component system for interface-based design of real-time systems:

1. First, we need to define an abstract component that describes the real-time properties of a concrete HW/SW system component. This entails defining proper abstractions for component inputs and outputs as well internal component relations that meaningfully relate abstract inputs to abstract outputs.

2. To derive the interface of an abstract component we then need to define interface variables as well as input and output predicates on these interface variables.

3. Finally, to enable dynamic adaptability, we need to establish the internal interface relations that relate incoming guarantees and assumptions to outgoing guarantees and assumptions of the interface.

The following Section 5.2.1 is dedicated to Step 1, while Section 5.2.2 is dedicated to Step 2. Establishing the internal interface relations for Step 3 is then the central part of Section 5.3.

5.2.1 Abstract Real-Time Components

In general, any abstract component of the MPA framework could serve as basis to establish corresponding Real-Time Interfaces. However, in

this chapter we will focus on a slightly reduced version of an abstract greedy processing component as introduced in Section 2.4. Compared to the abstract greedy processing component of Section 2.4, we will not consider an arrival curve output. This leads to a component as depicted in Figure 55(a). Moreover, we will model an event stream only with an upper arrival curve, denoted as α, and an associated maximum allowable delay d, and we will model a resource availability only with a lower service curve, denoted as β.

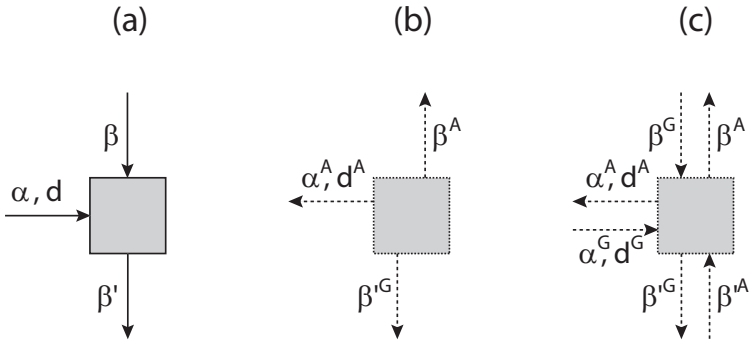


Fig. 55: (a) An abstract component, (b) its interface with assumptions on the inputs and guarantees on the output, and (c) its interface including the input guarantees and output assumptions of connected interfaces.

Following (2.10), we know that if an event stream with (upper) arrival curve α is processed by such an abstract component on a resource with (lower) availability β, then the remaining resources that are not consumed by the abstract component can be bounded by the (lower) service curve

$$\beta' = (\beta - \alpha) \overline{\otimes} 0 \stackrel{def}{=} RT(\beta, \alpha) \tag{5.2}$$

And following (2.11), we know that the maximum delay experienced by an event at such an abstract component is bounded by

$$d_{max} \leq Del(\alpha, \beta) \tag{5.3}$$

5.2.2 Interface Variables and Predicates

A real-time interface of an abstract component as introduced above has input and output variables related to event streams (arrival variables α and delay variables d), and resource availabilities (service variables β).

An arrival variable consists of an arrival curve α, and a delay variable consists of a maximum allowable event delay d. The output guarantee on an arrival variable contains the bound α^G, and the output predicate ϕ^O

guarantees $\alpha(\Delta) \leq \alpha^G(\Delta)$. The output guarantee on a delay variable on the other hand contains the bound d^G, and the output predicate ϕ^O guarantees $d \geq d^G$. Thus, the interface of an event stream, with the guarantees α^G and d^G, expresses that the event stream has an arrival curve less or equal α^G, and a maximum allowable delay larger or equal than d^G. The input assumptions on an event stream then contain the bounds α^A and d^A, and the input predicate ϕ^I reflects the assumptions $\alpha(\Delta) \leq \alpha^A(\Delta)$ and $d \geq d^A$. The value of a service variable consists of a service curve β, and the output guarantee on a service variable contains the bound β^G, and the output predicate ϕ^O guarantees $\beta(\Delta) \geq \beta^G(\Delta)$. The input assumption contains the bound β^A, and the input predicate ϕ^I reflects the assumption $\beta(\Delta) \geq \beta^A(\Delta)$. Figure 55(b) depicts the interface of the abstract component of Figure 55(a).

In order to determine whether two abstract components are compatible, we can check whether their interfaces are compatible. To this end, we need to check the relation $\phi^O \Rightarrow \phi^I$ for all connections, given the quantities depicted in Figure 55(c). Event stream connections are thereby compatible, if

$$(d^A \leq d^G) \quad \wedge \quad (\alpha^A(\Delta) \geq \alpha^G(\Delta)) \qquad \forall\, \Delta \geq 0 \tag{5.4}$$

while a service connection is compatible, if

$$\beta^A(\Delta) \leq \beta^G(\Delta) \qquad \forall\, \Delta \geq 0 \tag{5.5}$$

We can then generalize that two interfaces are compatible if (5.4) and (5.5) are true or satisfiable for all internal event stream and service connections, respectively, and if the input predicates of all open input variables are still satisfiable. The relations (5.4) and (5.5) are depicted in Figure 56(a) and 56(b), respectively.

5.3 A Component System with Interfaces

In Chapter 2, we introduced three types of models, models of the environment, models of resources, and models of tasks and components, that serve together as basic building blocks of a performance model for performance analysis within the MPA framework. To establish a component system with Real-Time Interfaces for interface-based design within the MPA framework, we accordingly distinguish between the same three types of models. The environment is modeled by so-called event stream components, resources are modeled by resource components, and application tasks and HW/SW components are modeled by a range of different processing components. Following, we define these components and establish their internal interface relations.

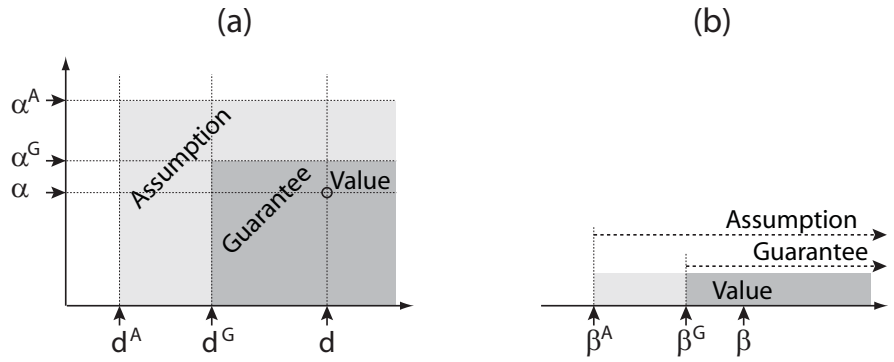


Fig. 56: Relation between interface assumptions, interface guarantees, and variable values. (a) At an event stream connection, and (b) at a service connection.

5.3.1 Event Stream Component

Def. 13: **(Event Stream Component)** *An* event stream component *models an event stream with a real-time delay constraint. The Real-Time Interface of an event stream component has a single arrival output variable and a single delay output variable, with the output guarantee*

$$\phi^O = (\alpha \leq \alpha^G) \wedge (d \geq d^G) \tag{5.6}$$

Through the output guarantee of its Real-Time Interface, an event stream component expresses that the load $\alpha(\Delta)$ that is emitted through its arrival output is always less or equal $\alpha^G(\Delta)$ for any time interval Δ, and that the delay requirement d for this load is larger or equal d^G. Figure 57(a) depicts the interface of an event stream component.

5.3.2 Resource Component

Def. 14: **(Resource Component)** *A* resource component *models a computing or communication resource. The Real-Time Interface of a resource component has a single service output variable with the output guarantee*

$$\phi^O = (\beta \geq \hat{\beta}^G) \tag{5.7}$$

Through the output guarantee of its Real-Time Interface, a resource components expresses that the service $\beta(\Delta)$, that is provided by the component on its service output, is always larger or equal $\beta^G(\Delta)$ for any time interval Δ. Figure 57(b) depicts the interface of a resource component.

5.3.3 Processing Component for FP Scheduling

Def. 15: **(Processing Component for FP Scheduling)** *A* processing component for preemptive fixed priority scheduling *models a task in a real-time system,*

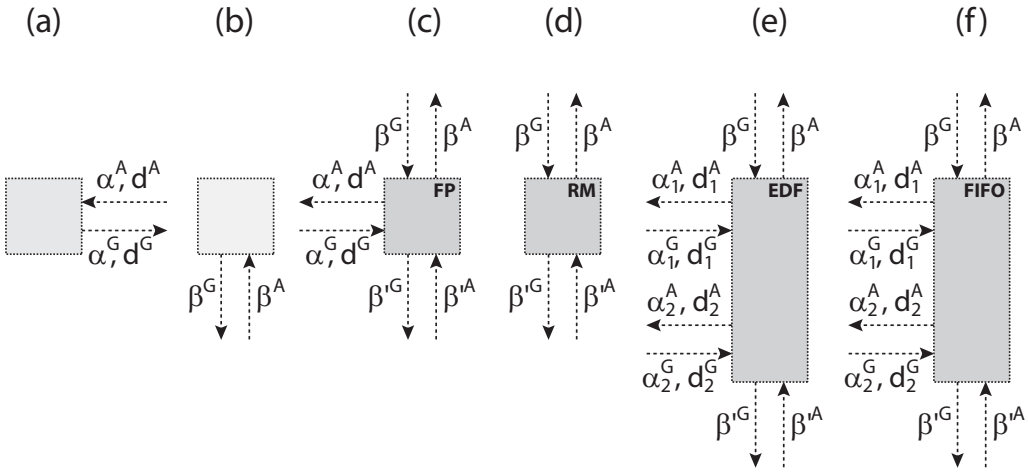


Fig. 57: (a) An event stream component, (b) a resource component, (c) a processing component for FP scheduling, (d) a processing component for RM scheduling, (e) a processing component for EDF scheduling, and (f) a processing component for FIFO scheduling.

that shares system services with a fixed priority scheduling strategy, and that uses the available system services to process real-time event streams. The Real-Time Interface of a processing component for FP scheduling has an arrival input variable, a delay input variable, a service input variable, and a service output variable. The interface has the input assumption

$$\phi^I = (\alpha \leq \alpha^A) \wedge (d \geq d^A) \wedge (\beta \geq \beta^A) \tag{5.8}$$

and the output guarantee

$$\phi^O = (\beta' \geq \beta'^G) \tag{5.9}$$

With the input assumptions and the output guarantees of its Real-Time Interface, a processing component for fixed priority scheduling expresses that whenever (a) the service $\beta(\Delta)$ that is provided to the component on its service input is larger or equal $\beta^A(\Delta)$ for any Δ, and (b) the load $\alpha(\Delta)$ that arrives at the component on its arrival input is less or equal $\alpha^A(\Delta)$ for any Δ and has a maximum required delay that is larger or equal d^A, then (i) the arriving load can be processed in real-time, i. e. with a guaranteed delay $d_{max} \leq d^A \leq d^G$, and (ii) the service $\beta'(\Delta)$ that is provided by the component on its service output is always larger or equal $\beta'^G(\Delta)$ for any time interval Δ, and can be used by lower-priority tasks. Figure 57(c) depicts the interface of a processing component for FP scheduling.

To enable dynamic adaptability of the interface of a processing component, we must next establish the internal interface relations that relate

the incoming guarantees and assumptions to outgoing guarantees and assumptions of the component interface.

The internal interface relations to compute β'^G and d^A can be directly derived from (5.2) and (5.3), respectively. To compute the input service assumption β^A on the other hand, we must consider the delay constraint

$$\beta^A(\Delta) \geq \alpha^G(\Delta - d^G) \tag{5.10}$$

and the resource constraint

$$\beta^A(\Delta) \geq \inf\left\{\beta \;:\; \beta'^A(\Delta) = \sup_{0 \geq \lambda \geq \Delta} \{\beta(\lambda) - \hat{\alpha}^G(\lambda)\}\right\} \tag{5.11}$$

The tightest β^A is then the maximum of both expressions. And to compute the input arrival assumption $\hat{\alpha}^A$, we must consider the delay constraint

$$\alpha^A(\Delta) \leq \beta^G(\Delta + d^G) \tag{5.12}$$

and the resource constraint

$$\alpha^A(\Delta) \leq \sup\left\{\alpha \;:\; \beta'^A(\Delta) = \sup_{0 \geq \lambda \geq \Delta} \{\beta^G(\lambda) - \alpha(\lambda)\}\right\} \tag{5.13}$$

The tightest α^A is then the minimum of both expressions.

To take into account the resource constraints (5.11) and (5.13), we must construct the inverse of the resource transformation $RT(\beta, \alpha)$ that is defined by (5.2). We can determine the inverse of (5.2) with respect to α as the largest $\alpha(\Delta)$ such that (5.2) holds, given β' and β

$$\begin{aligned}\alpha(\Delta) &= \beta(\Delta + \lambda) - \beta'(\Delta + \lambda) \quad \text{for } \lambda = \sup\{\tau : \beta'(\Delta + \tau) = \beta'(\Delta)\} \\ &\stackrel{def}{=} RT^{-\alpha}(\beta', \beta)\end{aligned} \tag{5.14}$$

and we can determine the inverse of (5.2) with respect to β as the smallest $\beta(\Delta)$ such that (5.2) holds, given β' and α

$$\begin{aligned}\beta(\Delta) &= \beta'(\Delta - \lambda) + \alpha(\Delta - \lambda) \quad \text{for } \lambda = \sup\{\tau : \beta'(\Delta - \tau) = \beta'(\Delta)\} \\ &\stackrel{def}{=} RT^{-\beta}(\beta', \alpha)\end{aligned} \tag{5.15}$$

Using these inverses, we can establish the complete set of interface relations in a processing component for FP scheduling:

$$\beta'^G = RT(\beta^G, \alpha^G) \tag{5.16}$$

$$\beta^A = \max\left\{\alpha^G(\Delta - d^G), RT^{-\beta}(\beta'^A, \alpha^G)\right\} \tag{5.17}$$

$$\alpha^A = \min\left\{\beta^G(\Delta + \hat{d}^G), RT^{-\alpha}(\beta'^A, \beta^G)\right\} \tag{5.18}$$

$$d^A = Del(\beta^G, \alpha^G) \tag{5.19}$$

5.3.4 Processing Component for RM Scheduling

Def. 16: **(Processing Component for RM Scheduling)** *A* processing component for rate monotonic scheduling *models a set of n tasks in a real-time system, that share system services with a rate monotonic scheduling strategy, and that use the available system services to process n real-time event streams. The Real-Time Interface of a processing component for RM scheduling has a service input variable, and a service output variable. The interface has the input assumption*

$$\phi^I = (\beta \geq \beta^A) \tag{5.20}$$

and the output guarantee

$$\phi^O = (\beta' \geq \beta'^G) \tag{5.21}$$

With the input assumption and the output guarantee of its Real-Time Interface, a processing component for rate monotonic scheduling expresses that whenever (a) the service $\beta(\Delta)$ that is provided to the component on its service input is larger or equal $\beta^A(\Delta)$ for any Δ, then (i) the n arriving loads can be processed in real-time, and (ii) the service $\beta'(\Delta)$ that is provided by the component on its service output is always larger or equal $\beta'^G(\Delta)$ for any time interval Δ, and can be used by lower-priority tasks. Figure 57(d) depicts the interface of a processing component for RM scheduling.

Rate monotonic scheduling [LL73] is a special instance of preemptive fixed priority scheduling for a set $W = \{T_i\}$ of n real-time event streams that can all be described by the periodic task model $T(p_i, e_i)$, where p_i is a period, and e_i is an execution time requirement with $e_i \leq p_i$, and that all have a relative delay requirement of $d_i = p_i$. Internally, such a processing component for RM scheduling can be modeled by n event stream components that are connected to n processing components for FP scheduling. The n event stream components have the output guarantee bounds $\alpha^G(\Delta) = \lceil \Delta / p_i \rceil \cdot e_i$ and $d^G = p_i$, and the n processing components for FP scheduling are interconnected by their service connection, ordered from top to down, according to the decreasing periods of their input event streams.

5.3.5 Processing Component for EDF Scheduling

Def. 17: **(Processing Component for EDF Scheduling)** *A* processing component for preemptive earliest deadline first scheduling *models a set of n tasks in a real-time system, that share system services with an earliest deadline first scheduling strategy, and that use the available system services to process n real-time event streams. The Real-Time Interface of a processing component for EDF scheduling has n arrival input variables and n corresponding delay input*

variables, a service input variable, and a service output variable. The interface has the input assumption

$$\phi^I = \bigwedge_{\forall i} \left((\alpha_i \leq \alpha_i^A) \wedge (d \geq d_i^A) \right) \wedge (\beta \geq \beta^A) \tag{5.22}$$

and the output guarantee

$$\phi^O = (\beta' \geq \beta'^G) \tag{5.23}$$

With the input assumptions and the output guarantees of its Real-Time Interface, a processing component for earliest deadline first scheduling expresses that whenever (a) the service $\beta(\Delta)$ that is provided to the component on its service input is larger or equal $\beta^A(\Delta)$ for any Δ, and (b) for all arrival inputs, the load $\alpha_i(\Delta)$ that arrives at the component on its i^{th} arrival input is less or equal $\alpha_i^A(\Delta)$ for any Δ, and has a maximum required delay that is larger or equal d_i^A, then (i) the arriving loads can be processed in real-time, i. e. with a respective guaranteed delay $d_{max,i} \leq d_i^A \leq d_i^G$, and (ii) the service $\beta'(\Delta)$ that is provided by the component on its service output is always larger or equal $\beta'^G(\Delta)$ for any time interval Δ, and can be used by lower-priority tasks. Figure 57(e) depicts the interface of a processing component for EDF scheduling.

In an EDF component, the total resource demand that the n arriving event streams generate can be bounded by the sum of their arrival curves. We therefore get the resource transformation of an EDF component, if we replace α with $\sum_{\forall i} \alpha$ in (5.2). From this, we can already derive the internal interface relation to compute β'^G for an EDF component.

To compute the input service assumption β^A, we must consider the delay constraints of all n arriving event streams

$$\beta^A(\Delta) \geq \sum_{\forall i} \alpha_i^G(\Delta - d_i^G) \tag{5.24}$$

as well as the resource constraint

$$\beta^A(\Delta) \geq \inf \left\{ \beta : \beta'^A(\Delta) = \sup_{0 \geq \lambda \geq \Delta} \left\{ \beta(\lambda) - \sum_{\forall i} \alpha_i^G(\lambda) \right\} \right\} \tag{5.25}$$

The tightest β^A is the maximum of both expressions.

To compute the input arrival assumption for the j^{th} arrival input α_j^A on the other hand, we first need to know, which share of the resource guarantee β^G is available to process the j^{th} arriving event stream in an EDF component. From (5.24) and (5.4) and (5.5), we know that

$$\sum_{\forall i} \alpha_i^A(\Delta - d_i^G) \leq \beta^G(\Delta) \tag{5.26}$$

and if we look at α_j^A of a single event stream, we get

$$\begin{aligned} \alpha_j^A(\Delta - d_j^G) &\leq \inf_{0 \leq \lambda} \left\{ \beta^G(\Delta + \lambda) - \sum_{\forall i \neq j} \alpha_i^A(\Delta - d_i^G + \lambda) \right\} \\ &\stackrel{def}{=} \beta_{EDF,j}^G \end{aligned} \tag{5.27}$$

The right side of (5.27) equals the share of the resource β^G that is available to process the j^{th} arriving event stream.

To compute α_j^A, we then need to consider the delay constraint

$$\alpha_j^A(\Delta) \leq \beta_{EDF,j}^G(\Delta + d^G) \tag{5.28}$$

and the resource constraint

$$\alpha_j^A(\Delta) \leq \sup \left\{ \alpha : \beta'^A(\Delta) = \sup_{0 \geq \lambda \geq \Delta} \left\{ \beta_{EDF,j}^G(\lambda) - \alpha(\lambda) \right\} \right\} \tag{5.29}$$

The tightest α_j^A is the minimum of both expressions.

Using the inverses of (5.2) that are defined by (5.14) and (5.15), we can then establish the complete set of interface relations in a processing component for EDF scheduling:

$$\beta'^G = RT(\beta^G, \sum_{\forall i} \alpha_i^G) \tag{5.30}$$

$$\beta^A = \max \left\{ \sum_{\forall i} \alpha_i^G(\Delta - d_i^G), RT^{-\beta} \left(\beta'^A, \sum_{\forall i} \alpha_i^G \right) \right\} \tag{5.31}$$

$$\alpha_j^A = \min \left\{ \beta_{EDF,j}^G(\Delta + d_j^G), RT^{-\alpha}(\beta'^A, \beta_{EDF,j}^G) \right\} \tag{5.32}$$

$$d_j^A = Del(\beta_{EDF,J}^G, \alpha_j^G) \tag{5.33}$$

5.3.6 Processing Component for FIFO Scheduling

Def. 18: (Processing Component for FIFO Scheduling) *A* processing component for first in first out scheduling *models a set of n tasks in a real-time system, that share system services with a first in first out scheduling strategy, and that use the available system services to process n real-time event streams. The Real-Time Interface of a processing component for FIFO scheduling has n arrival input variables and n corresponding delay input variables, a service input variable, and a service output variable. The interface has the input assumption*

$$\phi^I = \bigwedge_{\forall i} \left((\alpha_i \leq \alpha_i^A) \wedge (d \geq d_i^A) \right) \wedge (\beta \geq \beta^A) \tag{5.34}$$

and the output guarantee

$$\phi^O = (\beta' \geq \beta'^G) \tag{5.35}$$

With the input assumptions and the output guarantees of its Real-Time Interface, a processing component for first in first out scheduling expresses that whenever (a) the service $\beta(\Delta)$ that is provided to the component on its service input is larger or equal $\beta^A(\Delta)$ for any Δ, and (b) for all arrival inputs, the load $\alpha_i(\Delta)$ that arrives at the component on its i^{th} arrival input is less or equal $\alpha_i^A(\Delta)$ for any Δ, and has a maximum required delay that is larger or equal d_i^A, then (i) the arriving loads can be processed in real-time, i. e. with a respective guaranteed delay $d_{max,i} \leq d_i^A \leq d_i^G$, and (ii) the service $\beta'(\Delta)$ that is provided by the component on its service output is always larger or equal $\beta'^G(\Delta)$ for any time interval Δ, and can be used by lower-priority tasks. Figure 57(f) depicts the interface of a processing component for FIFO scheduling.

First in first out scheduling can be implemented as a special instance of earliest deadline first scheduling, where the same deadline is assigned to all processed event streams. And to ensure the delay requirements of all event streams, this deadline must equal the smallest deadline of all processed event streams, that is $D_{FIFO} = \min_{\forall i}\{D_i\}$. A processing component for FIFO scheduling can thus be modeled by a processing component for EDF scheduling, where the deadline for all n event streams is set to $D_i^* = D_{FIFO}$.

5.4 Applications and Experimental Results

Interface-based design with Real-Time Interfaces enables a number of applications that contribute to a simplified system design process, and some applications even lead to interesting on-line applications. Following, we introduce a number of applications of interface-based design within the MPA framework.

5.4.1 Application Scenario

Consider the real-time system depicted in Figure 58(a), that processes three real-time event streams R_1, R_2, and R_3, on three fully preemptable and independent tasks T_1, T_2, and T_3, that run on a processor P_I, that implements a preemptive fixed priority scheduling policy to share its available resources among the three tasks. The highest priority is thereby assigned to task T_1, while task T_3 runs with the lowest priority. The events on event stream R_1 arrive with a period $P_{R_1} = 4s$, and a jitter $J_{R_1} = 3s$, have an execution demand of $e_{R_1} = 3E8$ cycles, and a relative deadline $d_{R_1} = 1.5s$. On event stream R_2 on the other hand, the events arrive with a period $P_{R_2} = 2s$, a jitter $J_{R_2} = 10s$, and a minimum inter-arrival distance $D_{R_2} = 0.3s$, have an execution demand of $e_{R_2} = 1.5E8$ cycles, and a relative

deadline $d_{R_2} = 10s$. And event stream R_3 is specified by $P_{R_3} = 7s$, $J_{R_3} = 15s$, $D_{R_3} = 3s$, $e_{R_3} = 5E8$ cycles, and $d_{R_3} = 20s$. The processor is fully available to process the three tasks, and runs with a frequency $f_{P_I} = 350MHz$.

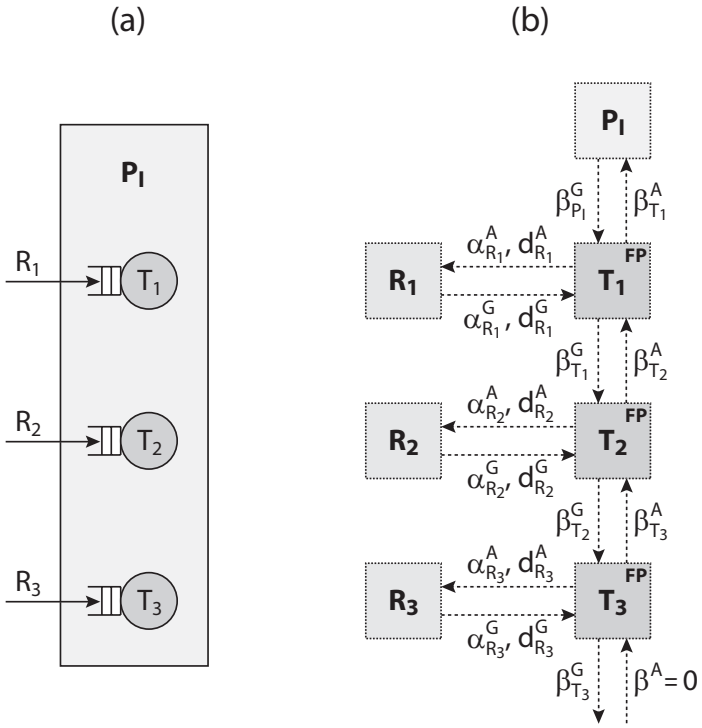


Fig. 58: (a) A stream processing real-time system, and (b) its Real-Time Interface model.

For interface-based design, we first need to specify the various system components and their interfaces. We model the processor with a resource component with a service guarantee $\beta^G(\Delta) = 3.5e8$ cycles·Δ. We further model the three event streams with three event stream components, with a respective delay guarantee $d^G_{R_i} = d_{R_i}$, and a load guarantee $\alpha_{R_i}(\Delta)$ that we compute from P_{R_i}, J_{R_i}, D_{R_i} and e_{R_i}, using (2.2), and multiplying the resulting curve with e_{R_i}. Finally, we model the three tasks using three processing components for FP scheduling. We then compose the interfaces of all these components to the complete system interface model, as depicted in Figure 58(b).

5.4.2 Interface-Based Schedulability Analysis

With interface-based design, schedulability analysis is done implicitly during system composition. Thus, since we can successfully compose all component interfaces of the application scenario, as depicted in Figure 58)(b), we are guaranteed that the system as specified is schedulable.

Figure 59 depicts the A/G arrival bounds at the three event stream connections within the system interface model. And as expected, we see that the load guarantees $\alpha^G_{R_i}(\Delta)$ at these connections are smaller than the respective the load assumptions $\alpha^A_{T_i}(\Delta)$, which complies to the requirement (5.4) for interface compatibility at event stream connections.

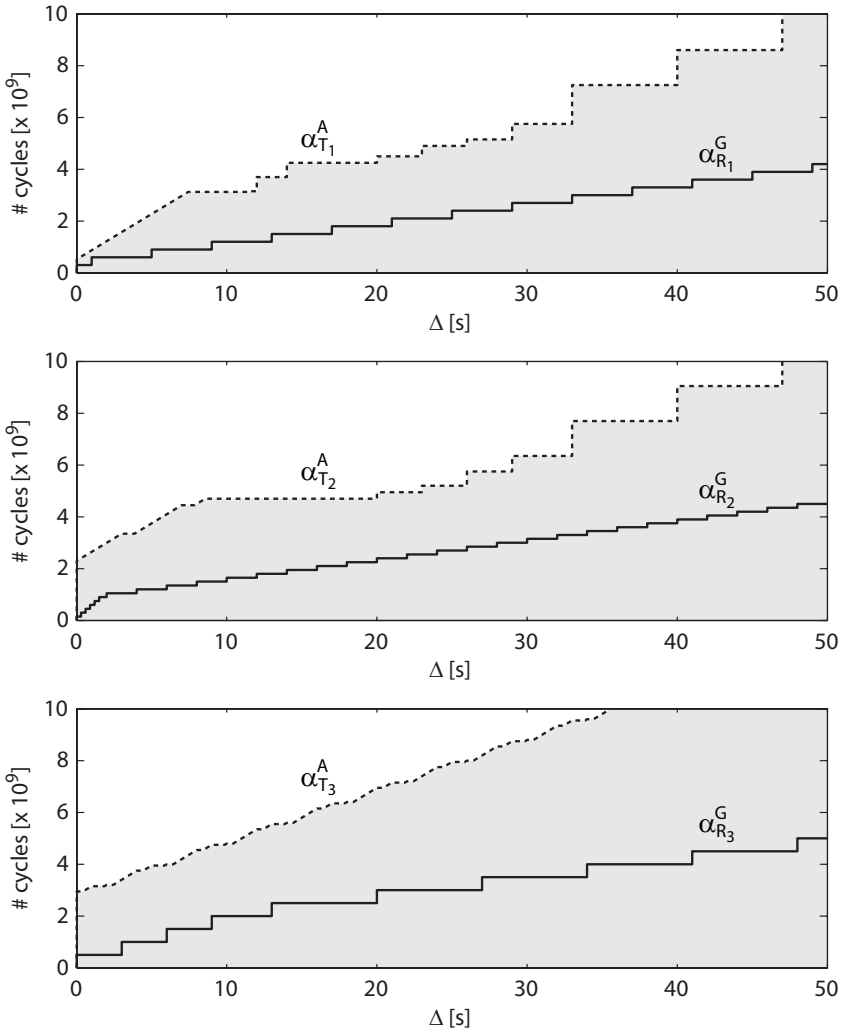


Fig. 59: The output arrival guarantees $\alpha^G_{R_i}$, and the input arrival assumptions $\alpha^A_{T_i}$, at the three event stream connections of the event streams R_1, R_2, and R_3.

5.4.3 Interface-Based System Design

Interface-based design of real-time systems benefits from all advantages of interface-based design as described in [dAH05]. These enable a variety of interesting design applications. The property of incremental design, together with the property of dynamic adaptability, allows for example to design a real-time system by first composing all event stream and processing components. This leads to an interface model with only one open service input with a service input assumption β^A_{tot}. And according to (5.5), it is guaranteed that the complete system is schedulable on any resource with a service output guarantee $\beta^G_{tot} \geq \beta^A_{tot}$. This means that once we know β^A_{tot}, we can directly decide whether a system is schedulable on a resource with a given service guarantee β^G_{tot}. In particular, we do not need to do any schedulability analysis, other than checking the relation $\beta^G_{tot} \geq \beta^A_{tot}$, that is the complete system schedulability information is encoded in β^A_{tot}.

Figure 60 depicts the A/G service bounds at the service connections within the system interface model. As expected, we see that the service guarantee $\beta^G_{P_I}(\Delta)$ at the connection between the resource component and the processing component of task T_1 is larger than the service assumption $\beta^A_{T_1}(\Delta)$, which complies to the requirement (5.5) for interface compatibility at service connections. Suppose now that a designer wants to exchange processor P_I for a faster processor P_{II} with a frequency $f_{P_{II}} = 600MHz$, that applies a TDMA policy to share its resources with other applications, and that allocates a slot of $1s$ within every cycle of length $2s$ to our application. To guarantee that this processor exchange retains the system schedulability, the designer only needs to check that the service guarantee $\beta^G_{P_{II}}(\Delta)$ of processor P_{II} is larger than the total service assumption $\beta^A_{T_1}(\Delta)$ of our application, which is true, as we can see in the same figure.

Analogously, by looking at the service input assumption $\beta^A_{T_1}$ of the complete system, a system designer can directly find the tightest possible service guarantee with $\beta^G_{P_I} \geq \beta^A_{T_1}$. By choosing the most economic processor, that still conforms to this tight service guarantee, he then obtains an economic real-time system that guarantees system schedulability, without being over-dimensioned. This design procedure stands in contrast to traditional system design, where resource components are chosen a priori, and performance analysis methods are used a posteriori to decide whether the system is schedulable or not. In this traditional approach, economic designs must be found by trial-and-error, i. e. by parameter sweeps or binary search.

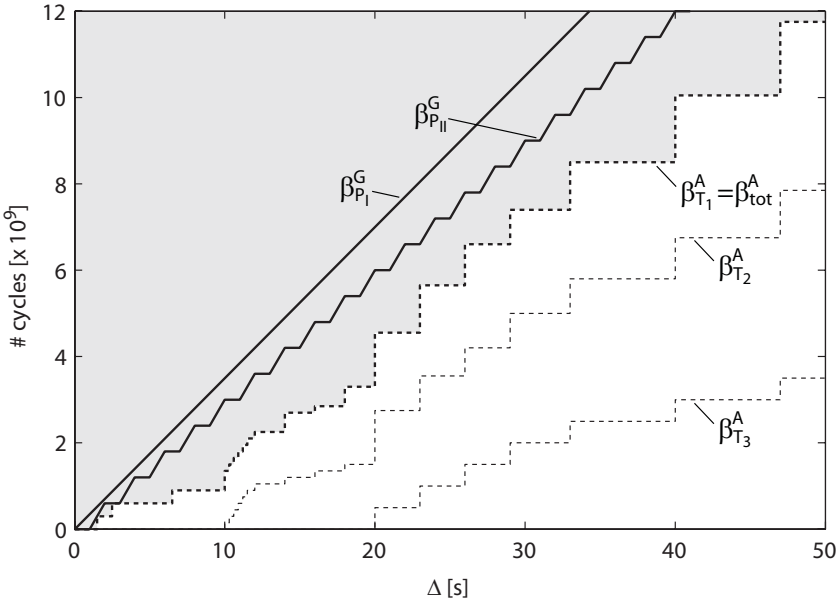


Fig. 60: The output service guarantees $\beta^G_{P_I}$ for a processor P_I that is fully available, and $\beta^G_{P_{II}}$ for a processor that provides a TDMA resources share, together with the input service assumptions $\beta^A_{T_1}$, $\beta^A_{T_2}$, and $\beta^A_{T_3}$, of the three processing components of the tasks T_1, T_2, and T_3. Note, of the three depicted input service assumptions, only $\beta^A_{T_1}$ needs to be considered to check schedulability.

5.4.4 Interface-Based System Adaption

After composition of the complete system, the assumtion bounds on all internal component connections specify the maximum arrival load, the minimum deadline, and the minimum service, respectively, that is allowable at the specific connection to keep the system schedulable. We can exploit this information to adapt for example the load on our system up to the maximum limit without trial-and-error, and therefore without the danger of rendering the system to become unschedulable. Or analogously, we could also directly reduce the service to our system down to the minimum, as we already mentioned above, or we could directly reduce the delay requirement to the minimum allowable deadline.

Figure 61 depicts the service assumption $\beta^A_{T_1}$ of the complete system. From this assumption, we learn that the processor P_I could lower its frequency from initially $f_{P_I} = 350MHz$ down to $f^*_{P_I} = 258MHz$. Figure 62 on the other hand, depicts the arrival assumption $\alpha^A_{T_2}$ of task T_2. We see that the arrival guarantee $\alpha^G_{R_2}$ of event stream R_2 does currently by far not exploit this assumption. We can therefore for example increase the

arrival rate of R_2, by changing its period from $P_{R_2} = 2s$ to $P^*_{R_2} = 1s$ (to retain the same burstiness, we decrease the jitter from $J_{R_2} = 10s$ down to $J^*_{R_2} = 4s$). Since the new arrival guarantee $\alpha^{G^*}_{R_2}$ still complies with $\alpha^A_{T_2}$, we are guaranteed that the system remains schedulable after this load adaption. Further, when we compute the delay assumption $d^A_{T_2} = 3s$ for the system after the load adaption, we learn that we could decrease the delay guarantee of R_2 from $d^G_{R_2} = 10s$ down to $d^{G^{**}}_{R_2} = 3s$. This will change the arrival assumption of T_2, and since the arrival guarantee $\alpha^{G^*}_{R_2}$ still complies with the new arrival assumption $\alpha^{A^{**}}_{R_2}$, we are guaranteed that the system remains also schedulable after this delay requirement adaption.

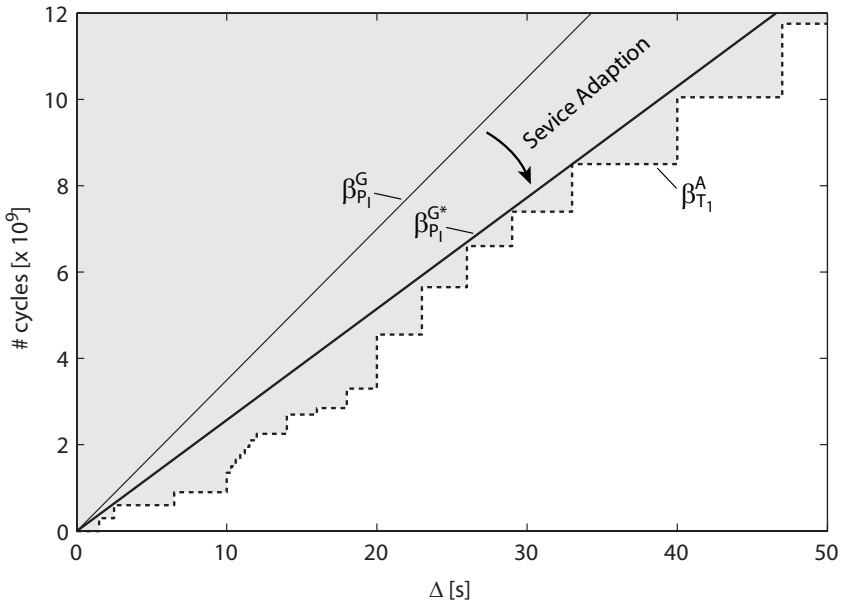


Fig. 61: The output service guarantee $\beta^G_{P_I}$ of processor P_I with $f_{P_I} = 350MHz$, and the output service guarantee $\beta^{G^*}_{P_I}$ after the service adaption to $f^*_{P_I} = 258MHz$, together with the input service assumption $\beta^A_{T_1}$ of task T_1, that encodes the input service assumption of the complete system.

These interface-based system adaptions would also be interesting to be conducted on-line on a real-time system. This would be particularly interesting in the area of power-aware real-time systems or real-time systems for QoS-applications, where we would deploy the interface model of the complete real-time system under consideration together with the concrete system.

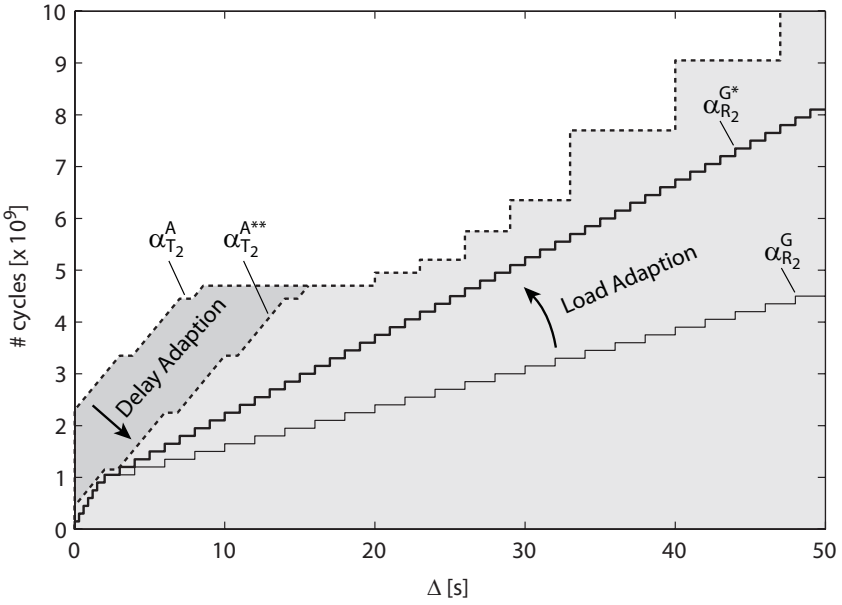


Fig. 62: The output arrival guarantee $\alpha^G_{R_2}$ of event stream R_2, and the output arrival guarantee $\alpha^{G^*}_{R_2}$ of event stream R_2 after increasing the arrival rate, together with the input arrival assumption $\alpha^A_{T_2}$ of task T_2, and the input arrival assumption $\alpha^{A^{**}}_{R_2}$ of task T_2 after decreasing the delay guarantee of R_2.

5.4.5 Interface-Based Admission Tests

Another application of Real-Time Interfaces are interface-based admission tests. For this, we add a dummy processing component with a connected dummy event stream component with $\alpha^G = 0$ and $d^G = \infty$ at every existing service interface connection of the system interface model. These additional process components, that add new event stream inputs to the system interface, have no influence to the compatibility of the initial system interface, since the load of added the event streams are zero. But the assumptions $\alpha^A_{T_4,i}$ and $d^A_{T_4,i}$ on the corresponding event stream connections contain a detailed specification of the load that could be processed by an additional task T_4, when scheduled with the priority i. When we then add a new task T_4 to process a periodic event stream R_4 with $P_{R_4} = 10s$, $e_{R_4} = 6E8$ cycles, and $d_{R_4} = 4s$, we only need to check the load guarantee $\alpha^G_{R_4}$, with the various load assumptions $\alpha^A_{T_4,i}$, to determine whether this load can be admitted, and if yes with what priority T_4 should run. From the bounds depicted in Figure 63, we learn that the event stream R_4 can be admitted, but only with a task T_4 that is scheduled at priority $p_{T_4} = 2$. Any other priority assignment of T_4 would render the system

unschedulable.

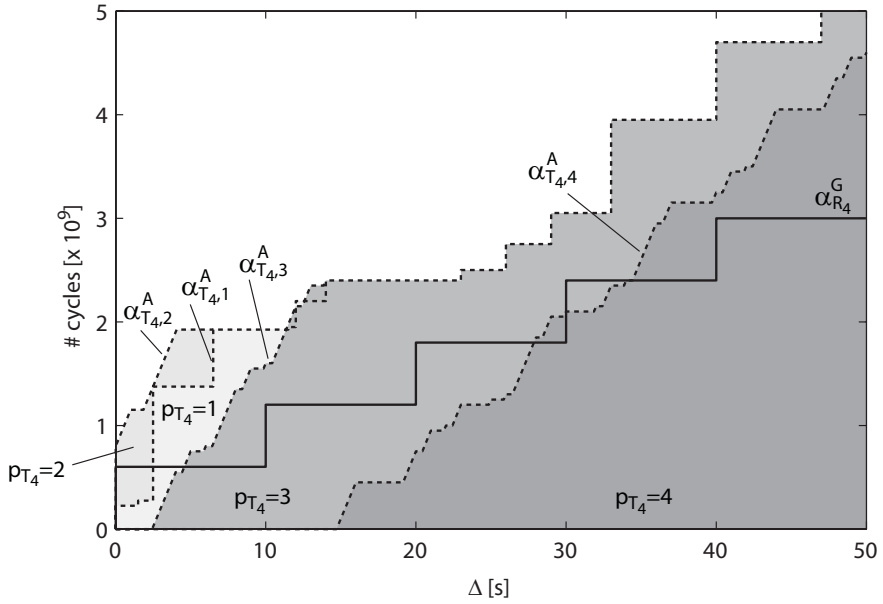


Fig. 63: Interface-based admission test.

5.5 Related Work

In [dAH01b] and [dAH05], de Alfaro and Henzinger first describe the principles of interface-based design that build the foundation for Real-Time Interfaces. Although de Alfaro and Henzinger mention A/G interfaces in their work, their interface theory research is mainly centered around the stateful interface language of interface automata [dAH01a], including extensions towards the use of time [dAHS02], or resources [CdAHS03]. With these interface languages, the temporal input/output behavior of a component is modeled by an automaton, and the automaton of a composite interface is constructed by pruning all violating states from the product of the component interface automata.

In [WRM+05], Wang et al. propose a method for component-based analysis and design of real-time system, that uses the notion and concept of traditional software component standards such as CORBA. Wang et al. propose extensions in the interface of such software components to include static real-time assumptions and guarantees. These static assumptions and guarantees are expressed using a restricted set of arrival curves, together with a deadline, and allow to model assumptions and

guarantees on the load that can be processed by a component. The proposed interfaces have however no notion of resource utilization.

In [SL04], Shin et al. propose a compositional scheduling framework to determine the schedulability of real-time systems with a set of applications that are scheduled hierarchically. In this framework, the resource demand of a single task is represented as a demand bound function **dbf**(Δ), and a scheduling component has as input the set of demand bound functions of all tasks that are scheduled by this component. Depending on the associated scheduling strategy, a scheduling component then determines the total demand to schedule all tasks and expresses this again as a demand bound function on its output. Scheduling components can then be composed hierarchically, and the complete system is schedulable if the demand of the scheduling component at the top of the hierarchy can be fulfilled by a dedicated resource that is represented with a supply bound function **sbf**(Δ). Although Shin et al. do not relate their work to interface-based design, it implicitly features several concepts known from this area. Demand bound functions express service assumptions of a scheduling component, and supply bound functions express service guarantees of a dedicated resource. A system is then schedulable, if the service assumption of the complete system is smaller than the service guarantee of the dedicated hardware.

In the area of real-time scheduling theory, several other methods were proposed that also use the concept of demand bound functions **dbf**(Δ) and supply bound functions **sbf**(Δ) for compositional schedulability analysis, see for example [Bar03], [SL03], or [AP04]. In general, with these functions, a component is considered to be schedulable, if **dbf**(Δ) $\leq$ **sbf**(Δ) $\forall\Delta$. The same concept also exists in the theory of Real-Time Interfaces, where the bound $\beta^A(\Delta)$ of the input assumption on a service connection can be interpreted as a demand bound function **dbf**(Δ), and the bound $\beta^G(\Delta)$ of the output guarantee on a service connection can be interpreted as a supply bound function **sbf**(Δ). Then, the compatibility requirement (5.5) on a service connection equals the above schedulability requirement **dbf**(Δ) $\leq$ **sbf**(Δ).

Common to all of this previous work in the area of real-time scheduling theory is the use of very restricted models to propagate constraints (resource demands) between different hierarchy layers. All methods use either a periodic or a bounded delay resource model, to model the resource demand at any given level of hierarchy. The actual resource demand at a given level of hierarchy is however typically more complex, and consequently a considerable abstraction overhead is introduced with every level of hierarchy. Moreover, the methods presented in [SL03, SL04, BL03] only consider systems that are triggered by strictly periodic event streams, while [AP04] considers only periodic event streams with jitter, and they

all do not explicitly express assumptions on the processed event streams.

In [HM06], Henzinger et al. propose an approach to interface-based design of real-time systems that relies on stateless A/G interfaces. In this approach, a single task sequence is represented by an interface that specifies assumptions on the input event stream and the available resources, and that gives guarantees about the output event stream. The input and output event streams are both modeled using a restricted class of arrival curves, and the former are further augmented with an associated delay. Assumptions on the resource availabilities on the other hand are modeled using so-called capacity functions [SL04], each representing a whole class of bounded delay resource models. The use of these capacity functions largely restricts the applicability of the method proposed by Henzinger et al., because for interface compatibility as defined in [HM06], the service assumption of a tasks sequence component, represented as a capacity function, must me smaller than the service guarantee of a resource component, also represented as a capacity function. This implicates that the whole class of bounded delay resource assumptions modeled by the capacity function of the tasks sequence component must be smaller than the whole class of bounded delay resource guarantees modeled by the capacity function of the resource component. In practice however, it is sufficient, if one single bounded delay assumption of the task sequence component is smaller than the corresponding bounded delay guarantee of the resource component. Moreover, the method presented by Henzinger et al. does not support independent implementability.

5.6 Discussion

Real-Time Interfaces connect the principles of interface-based design and Real-Time Calculus, and can therefore fall back to a wide range of already established results for both of these theories. It is also because of this, that the theory of Real-Time Interfaces is not confined to the interface models and component system presented in this chapter, but can instead be generalized into various directions.

In this chapter, the interfaces of components expose the assumptions and guarantees on the arriving loads, the associated minimum deadlines, and the service supplies. However, a component could also expose additional information on its interface. An abstract component could for example express an assumption b^A on the available buffer space to process the incoming event stream, and in return, a resource component could provide a guarantee b^G on the minimum available buffer space. The compatibility predicate (5.5) on a service connection would then be extended by a predicate $b^A \leq b^G$, and in the internal interface relations of the pro-

cessing components, an additional buffer space constraint would have to be considered, that would for example add an additional constraint to (5.17) and (5.18).

Another extension of Real-Time Interfaces would make them applicable for the design and analysis of distributed embedded systems. For this, we would need to extend the abstract components of Section 5.3, with an output that models an outgoing event stream, equal to the abstract components introduced in Chapter 2. A processing component would then have an additional output event stream connection, with a corresponding output guarantee, and an input assumption from the connected succeeding processing component. This additional assumption would add yet another constraint to the internal interface relations of a processing component. It must be noted that with such an extended component system it is possible to construct interface models with cyclic dependencies between the guarantees and the assumptions on an interface variable. In general, interface compatibility checking requires then a fixed point calculation. However for many practical system designs this may not be required.

Finally, the component system introduced in Section 5.3 could in principle also be extended with any other abstract components of the MPA framework, such as for example greedy shaper components or components with multiple inputs, as introduced in Chapter 3.

6

Design & Analysis of Systems with Hierarchical Scheduling

The advances in the field of computer architecture lead to increasingly powerful microprocessors, and triggered a trend towards higher integration of functionality in embedded systems design. Previously, embedded system architectures mostly employed low-cost microprocessors as the basis for separate Electronic Control Units (ECUs), each supporting a single hard real-time application. Modern embedded system architectures on the other hand are often comprised of a smaller number of more powerful microprocessors, with each supporting multiple hard real-time applications. This trend is mostly motivated by cost reductions, but also the reuse of legacy applications, as well the opportunity of functionality enhancements are driving factors.

The main question that arises when integrating a number of real-time applications onto a single microprocessor, is how to schedule these applications such that their individual timing requirements are not violated. The simplest method to compose several real-time applications onto a single resource would be to use a unique scheduling policy for all applications. Schedulability analysis of the integrated system could then be done using traditional analysis methods. However, this approach is typically not applicable, since the applications are often implemented by different vendors, and moreover because time and money constraints often force the re-use of already implemented applications. Moreover, this approach is alos not practical and scalable, since it does not allow independent implementation of independent applications. Consequently,

the problem then becomes how to integrate real-time applications with different individual scheduling policies onto a single resource, such that the individual timing requirements are not violated. While this problem is often addressed by introducing hierarchical schedulers, the difficulty then consists of the schedulability and performance analysis of the hierarchically scheduled applications, and of the parameter selection for the hierarchical schedulers.

This chapter introduces new components that enable interface-based design and analysis of systems with hierarchical scheduling within the MPA framework. In the following Section 6.1, a component is introduced that models a computation or communication resource with a TDMA sharing policy, and methods are presented for optimal parameter selection for such a component. In Section 6.2 on the other hand, components are introduced that model polling servers within the MPA framework, both for static as well as for dynamic scheduling policies, and methods are presented to simplify parameter selection for such servers in a real-time setting. The chapter concludes with a discussion in Section 6.3.

6.1 Time Division Multiple Access

In large distributed embedded systems, time division multiple access (TDMA) scheduling policies play an increasingly important role, and are often employed on backbone communication resources that interconnect a large number of embedded control units (ECU's). This trend can best be observed in the area of safety-critical automotive and avionic systems, where TDMA-based communication protocols such as TTP [KG93], or more recently the mixed TDMA/FTDMA-based FlexRay [BES+01, Fle] replace the formerly omnipresent CAN protocol. But also for communication on MpSoC's [HE05], as well as to provide QoS guarantees in network on chips [GDvM+03], TDMA gets increasingly important. And besides these communication centric applications, TDMA is also an interesting candidate to coordinate global resource sharing on a single processor, to enable composable and hierarchical scheduling.

The major advantages of TDMA-scheduled resources is the support of temporal composability, by clearly separating resources access of different subsystems. TDMA thus eliminates any sort of interference of unrelated subsystems with each other. Moreover, TDMA-scheduled resources have a very deterministic timing behavior, can be made fault tolerant, and support error detection, as well as error contention, i. e. a faulty subsystem does not affect the correct behavior of other subsystems. A major difficulty that arises however during the design process of systems with TDMA-scheduled resources is parameter selection. Customizable parameters are

typically the total bandwidth B of the resource, the cycle length c of the TDMA round, as well as the individual slot lengths s_i for the different service consumers of the TDMA-scheduled resource.

6.1.1 Performance Analysis

Consider a real-time system consisting of n applications that are executed on a resource with bandwidth B that controls resource access of the n applications using a TDMA policy. Analogously, we could consider a distributed real-time system with n communicating nodes, that communicate via a shared bus with bandwidth B, with a bus arbitrator that implements a TDMA policy. For both scenarios, the TDMA cycle length is denoted as $\bar{c}$ and can only take on values that are multiples of the cycle length quantum q_c. In every TDMA cycle, one single resource slot of length s_i is assigned to every application. In a realistic system, the slot lengths s_i can only take on values that are multiples of a slot length quantum q_s. We denote a quantized slot length as $\bar{s}_i = \lceil s_i/q_s \rceil \cdot q_s$. Further, every slot typically involves a slot overhead o_s, while the cycle itself involves a cycle overhead o_c. On communication resources, these overheads account for example for required network idle times between consecutive slots and cycles, CRC codes for channel fault detection, time synchronization data, or any other protocol related overhead. On computation resources, these overheads are typically smaller, and account for example for context switches. Depending on the different timing specifications, some bandwidth may remain unused in every communication cycle. Figure 64 depicts the timing specifications of this TDMA protocol.

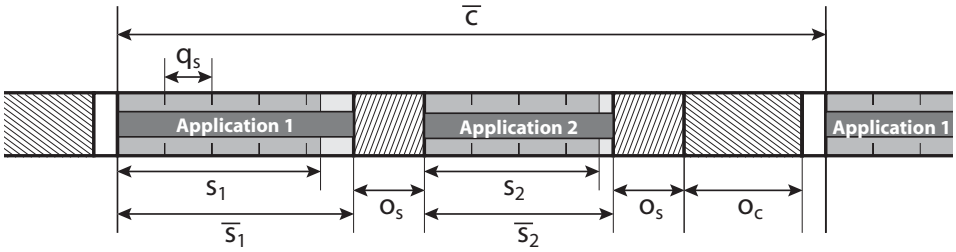


Fig. 64: TDMA protocol timing specifications.

Within the MPA framework, an abstract computation or communication resource that implements such a TDMA protocol can be modeled by a component as depicted in Figure 65(a), with the corresponding real-time interface depicted in Figure 65(b). The component has a parameter B, that determines the total bandwidth of the underlying resource, and the cycle length of the TDMA protocol is specified by the parameter $\bar{c}$. Fur-

ther, the component has n service outputs, that provide service with the output guarantees $\beta_i(\Delta) \geq \beta_i^G(\Delta)$ to the n applications with input assumptions $\beta_i^A(\Delta)$. Internally, the service output guarantee $\beta_i^G(\Delta)$ is determined from the slot length guarantee s_i^G that is assigned to the i^{th} application. Analogously, the service input assumption $\beta_i^A(\Delta)$ of an application can be transformed into a minimum slot length assumption s_i^A, as we will show in this section. The fulfillment of the service demand of an application can then be guaranteeing $s_i^G \geq s_i^A$.

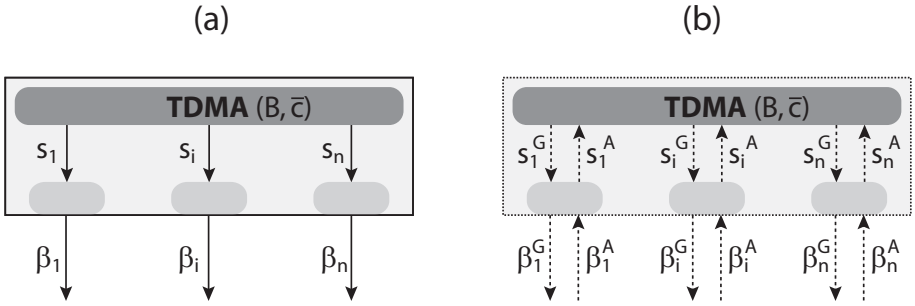


Fig. 65: (a) An abstract resource component with TDMA scheduling, and (b) its Real-Time Interface.

To determine the service guarantee $\beta_i^G(\Delta)$ to an application, we must consider that the i^{th} application may not have access to the resource during a time interval that is limited by $\Delta = \bar{c} - s_i^G$. After this interval however, the application is granted exclusive access to the resource during a time interval of length s_i^G. A resource can therefore not guarantee any service to a connected applicatoin during any time interval $0 \leq \Delta < \bar{c} - s_i^G$, but it can guarantee a service of $B(\Delta - (\bar{c} - s_i^G))$ in any time interval $\bar{c} - s_i^G \leq \Delta < \bar{c}$. This service guarantee can be expressed as

$$\beta_i^G(\Delta) = B \ \max\left\{\left\lfloor \frac{\Delta}{\bar{c}} \right\rfloor s_i^G, \Delta - \left\lceil \frac{\Delta}{\bar{c}} \right\rceil (\bar{c} - s_i^G)\right\} \tag{6.1}$$

or more compactly

$$\beta_i^G(\Delta) = B \sup_{0 \leq \lambda \leq \Delta} \left\{\lambda - \left\lceil \frac{\lambda}{\bar{c}} \right\rceil (\bar{c} - s_i^G)\right\} \tag{6.2}$$

According to (5.5), we then define that a system with n real-time applications that share a resource with a TDMA scheduling policy is *schedulable*, if this service guarantee to all applications is larger or equal than the respective service assumptions

$$\beta_i^G(\Delta) \geq \beta_i^A(\Delta) \quad \forall i, \forall \Delta \geq 0 \tag{6.3}$$

or equivalently, if $s_i^G \geq s_i^A \; \forall i$.

We further define that a real-time system with a TDMA scheduling policy is *feasible*, if the sum of the required slot lengths and the scheduling overhead is less or equal the cycle length, i. e. if

$$\bar{c} \geq \sum_{\forall i} s_i^A + o_c + no_s \tag{6.4}$$

Following this definition of feasibility, the slot length guarantee s_i^G to the i^{th} application can then be computed as

$$s_i^G = \bar{c} - \left(\sum_{\forall j \neq i} s_j^A + o_c + no_s \right) \tag{6.5}$$

Finally, we define the utilization σ_i^A as the quotient of the slot length s_i^A divided by the cycle length $\bar{c}$. Analogously to (6.4), a system with n real-time applications that share a resource with a TDMA scheduling policy is then feasible if the total utilization is less or equal one

$$\sum_{\forall i} \sigma_i^A + \frac{o_c + no_s}{\bar{c}} \stackrel{def}{=} \sigma_{tot}(\bar{c}) \leq 1 \tag{6.6}$$

For performance analysis within the MPA framework, a resource with a TDMA scheduling policy that serves n applications can be treated as n separate resources, each with a specific service guarantee $\beta_i^G(\Delta)$. Delay and backlog analysis of the respective real-time applications can then be conducted using (2.11) and (2.12).

6.1.2 Parameter Selection

Based on the powerful abstractions of service guarantees and service assumptions, and the explicit schedulability requirement (6.3), it is possible to determine optimal parameters for a TDMA scheduled resource.

6.1.2.1 Minimum Slot Time Allocation

Let us first determine the exact minimum time slot s_i^A that must be assigned to an application with service assumption $\beta_i^A(\Delta)$ to be schedulable on a TDMA-scheduled resource with bandwidth B and cycle length $\bar{c}$. For this, we need to construct the inverse of (6.1) with respect to s_i^G as the smallest s_i^G that leads to a service supply that fulfills the schedulability requirement (6.3)

$$s_i^A = \sup_{\Delta \geq 0} \left\{ \min \left\{ \frac{\beta_i^A}{B \left\lfloor \frac{\Delta}{\bar{c}} \right\rfloor}, \frac{\beta_i^A - B\Delta + B \left\lceil \frac{\Delta}{\bar{c}} \right\rceil \bar{c}}{B \left\lceil \frac{\Delta}{\bar{c}} \right\rceil} \right\} \right\} \tag{6.7}$$

This minimum time slot s_i^A is the smallest possible time slot allocation that guarantees a service supply $\beta_i^G(\Delta) \geq \beta_i^A(\Delta)$ on a TDMA resource with bandwidth B and slot length $\bar{c}$.

Ex. 10: *Consider an application A_1 consisting of a single task with period $p = 198ms$, jitter $j = 387ms$, and minimum inter-arrival distance $d = 48ms$. Further the task has an execution time of $e = 12ms$ and a relative deadline $D = 110ms$. Figure 66 shows the minimum slot length $s_{A_1}^A(\bar{c})$ assumed by this application, as a function of the TDMA cycle length $\bar{c}$.*

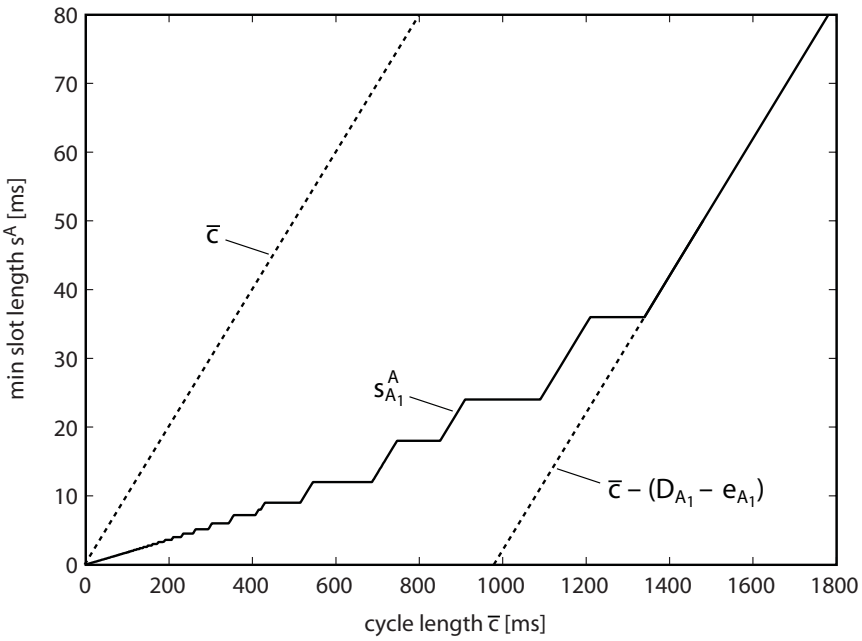


Fig. 66: Minimum required slot lengths $s_{A_1}^A$ for application A_1, together with the cycle length $\hat{c}$, and the lower bound $\bar{c} - (D_{A_1} - e_{A_1})$ to the slot length $s_{A_1}^A$.

We see that the minimum required slot length increases with increasing cycle length, and it is lower bounded by $s_{A_1}^A(\bar{c}) \geq \bar{c} - (D_{A_1} - e_{A_1})$, since the gap between two consecutive slots must never be larger than $D_{A_1} - e_{A_1}$. Figure 67 shows the minimum service guarantees $\beta_{A_1}^G(\Delta)$ for three different cycle lengths, computed by setting $s_{A_1}^G(\bar{c}) = s_{A_1}^A(\bar{c})$.

Using (6.7), we can now compute the minimum slot lengths for all applications in a real-time system with a TDMA-scheduled resource with bandwidth B and TDMA cycle length $\bar{c}$. If a slot allocation with these minimum slot lengths leads to a feasible system according to (6.4), i. e. if the sum of the minimum slot lengths plus the protocol overhead is smaller

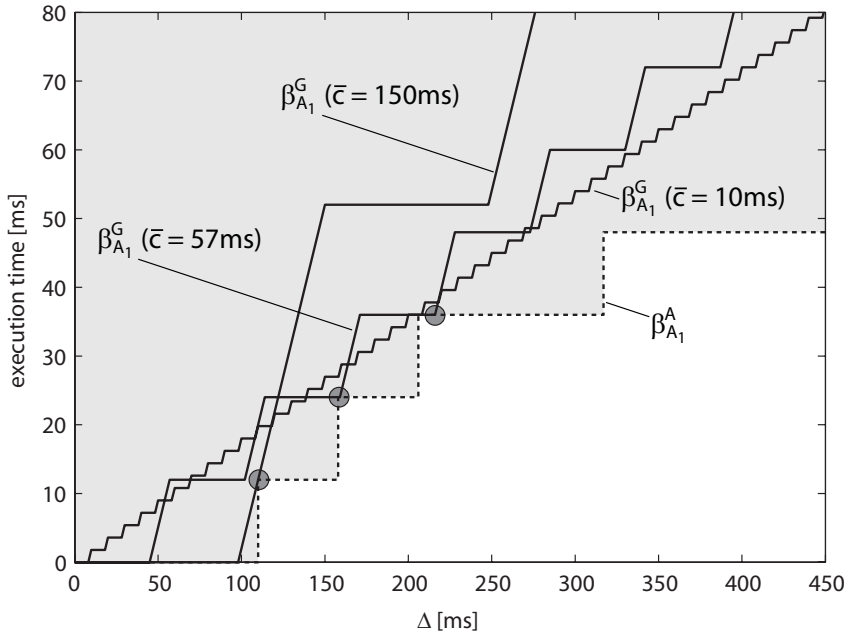


Fig. 67: The service assumption $\beta^A_{A_1}(\Delta)$ of application A_1, together with the minimum service guarantees $\beta^G_{A_1}(\Delta)$ for three different cycle lengths.

than the cycle length, then we can use $\bar{c}$ and $s_i = s_i^A(\bar{c})$ as TDMA settings. Otherwise, we are guaranteed that no feasible slot allocation exists for the cycle length $\bar{c}$.

6.1.2.2 Optimal Cycle Length

In a typical TDMA system, not only the slot lengths s_i, but also the cycle length $\bar{c}$ is a customizable parameter. To find an optimal cycle length for a TDMA scheduler, we first need to define an optimality criterion. One possible optimality criterion that we will use in the following, is the average remaining bandwidth $\sigma_r = 1 - \sigma_{tot}$. This remaining bandwidth could be distributed additionally to the existing applications, or it could be used to admit additional future load in a dynamic system.

To compute this remaining bandwidth, we need to consider that in order to account for up to m dynamically added applications in a dynamic system, we already need to include the slot overheads for these dynamically added applications. And further, for systems with a slot quantum, we also need to consider that only multiples of full slot quanti can be

assigned to existing or future applications

$$\bar{\sigma}_r(\bar{c}) = \left\lfloor \left(1 - \sum_{\forall i} \bar{\sigma}_i(\bar{c}) - \frac{(n+m)o_s + o_c}{\bar{c}} \right) \frac{\bar{c}}{q_s} \right\rfloor \frac{q_s}{\bar{c}} \tag{6.8}$$

In Section 6.1.3, we will see that the total utilization $\sigma_{tot}(\bar{c})$ as a function of the TDMA cycle length has a very complex and nonlinear behavior. To find the optimal cycle length with the maximum remaining bandwidth σ_r, we therefore have no choice but to compute σ_r for all possible values of $\bar{c}$. However, from Example 10 and Figure 66, we know that the required slot lengths are lower bounded by $s_i^A(\bar{c}) \geq \bar{c} - (D_i - e_i)$. It is therefore possible to find an upper bound to feasible cycle lengths

$$\bar{c}_{max} = \sup_{\bar{c} \geq 0} \left\{ \bar{c} \ : \ \bar{c} \geq \sum_{\forall i} \max(0, \bar{c} - (D_i - e_i)) \right\} \tag{6.9}$$

Therefore, σ_r only needs to be computed for $\bar{c}_{max}/q_c$ different values.

6.1.2.3 Minimum Total Service Bandwidth

At design time, the service bandwidth B of a resource is often also a customizable parameter. The minimum total service bandwidth B_{min} is the smallest possible service bandwidth B of a TDMA system with service assumptions β_i^A, for which feasible slot allocations s_i exists

$$B_{min} = \inf_{B \geq 0} \{ B : \sigma_{tot,min}(B) \leq 1 \} \tag{6.10}$$

with

$$\sigma_{tot,min}(B) = \inf_{\bar{c} \leq \bar{c}_{max}} \{ \sigma_{tot}(\bar{c}, B) \} \tag{6.11}$$

From (6.7), it can be seen that the minimum slot length s_i^A is monotonically decreasing with increasing service bandwidth B, i. e. $s_i^A(B + dB) \leq s_i^A(B)$. Because of this property, the total utilization σ_{tot} is also monotonically decreasing, and consequently also the minimum total utilization $\sigma_{tot,min}$ is monotonically decreasing with increasing service bandwidth B, i. e. $\sigma_{tot,min}(B + dB) \leq \sigma_{tot,min}(B)$. We can therefore find the minimum total service bandwidth B_{min} by binary search.

6.1.3 Experimental Results

Following, we analyze two real-time embedded systems with a number of independent applications that share a common resource with a TDMA scheduling policy.

6.1.3.1 TDMA-Based Communication

Consider a distributed real-time system, where 10 applications communicate via a shared bus, with a line speed of $1Mbps$, that implements a TDMA protocol for bus arbitration. Every applications A_i sends a real-time message stream M_i, that is specified with a period p_i, a jitter j_i, a minimum inter-arrival distance d_i, a total message size e_i, as well as a relative message delivery deadline D_i. The real-time message streams of the 10 applications are specified in Table 5. This system specification equals the specification of System 3 in the work of Hamann et al. [HE05], and is, according to Hamann et al., the most difficult system considered in their work.

	M_1	M_2	M_3	M_4	M_5	M_6	M_7	M_8	M_9	M_{10}
p [ms]	198	102	283	354	239	194	148	114	313	119
j [ms]	387	70	269	387	222	260	91	13	302	187
d [ms]	48	45	58	17	65	32	78	-	86	89
e [kb]	12	7	7	11	8	5	13	14	5	6
D [ms]	110	140	115	145	180	140	200	120	140	100

Tab. 5: Specification of the 10 message streams in the distributed embedded system.

To analyze this system, we compute $\sigma_{tot}(\bar{c})$ for $\bar{c} \in [0.1ms \ldots 600ms]$ with a cycle quantum $q_c = 100\mu s$. However, note that to find feasible TDMA parameters for this system, it would be sufficient to compute $\sigma_{tot}(\bar{c})$ for $\bar{c} \in [0.1ms \ldots 134.7ms]$, as can be computed from (6.9). Hamann et al. did not consider any protocol overhead or slot length quantization, and the corresponding results $\sigma_{tot}(\bar{c})$ are depicted in Figure 68. For a more realistic analysis, we also compute $\bar{\sigma}_{tot}(\bar{c})$, where we consider a slot length quantum $q_s = 10\mu s$, as well as protocol overheads of $o_s = 10\mu s$ in every slot, and $o_c = 20\mu s$ in every cycle. The resulting utilizations $\bar{\sigma}_{tot}(\bar{c})$ are also shown in Figure 68.

The grey shaded areas in Figure 68 visualize the ranges of cycle lengths $\bar{c}$, for which feasible TDMA parameters exist, i. e. for which $\bar{\sigma}_{tot}(\bar{c}) \leq 1$. When we compare this range to the results of $\sigma_{tot}(\bar{c})$, where we do not consider quantization effects and protocol overheads, we learn that considering these effects and overheads reduces the range of feasible cycle lengths $\bar{c}$ considerably. In general, if we do not consider quantization effects and protocol overhead, then the smallest possible $\bar{c}$ will always lead to feasible TDMA parameters, if the total bandwidth B is large enough. As soon as we consider quantization effects and protocol overhead however, arbitrary small values for $\bar{c}$ are not feasible anymore. But also arbitrary large values for $\bar{c}$ will not lead to feasible TDMA parameters, because the

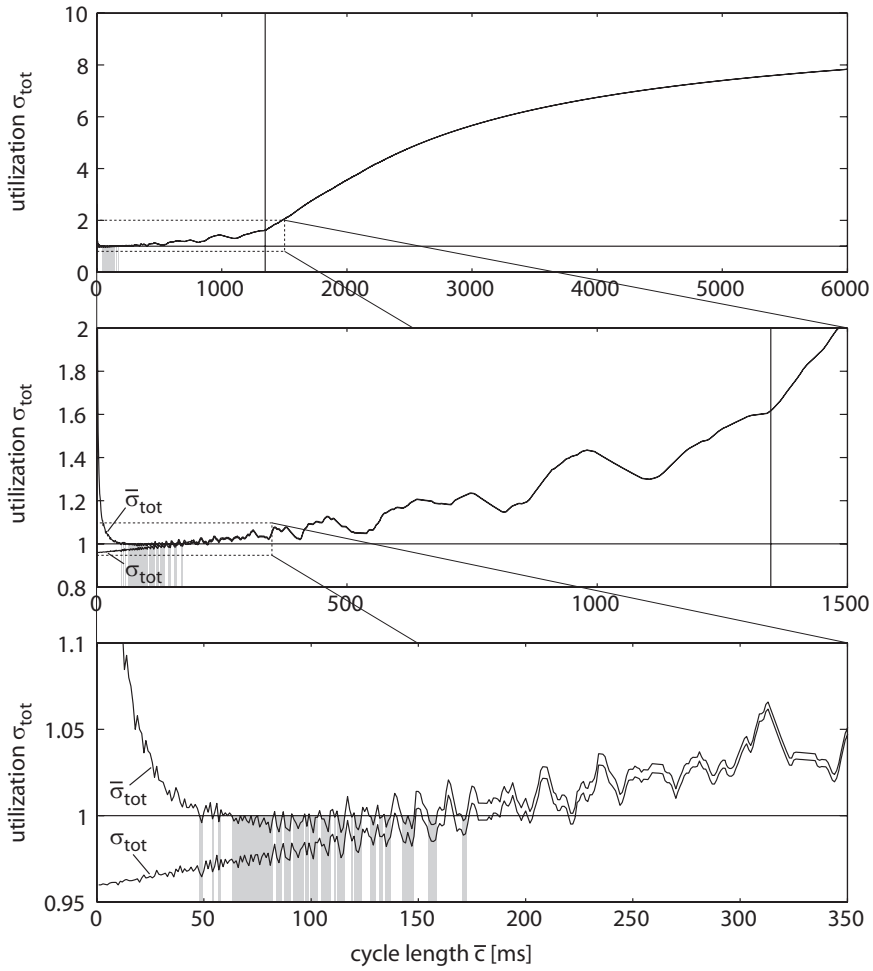


Fig. 68: Total utilization as a function of the TDMA cycle length. $\sigma_{tot}(\bar{c})$ is the total utilization without considering quantization effects or protocol overheads, while $\bar{\sigma}_{tot}(\bar{c})$ is the total utilization including these effects and overheads. The vertical line at $\bar{c} = 134.7ms$ marks the theoretical upper bound to feasible cycle lengths.

slot lengths are lower bounded by $s_i^A(\bar{c}) \geq \bar{c} - (D_i - e_i)$. $\sigma_{tot}(\bar{c})$ will therefore strive towards the number of applications in the system for large $\bar{c}$, i. e. towards 10 in our case. Feasible TDMA parameters thus only exist for cycle lengths $\bar{c}$ that are neither too small nor too large. As we can see in Figure 68, the total utilization $\bar{\sigma}_{tot}(\bar{c})$ as a function of the TDMA cycle length has a very complex and nonlinear behavior. Often, intervals of feasible cycle lengths are even non-contiguous. It is because of this behavior, that we need to compute $\bar{\sigma}_{tot}(\bar{c})$ for all possible values of $\bar{c}$, as we already mentioned in Section 6.1.2.2.

6.1.3.2 TDMA-Based Hierarchical Scheduling

Let us next consider a real-time embedded system with 7 applications that run on a single processor that implements a TDMA policy to share its service among the applications, as depicted in Figure 69. Application A_1 consists of the four tasks T_1–T_4 that are locally scheduled using an EDF scheduling policy, application A_2 also uses a local EDF scheduling policy to schedule the three tasks T_5–T_7. In application A_3 on the other hand, the three tasks T_8–T_{10} are locally scheduled using a FIFO scheduling policy, and in application A_4, the three tasks T_{11}–T_{13} are scheduled using a fixed priority policy, where T_{11} is assigned the highest priority, while T_{13} is assigned the lowest priority. Finally, the applications A_5–A_7 consist only of a single task each, namely T_{14}, T_{15}, and T_{16}, respectively. The 16 tasks are specified in Table 6, with their period p, jitter j, minimum inter-arrival time d, relative deadline D, and their execution demand e. The TDMA scheduler has a cycle quantum $q_c = 1ms$, a slot quantum $q_s = 0.5ms$, a slot overhead $o_s = 0.1ms$, and a cycle overhead $o_c = 0.1ms$.

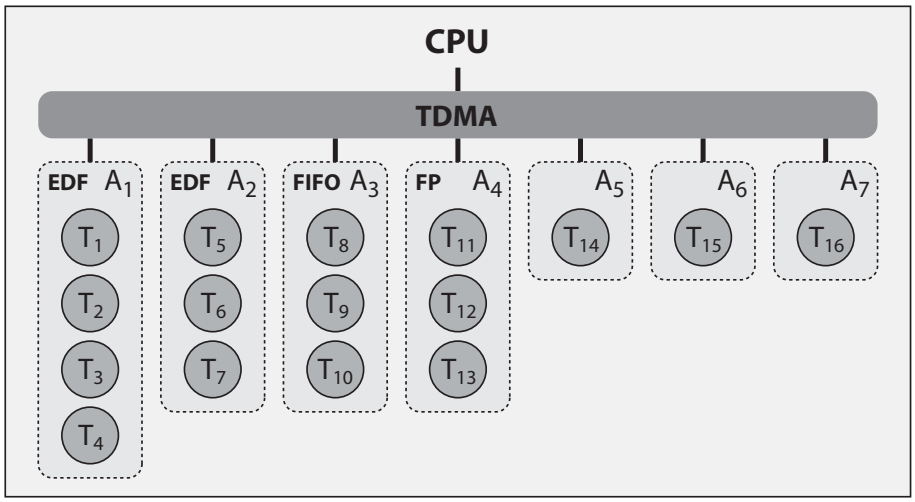


Fig. 69: Real-time system with 7 applications that are hierarchically scheduled using a TDMA-scheduler.

Using binary search, we first compute the minimum required processor speed, as described in Section 6.1.2.3. From this we learn, that a minimum processor speed of $f_{CPU} = 476.2MHz$ is required. With this processor speed, feasible TDMA settings exist for the two cycle lengths $\bar{c} = 49ms$ and $\bar{c} = 50ms$, and lead to a total utilization of $\bar{\sigma}_{tot} = 0.9959$, and $\bar{\sigma}_{tot} = 0.996$, respectively.

Next, suppose we use a processor with a frequency $f_{CPU} = 600MHz$, and we want to optimize slot and cycle lengths according to Section 6.1.2.2, such that 5 additional applications could be added at a later point of time.

		T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8
p	[ms]	196	245	105	147	231	308	275	234
j	[ms]	387	70	269	387	222	260	91	387
d	[ms]	48	-	58	17	65	-	-	48
e	[Mc]	7	4	6	1	8	2	3	5
D	[ms]	176	237	115	488	206	311	275	207
		T_9	T_{10}	T_{11}	T_{12}	T_{13}	T_{14}	T_{15}	T_{16}
p	[ms]	273	182	153	357	476	302	258	424
j	[ms]	70	269	80	70	177	967	719	257
d	[ms]	-	58	-	-	-	27	89	-
e	[Mc]	5	3	2	2	3	2	2	4
D	[ms]	178	198	153	423	556	511	371	315

Tab. 6: Specification of the 16 tasks in the hierarchically scheduled system. Note, Mc= 1*E*6 cycles.

For this we compute (6.8) up to $\bar{c}_{max} = 208ms$. The results are shown in Figure 70 and suggest to use a cycle length of $\bar{c} = 70ms$. This leads to a maximum remaining average bandwidth of $\bar{\sigma}_r = 0.1786$.

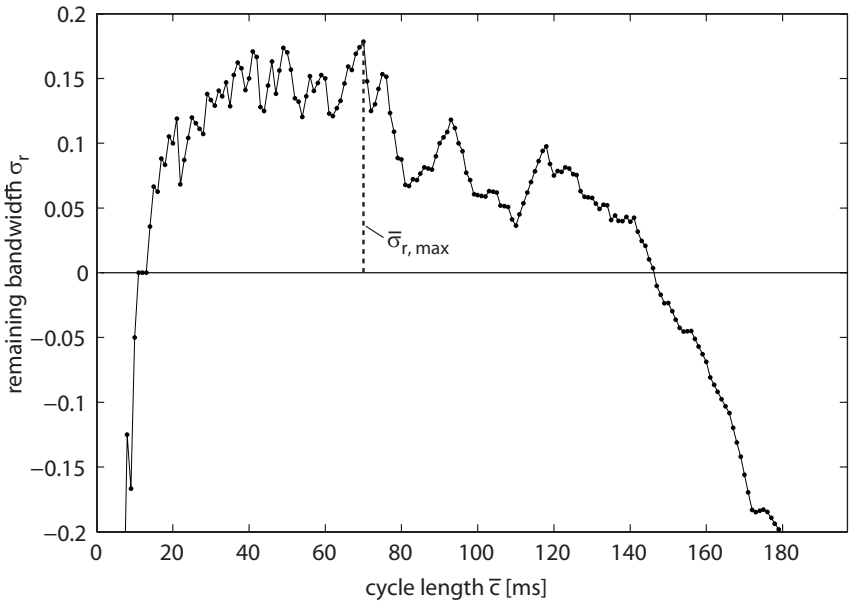


Fig. 70: Remaining average bandwidth in the hierarchically scheduled system.

6.1.4 Related Work

The problem of assigning optimal parameters to a TDMA scheduler was already studied extensively in the past, see e. g. [Inu79]. However, most research on this area concentrates on strictly periodic loads, and on TDMA protocols employed on communication resources. The problem of assigning optimal parameters for TDMA schedulers that are employed in hierarchically scheduled systems on the other hand, is not considered in most previous work.

In purely time-triggered systems, an optimal communication schedule that defines slot and cycle lengths can be constructed at design-time [Kop97], but in reality heuristics are often used to find a valid communication schedule due to the computational complexity of finding the optimal schedule.

Many large distributed embedded systems are however not anymore designed as purely time-triggered systems, but contain instead mixed time- and event-triggered components. Be it because of the coexistence of time- and event-triggered subsystems (clusters) that are connected with each other by bridges, as considered by Pop et al. in [PEP+04], or be it because of the existence of some event-triggered ECU's, as considered for example by Obermaisser in [Obe02].

When we get to such mixed time- and event-triggered systems, parameter selection for TDMA resources gets even more challenging. In [Obe05], Obermaisser proposes to choose slot lengths as a fraction of a fixed cycle length, such that every service consumer with event-triggered load receives an individual total bandwidth from the TDMA resource. While this method can be used for systems with non-real-time event-triggered loads, it is only applicable by trial-and-error for systems with real-time event-triggered loads, i. e. loads with deadline constraints. For such systems, Eles et al. [EDPP00] present a heuristic to assign slot lengths of a TDMA resource.

A method for slot as well as cycle length optimization based on evolutionary search techniques was presented by Hamann et al. in [HE05]. This method can be used to parameterize slot and cycle lengths of TDMA resources with fixed bandwidth, and it can handle real-time event-triggered loads with jitter and bursts. Since the method is based on evolutionary algorithms, it is however computationally expensive, and cannot guarantee a global optimal solution for a predetermined optimality criterion.

6.2 Polling Servers

Server-based scheduling is widely used in the area of real-time system design. Traditionally, it is mostly employed on systems with mixed hard real-time, as well as soft or non real-time applications, to schedule the soft or non real-time applications. Server based scheduling thereby typically improves the reactivity of these applications. However, server based scheduling can also be employed to enable hierarchical scheduling of hard real-time applications. Same as TDMA, server-based scheduling also supports temporal composability. However, in contrast to TDMA, server-based scheduling is typically more flexible, and often allows to exploit the available resources more efficiently, since a server only claims resources if there is workload ready to be processed.

In the area of classical real-time scheduling theory, a large range of different server algorithms were proposed, that typically trade off performance, computational complexity, memory requirement and implementation complexity. Some of the more prominent fixed priority servers include the polling server [LSS+87, SSL89], the deferrable server [LSS+87, SLS95], and the sporadic server [SSL89], but there exist also many others, such as for example various slack stealing servers [LRT92, RTL93, DTB93, TLS96], or the total bandwidth server [SB94, SB96], which is an example of a dynamic priority server. In [But97], Buttazzo presents an extensive overview of these and other server algorithms.

In this section, we concentrate on polling servers. A polling server can be thought of as a periodic task $T(p, e)$. When the task T is selected to run by the scheduler, it checks whether workload is waiting to be processed by the server. If yes, the server will provide e resources to process the waiting workload. But if no work is available for the server, the task will immediately finish, i. e. the server will not check for arriving work anymore until the next period starts.

6.2.1 Performance Analysis

From an applications point of view, polling servers can be considered as resources that deliver a service with a service guarantee β^G_{PS}. Performance metrics of the application, such as delay guarantees or buffer requirements, can thus be analyzed seamlessly within the MPA framework, using for example (2.11) or (2.12). Following, we will introduce abstract components for dynamic, as well as static polling servers.

6.2.1.1 Dynamic Polling Servers for EDF Scheduling

A polling server for dynamic scheduling (DPS) can be implemented by a periodic task $T(p, e)$ with an associated deadline $d = p$. Within the MPA framework, an abstract dynamic polling server for can be modeled by a component as depicted in Figure 71(a), with the corresponding real-time interface depicted in Figure 71(b). For an EDF component, a polling server

Fig. 71: (a) An abstract dynamic polling server component, and (b) its Real-Time Interface.

behaves like an arriving event stream with arrival curve $\alpha(\Delta) = \lceil \Delta/p \rceil \cdot e$ and a relative deadline $d = p$, and to components connected to the service output, the server is a resource that provides a resource supply of e during every interval of length p. We have however to consider two specialties. Firstly, a server might not provide any resources during one full period. This is the case if workload arrives just after the server task was selected to run by the scheduler, and was finished because there was no workload waiting to be processed. And secondly, under EDF scheduling it is hard to determine when exactly the server task receives the e resources within a period, and the DPS can therefore only guarantee to supply e resources at the end of every period p. We therefore establish the set of interface relations in a DPS as

$$\alpha^G = \left\lceil \frac{\Delta}{p} \right\rceil e \tag{6.12}$$

$$d^G = p \tag{6.13}$$

$$\beta^G_{DPS} = \max\left\{0, \left\lfloor \frac{\Delta - p}{p} \right\rfloor e\right\} \tag{6.14}$$

6.2.1.2 Static Polling Servers for FP Scheduling

A polling server for static scheduling (SPS) can also be implemented by a periodic task $T(p, e)$ with an associated maximum allowable delay $d = p$. Within the MPA framework, an abstract static polling server can then be modeled by a component as depicted in Figure 72(a), with the

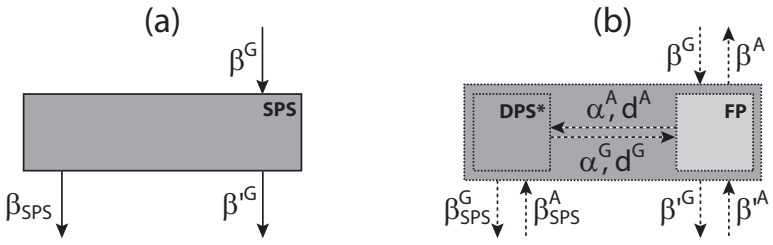


Fig. 72: (a) An abstract static polling server component, and (b) its Real-Time Interface.

corresponding real-time interface depicted in Figure 72(b). Internally, the abstract static polling server is built up using a processing component for FP scheduling, and a slightly modified version of an abstract dynamic polling server. In difference to a dynamic polling server, it is simpler to determine when the server task receives the e resources at the latest within a period. For an SPS we can therefore establish the improved service guarantee β^G_{PS}

$$\beta^G_{SPS} = \max\left\{0, \left\lfloor \frac{\Delta - p}{p} \right\rfloor e + \min\left\{e, \hat{\beta}^G\left(\Delta - \left\lfloor \frac{\Delta}{p} \right\rfloor p\right)\right\}\right\} \tag{6.15}$$

The interface relations for β'^G, β^A, α^A, d^A, α^G, and d^G on the other hand remain unchanged and are defined by (5.16), (5.17), (5.18), (5.19), (6.12), and (6.13), respectively.

6.2.1.3 Releasing Unused Server Capacity

Relation (6.12) validly upper bounds the workload α^G that polling servers generate on a system. But this bound is typically overly pessimistic, because polling servers will not claim resources if no workload is present to be processed by them. Hence, the workload $\alpha(\Delta)$ that a PS generates on a system is not only upper bounded by (6.12), but also by the sum the of the workloads $\alpha_i(\Delta + p)$ of all process components that are processed by the server and that are not hierarchically decoupled by another server

$$\alpha(\Delta) \leq \min\left\{\left\lceil \frac{\Delta}{p} \right\rceil e, \sum_{i \in PS} \alpha_i(\Delta + p)\right\} = \min\left\{\left\lceil \frac{\Delta}{p} \right\rceil e, \Sigma(\Delta + p)\right\} \tag{6.16}$$

To consider this new bound in the analysis of polling servers, we extend the service connections of all component interfaces with a new curve $\Sigma(\Delta)$, that represents the sum of all workloads $\alpha_i(\Delta)$ that are processed with the service $\beta(\Delta)$ that is available to a component, and that are not hierarchically decoupled by a server. The output guarantee and input

assumption on a connection then contain the additional bounds $\Sigma^G(\Delta)$ and $\Sigma^A(\Delta)$, respectively, with the relation $\Sigma^A(\Delta) \leq \Sigma^G(\Delta) \leq \Sigma(\Delta)$ that must be true for a compatible service connection of two interfaces.

To obtain the internal interface relation for this new bound Σ^G, we need to satisfy the resource constraint

$$\Sigma^G \leq \sup\left\{\Sigma \;:\; \hat{\alpha}^G(\Delta) = \min\left\{\left\lceil\frac{\Delta}{p}\right\rceil e, \Sigma(\Delta+p)\right\}\right\} \tag{6.17}$$

We can therefore establish the interface relations

$$\alpha^G = \min\left\{\left\lceil\frac{\Delta}{p}\right\rceil e, \Sigma^A\right\} \otimes \min\left\{\left\lceil\frac{\Delta}{p}\right\rceil e, \Sigma^A\right\} \tag{6.18}$$

$$\Sigma^G(\Delta+p) = \min^{-1}\left\{\alpha^A, \left\lceil\frac{\Delta-p}{p}\right\rceil e\right\} \tag{6.19}$$

with

$$\min^{-1}(\alpha_1,\alpha_2) = \alpha_1(\Delta+\lambda) \quad \text{for } \lambda = \inf\{\tau : \alpha_1(\Delta+\tau) \leq \alpha_2(\Delta+\tau)\} \tag{6.20}$$

And while (6.18) replaces (6.12) in the already established interface relations for polling servers, (6.19) complements the established interface relations.

Note, that since we extend the service connection of the abstract component interfaces with the new curve $\Sigma(\Delta)$, we need to establish interface relations for Σ'^G and Σ^A for all processing components of our component system. Further, we must also adapt the interface relations for α^A in the processing component interfaces. For the FP processing component interface, we get $\Sigma'^G = \Sigma^G - \alpha^G$ and $\Sigma^A = \Sigma'^A + \alpha^G$. And to get α^A we must now compute the minimum of (5.18) and the additional new bound $(\hat{\Sigma}^G - \hat{\Sigma}'^A)$. For the other process components, the extensions can be derived similarly.

6.2.2 Parameter Selection

Based on the explicit service and load guarantees and assumptions, and the explicit interface compatibility requirements (5.5) and (5.4) that guarantee schedulability, it is possible to determine valid pairs of server periods and corresponding server resource budgets for polling servers.

6.2.2.1 Minimum Resource Budget

The minimum resource budget for a given server period must be chosen, such that the service assumption of the application that is served by the polling server is satisfied. Following (5.5) we thus get the condition $\beta^G_{PS} \geq \beta^A_{PS}$. For a dynamic polling server with server period p and server

resource budget e, the server service guarantee is determined by (6.14), and when we construct the inverse of this condition with respect to e, we can compute the minimum required server resource budget

$$e^A_{min} = \sup_{\Delta \geq 2p} \left\{ \frac{\beta^A_{PS}}{\left\lfloor \frac{\Delta - p}{p} \right\rfloor} \right\} \qquad \text{if } \beta^A_{PS} = 0 \; \forall \Delta < 2p \tag{6.21}$$

The server is not feasible if the condition $\beta^A_{PS} = 0 \; \forall \Delta < 2p$ is not met.

For a static polling server, the server service guarantee is determined by (6.15). In principle it is again possible to construct the inverse of this more complex service guarantee with respect to e, to compute the minimum required server resource budget for a static polling server. However, (6.21) is often simpler to compute, and also leads to a sufficient, although not necessary bound for static polling servers.

6.2.2.2 Maximum Resource Budget

The maximum resource budget for a given server period must be chosen, such that the load assumption of the resource on which the polling server is running is satisfied. Following (5.4) we get the condition $\alpha^G_{PS} \leq \alpha^A_{PS}$. From (6.18), and knowing that $f \otimes f \leq f$, we can establish the sufficient, although not necessary condition

$$\left\lceil \frac{\Delta}{p} \right\rceil e \leq \alpha^A_{PS} \qquad \forall \Delta : \alpha^A_{PS}(\Delta) < \Sigma^A(\Delta) \tag{6.22}$$

Constructing the inverse of this condition with respect to e leads to the lower bound to the maximum allowable server resource budget

$$e^A_{max} = \inf_{\forall \Delta \, : \, \alpha^A_{PS}(\Delta) < \Sigma^A(\Delta)} \left\{ \frac{\alpha^A_{PS}}{\left\lceil \frac{\Delta}{p} \right\rceil} \right\} \tag{6.23}$$

6.2.3 Experimental Results

Consider a real-time system with a complex mix of hierarchically arranged static and dynamic scheduling. The system consists of a set of 10 real-time tasks T_1–T_{10} that are running on a single processor under the scheduling hierarchy policies depicted in Figure 73. The scheduling hierarchy uses a mix of FP, RM, and EDF scheduling, as well as a polling server for static scheduling ($p_{SPS} = 4, e_{SPS} = 1$) and a polling server for dynamic scheduling, for which we want to find valid parameters. The processor is fully available to process the 10 tasks, and all tasks are fully preemptable and independent from each other. The tasks are specified in Table 7 with a

period p, a jitter j, a minimum inter-arrival distance d, a relative deadline d, and a worst-case execution time e. The task set results in a total average processor utilization of 98.5%.

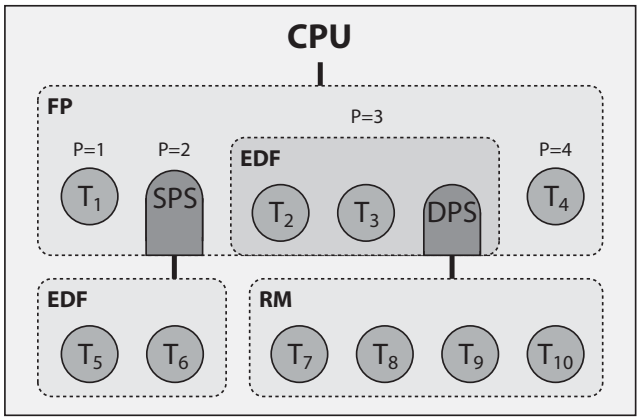


Fig. 73: Hierarchical scheduling scheme of a real-time system with a complex mix of hierarchically arranged static and dynamic scheduling policies.

	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8	T_9	T_{10}
p [ms]	5	10	25	100	25	20	12	16	20	30
j [ms]	0	5	0	2	80	0	0	0	0	0
d [ms]	-	-	-	-	5	-	-	-	-	-
e [ms]	0.2	1	1.5	40	2	2	0.5	0.75	1	2
D [ms]	2	10	15	150	50	30	12	16	20	30

Tab. 7: Specification of the 10 tasks in the hierarchically scheduled system.

To determine a valid pair of server parameters (p_{DPS}, e_{DPS}) for the dynamic polling server in this system, we first build the interface model of the system, as depicted in Figure 74. We then use (6.21) to compute the minimum required server resource budget $e^A_{DPS,min}(p_{DPS})$ for all server periods $p_{DPS} \in [0.1ms \ldots 7ms]$ in steps of $0.1ms$. The result is depicted in Figure 75, and any parameter pair (p_{DPS}, e_{DPS}) in the area above the curve $e^A_{DPS,min}$ results in a dynamic polling server that satisfies the service requirement (5.5). Further, we use (6.23) to compute the maximum allowed server resource budget $e^A_{DPS,max}(p_{DPS})$ for all server periods $p_{DPS} \in [0.1ms \ldots 7ms]$ in steps of $0.1ms$. The result is again depicted in Figure 75, and any parameter pair (p_{DPS}, e_{DPS}) in the area below the curve $e^A_{DPS,max}$ results in a dynamic polling server that satisfies the load requirement (5.4). Consequently, any parameter pair (p_{DPS}, e_{DPS}) that lies both, above $e^A_{DPS,min}$, and

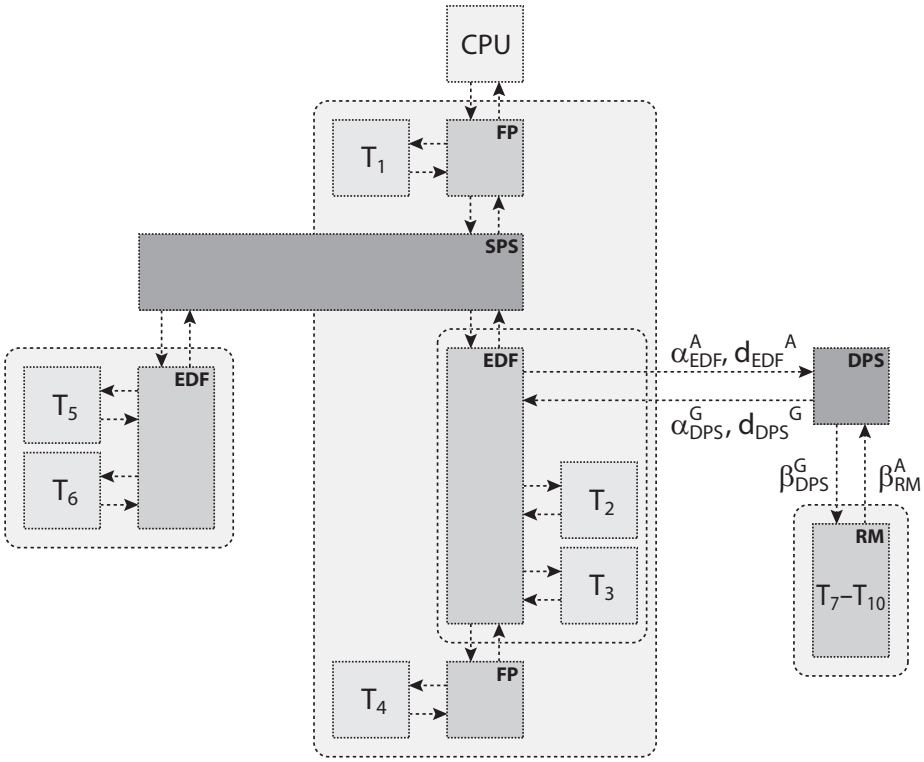


Fig. 74: Real-Time Interface model of the system with the scheduling hierarchy depicted in Figure 73.

below $e^A_{DPS,max}$, results in a dynamic polling server that satisfies both, the service requirement (5.5), as well as the load requirement (5.4).

From the area of valid server parameter pairs, let us choose $p_{DPS} = 3ms$, and $e_{DPS} = 1ms$, the corresponding point is indicated in Figure 75. For this server parameter pair, Figure 76 depicts the service assumption β^A_{RM} of the four tasks T_7–T_{10}, together with the service guarantee β^G_{DPS} of the dynamic polling server, and Figure 77 depicts the load assumption α^A_{EDF} of the EDF scheduled component, together with the load guarantee α^G_{DPS} of the dynamic polling server.

In recent work on hierarchical scheduling, hierarchy is typically achieved by servers that are implemented as simple periodic tasks with period p_{PS} and execution time e_{PS}, see e. g. [SL03] or [SL04]. Such an implementation would generate a load with a load guarantee as defined by (6.12). In Figure 77, $\alpha^G_{DPS,full}$ depicts this arrival guarantee, and as we see, this guarantee does not comply with the arrival assumption α^A_{EDF} of the EDF scheduled component. Therefore, such a server implementation would render the system to be not schedulable. However, when we consider unused server capacity that a real polling server would release, the

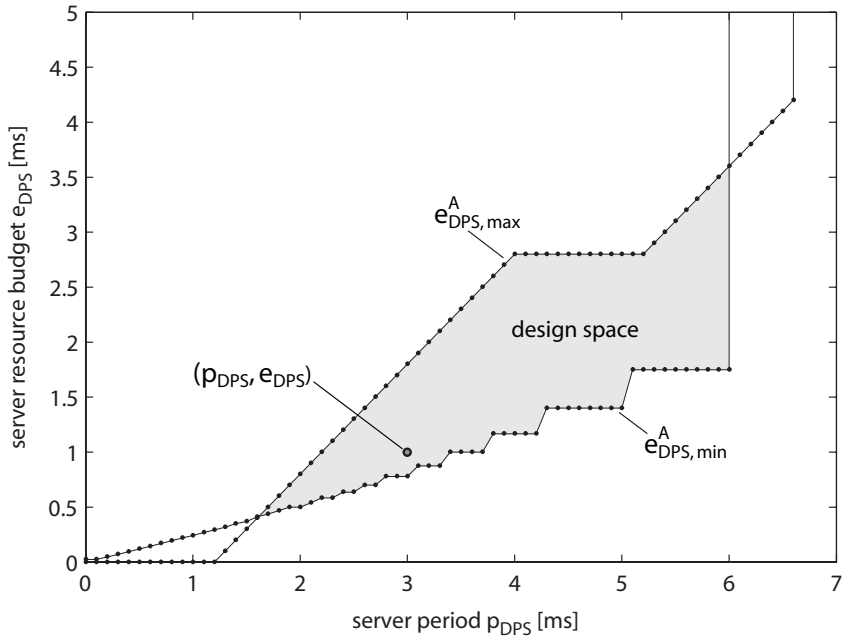


Fig. 75: The minimum required server resource budget $e^A_{DPS,min}(p_{DPS})$, and the maximum allowed server resource budget $e^A_{DPS,max}(p_{DPS})$. The grey area depicts the resulting design space of valid server parameter pairs.

system is in fact schedulable, as we can see since $\alpha^G_{DPS} \leq \alpha^A_{EDF}$ in Figure 77.

6.2.4 Related Work

In the area of performance analysis of systems with server-based scheduling, Kuo and Li [KL99] first introduced analysis of hierarchical fixed priority scheduling, building upon the work of Deng and Liu [DL97]. Kuo and Li consider the use of sporadic servers to execute applications, and use the techniques of Liu and Layland [LL73] to provide a simple utilisation based schedulability test. This test is however only applicable if each server period in the system is the greatest common divisor (gcd) or a divisor of the gcd of all the tasks in an application.

This limitation is overcome by Seawong et al. [SRLK02] that introduce a response time analysis for hierarchical systems using deferrable servers or sporadic servers to schedule a set of hard real-time applications. The analysis of Seawong et al. assumes that the capacity of a server is always made available at the end of its period. This leads to a sufficient but not necessary schedulability analysis.

Davis and Burns [DB05] address this problem and claim to introduce

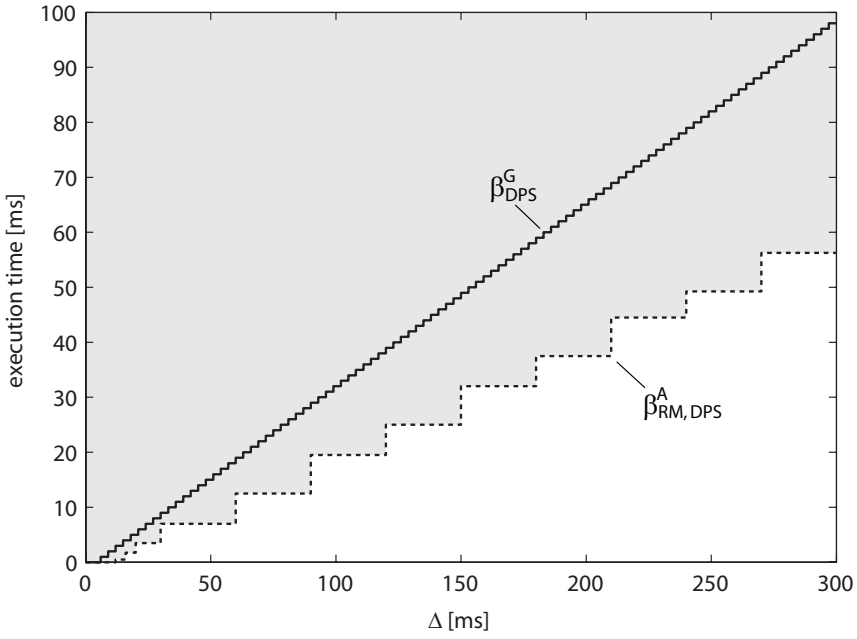


Fig. 76: The service assumption β^A_{RM} of the four tasks T_7–T_{10}, together with the service guarantee β^G_{DPS} of the dynamic polling server with $p_{DPS} = 3ms$, and $e_{DPS} = 1ms$.

an exact schedulability test for hierarchical systems with polling servers, deferrable servers, or sporadic servers. In their work, Davis and Burns come to the conclusion that the simple polling server outperforms both the sporadic and the deferrable server when the metric is application task schedulability. However, the response time analysis of Davis and Burns is only exact under the constraint that an idle task or a set of non real-time background tasks exist in every application, to consume the server capacity if no hard real-time tasks are present to be processed. But if every application contains an idle task that executes at a background priority level when all other tasks are inactive, then the choice of server algorithms has no influence, since both, the deferrable and sporadic server will behave exactly as the periodic server. That is, since there is always an idle task to be processed, the capacity of the deferrable server will never be deferred, and the sporadic server will always use its full capacity for the idle task if no other tasks are waiting.

Lipari and Bini [BL03] present a response time analysis that is more similar to the work presented in this thesis. They represent the service provided by a server as a so-called characteristic function, which is equivalent to a service curve. Based on this characteristic function, Lipari and Bini investigate a server that provides the service of a periodic resource

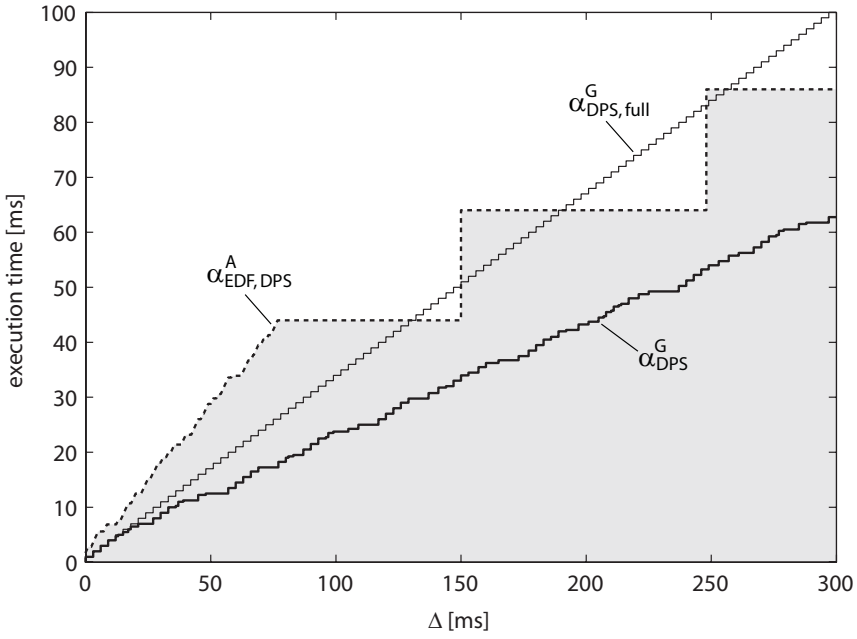


Fig. 77: The load assumption α^A_{EDF} of the EDF component, together with the load guarantee α^G_{DPS} of the dynamic polling server with $p_{DPS} = 3ms$, and $e_{DPS} = 1ms$. $\alpha^G_{DPS,full}$ denotes the arrival guarantee if the release of unused server capacity is not considered.

model. This model also assumes that the capacity of a server is always made available at the end of its period. Lipari and Bini also investigate the problem of server parameter selection. For this they approximate the service provided by their periodic server with a bounded delay resource model, and the delay and the speed factor are then used to determine the delay and capacity of their periodic server. The presented method is however overly pessimistic. First, because it cannot consider that different tasks in an application may have different deadlines, and secondly because it relies on a very pessimistic delay analysis. Instead of using a bound similar to (2.11), Lipari and Bini compute the maximum delay in their server schedulability analysis as $d_{max} \leq \inf\{\tau \geq 0 : \alpha^u(\tau) \leq \beta^l(\tau)\}$.

Almeida et al. [Alm03, AP04] builds on the work of Lipari and Bini, and introduces a response time analysis for a server with jitter, thus allowing a limited generalization of the server model of Lipari and Bini. Almeida et al. also investigate the problem of server parameter selection, and propose binary search to determine the capacity of a periodic server. However, the method presented by Almeida et al. suffers from the same pessimism as the method of Lipari and Bini.

The methods proposed by Shin and Lee [SL03, SL04] finally overcome this pessimism of the previous two methods, primarily by introducing the concept of demand bound functions to represent the workload demand of the tasks in an application, but also by using the correct delay bound (2.11) in their server schedulability analysis. The work of Shin and Lee is however also confined to servers with a periodic or a bounded delay resource model.

6.3 Discussion

Service assumption curves $\beta^A(\Delta)$ prove to be a very powerful model within the analysis of real-time systems with hierarchical scheduling, as they allow to precisely capture the scheduling information of a complete application. And together with the explicit service guarantees $\beta^G(\Delta)$ of the various servers, and with the exact schedulability condition $\beta^A(\Delta) \leq \beta^G(\Delta)\ \forall\Delta$, they also build the basis for the server parameter selection methods presented in this section.

Compared to the related performance analysis methods for hierarchical systems that were discussed in the previous section, the performance analysis method introduced in this thesis is the first to consider the release of unused server capacity for the schedulability analysis of systems with polling servers. This allows to correctly classify systems as schedulable that would otherwise be classified incorrectly as not schedulable. A designer must however consider that incorporating unused server capacity within the schedulability analysis threatens the temporal composability of hierarchically decoupled applications, since resource access of the different applications is not clearly separated anymore. But once a designer is aware of this fact, he is free to choose whether or not to incorporate the release of unused server capacity in the schedulability analysis. But not only service assumption curves $\beta^A(\Delta)$ prove to be useful , but also load assumption curves $\alpha^A(\Delta)$. These allow to compute a maximum capacity bound for polling servers with a given period, incorporating the schedulability constraints of the complete system. None of the discussed related methods have means to compute this maximum capacity bound, instead they all only compute the minimum capacity bound.

In this section, we introduced performance analysis and parameter selection methods for systems with hierarchical scheduling with TDMA, and with dynamic as well as static polling servers. The underlying principles to these methods are however not confined to these specific server instances. Instead it is in principle possible to extend the presented methods to other server types by defining appropriate abstract components, their interfaces, and the corresponding internal relations.

Part III

Tool Support

7

Efficient Computation of Real-Time Calculus

Most methods presented in this thesis employ real-time calculus to perform computations on arrival curves, service curves, or other similar curves, often collectively referenced to as so-called variability characterization curves (VCCs). While real-time calculus provides compact mathematical representations for all the different curve operations, their practical computation is typically more involved. The major problem arises from the fact, that the various VCCs are defined for the infinite range of positive real numbers $\Delta \in \mathbb{R}_{\geq 0}$. However, for practical computation we require that the VCCs have a finite representation, and that applying any of the curve operations leads to a result in a finite time. Moreover, we require that the resulting VCC is again defined for the infinite range of positive real numbers $\Delta \in \mathbb{R}_{\geq 0}$, but that it nevertheless has again a finite representation. To overcome this problem, VCCs are often described by a finite set of piecewise linear curve segments. However, many arrival and service curves that represent practical event stream models or resource models require an infinite set of piecewise linear curve segments to be represented. Examples are the arrival curves to represent a periodic event stream as depicted in Figure 5(a), or the service curves to represent a periodic resource model as depicted in Figure 6(d). And although all these curves can be safely approximated by a finite set of piecewise linear curve segments, as for example depicted in Figure 78, this is in general not desired, as this approximation leads to overly pessimistic performance analysis results, as can also be seen in Figure 78.

This chapter introduces a compact representation for special classes of

variability characterization curves that are either defined by a finite or an infinite set of piecewise linear curve segments, and it presents methods to compute various curve operations on these compact curve descriptions. In the following Section 7.1, a classification for VCCs is introduced, and the classes of VCCs that can be represented by the introduced curve description are identified. Further, it is shown that these classes of VCCs cover a large part of the VCCs that are or practical relevance in the area of modular performance analysis and interface-based design for embedded real-time systems. In Section 7.2, the compact representation for VCCs is then introduced, and Section 7.3 presents methods to compute various curve operations on these compact curve descriptions. The chapter concludes with a discussion in Section 7.4.

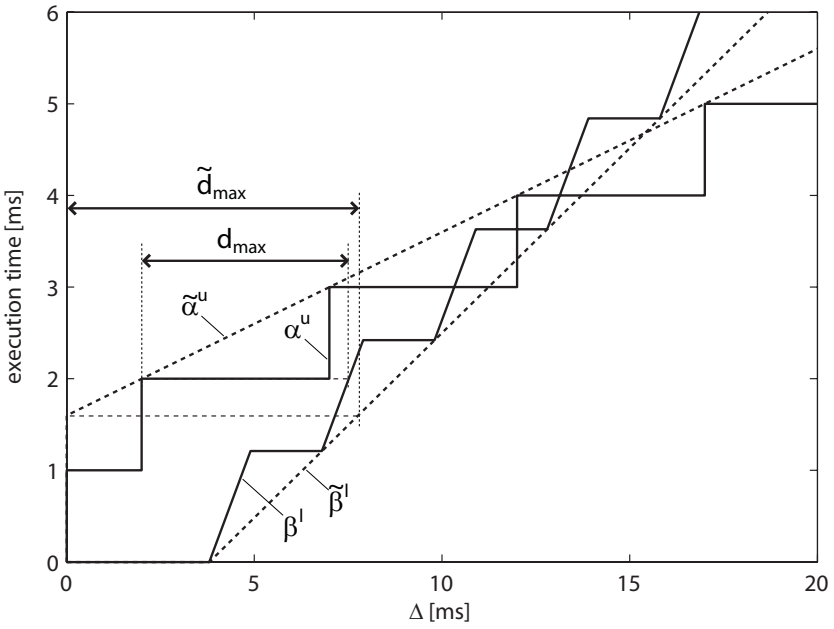


Fig. 78: The upper arrival curve α^u of a periodic event stream with jitter, and the lower service curve β^l of a periodic service. Following (2.11), the maximum horizontal distance d_{max} denotes the maximum delay experienced by an event on the event stream, when processed by a greedy processing component on a resource with the periodic service β^l. $\tilde{\alpha}^u$ and $\tilde{\beta}^l$ depict the finite piecewise linear approximation of α^u and β^l, and $\tilde{d}_{max}$ depicts the maximum horizontal distance between these approximated curves. In this example, $\tilde{d}_{max}$ overestimates d_{max} by 40.8%.

7.1 Classification of VCCs

The general concept of variability characterization curves was first introduced by Maxiaguine et al. in [MZCW04], to collectively define a special class of functions, of which also arrival curves, service curves, or workload curves, are instances.

7.1.1 Variability Characterization Curves

Variability characterization curves allow to precisely quantify best-case and worst-case variability on wide-sense increasing functions.

Def. 19: **(Variability Characterization Curve)** *For a given wide-sense increasing function r, and let $r[s,t)$ denote the increase of r in the interval from s to t, including s, and excluding t, for all $s,t \in \mathbb{R}$ with $s \leq t$. Upper and lower variability characterization curves (VCCs) of r are defined as curves $\rho^u(\Delta)$ and $\rho^l(\Delta)$, where $r[s,t)$, $\rho^u(\Delta)$, and $\rho^l(\Delta)$ are related to each other by the following inequality*

$$\rho^l(t-s) \leq r[s,t) \leq \rho^u(t-s) \quad \forall s,t \in \mathbb{R}, s \leq t \tag{7.1}$$

with $\rho^u(0) = \rho^l(0) = 0$.

Note that this definition is more general than the original definition of variability characterization curves by Maxiaguine et al. in [MZCW04]. In the original definition, the interval length $(t-s)$ is restricted to natural numbers $(t-s) \in \mathbb{Z}_{\geq 0}$, while the above definition allows $(t-s) \in \mathbb{R}_{\geq 0}$. From this also follows directly that VCCs following the above definition are defined on the full range of positive real numbers $\Delta \in \mathbb{R}_{\geq 0}$.

To realize the relation of VCCs to arrival and service curves, suppose that $r[s,t)$ denotes the number of events that arrive on an event stream in the time interval from s to t. The corresponding variability characterization curves $\rho^u(\Delta)$ and $\rho^l(\Delta)$ are then arrival curves of this event stream. Likewise, if $r[s,t)$ denotes the amount of available service on a resource in the time interval from s to t, then the corresponding variability characterization curves $\rho^u(\Delta)$ and $\rho^l(\Delta)$ are service curves of this resource.

7.1.2 Classification Scheme

To clearly define the set of VCCs that can be handled with the models and methods presented in this chapter, we propose a hierarchical classification scheme for VCCs as depicted in Figure 79.

At the top level of this classification scheme, we distinguish between *piecewise linear VCCs* and *not piecewise linear VCCs*. For the remainder of this chapter, we only focus on piecewise linear VCCs. The class of piecewise linear VCCs can further be divided into *finite VCCs* that consist of a

finite set of linear pieces, and into *infinite VCCs* that consist of an infinite set of linear pieces. We further divide this last class into three subclasses. *Periodic piecewise linear VCCs* consist of a finite set of linear pieces that is repeated periodically with a constant offset between consecutive repetitions. *Regular piecewise linear VCCs* on the other hand consist of a finite set of linear pieces that is eventually followed by a second finite set of linear pieces that is repeated periodically, again with a constant offset between consecutive repetitions. And finally, *irregular piecewise linear VCCs* consist of an infinite set of linear pieces without any regular periodicity.

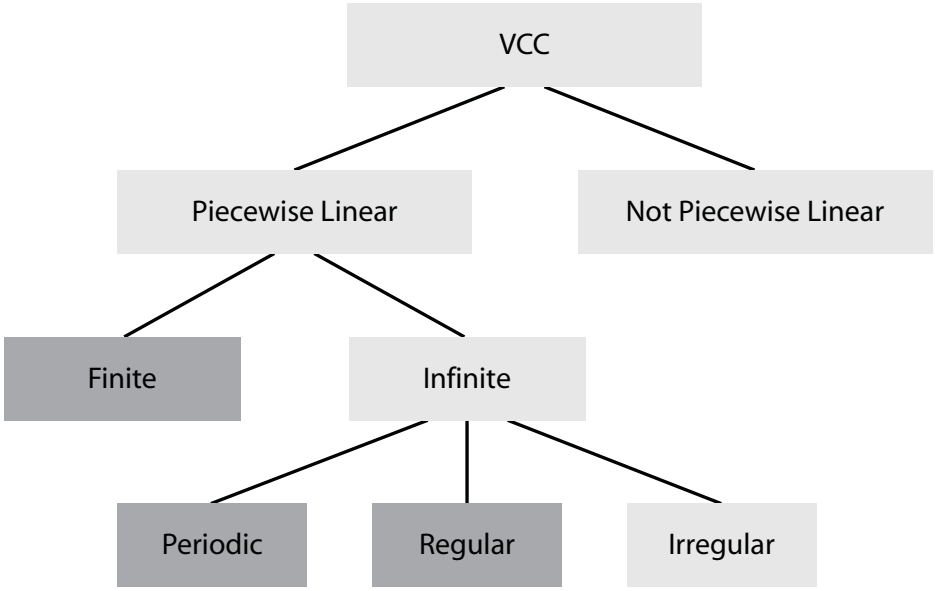


Fig. 79: Classification scheme for VCCs. The dark grey classes can be handled by the models and method presented in this chapter.

The models and methods presented in this chapter allow to handle the classes of finite piecewise linear VCCs, periodic piecewise linear VCCs, and regular piecewise linear VCCs. Examples of VCCs of these three classes are depicted in Figure 80(a), Figure 80(b), and Figure 80(c), respectively.

7.1.3 Practically Relevant Classes of VCCs

For the following discussion, we focus on VCCs that represent arrival curves or service curves, as these are the most relevant VCCs within the MPA framework.

Let us first investigate, whether practically relevant arrival and service curves are piecewise linear or not. In the case of arrival curves, $r[s,t)$

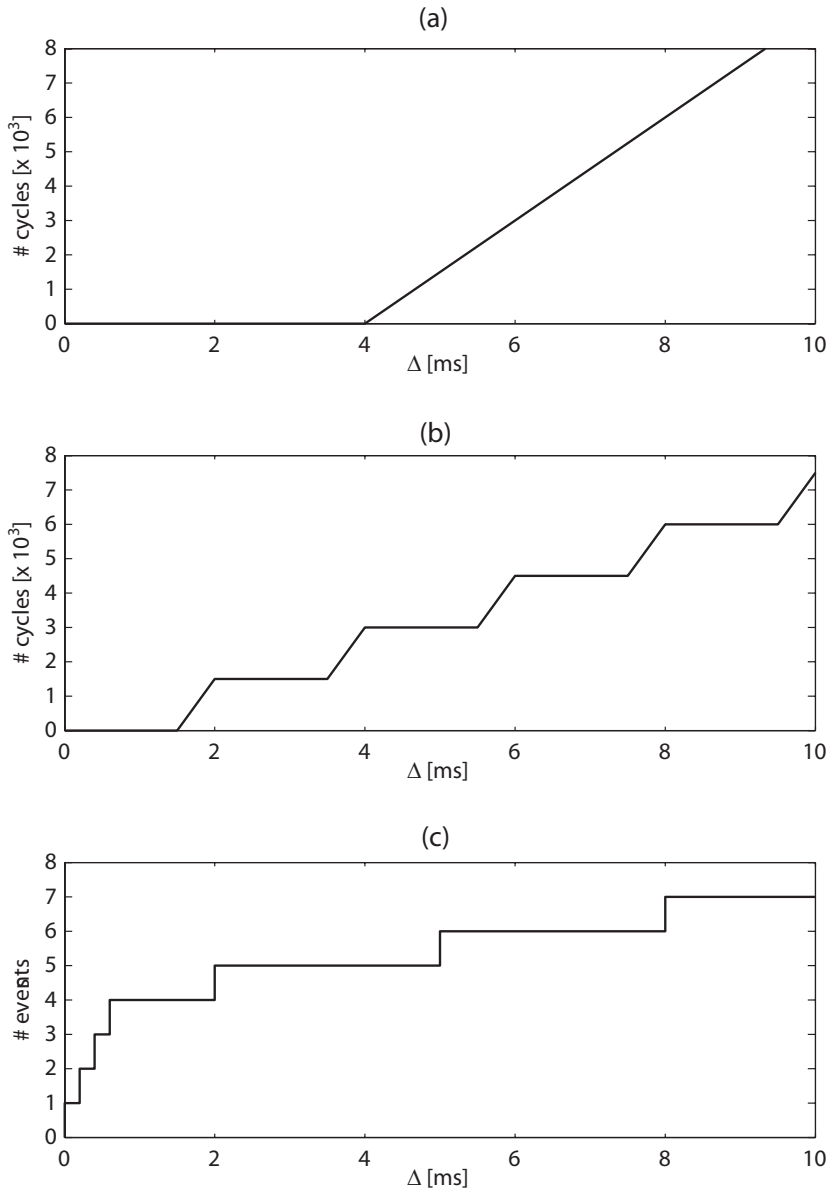


Fig. 80: (a) A finite piecewise linear VCC. (b) A periodic piecewise linear VCC. (c) A regular piecewise linear VCC. The curves are specified in Example 11.

denotes the number of events that arrive on an event stream. And since events are atomic units, it follows that $r[s,t) \in \mathbb{Z}_{\geq 0}$. Consequently, the resulting arrival curves are piecewise constant, and hence piecewise linear. In principle, the same line of thoughts can be made for service curves, as the basic unit of resource availability is typically also atomic, be it for example a clock cycle on a CPU, or a transmittable bit on a communication

line. However, since these atomic units are often too fine-grained for a practical analysis, fluid models are often used to describe resource availability. To then obtain piecewise linear service curves, we must require that the fluid resource availability is piecewise constant over time. Computation or communication resources typically fulfill this requirement. Now that we derived that practically relevant arrival and service curves are piecewise linear it has also to be noted that any not piecewise linear VCC could be safely approximated by a piecewise linear VCC.

Next let us investigate arrival curves. To analyze the timing properties of a hard real-time system, a deterministic timing specification of the arriving event streams for which the timing requirements must be satisfied is mandatory. Most results in the area of real-time scheduling are derived for event streams that are either periodic or sporadic, possibly containing some jitter. These timing specifications can all be captured by the common set of standard event arrival patterns described in Example 1 in Chapter 2, that specifies an event stream with a parameter triple (p, j, d). To obtain corresponding upper arrival curves (2.3) can be used, while lower arrival curves are either zero for sporadic events streams, or can otherwise be obtained from (2.2). From inspection of (2.2) and (2.3) follows that the arrival curves of a periodic event stream are periodic piecewise linear VCCs, while the arrival curves of a periodic event stream with jitter or burst are regular piecewise linear VCCs. Moreover, to our best knowledge there exists no practically relevant deterministic timing specification that can not be captured by arrival curves that are either finite, periodic, or regular piecewise linear VCCs.

To analyze a hard real-time system, not only requires a deterministic timing specification of the arriving event streams, but also of the resource availability over time. Most results in the area of real-time scheduling assume that a resource is fully available to the task set that must be scheduled. But to enable schedulability analysis of hierarchically scheduled systems, also other resource availability models were proposed, most notably the bounded delay resource model [MFC01], and the periodic resource model [SL03]. The corresponding service curves are depicted in Figure 6, and as we see there, the service curves for a fully available resource as well as for a bounded delay resource are finite piecewise linear VCCs, while the service curves for a periodic resource are regular piecewise linear VCCs. Again, there exists to our best knowledge no practically relevant deterministic resource availability specification that can not be captured by service curves that are either finite, periodic, or regular piecewise linear VCCs.

Finally, let us investigate arrival and service curves that are used to analyze the timing properties of soft real-time systems. For these systems, it is often convenient to specify the timing of event streams and

the resource availability by simulation or measurement traces. Using a sliding window approach, arrival and service curves can then be generated from these traces. However, since any simulation or measurement trace is of finite length, arrival and service curves could only be determined for intervals up to the trace length. To overcome this problem, the traces are artificially extended to infinite length, by periodic repetition. Consequently, the corresponding arrival or service curves are periodic piecewise linear VCCs.

After ensuring that the VCCs at the input of a performance model within the MPA framework are finite, periodic, or regular piecewise linear VCCs, we must also ensure that the VCCs within the performance model are finite, periodic, or regular piecewise linear VCCs. That is, we must ensure that any curve operation with finite, periodic or regular piecewise linear VCCs as operands yields again to a finite, periodic or regular piecewise linear VCC as result. In Section 7.3 we will see that this is the case. The resulting curve will thereby typically either be finite, or it will have a periodic behavior with a period equalling either of the operands periods, or equalling the hyperperiod of the operands periods.

7.2 A Compact Representation for VCCs

To compactly describe finite, periodic, and regular piecewise linear VCCs, we first introduce the concept of curve segment sequences. VCCs are then represented by either one or two curve segment sequences, accompanied by a set of additional parameters.

7.2.1 Curve Segment Sequences

A single linear segment of a piecewise linear curve can conveniently be described by a parameter triple.

Def. 20: **(Curve Segment)** *A curve segment is a triple* $\sigma = \langle x, y, s \rangle$ *with* $x \in \mathbb{R}_{\geq 0}$ *and* $y, s \in \mathbb{R}$, *that specifies a straight line in the cartesian coordinate system that passes through the point* (x, y) *and has a slope* s.

A sequence of piecewise linear segments of a piecewise linear curve can then be described by a sequence of curve segments.

Def. 21: **(Curve Segment Sequence)** *A curve segment sequence* Σ *is an ordered list of curve segments*

$$\Sigma = \langle \langle x_1, y_1, s_1 \rangle, \langle x_2, y_2, s_2 \rangle, \ldots, \langle x_m, y_m, s_m \rangle \rangle \tag{7.2}$$

A curve segment sequence is called proper, if $x_i < x_{i+1}$ *holds for all* $1 \leq i < m$.

And from a curve segment sequence, we can then derive upper as well as lower piecewise linear VCCs. An upper piecewise linear VCC corresponding to a curve segment sequence Σ can thereby be determined by the operator $F^u_\Sigma(\Delta)$ defined as

$$F^u_\Sigma(\Delta) = \begin{cases} 0 & \text{if } 0 \le \Delta \le x_1 \\ y_i + ((x - x_i) * s_i) & \text{if } x_i < \Delta \le x_{i+1} \\ y_m + ((x - x_m) * s_m) & \text{if } x_m < \Delta \end{cases} \tag{7.3}$$

A lower piecewise linear VCC corresponding to a curve segment sequence Σ on the other hand can be determined by the operator $F^l_\Sigma(\Delta)$ defined as

$$F^l_\Sigma(\Delta) = \begin{cases} 0 & \text{if } 0 \le \Delta < x_1 \\ y_i + ((x - x_i) * s_i) & \text{if } x_i \le \Delta < x_{i+1} \\ y_m + ((x - x_m) * s_m) & \text{if } x_m \le \Delta \end{cases} \tag{7.4}$$

Next, let us define the concatenation operator $\sqcup$ for two curve segment sequences Σ_A and Σ_B as

$$\begin{aligned} \Sigma_A \sqcup \Sigma_B \;=\; & \Big\langle \langle x_{1,A}, y_{1,A}, s_{1,A} \rangle, \ldots, \langle x_{j,A}, y_{j,A}, s_{j,A} \rangle, \\ & \langle x_{1,B}, y_{1,B}, s_{1,B} \rangle, \ldots, \langle x_{m_B,B}, y_{m_B,B}, s_{m_B,B} \rangle \Big\rangle \end{aligned} \tag{7.5}$$

with

$$j = \max_{1 \le j \le m_A} \left\{ j : x_{j,A} < x_{1,B} \right\}$$

That is, $\Sigma_A \sqcup \Sigma_B$ consists of all curve segments of Σ_B, preceded by the curve segments of Σ_A that start at an x-coordinate $x_{A,j} < x_{B,1}$.

Finally, let us define the shift operator $\oplus$ for a curve segment sequence Σ and a shift vector (d_x, d_y) with $d_x, d_y \in \mathbb{R}$ as

$$\Sigma \oplus (d_x, d_y) = \Big\langle \langle x_1 + d_x, y_1 + d_y, s_1 \rangle, \ldots, \langle x_m + d_x, y_m + d_y, s_m \rangle \Big\rangle \tag{7.6}$$

7.2.2 Compact VCCs

Using the concept of curve segments and curve segment sequences, we can now define compact VCCs.

Def. 22: (Compact Variability Characterization Curve) *A compact VCC ν is a tuple*

$$\nu = \left\{ \Sigma_A, \Sigma_P, p_x, p_y, x_{p0}, y_{p0} \right\} \tag{7.7}$$

where Σ_A is a curve segment sequence describing a possibly existing irregular start sequence of a VCC, and Σ_P is a curve segment sequence describing a possibly existing regularly repeated sequence of a VCC. If Σ_P is not an empty curve segment sequence, then the regular part of the VCC is defined by the period p_x and the vertical offset p_y between two consecutive repetitions of Σ_P, and the first occurrence of the regular sequence Σ_P starts at (x_{p0}, y_{p0}).

In this compact VCC, we call Σ_A the aperiodic curve part of the VCC, and we call Σ_P the periodic curve part of the VCC. To derive the upper or lower VCC corresponding to a compact VCC ν, we first build a curve segment sequence Σ_ν defined as

$$\Sigma_\nu = \Sigma_A \sqcup \left\{ \bigsqcup_{i \in \mathbb{Z}_{\geq 0}} \Sigma_P \oplus (x_{p0} + ip_x, y_{p0} + ip_y) \right\} \tag{7.8}$$

and we then apply (7.3) and (7.4) to this possibly infinite curve segment sequence. When we apply (7.3) or (7.4) to a curve segment sequence Σ_ν of a compact VCC ν, we abbreviated write $F^u_\nu(\Delta)$ and $F^l_\nu(\Delta)$ in the following.

We can describe a finite piecewise linear VCC by a compact VCC ν with $\Sigma_P = \varnothing$ and $x_{A,1} = 0$. A periodic piecewise linear VCC can be described by a compact VCC ν with $\Sigma_A = \varnothing$, $x_{P,1} = 0$, and $x_{p0} = 0$. And a regular piecewise linear VCC can be described by a compact VCC ν with $x_{A,1} = 0$, $x_{P,1} = 0$, and $x_{p0} > 0$.

Ex. 11: *The lower service curve of a bounded delay resource with delay $d = 4ms$ and bandwidth $B = 1.5MHz$, depicted in Figure 80(a) can be described by the finite compact VCC ν_A defined as*

$$\nu_A = \left\{ \left\langle \langle 0,0,0\rangle, \langle 4,0,1.5\rangle \right\rangle, \varnothing, 0, 0, 0, 0 \right\}$$

The lower service curve of a TDMA resource with bandwidth $B = 3MHz$, cycle length $c = 2ms$, and slot length $s = 0.5ms$, depicted in Figure 80(b) on the other hand can be described by the periodic compact VCC ν_B defined as

$$\nu_B = \left\{ \varnothing, \left\langle \langle 0,0,0\rangle, \langle 1.5,0,3\rangle \right\rangle, 2, 1.5, 0, 0 \right\}$$

And the upper arrival curve of a periodic event stream with period $p = 3ms$, jitter $j = 10ms$ and minimum inter-arrival distance $d = 0.2ms$, depicted in Figure 80(c) can be described by the regular compact VCC ν_C defined as

$$\nu_C = \left\{ \left\langle \langle 0,1,0\rangle, \langle 0.2,2,0\rangle, \langle 0.4,3,0\rangle, \langle 0.6,4,0\rangle \right\rangle, \left\langle \langle 0,0,0\rangle \right\rangle, 3, 1, 2, 5 \right\}$$

7.3 Operations on Compact VCCs

To employ compact VCCs within the MPA framework, we must be able to compute various curve operations on these compact VCCs. We first introduce a general method to compute curve operations on compact VCCs, and we then show how to apply this method to compute a set of unary and binary operations on compact VCCs.

7.3.1 Method

When we closely investigate the various curve operators, we observe that the result of all operators, when applied to finite, periodic, or regular piecewise linear VCCs, is again a finite, periodic, or regular piecewise linear VCC. Hence it suffices to compute the result of the various operators up to an upper bound, after which we know that the result is either one linear piece, or a periodic repetition of a part we already computed. The following four step approach presents a general method to compute curve operators on compact VCCs.

1. Determine whether the resulting VCC is finite or not. If the resulting VCC is not finite, determine the period $p_{x,C}$, and the offset $p_{y,C}$ of the resulting periodic or regular VCC.

2. Determine an upper bound x_{unfold} up to which the resulting curve must be computed.

3. Compute the resulting curve up to x_{unfold}.

4. If the resulting VCC is finite, the aperiodic curve part of the resulting compact VCC is determined from the result in the range $0 \leq x \leq x_{unfold}$. If the resulting VCC is periodic or regular on the other hand, then the aperiodic curve part of the resulting compact VCC is determined from the result in the range $0 \leq x < x_{unfold} - p_{x,C}$, and the periodic curve part is determined from the result in the range $x_{unfold} - p_{x,C} \leq x < x_{unfold}$.

After applying this four step approach, it is often possible to optimize the resulting compact VCC, because the parts of the aperiodic curve part could also be captured already by the periodic curve part.

The remainder of this section concentrates on determining the parameters $p_{x,C}$, $p_{y,C}$, and x_{unfold} for the various curve operators. Knowing these parameters, the above four step approach can directly be applied to compute the various curve operators.

7.3.2 Unary Operators

Within the MPA framework, we identify three unary operators that are important for performance analysis.

7.3.2.1 Floor and Ceil

The floor $\lfloor \nu \rfloor$ and the ceil $\lceil \nu \rceil$ of a VCC are important to compute the arrival curves of a discrete event stream. The following parameters are valid for both operators.

If the compact VCC ν is finite, then $p_{x,C}$, $p_{y,C}$, and x_{unfold} are

$$p_{x,C} = 1/s_m \tag{7.9}$$
$$p_{y,C} = 1 \tag{7.10}$$
$$x_{unfold} = x_m + p_{x,C} \tag{7.11}$$

If the compact VCC ν is periodic or regular, then we must first determine the numerator $f_n \in \mathbb{Z}_{>0}$ and the denominator $f_d \in \mathbb{Z}_{>0}$ of the fraction $\frac{f_n}{f_d} = p_y$. Then $p_{x,C}$, $p_{y,C}$, and x_{unfold} can be determined as

$$p_{x,C} = f_d \cdot p_x \tag{7.12}$$
$$p_{y,C} = f_d \cdot p_y \tag{7.13}$$
$$x_{unfold} = x_{p0} + p_{x,C} \tag{7.14}$$

7.3.2.2 Scaling

Scaling a VCC ν with a factor f is important to transform event-based to resource-based curves and vice-versa. Then parameters $p_{x,C}$, $p_{y,C}$, and x_{unfold} to compute $f * \nu$ can be determined as

$$p_{x,C} = p_x \tag{7.15}$$
$$p_{y,C} = f \cdot p_x \tag{7.16}$$
$$x_{unfold} = x_{p0} + p_{x,C} \tag{7.17}$$

7.3.3 Binary Operators

Within the MPA framework, we further identify eight binary operators that are important for performance analysis. These eight binary operators can be grouped into four pairs of two closely related operators.

To compute x_{unfold} for a binary operation with the two VCCs ν_A and ν_B as argument, it is often important to compute the largest $x_{cross,max}$ at which these two argument VCCs cross. It is however also sufficient, and typically computationally less expensive, to compute an upper bound x_{cross} to this point. For this we define an upper and a lower linear bound to a compact VCC as follows.

For a finite VCC, its upper and lower bound equal its last curve segment

$$\eta^u = \eta^l = \langle x_m, y_m, s_m \rangle = \langle x_\eta, y_\eta, s_\eta \rangle \tag{7.18}$$

For a periodic or regular VCC, its upper linear bound is computed as

$$\eta^u = \langle x_{p0}, y_{p0} + y_{\sigma^u}, p_y/p_x \rangle = \langle x_\eta, y^u_\eta, s_\eta \rangle \tag{7.19}$$

with

$$y_{\eta^u} = \sup_{0 \le \Delta \le p_x} \left\{ F^u_{\Sigma_p}(\Delta) - \Delta \cdot \frac{p_y}{p_x} \right\} \tag{7.20}$$

And its lower linear bound is computed as

$$\eta^l = \langle x_{p0}, y_{p0} + y_{\eta^l}, p_y/p_x \rangle = \langle x_\eta, y^l_\eta, s_\eta \rangle \tag{7.21}$$

with

$$y_{\eta^l} = \inf_{0 \le \Delta \le p_x} \left\{ F^l_{\Sigma_P}(\Delta) - \Delta \cdot \frac{p_y}{p_x} \right\} \tag{7.22}$$

If the slopes of the linear bounds of the two argument VCCs ν_A and ν_B are equal, that is if $s_{\eta_A} = s_{\eta_B}$, then we call ν_A and ν_B *parallel* VCCs. Further, if there exists a crossing point of ν_A and ν_B after $\max\{x_{\eta_A}, x_{\eta_B}\}$, that is if $\exists x > \max\{x_{\eta_A}, x_{\eta_B}\} : \nu_A(x) = \nu_B(x)$, then we call ν_A and ν_B *interleaving* VCCs.

Using the above definitions, the upper bound to a possible cross point of two VCCs with $s_{\eta_A} < s_{\eta_B}$ can be computed as

$$x_{cross} = \max \left\{ \frac{y^l_{\eta_A} - y^u_{\eta_B} - (s_{\eta_A} * x_{\eta_A}) + (s_{\eta_B} * x_{\eta_B})}{s_{\eta_B} - s_{\eta_A}}, x_{\eta_A}, x_{\eta_B} \right\} \tag{7.23}$$

For two parallel and non-interleaving VCCs on the other hand, we define x_{cross} as

$$x_{cross} = \max \left\{ x_{\eta_A}, x_{\eta_B} \right\} \tag{7.24}$$

and for two interleaving VCCs, x_{cross} is not defined.

Next we define an operator $hp(p_{x,A}, p_{x,B})$ that computes the hyperperiod of two periods $p_{x,A} > 0$ and $p_{x,B} > 0$ as their least common multiple. If either of the two periods is zero, then $hp(p_{x,A}, p_{x,B})$ returns the non-zero period, and if both periods are zero, then $hp(p_{x,A}, p_{x,B})$ returns also zero.

Finally, in the following discussion of all commutative operators, we assume without loss of generality that the argument VCCs are arranged such that $s_{\eta_A} < s_{\eta_B}$ if ν_A and ν_B are not parallel, or such that $\nu_A(x) < \nu_B(x)\ \forall x > \max\{x_{\eta,A}, x_{\eta,B}\}$ if ν_A and ν_B are not interleaving.

7.3.3.1 Addition and Substraction

The parameters $p_{x,C}$, $p_{y,C}$, and x_{unfold} to compute the addition of two VCCs $\nu_A + \nu_B$ can be determined as

$$p_{x,C} = hp(p_{x,A}, p_{x,B}) \tag{7.25}$$

$$x_{unfold} = \max\{x_{\eta,A}, x_{\eta,B}\} + p_{x,C} \tag{7.26}$$

The substraction of two VCCs can be computed as $\nu_A - \nu_B = \nu_A + (-\nu_B)$, using the unary scaling operator.

7.3.3.2 Minimum and Maximum

The parameters $p_{x,C}$, $p_{y,C}$, and x_{unfold} to compute the minimum of two non-interleaving VCCs ν_A and ν_B can be determined as

$$p_{x,C} = p_{x,A} \tag{7.27}$$
$$x_{unfold} = x_{cross} + p_{x,C} \tag{7.28}$$

and if ν_A and ν_B are interleaving VCCs, $\min\{\nu_A, \nu_B\}$ can be computed with

$$p_{x,C} = hp(p_{x,A}, p_{x,B}) \tag{7.29}$$
$$x_{unfold} = \max\{x_{\eta,A}, x_{\eta,B}\} + p_{x,C} \tag{7.30}$$

The maximum of two VCCs can then be computed as $\max\{\nu_A, \nu_B\} = -\min\{-\nu_A, -\nu_B\}$.

7.3.3.3 Convolution

The parameters $p_{x,C}$, $p_{y,C}$, and x_{unfold} to compute the max-plus convolution $\overline{\otimes}$ of two non-parallel VCCs ν_A and ν_B can be determined as

$$p_{x,C} = p_{x,B} \tag{7.31}$$
$$x_{unfold} = \max\{x_{\eta,A} + x_{\eta,B} + hp(p_{x,A}, p_{x,B}), x_{cross} + p_{x,C}\} \tag{7.32}$$

and if ν_A and ν_B are parallel VCCs, then $\nu_A \,\overline{\otimes}\, \nu_B$ can be computed with

$$p_{x,C} = hp(p_{x,A}, p_{x,B}) \tag{7.33}$$
$$x_{unfold} = x_{\eta,A} + x_{\eta,B} + hp(p_{x,A}, p_{x,B}) \tag{7.34}$$

The min-plus convolution of two VCCs can then be computed as $\nu_A \otimes \nu_B = -(-\nu_A \,\overline{\otimes}\, -\nu_B)$.

7.3.3.4 Deconvolution

The parameters $p_{x,C}$, $p_{y,C}$, and x_{unfold} to compute the noncommutative min-plus deconvolution $\oslash$ of two non-parallel VCCs ν_A and ν_B can be determined as

$$p_{x,C} = p_{x,A} \tag{7.35}$$
$$x_{unfold} = x_{\eta,A} + x_{\eta,B} + hp(p_{x,A}, p_{x,B}) \tag{7.36}$$

Finally, the max-plus deconvolution of two VCCs can then be computed as $\nu_A \,\overline{\oslash}\, \nu_B = -(-\nu_A \oslash -\nu_B)$.

7.4 Discussion

The compact VCCs introduced in this chapter allow to efficiently and precisely compute various curve operators on VCCs that are defined by an infinite number of piecewise linear curve segments. Before the introduction of compact VCCs, the results of such curve operations could only be computed by either using finite piecewise linear approximations of the argument VCCs, or by sampling the argument VCCs up to a finite length. Neither of these methods was however very practical to apply within the MPA framework. Using approximations within the MPA framework leads to overly pessimistic performance analysis results, and using sampled VCCs within the MPA framework requires much care, because the number of valid samples in a sampled result curve is sometimes smaller than the number of samples in the original argument curves. This effect is especially observed when computing any of the convolution or deconvolution operators, and requires that the initial sampled input curves to a performance model consist of a large number of samples, consequently leading to long computation times.

In contrast to these formerly used methods, employing compact VCCs within the MPA framework allows efficient and precise performance analysis. However, the required computing power to compute the various curve operators on compact VCCs depends largely on the number of curve segments in the aperiodic and periodic parts of the argument VCCs, as well as on the periods of periodic or regular argument VCCs. Computing power demand particularly increases if the periods of two periodic or regular argument VCCs are relative prime to each other, or if they differ by several magnitudes. That said, it is then always possible to trade off precision for required computing power, by selectively using linearly or periodically approximated compact VCCs.

8

The RTC Toolbox

The Real-Time Calculus (RTC) Toolbox [WT] is a toolbox for MATLAB [Mat] that enables MPA framework-based performance analysis and interface-based design of embedded real-time systems within MATLAB. At its core, the RTC Toolbox provides a MATLAB type for variability characterization curves (VCCs), and an implementation of a large set of real-time calculus curve operations. Built around this core, the RTC Toolbox provides libraries to perform modular performance analysis and interface-based design, and to visualize VCCs and related data.

This chapter provides an introduction to the RTC Toolbox. The following Section 8.1 first presents an overview to the software architecture of the RTC Toolbox. This is followed by an application example of the RTC Toolbox for performance analysis of an embedded real-time system in Section 8.2. The chapter concludes with a discussion in Section 8.3.

8.1 Software Architecture

In the high-level software architecture of the RTC Toolbox that is depicted in Figure 81, we can identify that the RTC Toolbox internally consists of two major software components: a kernel that is implemented in Java, and a set of MATLAB libraries that connect the Java kernel to the MATLAB command line interface.

The first principal component of the Java kernel is a class that implements an object representing a compact VCC, as defined in Section 7.2. Internally, an object of this compact VCC class consists of two objects that

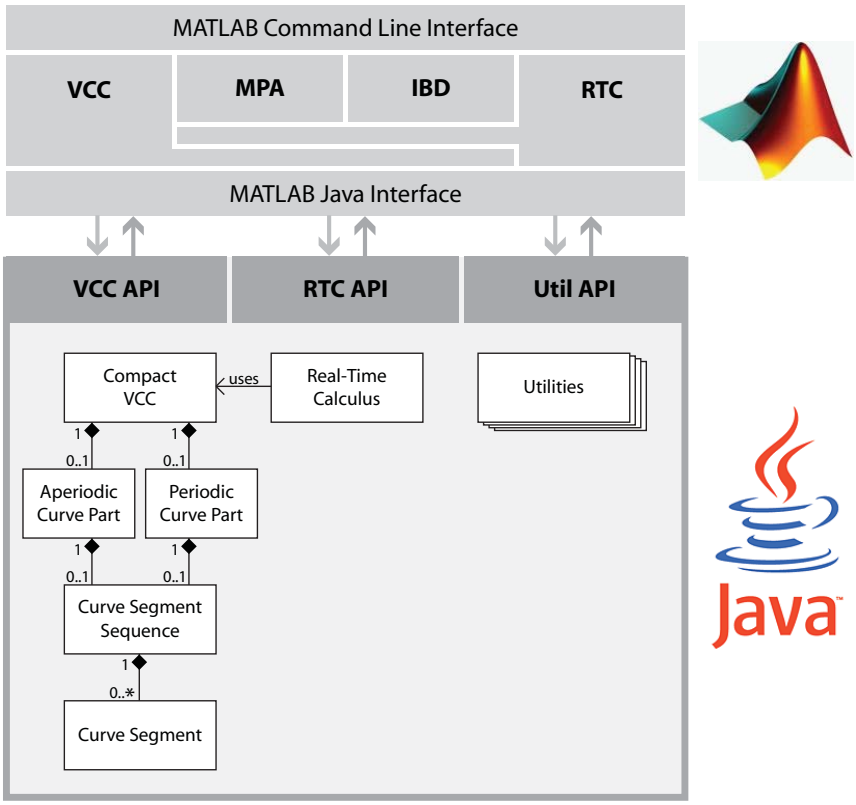


Fig. 81: Software architecture of the RTC Toolbox.

represent the compact VCC's aperiodic and periodic curve part, respectively, and that both internally consist again of an object that represents a curve segment sequence as an ordered list of curve segment objects. This second principal component of the Java kernel is then a class that implements the real-time calculus curve operators for compact VCCs that are represented as objects of the compact VCC class. These two principal classes are supplemented by a set of classes that provide various utilities. The Java kernel then provides a well-defined API that provide methods to create compact VCCs, to compute real-time calculus operations on these compact VCCs, and to access parts of the utilities.

From within MATLAB, the Java kernel is accessed via the MATLAB Java Interface. This access is however completely hidden from the user, that accesses the Java kernel instead via MATLAB functions that are provided by the VCC library and the RTC library. The VCC library mainly provides functions to create VCCs that are a self-contained MATLAB data types, internally represented as a Java object. Further, the VCC library provides functions to plot VCCs. The RTC library on the other hand provides wrapper functions for all the real-time calculus curve operators

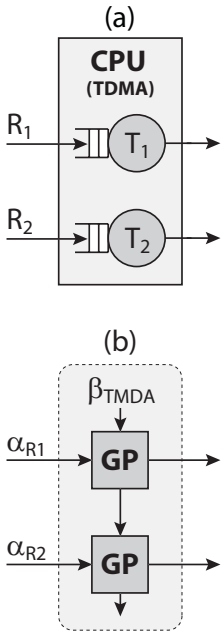
that are implemented by the Java kernel. And through the use of operator overloading, the functions of the RTC library allow to perform operations on VCCs similar as on any other data type within MATLAB.

The RTC Toolbox further provides two libraries that are built on top of the VCC and the RTC library and that provide a set of functions that facilitate the use of the RTC Toolbox for modular performance analysis and interface-based design. The MPA library provides functions to create commonly used arrival curves and service curves, as well as functions to conveniently compute and analyze the various abstract components within the MPA framework. The IBD library on the other hand provides functions to compute the various input assumptions and output guarantees of the abstract components within the MPA framework, thus facilitating interface-based design.

8.2 Case Study

In this section we illustrate how the RTC Toolbox can be used to analyze an MPA performance model. For this we consider the small embedded system depicted in Figure 82(a), that processes two real-time event streams R_1 ($p_{R_1} = 50ms$, $j_{R_1} = 200ms$, and $d_{R_1} = 1ms$) and R_2 ($p_{R_2} = 70ms$, and $j_{R_2} = 10ms$) on two fully preemptable and independent tasks T_1 and T_2 that run as an application on a CPU with a clock frequency of $1GHz$. The CPU also hosts various other applications that are not depicted in the system model. At the top level, the CPU implements a TDMA scheduler to control the resource distribution to the various applications, and within the TDMA cycle length of $40ms$, a slot of length $10ms$ is allocated to process the application that consists of T_1 and T_2. Within the application, T_1 and T_2 share the available service using a preemptive fixed priority scheduling policy, where the higher priority is assigned to T_1. The execution of T_1 requires $4E6$ cycles, and the execution of T_2 requires $5E6$ cycles.

The MPA performance model of the above specified system is depicted in Figure 82(b), and the code listing in Figure 82 depicts the M-code to analyze the performance model with MATLAB and the RTC Toolbox. On line 2 and 3, the function `pjd` is used to create the arrival curves for the two events streams, and on line 6, the function `tdma` creates the TDMA service curves. Note that the bandwidth is set to $1E6$, because the time basis is $1ms$. On lines 13 and 14, the function `gpc` computes the output of the greedy processing components, and on lines 17 and 18, the delays of the two event streams are computed with the function `del`. Finally, the command on line 21 plots the remaining service curves at the output of the performance model.




```
 1 % Arrival curves
 2 a0_R1   = pjd(50,200,1);
 3 a0_R2   = pjd(70,10,0);
 4
 5 % Service curves
 6 b0_TDMA = tdma(10,40,1e6);
 7
 8 % Execution demands
 9 e_T1    = 4e6;
10 e_T2    = 5e6;
11
12 % Compute output curves
13 [a1_R1 b1_TDMA] = gpc(a0_R1, b0_TDMA, e_T1);
14 [a1_R2 b2_TDMA] = gpc(a0_R2, b1_TDMA, e_T2);
15
16 % Compute delays
17 delay_S1   = del(a0_R1, b0_TDMA, e_T1);
18 delay_S2   = del(a0_R2, b1_TDMA, e_T2);
19
20 % Plot the remaining service curves
21 plot(b2_TDMA,600);
```

Fig. 82: System model (a) and performance model (b) of the case study system, together with the MATLAB M-code to analyze the performance model with the RTC Toolbox.

8.3 Discussion

The RTC Toolbox and the underlying compact VCCs prove to be powerful enough to analyze and design embedded real-time systems with the MPA framework. All results in this thesis were computed with the RTC Toolbox, and all plots of VCCs were created with it as well. Moreover, the RTC Toolbox was also successfully applied already to compute the results in [WT05b, WTVL05, WT05a, WT06b, WMT06, WT06a, CLS+06].

The execution time to analyze a performance model with the RTC Toolbox varies with the complexity of the model and the VCCs. In general, it is however in the magnitude of milliseconds or seconds.

For the future, it is planned to integrate the functionality of the RTC Toolbox into Simulink. This would allows to graphically compose performance models, what would largely increase the user friendliness of the RTC Toolbox. A proof-of-concept-implementation was already successfully accomplished.

Conclusions

The aim of this thesis was to show that it is possible to formally analyze complex distributed embedded real-time systems with a modular and extensible framework for system level performance analysis that enables the efficient computation of correct and accurate performance analysis results, and that can seamlessly be embedded into an embedded systems' design process. Moreover, we wanted to show that it is possible to extend the same framework to actively support system design through interface-based design methodologies. To achieve this goal, we extended the Modular Performance Analysis framework into various directions. In particular, we would like to point out the following main contributions of this work.

- We presented an extensive case study of a distributed in-car navigation system, where the MPA framework was used to answer design questions that typically arise in the early design phase of such a system. This case study demonstrated how performance analysis with the MPA framework can be seamlessly embedded into a UML-based design process, and it presented the first application of sensitivity analysis within the MPA framework.

- We introduced three new abstract components for the MPA framework that model greedy shapers, and tasks with multiple inputs, respectively, and we addressed the challenge to modeling and analyzing different processing semantics within a complex embedded system.

- We introduced Type Rate Curves, Event Sequence Automata, Workload Variability Automata, and Workload Correlation Curves, together with corresponding methods to address the challenges of complex inputs, variable execution demands, and workload correlations in complex embedded systems.

- We introduced the theory of Real-Time Interfaces that connects the principles of Real-Time Calculus and interface-based design, and we developed a component system with Real-Time Interfaces, that

enables interface-based embedded real-time system design within the MPA framework.

- We introduced two new abstract components for the MPA framework that model a TDMA scheduler, and a polling server, respectively, and that address the challenge of modeling, analyzing, and designing hierarchical scheduling policies in a complex embedded system. Moreover, we developed methods for optimal parameter selection of TDMA schedulers, and to support parameter selection for polling servers.

- We introduced a compact representation for a special class of variability characterization curves, together with methods to efficiently compute various Real-Time Calculus curve operations on these compact variability characterization curves, in order to efficiently conduct system level performance analysis and interface-based design within the MPA framework.

- We introduced the Real-Time Calculus (RTC) Toolbox for MATLAB to support system level performance analysis and interface-based design of embedded real-time systems with the MPA framework.

Although we could show that it is possible to formally analyze complex distributed embedded real-time systems, the question remains how relevant these methods and the results obtained with them are in the design process of realistic systems in an industrial environment. When we reflect on this question, we identify that the relevance depends on three factors: the quality of the analysis method, the quality of the available input data, and the hardness of the design requirements.

There are many factors that determine the quality of an analysis method. Some of which are correctness, accuracy, embeddability, modularity, and efficiency, as specified in Section 1.1.4. In this thesis, we further developed and extended the MPA framework, aiming at increasing its quality regarding these factors. And although we could improve the framework towards various directions, one of the major challenges that still needs to be solved is the incorporation of timing correlations in the analysis of distributed systems. Consequently, the analysis results of systems with a large amount of timing correlated event streams may be overly pessimistic, and are thus of decreased relevance.

But even if a formal analysis method is able to consider all details of a system, its analysis results are only as good as the available input data. The question here is always how good the various models that are used in the system analysis correlate with their concrete counterparts.

How detailed is the behavior of a communication protocol modeled, or how good is the model of a scheduling policy that is implemented in a particular concrete real-time operating system? And the major problem typically is to model the environment. Is a periodic event-stream really strictly periodic in reality, or how large may the jitter of an event-stream be in the worst-case? Often it is hard or even impossible to formally determine the specification of an environment. However a designer of a hard real-time system must always consider that the adherence to design specifications can only be guaranteed for a specified input. And if this input specification does not cover the worst-case environment behavior, the guarantee is invalid.

Finally, an important factor that threatens the relevance of formal performance analysis methods is the significance of worst-case and best-case bounds in modern computing systems. These systems are often designed to optimize average performance instead of worst-case performance. And the applied technologies to improve average performance, such as caches, branch predictors, and others, not only improve the average performance, but they often drastically downgrade the worst-case performance, such that the worst-case performance in these systems is often by magnitudes worse than the average performance. Moreover these technologies often also threaten the predictability of a system's performance in general. And even if such a system remain predictable, the worst-case and best-case bounds obtained on their performance are often irrelevant to a designer, even if they are correct and accurate. Consequently, it would be very interesting to augment the MPA framework with statistical methods, to tackle also design questions regarding the average performance of an embedded system.

Outlook

The Modular Performance Analysis framework proves to be a powerful and extensible framework for formal system level performance analysis of complex distributed embedded real-time systems. And although we considerably extended and further developed the MPA framework in this thesis, there exists still much potential for further extensions and developments. Based on the work presented in this thesis, and on the extensive experience we gained on the MPA framework, we identify three main areas of future research and development.

In the area of further developing the theoretical foundations and capabilities of the MPA framework we would like to highlight three challenges. First, it would be interesting to develop models and methods to address the challenge of timing correlations in complex embedded systems. Next, the problem of cyclic dependencies and fixed-point calculation within performance models needs to be investigated thoroughly. And finally, it would be interesting to conduct research on how to model and analyze dynamic systems with feedback and state-dependant behavior.

In the area of extending the modeling and analysis capabilities of the MPA framework, there exists a large potential to develop various additional abstract components. To name only a few, it would for example be interesting to develop abstract components for round robin scheduling, for deferrable servers, for smoothers, for schedulers with timeout, for external memory, or for components with blocking mutually exclusive access.

Finally, there exists a large potential in extending the theory of Real-Time Interfaces, and in extending the corresponding component system for interface-based design of embedded real-time systems. In particular, it would be interesting to extend the framework to be applicable for distributed systems, by introducing components with outgoing arrival curves. Moreover, the component system could be extended to consequently work with both, the upper, and the lower arrival and service curves at all connections.

Bibliography

[AB98] A. Atlas and A. Bestavros. Statistical rate monotonic scheduling. In *Proceedings of the 19th IEEE Real-Time Systems Symposium (RTSS)*, page 123, Washington, DC, USA, 1998. IEEE Computer Society.

[AD94] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[AFM+02] T. Amnell, E. Fersman, L. Mokrushin, P. Pettersson, and W. Yi. Times – A Tool for Modelling and Implementation of Embedded Systems. In *TACAS '02: Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 460–464, London, UK, 2002. Springer-Verlag.

[Aga92] A. Agarwal. Performance Tradeoffs in Multithreaded Processors. *IEEE Transactions on Parallel and Distributed Systems*, 3(5):525–539, September 1992.

[Alm03] L. Almeida. Response-Time Analysis and Server Design for Hierarchical Scheduling. In *Proceedings of the Work-in-Progress session of RTSS*, 2003.

[AP04] L. Almeida and P. Pedreira. Scheduling within temporal partitions: response-time analsysis and server design. In *Proceedings of the Fourth ACM International Converence on Embedded Software*, 2004.

[Bar98] S. K. Baruah. A general model for recurring real-time tasks. In *Proceedings of the IEEE Real-Time Systems Symposium (RTSS)*, pages 114–122, 1998.

[Bar03] S. K. Baruah. Dynamic- and static-priority scheduling of recurring real-time tasks. *Real-Time Systems*, 24(1):93–128, 2003.

[BBB+03] L. Benini, D. Bertozzi, D. Bruni, N. Drago, F. Fummi, and M. Poncino. SystemC cosimulation and emulation of multiprocessor SoC designs. *IEEE Computer*, 36(4):53–59, 2003.

[BCGM99] S. K. Baruah, D. Chen, S. Gorinsky, and A. K. Mok. Generalized multiframe tasks. *Real-Time Systems*, 17(1):5–22, 1999.

[BCOQ92] F. Bacelli, G. Cohen, G. J. Olsder, and J.-P. Quadrat. *Synchronization and Linearity: An Algebra for Discrete Event Systems*. Wiley Series in Probability and Mathematical Statistics. John Wiley & Sons Ltd, August 1992.

[BDL] G. Behrmann, A. David, and K. G. Larsen. A Tutorial on UppAal. In *Marco Bernardo and Flavio Corradini, editors, Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004, number 3185 in LNCS*, pages 200–236, September.

[BES+01] J. Berwanger, C. Ebner, A. Schedl, R. Belschner, S. Fluhrer, P. Lohrmann, E. Fuchs, D. Millinger, M. Sprachmann, F. Bogenberger, et al. FlexRay: The communication system for advanced automotive control systems. *SAE transactions*, 110(7):303–314, 2001.

[BL03] E. Bini and G. Lipari. Resource partitioning among real-time applications. In *Proc. of EUROMICRO Conference on Real-Time Systems (ECRTS)*, 2003.

[BRH90] S. K. Baruah, L. E. Rosier, and R. R. Howell. Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor. *Real-Time Systems*, 2(4):301–324, 1990.

[But97] G.C. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Kluwer Academic Publishers, Boston, 1997.

[CdAHS03] A. Chakrabarti, L. de Alfaro, T.A. Henzinger, and M. Stoelinga. Resource interfaces. In *EMSOFT 03: Embedded Software*, Lecture Notes in Computer Science 2855, pages 117–133. Springer-Verlag, 2003.

[CDQV85] G. Cohen, D. Dubois, J. P. Quadrat, and M. Voit. A linear-system-theoretic view of discrete-event processes and its use for performance avaluation in manufacturing. *IEEE Transactions on Automatic Control*, AC-30(3):210–220, March 1985.

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

[CLS+06] S. Chakraborty, Y. Liu, N. Stoimenov, L. Thiele, and E. Wandeler. Interface-Based Rate Analysis of Embedded Systems. In *27th IEEE Real-Time Systems Symposium (RTSS)*, December 2006. Under Review.

[Cru91] R. L. Cruz. A calculus for network delay. *IEEE Transactions on Information Theory*, 37(1):114–141, 1991.

[CT05] S. Chakraborty and L. Thiele. A new task model for streaming applications and its schedulability analysis. In *Design, Automation and Test in Europe (DATE)*, pages 486–491, 2005.

[dAH01a] L. de Alfaro and T. A. Henzinger. Interface Automata. In *Proc. Foundations of Software Engineering*, pages 109–120. ACM Press, 2001.

[dAH01b] L. de Alfaro and T. A. Henzinger. Interface Theories for Component-Based Design. In *EMSOFT 01: Embedded Software*, Lecture Notes in Computer Science 2211, pages 148–165. Springer-Verlag, 2001.

[dAH05] L. de Alfaro and T. A. Henzinger. Interface-based Design. In *To appear in the Proceedings of the 2004 Marktoberdorf Summer School*. Kluwer, 2005.

[dAHS02] L. de Alfaro, T. A. Henzinger, and M. Stoelinga. Timed Interfaces. In *EMSOFT 02: Embedded Software*, Lecture Notes in Computer Science 2491, pages 108–122. Springer-Verlag, 2002.

[DB05] R. I. Davis and A. Burns. Hierarchical Fixed Priority Pre-Emptive Scheduling. In *Proceedings of the 26th IEEE Real-Time Systems Symposium (RTSS).*, pages 389–398, 2005.

[DL97] Z. Deng and J.W.S. Liu. Scheduling real-time applications in an open environment. In *Proceedings of the 18th IEEE Real-Time Systems Symposium (RTSS)*, pages 308–319, 1997.

[DTB93] R. I. Davis, K. W. Tindell, and A. Burns. Scheduling slack time in fixed priority pre-emptive systems. In *Proceedings of the Real-Time Systems Symposium*, pages 222–231, 1993.

[EDPP00] P. Eles, A. Doboli, P. Pop, and Z. Peng. Scheduling with bus access optimization for distributed embeddedsystems. *IEEE Transactions on VLSI Systems*, 8(5):472–491, 2000.

[EWY99] C. Ericsson, A. Wall, and W. Yi. Timed Automata as Task Models for Event-Driven Systems. In *RTCSA '99: Proceedings of the Sixth International Conference on Real-Time Computing Systems and Applications*, page 182, Washington, DC, USA, 1999.

[Fle] FlexRay.
`http://www.flexray.com`.

[FM03] P. Fortier and H. Michel. *Computer Systems Performance Evaluation and Prediction*. Digital Press, August 2003.

[FW03] M.A. Franklin and T. Wolf. A network processor performance and design model with benchmark parameterization. In P. Crowley, M.A. Franklin, H. Hadimioglu, and P.Z. Onufryk, editors, *Network Processor Design: Issues and Practices, Volume 1*, chapter 6, pages 117–140. Morgan Kaufmann Publishers, 2003.

[GDvM+03] K. Goossens, J. Dielissen, J. van Meerbergen, P. Poplavko, A. Rădulescu, E. Rijpkema, E. Waterlander, and P. Wielage. Guaranteeing the quality of services in networks on chip. In *Networks on chip*, pages 61–82. Kluwer Academic Publishers, Hingham, MA, USA, 2003.

[GGPD01] M. González Harbour, J.J. Gutiérrez García, J.C. Palencia Gutiérrez, and J.M. Drake Moyano. MAST: Modeling and Analysis Suite for Real Time Applications. In *Proceedings of 13th Euromicro Conference on Real-Time Systems*, pages 125–134, Delft, The Netherlands, 2001. IEEE Computer Society Press.

[GH03] J. C. Palencia Gutiérrez and M. González Harbour. Offset-based response time analysis of distributed systems scheduled under EDF. In *Proceedings of the 15th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 3–12, 2003.

[GLMS02] T. Grötker, S. Liao, G. Martin, and S. Swan. *System Design with SystemC*. Kluwer Academic Publishers, Boston, MA, USA, May 2002.

[Gri04] M. Gries. Methods for evaluating and covering the design space during early design development. *VLSI Journal*, 38(2):131–183, 2004.

[GSE+98] S. Gringeri, K. Shuaib, R. Egorov, A. Lewis, B. Khasnabish, and B. Basch. Traffic shaping, bandwidth allocation, and quality assessment for MPEG video distribution over broadband networks. *IEEE Networks*, 12(6):94–107, 1998.

[GVNG94] D. D. Gajski, F. Vahid, S. Narayan, and J. Gong. *Specification and Design of Embedded Systems*. Prentice Hall, Englewood Cliffs, N.J., 1994.

[HE05] A. Hamann and R. Ernst. TDMA Time Slot and Turn Optimization with Evolutionary Search Techniques. In *Design, Automation and Test in Europe (DATE 2005)*, 2005.

[HHJ+05] R. Henia, A. Hamann, M. Jersak, R. Racu, K. Richter, and R. Ernst. System level performance analysis – the symta/s approach. *IEE Proceedings – Computers and Digital Techniques*, 152(2):148–166, 2005.

[HKA+01] C. J. Hughes, P. Kaul, S. V. Adve, R. Jain, C. Park, and J. Srinivasan. Variability in the execution of multimedia applications and implications for architecture. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 254–265. ACM Press, 2001.

[HM06] T. A. Henzinger and S. Matic. An interface algebra for real-time components. In *Proceedings of the 12th Annual Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE Computer Society Pres, 2006.

[HV06] M. Hendriks and M. Verhoef. Timed automata based analysis of embedded system architectures. In *Workshop on Parallel and Distributed Real-Time Systems – WPDRTS 2006*, 2006.

[IEE98] IEEE/EIA. *ISO/IEC 12207:1995 Standard for Information Technology – Software life cycle processes*. The Institute of Electrical and Electronics Engineers, Inc., March 1998.

[Inu79] T. Inukai. An Efficient SS/TDMA Time Slot Assignment Algorithm. *IEEE Transactions on Communications*, 27(10):1449–1455, 1979.

[JE03] M. Jersak and R. Ernst. Enabling Scheduling Analysis of Heterogeneous Systems with Multi-Rate Data Dependencies and Rate Intervals. In *Proceedings 40th Design Automation Conference (DAC)*, June 2003.

[Jer05] M. Jersak. *Compositional Performance Analysis for Complex Embedded Applications*. PhD thesis, Technical University of Braunschweig, 2005.

[JHE04] M. Jersak, R. Henia, and R. Ernst. Context-aware performance analysis for efficient embedded systems design. In *Proc. 7th Design, Automation and Test in Europe (DATE)*, 2004.

[JRE04] M. Jersak, K. Richter, and R. Ernst. Performance analysis for complex embedded applications. *International Journal of Embedded Systems, Special Issue on Codesign for SoC*, 2004.

[Kar78] R. M. Karp. A characterization of the minimum cycle mean in a digraph. *Discrete Mathematics*, (23):309–311, 1978.

[KDVW97] B. Kienhuis, E. Deprettere, K. Vissers, and P. van der Wolf. An approach for quantitative analysis of application-specific dataflow architectures. In *ASAP '97: Proc. of the IEEE International Conference on Application-Specific Systems, Architectures and Processors*, page 338, Washington, DC, USA, 1997. IEEE Computer Society.

[KG93] H. Kopetz and G. Grunsteidl. TTP - A time-triggered protocol for fault-tolerant real-timesystems. pages 524–533, 1993.

[KL99] T.W. Kuo and C.H. Li. A Fixed-Priority-Driven Open Environment for Real-Time Applications. In *Proceedings of the 20th IEEE Real-Time Systems Symposium*, 1999.

[KM98] A. Kalavade and P. Moghé. A tool for performance estimation of networked embedded end-systems. In *Proceedings of the 35th Conference on Design Automation Conference (DAC)*, pages 257–262. ACM/IEEE, June 1998.

[Kop97] H. Kopetz. *Real-Time Systems - Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, 1997.

[KPBT06] S. Künzli, F. Poletti, L. Benini, and L. Thiele. Combining simulation and formal methods for system-level performance analysis. In *Proc. Design, Automation and Test in Europe (DATE)*, March 2006.

[Leh96] J. P. Lehoczky. Real-time queueing theory. In *Proceedings of the 17th IEEE Real-Time Systems Symposium (RTSS)*, page 186, Washington, DC, USA, 1996. IEEE Computer Society.

[Lin98] C. Lindemann. *Performance Modelling with Deterministic and Stochastic Petri Nets*. John Wiley and Sons, 1998.

[LL73] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.

[LMCO04] Y. Liu, A. Maxiaguine, S. Chakraborty, and W. T. Ooi. Processor frequency selection for SoC platforms for multimedia applications. In *Proceedings of the 25th IEEE International Real-Time Systems Symposium (RTSS)*, pages 336–345, Lisbon, Portugal, December 2004. IEEE Computer Society.

[LRD01] K. Lahiri, A. Raghunathan, and S. Dey. System level performance analysis for designing on-chip communication architectures. *IEEE Transactions on Computer Aided-Design of Integrated Circuits and Systems*, 20(6):768–783, 2001.

[LRT92] J.P. Lehoczky and S. Ramos-Thuel. An Optimal Algorithm for Scheduling Soft-Aperiodic Tasks in Fixed-Priority Preemptive Systems. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 110–123, 1992.

[LSS+87] J.P. Lehoczky, L. Sha, J.K. Strosnider, et al. Enhanced Aperiodic Responsiveness in Hard Real-Time Environments. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 261–270, 1987.

[LT01] J. Y. Le Boudec and P. Thiran. *Network Calculus - A Theory of Deterministic Queuing Systems for the Internet*. Number 2050 in Lecture Notes in Computer Science (LNCS). Springer Verlag, 2001.

[LW82] J. Leung and J. W. Withehead. On the complexity of fixed priority scheduling of periodic real-time tasks. *Performance Evaluation*, 2(4), 1982.

[Mat] Matlab.
`http://www.mathworks.com`.

[Max] SoC Designer with MaxSim Technology (ARM).
`http://www.arm.com/products/DevTools/MaxSim.html`.

[Max05] A. Maxiaguine. *Modeling Multimedia Workloads for Embedded System Design*. PhD thesis, ETH Zurich, October 2005.

[MC97] A. K. Mok and D. Chen. A multiframe model for real-time tasks. *IEEE Transactions on Software Engineering*, 23(10):635–645, 1997.

[MEP04] S. Manolache, P. Eles, and Z. Peng. Schedulability analysis of applications with stochastic task execution times. *Transactions on Embedded Computing Systems*, 3(4):706–735, 2004.

[MFC01] A. K. Mok, A. X. Feng, and D. Chen. Resource partition for real-time systems. In *Proceedings of the Real-Time Systems Symposium (RTSS)*, pages 75– 84. IEEE Computer Society, 2001.

[MKT04] A. Maxiaguine, S. Künzli, and L. Thiele. Workload characterization model for tasks with variable execution demand. In *Design, Automation and Test in Europe (DATE)*, pages 1040–1045, Paris, France, February 2004. IEEE Computer Society.

[MLCO04] A. Maxiaguine, Y. Liu, S. Chakraborty, and W. T. Ooi. Identifying ẗepresentativeẅorkloads in designing mpsoc platforms for media processing. In *2nd Workshop on Embedded Systems for Real-Time Multimedia (ESTIMedia)*, pages 41– 46, Stockholm, Sweden, September 2004. IEEE Press.

[MZCW04] A. Maxiaguine, Yongxin Zhu, S. Chakraborty, and Weng-Fai Wong. Tuning SoC platforms for multimedia processing: identifying limits and tradeoffs. In *Proceedings of the 2nd IEEE/ACM/IFIP International Conference on Hardware/software Codesign and System Synthesis (CODES+ISSS)*, pages 128–133. ACM Press, 2004.

[Obe02] R. Obermaisser. CAN Emulation in a Time-Triggered Environment. In *Proceedings of the 2002 IEEE International Symposium on Industrial Electronics (ISIE)*. IEEE, 2002.

[Obe05] R. Obermaisser. *Event-Triggered and Time-Triggered Control Paradigms*, volume 22 of *Real-Time Systems Series*. Springer, 2005.

[PEP00] P. Pop, P. Eles, and Z. Peng. Performance estimation for embedded systems with data and control dependencies. In *Proceedings of the 8th International Workshop on Hardware/Software Co-Design (CODES)*, pages 62–66, 2000.

[PEP02] T. Pop, P. Eles, and Z. Peng. Holistic scheduling and analysis of mixed time/event-triggered distributed embedded systems. In *Proceedings of the 10th International Symposium on Hardware/Software Codesign (CODES)*, pages 187–192, 2002.

[PEP03] P. Pop, P. Eles, and Z. Peng. Schedulability analysis and optimization for the synthesis of multi-cluster distributed embedded systems. In *Design, Automation and Test in Europe (DATE)*, pages 10184–10189, 2003.

[PEP+04] Paul Pop, Petru Eles, Zebo Peng, Viacheslav Izosimov, Magnus Hellring, and Olof Bridal. Design Optimization of Multi-Cluster Embedded Systems for Real-Time Applications. In *Design, Automation and Test in Europe (DATE 2004)*, pages 1028–1033, 2004.

[PG93] A. K. Parekh and R. G. Gallager. A generalized processor sharing approach to flow control in integrated services networks: The single-node case. *IEEE/ACM Transactions on Networking*, 1(3):344–357, June 1993.

[PG94] A. K. Parekh and R. G. Gallager. A generalized processor sharing approach to flow control in integrated services networks: The multiple node case. *IEEE/ACM Transactions on Networking*, 2:137–150, April 1994.

[RBGW97] J. Rexford, F. Bonomi, A. Greenberg, and A. Wong. Scalable architectures for integrated traffic shaping and link scheduling in high-speed ATM switches. *IEEE Journal on Selected Areas in Communications*, 15(5):938–950, 1997.

[RE02] K. Richter and R. Ernst. Event model interfaces for heterogeneous system analysis. In *Proc. 5th Design, Automation and Test in Europe (DATE)*, page 506. IEEE Computer Society, March 2002.

[Ric05] K. Richter. *Compositional Performance Analysis*. PhD thesis, Technical University of Braunschweig, 2005.

[RJE03] K. Richter, M. Jersak, and R. Ernst. A formal approach to MpSoC performance verification. *IEEE Computer*, 36(4), 2003.

[RJE05] R. Racu, M. Jersak, and R. Ernst. Applying sensitivity analysis in real-time distributed systems. In *11th IEEE Real-Time Technology and Applications Symposium (RTAS)*, San Francisco, USA, 2005.

[RTL93] S. Ramos-Thuel and JP Lehoczky. On-line scheduling of hard deadline aperiodic tasks infixed-priority systems. In *Proceeding of the Real-Time Systems Symposium.*, pages 160–171, 1993.

[RZJE02] K. Richter, D. Ziegenbein, M. Jersak, and R. Ernst. Model composition for scheduling analysis in platform design. In *Proceedings 39th Design Automation Conference (DAC)*, June 2002.

[SAÅ+04] L. Sha, T. F. Abdelzaher, K.-E. Årzén, A. Cervin, T. P. Baker, A. Burns, G. Buttazzo, M. Caccamo, J. P. Lehoczky, and A. K. Mok. Real time scheduling theory: A historical perspective. *Real-Time Systems*, 28(2-3):101–155, 2004.

[SB94] M. Spuri and GC Buttazzo. Efficient aperiodic service under earliest deadline scheduling. In *Proceedings of the Real-Time Systems Symposium (RTSS)*, pages 2–11, 1994.

[SB96] M.S. Spuri and G.S. Buttazzo. Scheduling aperiodic tasks in dynamic priority systems. *Real-Time Systems*, 10(2):179–210, 1996.

[Sea] Seamless Hardware/Software Co-Verification, Mentor Graphics.
`http://www.mentor.com/seamless/`.

[Sim] SimpleScalar Tool Set.
`http://www.simplescalar.com/`.

[SJNL05] H. Schiøler, J. Jessen, J. Dalsgaard Nielsen, and K. G. Larsen. CyNC - towards a general tool for performance analysis of complex distributed real-time systems. In *Proceedings of the WiP Session of the 17th EUROMICRO Conference on Real-Time Systems (ECRTS 05)*, pages 61–64. IEEE, 2005.

[SL03] I. Shin and I. Lee. Periodic resource model for compositional real-time guarantees. In *Proceedings of the Real-Time Systems Symposium (RTSS)*, pages 2–13. IEEE Press, 2003.

[SL04] I. Shin and I. Lee. Compositional Real-Time Scheduling Framework. In *Proceedings of the Real-Time Systems Symposium (RTSS)*, pages 57–67. IEEE Press, 2004.

[SLR86] L. Sha, J. P. Lehoczky, and R. Rajkumar. Solutions for some practical problems in prioritised preemptive scheduling. In

Proceedings of the IEEE Real-Time Systems Symposium (RTSS), pages 181–191, 1986.

[SLS95] J. K. Strosnider, J. P. Lehoczky, and L. Sha. The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environments. *IEEE Transactions on Cumputers*, 44(1):73–91, 1995.

[SRLK02] S. Saewong, R. R. Rajkumar, J. P. Lehoczky, and M. H. Klein. Analysis of hierar hical fixed-priority scheduling. In *Proceedings of the 14th Euromicro Conference on Real-Time Systems*, pages 152–160, 2002.

[SSL89] B. Sprunt, L. Sha, and J. Lehoczky. Aperiodic task scheduling for Hard-Real-Time systems. *Real-Time Systems*, 1(1):27–60, 1989.

[Sym] SymTA/S Tool Suite. `http://www.symtavision.com/`.

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

[Sysb] System Studio (Synopsis). `http://www.synopsis.com/products/cocentric_studio/`.

[TBW95] K. Tindell, A. Burns, and A.J. Wellings. Calculating controller area networks (can) message response times. *Control Engineering Practice*, 3(8):1163–1169, 1995.

[TC94] K. Tindell and J. Clark. Holistic schedulability analysis for distributed hard real-time systems. *Microprocessing & Microprogramming*, 40(2-3):117–134, 1994.

[TCN00] L. Thiele, S. Chakraborty, and M. Naedele. Real-time calculus for scheduling hard real-time systems. In *Proc. IEEE International Symposium on Circuits and Systems (ISCAS)*, volume 4, pages 101–104, 2000.

[TDS+95] T.-S. Tia, Z. Deng, M. Shankar, M. Storch, J. Sun, L.-C. Wu, and J. W.-S. Liu. Probabilistic performance guarantee for real-time tasks with varying computation times. In *Proceedings of the IEEE Real-Time Technology and Applications Symposium (RTAS)*, pages 164 – 173. IEEE Computer Society, 1995.

[Tim] TimesTool. `http://www.timestool.com/`.

[TKZ04] L. Thiele, S. Künzli, and E. Zitzler. A Modular Design Space Exploration Framework for Embedded Systems. *IEE Proceedings Computers & Digital Techniques*, 2004. Special Issue on Embedded Microelectronic Systems.

[TLS96] T.S.N. Tia, J.W.S.N. Liu, and M.N. Shankar. Algorithms and optimality of scheduling soft aperiodic requests in fixed-priority preemptive systems. *Real-Time Systems*, 10(1):23–43, 1996.

[TW05] L. Thiele and E. Wandeler. Performance Analysis of Embedded Systems. In *The Embedded Systems Handbook*. CRC Press, 2005.

[Upp] UppAal. `http://www.uppaal.com/`.

[VCC] The Cadence Virtual Component Co-design (VCC). `http://www.cadence.com/products/vcc.html`.

[WMT06] E. Wandeler, A. Maxiaguine, and L. Thiele. Performance Analysis of Greedy Shapers in Real-Time Systems. In *Design, Automation and Test in Europe (DATE)*, pages 444–449, March 2006.

[WRM+05] S. Wang, S. Rho, Z. Mai, R. Bettati, and W. Zhao. Real-time component-based systems. In *Proceedings of the 11th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 428–437. IEEE Press, 2005.

[WT] E. Wandeler and L. Thiele. Real-Time Calculus (RTC) Toolbox. `http://www.mpa.ethz.ch/Rtctoolbox`.

[WT05a] E. Wandeler and L. Thiele. Real-Time Interfaces for Interface-Based Design of Real-Time Systems with Fixed Priority Scheduling. In *5th ACM Conference on Embedded Software (EMSOFT)*, pages 80–89, September 2005.

[WT05b] E. Wandeler and L. Thiele. Workload Correlations in Multi Processor Hard Real-Time Systems. *Journal of Computer ans System Sciences, Special Issue on Real-Time and Embedded Systems*, 2005. In press.

[WT06a] E. Wandeler and L. Thiele. Interface-Based Design of Real-Time Systems with Hierarchical Scheduling. In *12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 243–252, April 2006.

[WT06b] E. Wandeler and L. Thiele. Optimal TDMA Time Slot and Cycle Length Allocation for Hard Real-Time Systems. In *11th Asia South Pacific Design Automation Conference (ASP-DAC)*, pages 479–484, January 2006.

[WTVL05] E. Wandeler, L. Thiele, M. Verhoef, and P. Lieverse. System Architecture Evaluation Using Modular Performance Analysis - A Case Study. *Software Tools for Technology Transfer*, 2005. In press.

[YW95] T.-Y. Yen and W. Wolf. Performance estimation for real-time distributed embedded systems. In *ICCD '95: Proceedings of the 1995 International Conference on Computer Design*, pages 64–71, Wahsington, DC, USA, 1995. IEEE Computer Society.

A

Real-Time Calculus

Real-Time Calculus is relies on min-plus calculus and max-plus calculus that both define a special algebra (the min-plus dioid and max-plus dioid, respectively). Traditionally, we are used to work with the algebraic structure $(\mathbb{R}, +, \times)$, i. e. with the set of reals endowed with the operations of addition and multiplication, that possess a number of properties such as associativity, commutativity, or distributivity.

In difference to this, min-plus calculus works with an algebraic structure $(\mathbb{R} \cup \infty, \vee, +)$. Here, the operation of addition becomes the computation of the infimum (or the minimum), and the operation of multiplication becomes the addition. Most axioms known from conventional algebra still apply to this algebraic structure. And in max-plus calculus, the infimum and minimum are simply replaced by supremum and maximum.

For more information on min-plus and max-plus calculus see also [LT01] and [BCOQ92].

A.1 Convolutions and Deconvolutions

In the MPA framework, we often need to compute convolutions and deconvolutions defined in min-plus and max-plus calculus.

The min-plus convolution $\otimes$ and the min-plus deconvolution $\oslash$ of two functions f and g are defined as [LT01]:

$$(f \otimes g)(\Delta) = \inf_{0 \leq \lambda \leq \Delta} \{f(\Delta - \lambda) + g(\lambda)\} \tag{A.1}$$

$$(f \oslash g)(\Delta) = \sup_{\lambda \geq 0} \{f(\Delta + \lambda) - g(\lambda)\} \tag{A.2}$$

The max-plus convolution $\overline{\otimes}$ and the max-plus deconvolution $\overline{\oslash}$ of two functions f and g on the other hand are defined as [LT01]:

$$(f \,\overline{\otimes}\, g)(\Delta) = \sup_{0 \le \lambda \le \Delta} \{f(\Delta - \lambda) + g(\lambda)\} \tag{A.3}$$

$$(f \,\overline{\oslash}\, g)(\Delta) = \inf_{\lambda \ge 0} \{f(\Delta + \lambda) - g(\lambda)\} \tag{A.4}$$

A.2 Sub-Additivity and Sub-Additive Closure

A curve f is *sub-additive*, if

$$f(a) + f(b) \ge f(a + b) \quad \forall a, b \ge 0 \tag{A.5}$$

The sub-additive *closure* $\overline{f}$ of a curve f is the largest sub-additive curve with $\overline{f} \le f$ and is computed as

$$\overline{f} = \min\{f, (f \otimes f), (f \otimes f \otimes f), \ldots\} \tag{A.6}$$

If f is interpreted as an arrival curve, then any trace R that is upper bounded by f is also upper bounded by the sub-additive closure $\overline{f}$.

A.3 Selected Properties

Lem. 2: *Given non-negative monotonic increasing functions* $f, g, h, j : \mathbb{R} \mapsto \mathbb{R}$ *with* $f(t) = g(t) = h(t) = j(t) = 0$ *for all* $t \le 0$. *Then*

$$(f \otimes g)\overline{\oslash}(h \otimes j) \ge (f \overline{\oslash} h) \otimes (g \overline{\oslash} j)$$

Proof. We have by definition of the operators:

$$(f \otimes g)\overline{\oslash}(h \otimes j) = \inf_{\lambda \ge 0} \sup_{0 \le b \le \lambda} \inf_{0 \le a \le \Delta + \lambda} [f(a) - h(b) + g(\Delta + \lambda - a) - j(\lambda - b)]$$

We will consider three cases depending on the value of a:

- $b \le a \le b + \Delta$:

$$\begin{aligned}
&\inf_{\lambda \ge 0} \sup_{0 \le b \le \lambda} \inf_{b \le a \le b + \Delta} [f(a) - h(b) + g(\Delta + \lambda - a) - j(\lambda - b)] \\
=&\inf_{\lambda \ge 0} \sup_{0 \le b \le \lambda} \inf_{0 \le a' \le \Delta} [f(a' + b) - h(b) + g(\Delta + \lambda - a' - b) - j(\lambda - b)] \\
\ge&\inf_{\lambda \ge 0} \inf_{0 \le b \le \lambda} \inf_{0 \le a' \le \Delta} [f(a' + b) - h(b) + g(\Delta + \lambda - a') - j(\lambda)] \\
\ge&\inf_{\lambda \ge 0} \inf_{0 \le b} \inf_{0 \le a' \le \Delta} [f(a' + b) - h(b) + g(\Delta + \lambda - a') - j(\lambda)] \\
=&\inf_{0 \le a' \le \Delta} [\inf_{0 \le b}(f(a' + b) - h(b)) + \inf_{\lambda \ge 0}(g(\Delta + \lambda - a') - j(\lambda))] \\
=&(f \overline{\oslash} h) \otimes (g \overline{\oslash} j)
\end{aligned}$$

Note that we used variable substitutions and the relation

$$\sup_a\{u(a) + v(a)\} \geq u(a_0) + \inf_a\{v(a)\}$$

- $b + \Delta \leq a \leq \Delta + \lambda$:

$$\begin{aligned}
&\inf_{\lambda\geq 0}\sup_{0\leq b\leq\lambda}\inf_{b+\Delta\leq a\leq\Delta+\lambda}[f(a) - h(b) + g(\Delta + \lambda - a) - j(\lambda - b)]\\
=&\inf_{\lambda\geq 0}\sup_{0\leq b\leq\lambda}\inf_{0\leq a'\leq\lambda-b}[f(a' + b + \Delta) - h(b) + g(\lambda - b - a') - j(\lambda - b)]\\
=&\inf_{\lambda\geq 0}\sup_{0\leq b'\leq\lambda}\inf_{0\leq a'\leq b'}[f(a' + \Delta + \lambda - b') - h(\lambda - b') + g(b' - a') - j(b')]\\
\geq&\inf_{\lambda\geq 0}\inf_{0\leq b'\leq\lambda}\inf_{0\leq a'\leq b'}[f(a' + \Delta + \lambda - b') - h(\lambda - b')]\\
\geq&\inf_{0\leq b}\inf_{0\leq a'}[f(a' + \Delta + b) - h(b)]\\
=&\inf_{0\leq b}[f(\Delta + b) - h(b)]\\
=&f\overline{\oslash}h
\end{aligned}$$

- $0 \leq a \leq b$:

$$\begin{aligned}
&\inf_{\lambda\geq 0}\sup_{0\leq b\leq\lambda}\inf_{0\leq a\leq b}[f(a) - h(b) + g(\Delta + \lambda - a) - j(\lambda - b)]\\
=&\inf_{\lambda\geq 0}\sup_{0\leq b\leq\lambda}\inf_{0\leq a'\leq b}[f(b - a') - h(b) + g(\Delta + a' + \lambda - b) - j(\lambda - b)]\\
\geq&\inf_{\lambda\geq 0}\inf_{0\leq b'\leq\lambda}\inf_{0\leq a'\leq b}[g(\Delta + a' + \lambda - b) - j(\lambda - b)]\\
\geq&\inf_{0\leq\lambda}\inf_{0\leq a'}[g(\Delta + a' + \lambda) - j(\lambda)]\\
=&\inf_{0\leq\lambda}[g(\Delta + \lambda) - j(\lambda)]\\
=&g\overline{\oslash}j
\end{aligned}$$

Now we have

$$\begin{aligned}
(f \otimes g)\overline{\oslash}(h \otimes j) &\geq \min\{(f\overline{\oslash}h) \otimes (g\overline{\oslash}j), f\overline{\oslash}h, g\overline{\oslash}j\}\\
&=(f\overline{\oslash}h) \otimes (g\overline{\oslash}j)
\end{aligned}$$

□

A.4 Greedy Processing Component

This section contains the proofs for the real-time calculus relations of a greedy processing component as used in the MPA framework, and as

introduced in Example 3 in Chapter 2. The following proofs require arrival and service curves as defined in this thesis, that are based on differential arrival and service functions $R[s,t)$ and $C[s,t)$ that extend to the whole time domain. With the arrival and service curves defined by Cruz [Cru91] and by Le Boudec and Thiran [LT01], that are based on cumulative arrival and service functions $R(\tau)$ and $C(\tau)$ that are only defined for the positive time domain, the following proofs are not valid.

A.4.1 Basic Function Processing

The time domain is defined by $\tau \in \mathbb{R}$. The sum of events in the time interval $s \leq \tau < t$ or $\tau \in [s,t)$ is described by the differential arrival function $R[s,t)$, as defined in Chapter 2 For a zero-length interval, i. e. $s = t$, we set $R[s,t) = 0$. In a similar way, the resource availability is characterized by the service function $C[s,t)$ which here denotes the number of events that could be processed in $\tau \in [s,t)$, again with $C[s,s) = 0$.

Let us first rephrase the basic function processing by a greedy processing component.

Def. 23: **(Event Processing)** *Given a resource node with backlog $B(s)$ at some time s and with a service function described by $C[s,t)$. An event stream described by $R[s,t)$ is processed by this node. Then the remaining resources described by $C'[s,t)$ and processed event stream described by $R'[s,t)$, satisfy the following relations for all s,t with $s \leq t$:*

$$R'[s,t) = C[s,t) - C'[s,t) \tag{A.7}$$

$$C'[s,t) = \sup_{s \leq u \leq t} \{C[s,u) - R[s,u) - B(s), 0\} \tag{A.8}$$

The first equation states the conservation of events, i. e. the incoming resource stream is partitioned into the outgoing event and resource streams. The second equation is derived from the fact that the outgoing resource stream (i) is nonnegative and (ii) is the minimal value that satisfies $C'[s,t) \geq C[s,u) - (R[s,u) + B(s))$ for all u with $s \leq u \leq t$. Here, $R[s,u)+B(s)$ denotes the events available for processing within the interval $[s,u)$.

Using the abbreviation $\{x\}^+ = \sup\{x, 0\}$, we can also write

$$C'[s,t) = \sup_{s \leq u \leq t} \{C[s,u) - R[s,u) - B(s)\}^+ \tag{A.9}$$

In addition, we can define the backlog $B(s)$ at time s. We have for all $s \leq t$

$$B(t) - B(s) = R[s,t) - R'[s,t) \tag{A.10}$$

The proofs of the basic real-time relations for the greedy processing component need an assumption on the backlog. In particular, we require

without loss of generality that for any time τ, there exists an earlier time $p \leq \tau$ such that $B(p) = 0$, i. e. the buffer for events was empty at some time $p < t$.

The arrival and service curves are defined as in Chapter 2. In particular, we have

$$\alpha^l(t-s) \leq R[s,t) \leq \alpha^u(t-s) \quad \forall s \leq t \tag{A.11}$$

$$\beta^l(t-s) \leq C[s,t) \leq \beta^u(t-s) \quad \forall s \leq t \tag{A.12}$$

In addition, we will sometimes suppose that the upper and lower curves are sub-additive and super-additive, respectively:

$$\alpha^u(t) + \alpha^u(s) \geq \alpha^u(t+s) \qquad \text{(sub-additive)} \tag{A.13}$$

$$\alpha^l(t) + \alpha^l(s) \leq \alpha^l(t+s) \qquad \text{(super-additive)} \tag{A.14}$$

The same relations also hold for the service curves β.

A.4.2 Remaining Service

Cor. 1: **(Upper and Lower Remaining Service)** *Given an event stream described by the arrival curves α^u, α^l and a resource described by the service curves β^u, β^l. Then the remaining service is bounded by the curves*

$$\beta^{l'}(\Delta) = \sup_{0 \leq \lambda \leq \Delta} \left\{ \beta^l(\lambda) - \alpha^u(\lambda) \right\} \quad \forall 0 \leq \Delta \tag{A.15}$$

$$\beta^{u'}(\Delta) = \inf_{\Delta \leq \lambda} \left\{ \beta^u(\lambda) - \alpha^l(\lambda) \right\}^+ \quad \forall 0 \leq \Delta \tag{A.16}$$

Proof. Note that we suppose the existence of an arbitrarily small time $p \in \mathbb{R}$ such that the backlog satisfies $B(p) = 0$.

To prove (A.15), we require $s \geq p$ and $t \geq p$ in the following. Then, we can write for all $p \leq s \leq t$:

$$\begin{aligned} C'[s,t) &= C'[p,t) - C'[p,s) \\ &= \sup_{p \leq a \leq t} \{C[p,a) - R[p,a)\}^+ - \sup_{p \leq b \leq s} \{C[p,b) - R[p,b)\}^+ \end{aligned}$$

Since $C[p,p) - R[p,p) = 0$, the suprema are nonnegative and we can omit the maximization with zero:

$$\begin{aligned} &= \sup_{p \leq a \leq t} \{C[p,a) - R[p,a)\} - \sup_{p \leq b \leq s} \{C^*[p,b) - R[p,b)\} \\ &= \inf_{p \leq b \leq s} \left\{ \sup_{p \leq a \leq t} \{(C[p,a) - C[p,b)) - (R[p,a) - R[p,b))\} \right\} \end{aligned}$$

We know that $a \geq b$ because of $t \geq s$ and we can write:

$$= \inf_{p \leq b \leq s} \left\{ \sup_{0 \leq a-b \leq t-b} \{C[b,a) - R[b,a)\} \right\}$$

Using $\lambda = a - b$, the definitions of arrival and service curves and some arguing about the infimum and supremum operators finally yields:

$$\geq \inf_{p \leq b \leq s} \left\{ \sup_{0 \leq \lambda \leq t-b} \left\{ \beta^l(\lambda) - \alpha^u(\lambda) \right\} \right\}$$
$$\geq \sup_{0 \leq \lambda \leq t-s} \left\{ \beta^l(\lambda) - \alpha^u(\lambda) \right\}$$

To prove (A.16), we again suppose $p \leq s \leq t$ and $B(p) = 0$ for an arbitrarily small time p.

$$C'[s,t) = C'[p,t) - C'[p,s)$$
$$= \sup_{p \leq a \leq t} \left\{ \inf_{p \leq b \leq s} \{C[b,a) - R[b,a)\} \right\}$$

Using $a - b \geq 0$ and the fact that $C'[s,t) = 0$ if $a \leq s$ we can write:

$$= \sup_{s \leq a \leq t} \left\{ \inf_{a-s \leq a-b \leq a-p} \{C[b,a) - R[b,a)\} \right\}^+$$

Taking into account the substitution $\lambda = a - b$ and the definitions of the arrival and service curves we obtain:

$$\leq \sup_{s \leq a \leq t} \left\{ \inf_{a-s \leq \lambda \leq a-p} \left\{ \beta^u(\lambda) - \alpha^l(\lambda) \right\} \right\}^+$$

We can argue on the bounds of the operators and obtain:

$$\leq \sup_{s \leq a \leq t} \left\{ \inf_{t-s \leq \lambda \leq -p} \left\{ \beta^u(\lambda) - \alpha^l(\lambda) \right\} \right\}^+$$
$$= \inf_{t-s \leq \lambda \leq -p} \left\{ \beta^u(\lambda) - \alpha^l(\lambda) \right\}^+$$

Finally, using the fact that p can be arbitrarily small, we find:

$$= \inf_{t-s \leq \lambda} \left\{ \beta^u(\lambda) - \alpha^l(\lambda) \right\}^+$$

□

In a similar way, we can derive bounds on the processed events.

A.4.3 Processed Events

Let us first derive an upper bound on the processed-event stream.

Cor. 2: **(Upper Bound on Processed Events)** *Given an event stream which is described by the arrival curves α^u, α^l and a resource described by the service curves β^u, β^l. Then the processed-event stream is bounded from above by the curve*

$$\alpha^{u'}(\Delta) = \min\left\{\sup_{0\le\lambda}\left\{\inf_{0\le\mu\le\lambda+\Delta}\{\alpha^u(\mu)+\beta^u(\lambda+\Delta-\mu)\}-\beta^l(\lambda)\right\}, \beta^u(\Delta)\right\} \tag{A.17}$$

Proof. Again, we suppose $B(p) = 0$ for an arbitrarily small time p. Then, we can write for all $p \le s \le t$:

$$R'[s,t) = R'[p,t) - R'[p,s)$$

$$= \sup_{p\le b\le s}\{C[p,b) - R[p,b)\}^+ - \sup_{p\le a\le t}\{C[p,a) - R[p,a)\}^+ + C[p,t) - C[p,s)$$

Since $C[p,p) - R[p,p) = 0$, the suprema are nonnegative and we can omit the maximization with zero:

$$= \sup_{p\le b\le s}\{C[p,b) - R[p,b)\} - \sup_{p\le a\le t}\{C[p,a) - R[p,a)\} + C[p,t) - C[p,s)$$

$$= \sup_{p\le b\le s}\left\{\inf_{p\le a\le t}\{R[b,a) + C[a,t) - C[b,s)\}\right\}$$

We now introduce the variable substitutions $\lambda = s-b$ and $\mu = a+\lambda-s$. In addition, we know that $a \ge b$. Therefore, we also know that $\mu \ge 0$. Using these substitutions and the definition of upper and lower curves yields:

$$= \sup_{0\le\lambda\le s-p}\left\{\inf_{0\le\mu\le\lambda+(t-s)}\{R[s-\lambda,\mu-\lambda+s) + C[\mu-\lambda+s,t) - C[s-\lambda,s)\}\right\}$$

$$\le \sup_{0\le\lambda\le s-p}\left\{\inf_{0\le\mu\le\lambda+(t-s)}\{\alpha^u(\mu)+\beta^u(\lambda+(t-s)-\mu)\}-\beta^l(\lambda)\right\}$$

As p can be arbitrarily small, we obtain:

$$\le \sup_{0\le\lambda}\left\{\inf_{0\le\mu\le\lambda+(t-s)}\{\alpha^u(\mu)+\beta^u(\lambda+(t-s)-\mu)\}-\beta^l(\lambda)\right\}$$

□

Now we derive a lower bound for the processed-event stream.

Cor. 3: **(Lower Bound on Processed Events)** *Given an event stream which is described by the arrival curves α^u, α^l and a resource described by the service curves β^u, β^l. Then the processed-event stream is bounded from below by the curve*

$$\alpha^{l'}(\Delta) = \min\left\{\inf_{0\le\mu\le\Delta}\left\{\sup_{0\le\lambda}\left\{\alpha^l(\mu+\lambda)-\beta^u(\lambda)\right\}+\beta^l(\Delta-\mu)\right\}, \beta^l(\Delta)\right\} \quad \text{(A.18)}$$

Proof. Using the same assumptions and arguments as in the proof of the upper processed-event curve $\alpha^{u'}$, we can write for all $p < s < t$:

$$\begin{aligned} R'[s,t) &= R'[p,t) - R'[p,s) \\ &= \sup_{p\le b\le s}\{C[p,b) - R[p,b)\} - \sup_{p\le a\le t}\{C[p,a) - R[p,a)\} \\ &\quad + C[p,t) - C[p,s) \\ &= \inf_{p\le a\le t}\left\{\sup_{p\le b\le s}\{R[b,a) - C[b,s) + C[a,t)\}\right\} \end{aligned}$$

We now introduce the variable substitutions $\lambda = s - b$ and $\mu = a - s$. In addition, we know that $a \ge b$. Therefore, we also know that $\mu + \lambda \ge 0$. In addition, if $a < s$ (or $\mu > 0$) then $a = b$ and therefore, $R'[s,t) = C[s,t)$. Using these substitutions and the definition of upper and lower curves yields:

$$\begin{aligned} &= \inf_{p-s\le\mu\le t-s}\left\{\sup_{0\le\lambda\le s-p}\{R[s-\lambda,\mu+s) - C[s-\lambda,s) + C[\mu+s,t)\}\right\} \\ &= \min\left\{\inf_{0\le\mu\le t-s}\left\{\sup_{0\le\lambda\le s-p}\{R[s-\lambda,\mu+s) - C[s-\lambda,s) + C[\mu+s,t)\}\right\}, \right. \\ &\qquad \left. C[s,t)\right\} \end{aligned}$$

Using the definitions of the arrival and service curves and noting that p can be arbitrarily small, we obtain:

$$\ge \min\left\{\inf_{0\le\mu\le t-s}\left\{\sup_{0\le\lambda}\left\{\alpha^l(\mu+\lambda)-\beta^u(\lambda)\right\}+\beta^l((t-s)-\mu)\right\}, \beta^l(t-s)\right\}$$

□

List of Publications

The following list summarizes the publications which are based on this thesis. The pertinent chapters of this thesis, which are the source of the publications, are given in brackets.

E. Wandeler, L. Thiele, M. Verhoef, and P. Lieverse. System Architecture Evaluation Using Modular Performance Analysis - A Case Study. In *1st International Symposium on Leveraging Applications of Formal Methods (ISoLA)*, October 2004.
(Chapter 2)

E. Wandeler, L. Thiele, M. Verhoef, and P. Lieverse. System Architecture Evaluation Using Modular Performance Analysis - A Case Study. *Software Tools for Technology Transfer*, 2005. In press.
(Chapter 2)

L. Thiele, E. Wandeler, and S. Chakraborty. A Stream-Oriented Component Model for Performance Analysis of Multiprocessor DSPs. *IEEE Signal Processing Magazine, Special Issue on Hardware/Software Co-design for DSP*, 22(3):38–46, May 2005.
(Chapter 2)

E. Wandeler, A. Maxiaguine, and L. Thiele. Performance Analysis of Greedy Shapers in Real-Time Systems. In *Design, Automation and Test in Europe (DATE)*, pages 444–449, March 2006.
(Chapter 3)

E. Wandeler, A. Maxiaguine, and L. Thiele. Quantitative Characterization of Event Streams in Analysis of Hard Real-Time Applications. In *10th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 450–459, May 2004.
(Chapter 4)

E. Wandeler, A. Maxiaguine, and L. Thiele. Quantitative Characterization of Event Streams in Analysis of Hard Real-Time Applications. *Real-Time Systems*, 29(2):205–225, March 2005.
(Chapter 4)

E. Wandeler and L. Thiele. Abstracting Functionality for Modular Performance Analysis of Hard Real-Time Systems. In *10th Asia South Pacific Design Automation Conference (ASP-DAC)*, pages 697–702, January 2005.
(Chapter 4)

E. Wandeler and L. Thiele. Characterizing Workload Correlations in Multi Processor Hard Real-Time Systems. In *11th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 46–55, March 2005.
(Chapter 4)

E. Wandeler and L. Thiele. Workload Correlations in Multi Processor Hard Real-Time Systems. *Journal of Computer ans System Sciences, Special Issue on Real-Time and Embedded Systems*, 2005. In press.
(Chapter 4)

E. Wandeler and L. Thiele. Real-Time Interfaces for Interface-Based Design of Real-Time Systems with Fixed Priority Scheduling. In *5th ACM Conference on Embedded Software (EMSOFT)*, pages 80–89, September 2005.
(Chapter 5)

E. Wandeler and L. Thiele. Interface-Based Design of Real-Time Systems with Hierarchical Scheduling. In *12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 243–252, April 2006.
(Chapters 5 and 6)

E. Wandeler and L. Thiele. Optimal TDMA Time Slot and Cycle Length Allocation for Hard Real-Time Systems. In *11th Asia South Pacific Design Automation Conference (ASP-DAC)*, pages 479–484, January 2006.
(Chapter 6)

E. Wandeler and L. Thiele. Real-Time Calculus (RTC) Toolbox.
`http://www.mpa.ethz.ch/Rtctoolbox`.
(Chapters 7 and 8)

It follows a list of publications that are not covered in this thesis.

L. Thiele and E. Wandeler. Performance Analysis of Embedded Systems. In *The Embedded Systems Handbook*. CRC Press, 2005.

L. Thiele, E. Wandeler, and N. Stoimenov. Real-Time Interfaces for Composing Real-Time Systems. In *6th ACM Conference on Embedded Software (EMSOFT)*, October 2006. To appear.

S. Chakraborty, Y. Liu, N. Stoimenov, L. Thiele, and E. Wandeler. Interface-Based Rate Analysis of Embedded Systems. In *27th IEEE Real-Time Systems Symposium (RTSS)*, December 2006. Under Review.

E. Wandeler, J. Janneck, E. A. Lee, and L. Thiele. Counting Interface Automata and Their Application in Static Analysis of Actor Models. *Journal on Software and Systems Modeling*, 2006. Under review.

E. Wandeler, J. Janneck, E. A. Lee, and L. Thiele. Counting Interface Automata and Their Application in Static Analysis of Actor Models. In *3rd International Conference on Software Engineering and Formal Methods (SEFM)*, pages 106–115, September 2006.

Curriculum Vitae

Name	Ernesto Wandeler
Date of Birth	1 December 1978
Citizen of	Gansingen (AG)
Nationality	Swiss

Education:

2003–2006	ETH Zurich, Computer Engineering and Networks Laboratory Doctor Thesis under the Supervision of Prof. Dr. L. Thiele
2001–2003	UNITECH International Graduation with UNITECH International Certificate
1997–2003	ETH Zurich, Department of Electrical Engineering and Information Technology Studies in Electrical Engineering and Information Technology 6 Months Visiting Scholar at UC Berkeley, USA 9 Months UNITECH Scholar at Imperial College, UK Graduation as Dipl. El.-Ing. ETH with Distinction Awarded the ”Willi Studer Preis” 2003 by ETH Zurich Awarded the ETH Medal for the Diploma Thesis by ETH Zurich
1989–1997	Mathematics and Science Gymnasium Basel Graduation with Matura Type C Awarded the ”Basler Maturandenpreis” by Novartis Ltd.

Professional Experience:

2003–2006	Research & Teaching Assistant at ETH Zurich
2001–2002	Software Engineer at SchlumbergerSema, Beijing, P.R. China
2000	Software Engineer at Power-One, Boston, USA
2000–2001	Substitute Teacher at Kirschgarten Gymnasium, Basel
1998–2000	Graduate Lecturer at ETH Zurich
1998	Engineering Trainee at Novartis, Basel