

Diss. ETH No.

# Efficiency and predictability in resource sharing multicore systems

A dissertation submitted to  
ETH Zurich

for the degree of  
Doctor of Sciences

presented by  
ANDREAS SCHRANZHOFER  
Dipl.-Ing., B.Sc. TU Graz  
born December 23, 1980  
citizen of Austria

accepted on the recommendation of  
Prof. Dr. Lothar Thiele, examiner  
Prof. Dr. Marco Caccamo, co-examiner

2011





Institut für Technische Informatik und Kommunikationsnetze  
Computer Engineering and Networks Laboratory

---

TIK-SCHRIFTENREIHE NR. 123

Andreas Schranzhofer

# Efficiency and predictability in resource sharing multicore systems



Eidgenössische Technische Hochschule Zürich  
Swiss Federal Institute of Technology Zurich

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

Diss. ETH No.

Prof. Dr. Lothar Thiele, examiner  
Prof. Dr. Marco Caccamo, co-examiner

Examination date: March 8, 2011

to Anna,  
and the little one we don't know yet.



# Contents

<b>Abstract</b>	<b>i</b>
<b>Zusammenfassung</b>	<b>iii</b>
<b>Acknowledgements</b>	<b>v</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Multicore and multiprocessor systems . . . . .	2
1.2 Power efficiency and adaptivity . . . . .	3
1.3 Timing predictability and adaptivity . . . . .	5
1.4 Contributions and Thesis Outline . . . . .	6
<b>2 Allocation on heterogeneous MPSoCs</b>	<b>13</b>
2.1 Introduction . . . . .	14
2.2 Related Work . . . . .	16
2.3 System Model . . . . .	18
2.3.1 Hardware Model . . . . .	18
2.3.2 Application and Scenario Specification . . . . .	20
2.3.3 Problem Definition . . . . .	23
2.3.4 Hardness . . . . .	26
2.4 Global static power-aware mapping problem . . . . .	27
2.4.1 Initial Solutions . . . . .	27
2.4.2 Task Remapping . . . . .	30
2.4.3 Complexity . . . . .	35
2.5 Dynamic power-aware scenario-mapping problem . . . . .	35
2.5.1 Scenario Sequence Generation . . . . .	35
2.5.2 Deriving templates and the hardware platform . . . . .	36
2.5.3 Online mapping . . . . .	40
2.5.4 Templates for different probability distributions . . . . .	41
2.6 Performance Evaluation . . . . .	41
2.6.1 Simulation Setup . . . . .	42
2.6.2 Global static power-aware mapping . . . . .	43
2.6.3 Dynamic power-aware scenario-mapping . . . . .	47
2.7 Chapter Summary . . . . .	51

<b>3</b>	<b>Interference in Resource Sharing MPSoCs</b>	<b>53</b>
3.1	Introduction . . . . .	54
3.2	Related Work . . . . .	57
3.3	System Model . . . . .	59
3.3.1	Superblock Models . . . . .	59
3.3.2	Resource Access Models . . . . .	63
3.3.3	Model of the Shared Resource . . . . .	65
3.4	Interference on shared resources . . . . .	65
3.4.1	Analysis Overview . . . . .	66
3.4.2	Analysis Methodolgy . . . . .	69
3.4.3	Sequential execution of superblocks . . . . .	70
3.4.4	Time-triggered execution of superblocks . . . . .	76
3.4.5	Resulting arrival curve . . . . .	77
3.5	Chapter Summary . . . . .	78
<b>4</b>	<b>Static arbitration on shared resources</b>	<b>81</b>
4.1	Introduction . . . . .	82
4.2	Related Work . . . . .	83
4.3	System Model . . . . .	85
4.3.1	Models of Tasks and Processing Elements . . . . .	85
4.3.2	Model of the Shared Resource . . . . .	85
4.4	Analysis Overview . . . . .	87
4.5	Worst-Case Completion Time of A Phase . . . . .	89
4.5.1	Terminologies and notations used for analysis . . . . .	89
4.5.2	Worst-case completion time for a dedicated phase . . . . .	91
4.5.3	Worst-case completion time for a general phase . . . . .	92
4.6	Timing Analysis for Superblocks and Tasks . . . . .	99
4.7	Schedulability among Models . . . . .	100
4.8	Experimental Results . . . . .	102
4.9	Chapter Summary . . . . .	107
<b>5</b>	<b>Hybrid arbitration on shared resources</b>	<b>109</b>
5.1	Introduction . . . . .	110
5.2	Related Work . . . . .	112
5.3	System Model . . . . .	113
5.3.1	Models of Tasks and Processing Elements . . . . .	113
5.3.2	Model of the Shared Resource . . . . .	114
5.4	Analysis Overview . . . . .	115
5.5	Analysis Methodology . . . . .	118
5.6	Analysis for a single phase . . . . .	121
5.6.1	Initialization Stage . . . . .	122
5.6.2	Analysis Stage . . . . .	123
5.6.3	Finalization . . . . .	127



5.6.4	Complexity . . . . .	129
5.7	Analysis for Superblocks and Tasks . . . . .	129
5.8	Simulations . . . . .	130
5.9	Chapter Summary . . . . .	133
<b>6</b>	<b>Conclusion</b>	<b>135</b>
6.1	Contribution . . . . .	135
6.1.1	Adaptive power-aware multiprocessor design . . . . .	135
6.1.2	Task models and interference in resource sharing systems . . . . .	136
6.1.3	Towards timing predictability in resource sharing systems . . . . .	136
6.2	Outlook . . . . .	137
6.2.1	Reducing Interference or Controlled Interference . . . . .	137
6.2.2	Massive multicore systems . . . . .	138
6.2.3	Multiple shared resources . . . . .	138
	<b>References</b>	<b>139</b>
<b>A</b>	<b>Toolbox</b>	<b>149</b>
A.1	Representing Interference . . . . .	149
A.1.1	Sequential execution of superblocks . . . . .	150
A.1.2	Time-triggered execution of superblocks . . . . .	151
A.2	Worst-Case Analysis for static arbitration . . . . .	152
A.3	Worst-Case Analysis for hybrid arbitration . . . . .	155
A.4	Integration with Project Partners . . . . .	157
	<b>List of Publications</b>	<b>159</b>
	<b>Curriculum Vitae</b>	<b>161</b>



# Abstract

Multiprocessor and multicore systems are of increasing importance for industrial applications in the control, automotive and avionic domain. Applications in these domains often perform mission critical functionalities. In other words, these applications perform tasks with hard real-time constraints. The demand on computational resources grows with the computational complexity of these functionalities, e.g., control algorithms. Traditionally, these requirements for computational resources have been met with higher execution frequencies and instruction level parallelism.

In order to satisfy rising computational demands, an industry wide shift to parallel computing, and thus multicore and multiprocessor systems, has taken place. The increased performance requirements are met by parallelization of threads and tasks on multiprocessor and multicore systems. Communication among tasks on different processing elements leads to contention on the communication fabric, e.g., buses or shared memories, and thus, to additional delays. This limits the gain that can be achieved by parallelization. Consider a system with multiple concurrently executing tasks on different processing elements. Furthermore, consider that each task requests access to the shared memory. Then it is clear that tasks have to wait their turn for accessing the shared resource, and thus experience a delay. This interdependency of tasks limits performance in general, and increases the complexity of deriving hard real-time guarantees in particular.

In this thesis, we propose methods to design and analyze multicore systems with respect to power efficiency and timing predictability. The major contributions are as follows:

- We propose an algorithm that optimizes the mapping of tasks to processing elements. It can handle applications that are characterized by multiple concurrently executing multi-mode applications. Each mode is associated with an execution probability. The target hardware platform is given as a library of processing element types. Each processing element type is specified with power and performance characteristics. The proposed algorithm derives mappings that minimize the average expected power consumption. We consider systems with dynamic and static (leakage) power consumption.

- We propose a dynamic power aware mapping approach by providing a set of pre-computed template mappings. Template mappings are computed offline, based on the set of feasible system states (denoted scenarios) and are stored on the system. A manager monitors system state transitions and chooses an appropriate template mapping to assign new tasks to their respective processing elements.
- We propose a task model, where tasks are sequences of superblocks. A superblock is a functional block that has a unique entry and exit point. Superblocks are associated with worst-case execution times (WCETs) and worst-case communication demands. We propose different resource access models, that exhibit differing amounts of uncertainty with respect to the resource access pattern of a superblock.
- A worst-case analysis framework for resource sharing systems is proposed that takes contention on the shared resource into account. The framework allows to analyze systems with static or adaptive arbitration policies on the shared resource. We apply the analysis to two protocols, namely Time Division Multiple Access (TDMA), as an example for static arbitration, and FlexRay, as an example for adaptive arbitration.
- We derive worst-case response time (WCRT) bounds for the proposed resource access models and show that separating communication and computation is advantageous to achieve timing predictable systems, i.e., tight WCRT guarantees.

# Zusammenfassung

In der Automobil-, Avionik- und Automatisierungsindustrie gewinnen Multiprozessor- und Multicoresysteme immer mehr an Bedeutung. Die Anwendungen werden komplexer und zahlreicher, und viele führen zeitkritische Funktionen aus. In anderen Worten, die Anwendungen haben strikte Echtzeitbedingungen. Mit der Komplexität der Aufgaben steigt auch deren Rechenintensität. Traditionellerweise begegnete man diesem Problem mit erhöhten Ausführungsfrequenzen und der Parallelisierung von Instruktionen.

Um den Anforderungen an Rechenleistung weiterhin gerecht werden zu können wird vermehrt auf Parallelisierung, also auf Multiprozessor- und Multicoresysteme, gesetzt. Neben Instruktionen werden auch Ausführungsthreads und Aufgaben (Tasks) parallelisiert, was zu einer weiteren Steigerung der Rechenkapazitäten führt. Kommunikation zwischen Tasks auf verschiedenen Prozessoren führt jedoch zu Zugriffskonflikten auf der gemeinsam genutzten Ressource, z.B. auf Bussen oder dem Hauptspeicher. Dieser Effekt wird als "Contention" bezeichnet und führt zu zusätzlichen Verzögerungen bei der Ausführung von Tasks. Daraus folgt, dass die Steigerung der Rechenkapazität aufgrund von Parallelisierung durch die notwendig werdende Kommunikation zwischen Tasks begrenzt ist. Man stelle sich ein System mit mehreren nebenläufigen Tasks auf verschiedenen Prozessoren vor, wobei jeder Task auf den gemeinsamen Speicher zugreifen will. Zugriffe der einzelnen Tasks werden der Reihe nach abgearbeitet, das heißt die Ausführungszeiten der Speicherzugriffe eines Tasks hängen von der Anzahl der Speicherzugriffe aller anderen Tasks ab. Diese Abhängigkeit der Tasks limitiert die Steigerung der Rechenleistung und erhöht die Komplexität von Analyseverfahren zur Bestimmung der ungünstigsten Ausführungszeit (worst case execution time).

Konkret werden in dieser Dissertation die folgenden Beiträge bezüglich dem Entwurf und der Analyse von Multicoresystemen im Hinblick auf Effizienz und Vorhersagbarkeit des Zeitverhaltens präsentiert:

- Wir schlagen einen Algorithmus zur Zuweisung von Aufgaben (Tasks) an Prozesselemente vor, der mehrere nebenläufige Applikationen mit mehreren Ausführungsmodi berücksichtigt. Jeder Modus hat eine

Ausführungswahrscheinlichkeit und die Hardware Plattform ist als Bibliothek von Prozessortypen bekannt. Der Algorithmus minimiert die durchschnittlich erwartete Leistungsaufnahme, unter der Annahme dass Leistungsaufnahme aufgrund von Leckspannung nicht vernachlässigt werden kann.

- Wir präsentieren eine Methode um die Aufgabenzuweisung zur Laufzeit zu bestimmen. Dazu werden eine Reihe von Musterzuweisungen zur Entwurfszeit berechnet und auf dem System als Bibliothek gespeichert. Die Berechnung berücksichtigt dabei verschiedene Systemzustände. Zur Laufzeit wird dann eine Aufgabenzuweisung, welche dem aktuellen Systemzustand entspricht, ausgewählt.
- Wir schlagen ein Model vor, in welchem Aufgaben als Sequenzen von Superblöcken betrachtet werden. Superblöcke werden mit der ungünstigsten Ausführungszeit (worst-case execution time) und der ungünstigsten Anzahl von Ressourcenzugriffen (worst-case number of access requests) spezifiziert. Ein Superblock ist eine funktionale Einheit, welche eine eindeutige Eintritts- und Austrittsmarke hat. Wir schlagen verschiedene Ressourcenzugriffsmodelle vor, welche sich durch einen unterschiedlichen Grad an Unsicherheit in Bezug auf das ungünstigste Verhalten von Superblocks unterscheiden. Die Modelle reichen vom "dedicated model", in welchem dedizierte Phasen für Kommunikation bzw. Berechnung reserviert sind, bis zum "general model", in welchem Kommunikation und Berechnung zu jeder Zeit und in jeder Reihenfolge stattfinden können.
- Wir präsentieren eine Berechnungsmethodik die das ungünstigste Szenario (worst-case) berechnet. Die Methodik kann statische sowie adaptive Zuweisungsverfahren der gemeinsam genutzten Ressource berücksichtigen. Als statisches Verfahren wird Time Division Multiple Access (TDMA), ein Multiplexverfahren mit statisch definierten Zeitscheiben, verwendet. Das FlexRay Zuweisungsverfahren, welches vor allem in der Automobilindustrie Anwendung findet, wird als adaptives Verfahren verwendet.
- Wir berechnen die ungünstigsten Ausführungszeiten für die vorgeschlagenen Ressourcenzugriffsmodelle und zeigen dass die Trennung von Berechnung und Kommunikation zu den besten Ausführungszeiten führt. Wir empfehlen das Modell DSS um im Zeitverhalten vorher-sagbare Systeme zu entwerfen.

## Acknowledgements

I would like to express my gratitude to Prof. Dr. Lothar Thiele for giving me the opportunity to work on this thesis. Thank you for your constant support, your patience and your indispensable advise.

Thanks to Prof. Dr. Marco Caccamo for co-examining my thesis and the fruitful cooperation over the last years.

I would like to thank my collaborator Prof. Dr. Jian-Jia Chen, with whom I worked very closely and who significantly contributed to the work presented in this thesis. Thanks also to Prof. Dr. Rodolfo Pellizzoni for the successful cooperation and our almost perfect acceptance rate.

I feel gratefulness for my colleges at TIK. I met many new friends here and I thank you for making this time so unforgettable. Thank you for running through the woods with me, for the parties, for the "Zvieri". Simply for all the fun we had together.

I would like to thank my parents, Andreas and Gertraud, for their never ending support and my sister Lucia and her daughter Sophia, my little god-daughter.

Last but not least, I would like to thank the most important person in my life. Thank you Anna, I love you. This thesis is dedicated to you and our unborn child.

Research presented in this thesis was funded in part by the European Community's Seventh Framework Programme FP7/2007-2013 under grant agreement no. 216008 project Predator.





# 1

## Introduction

This thesis presents novel approaches to design and analyze resource sharing real-time systems. Multicore and multiprocessor systems are of increasing significance in the design of embedded systems for industrial applications in general, and for the automotive and avionic industry in particular. A significant share of these systems perform safety critical functionalities, and thus functional as well as timing constraints have to be satisfied. Traditionally, the increased demand of computational resources was satisfied by higher execution frequencies and instruction level parallelism. Higher frequencies cause heat generation, and thus power dissipation. In order to satisfy rising computational demands, an industry wide shift to parallel computing, and thus multicore and multiprocessor systems, has taken place.

As a result, new design and analysis methods for real-time systems need to be developed. Firstly, shared resources, such as communication buses or shared memories, become the new performance bottleneck with respect to timing. Multiple concurrently executing applications on different processing elements request the same resource at the same time. This leads to contention on the shared resource, and thus increased delays. Hence, the response time of an application or a task might be significantly increased compared to the execution time of the same application and task on single processor systems. Secondly, the distribution of applications and tasks onto a multicore and multiprocessor system has a significant influence on both, timing behavior and efficiency (e.g., in terms of power consumption).

We propose an approach to assign applications to processing elements of a heterogeneous multiprocessor platform. The proposed approach takes power efficiency and utilization bounds on the processing elements into consideration. Furthermore, we propose models to access shared resources and

analyze their behavior in terms of timing predictability for static and adaptive arbitration policies on the shared resource.

In Section 1.1, we present a survey of current design approaches for multicore and multiprocessor systems, in terms of efficiency and timing predictability. In Section 1.2 we discuss the challenges and benefits of shared resources in terms of efficiency and adaptivity. Section 1.3 introduces the notion of predictability and presents the associated challenges related to resource sharing. The contributions and an outline of this thesis are given in Section 1.4.

## 1.1 Multicore and multiprocessor systems

Embedded systems, such as telecommunication and multimedia applications, are deeply embedded into our society, which is often denoted as information society. An increasing number of areas are controlled, monitored or operated by embedded systems. The applications are diverse, but can be characterized by an increasing demand on computational resource, stringent constraints on power consumption and timing constraints. In order to meet these constraints, thread- and task-level parallelism is required in addition to instruction level parallelism. Multicore and multiprocessor systems satisfy these demands.

In this thesis, we focus on on-chip parallelism as in Multiprocessor Systems-on-Chip (MPSoC) and multicore architectures. In MPSoCs, multiple processing elements are on the same chip die, and are connected via an on-chip interconnect. These systems are usually optimized for a particular application domain, such as telecommunication or multimedia applications. As a result, MPSoCs are often composed of heterogeneous processing elements. Each processing element in the MPSoC is optimized for a specific functionality, and their composition is optimized towards a certain application domain. Multicore processing elements consist of multiple computing cores, that execute concurrently and are tightly integrated. That is, typically each core has a local memory (e.g., L1-cache), while multiple cores share a common memory (e.g., L2-cache). This memory is local to the multicore processing element, but shared among its individual processing cores. Multiprocessor systems are composed of multiple central processing units (CPU) that share a (hierarchical) communication fabric (e.g., a bus). Cache, memory and I/O peripherals are accessed via this communication fabric. Each processing element in this system might be a single-core or a multicore architecture.

Multiprocessor Systems-on-Chip (MPSoCs) and multicore platforms have been widely applied for modern computer systems for the last decade. Wolf et al. [WJM08] give a thorough survey of MPSoC architectures. Among

others, they consider the effect of architecture on "Performance and Power Efficiency" and on "Real-Time Performance".

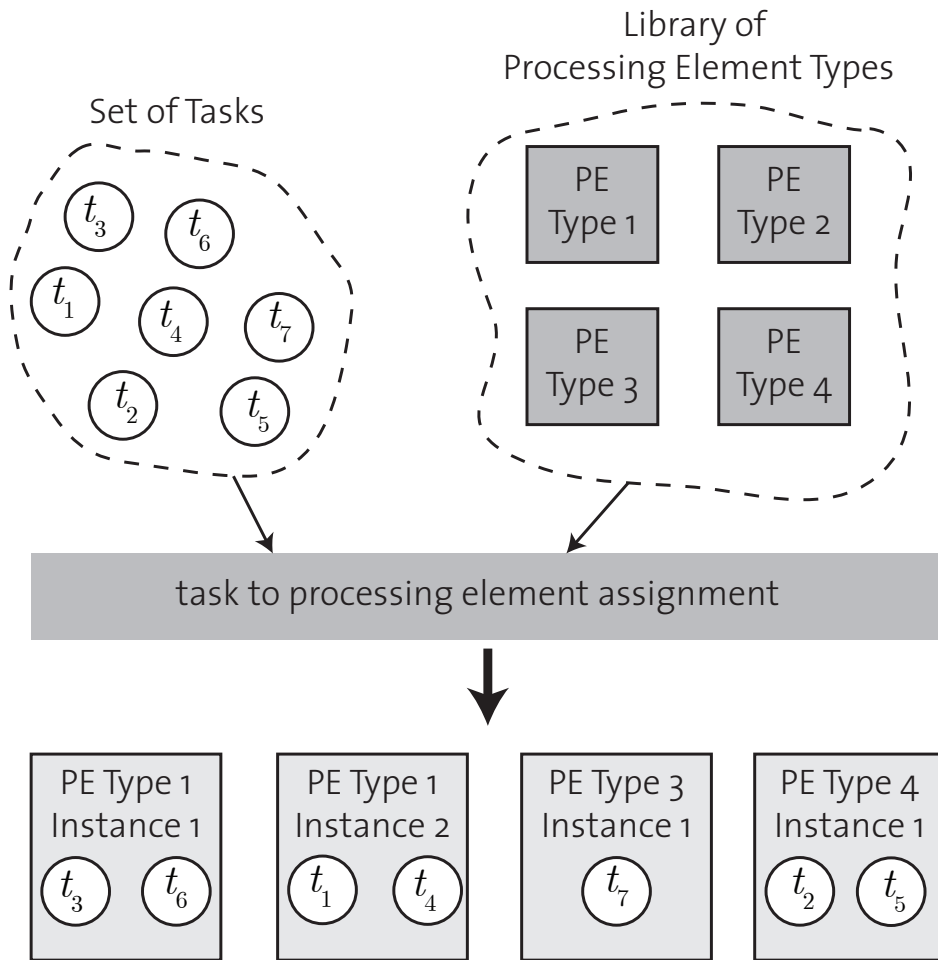
Multiprocessor systems for real-time systems have been studied in recent years in great detail. One of the most influential work, by Liu et al. [LL73], dates back as far as 1973. Several works target optimal task scheduling on multiprocessor systems, e.g., [AE10], [LESL10] and [Raj91]. An overview of resource access protocols in multiprocessor systems is presented by Brandenburg et al. [BCB<sup>+</sup>08].

## 1.2 Power efficiency and adaptivity

Embedded systems are often battery powered, like telecommunication handsets. Increased power consumption results in increased costs and space requirements, and thus systems in the automotive and avionic industry are required to be as power efficient as possible. In the following, we outline a number of parameters that influence power efficiency.

Firstly, the assignment of tasks to processing elements defines the utilization, and thus the dynamic power consumption of the processing element. Other parameters also influence the expected power consumption of a system. The architecture (heterogeneous or homogeneous), the manufacturing process (65nm, 45nm or 32nm) and the available dynamic power management (DVS, DFS) are key parameters of the power model. In heterogeneous multiprocessor systems, it is important to determine the type of processing element that fits the requirements and characteristics of an application. Figure 1.1 shows an example, where a set of tasks and a library of available processing unit types are given. A mapping approach needs to consider the power model, the available dynamic power management setup and the features of the tasks themselves in order to optimize the energy consumption. Wolf et al. [WJM08] give an overview of the historical development of MPSoCs and related design challenges, while Jalier et al. [JLJ<sup>+</sup>10] provide a comparison between homogeneous and heterogeneous multiprocessors systems for a telecommunication application.

In addition, the manufacturing process influences the power model. The power consumption of a processing element can be represented by (1) its dynamic power consumption and (2) its static, or leakage, power consumption. Dynamic power consumption is related to the utilization of a processing element, while static power is consumed as soon as a processing element is switched on, i.e., without performing any computation. Depending on the manufacturing process, the dynamic or the static part is the major contributor towards the overall power consumption of a processing element. Currently the 32nm manufacturing process is state-of-the-art, with the 22nm technology expected to be introduced by 2015 [fS]. Chips produced with



**Figure 1.1:** Task-to-processing element type assignment.

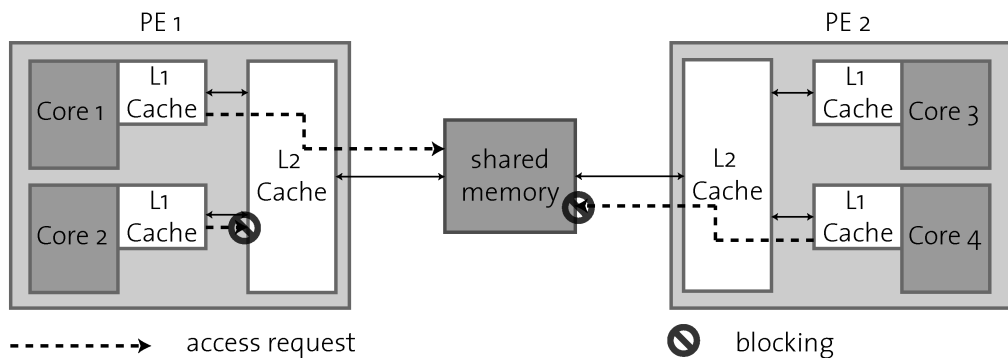
these manufacturing processes exhibit an increased significance of static power, i.e., the static power consumption exceeds the dynamic power consumption, even if the processor is fully utilized. Atitallah et al. give a survey of energy estimation techniques in [ANG<sup>+</sup>06].

Therefore, mapping approaches need to be redesigned to take the relation between dynamic and static power consumption into consideration. Dynamic power management protocols, such as dynamic frequency scaling (DFS) or dynamic voltage scaling (DVS), are used to reduce the dynamic power consumption, see [CK07a] [CRG08] [Che08]. However, when static power consumption cannot be neglected, techniques that switch off unused architectural elements (such as processing elements) gain increasing interest. These dynamic power management (DPM) techniques are particularly applied to CPUs and pose complex optimizations problems. Especially if real-time properties need to be satisfied.

Secondly, the ability of reassigning tasks and applications at runtime, to processing elements with reduced power consumption or increased computational resources, allows a system to adapt to changing environmental conditions. Adaptivity is of crucial importance for systems that exhibit mode-changes or need to adapt to different usage patterns. For example, mobile phones need to synchronize to base stations or fuel injection control applications need to increase their execution frequency with increasing speed. Jalier et al. [JLS<sup>+</sup>10] investigate dynamic mapping of a software defined radio (SDR) application, while Kim et al. [KSME05] consider dynamic mapping in order to save energy. Carvalho et al. [CMCM09] compare static and dynamic mapping approaches with respect to expected performance.

### 1.3 Timing predictability and adaptivity

The introduction of distributed and parallel computing, i.e., multicore and multiprocessor systems, yields additional challenges in terms of timing predictability. Mutually independent applications and tasks that execute concurrently result in an increased performance. However, accessing a shared resource, either due to necessary communication or synchronization between tasks, yields contention on the resource and results in additional blocking times. As a result, the available parallelism is reduced. See Figure 1.2 for an example of two multicore processors sharing a main memory. Once one processing element gains access to this main memory, any other device requesting access is blocked, and hence experiences a delay.



**Figure 1.2:** Blocking due to interference on the shared resource.

For applications in the automotive and avionic domain timely execution of applications is of utmost importance. The response time of tasks depend on processing capabilities of the processors and controllers, but also on delays that result from reading or writing to memory or other peripherals. In this thesis, we define timing predictability in relation to the worst-case response time (WCRT) of a system.

- A system that allows to derive its WCRT, is considered a timing predictable system.
- A system that allows to derive tight WCRTs is more predictable than a system that only allows to derive an untight WCRT.
- A system that does not allow to derive a meaningful WCRT is considered to be not timing predictable.

As a result, timing predictability of a resource sharing system depends on the protocol to access the shared resources. The amount of uncertainty in the protocol determines the untightness of a WCRT analysis approach. This uncertainty depends on (1) the analysis method and its underlying abstractions and assumptions and (2) the architecture and the task model of the system. With increasing uncertainty, the search space to find the worst-case grows and so does the computational complexity of corresponding analysis methodologies.

However, for applications in the aforementioned areas, these analysis methodologies are of crucial importance. First, in order to avoid costly warranty payments and re-design efforts, and second to be able to guarantee the correct behavior of safety critical tasks.

Thiele et al. [TW04] introduce the term timing predictability and describe some prerequisites, that are required in order to achieve a timing predictable system. Wilhelm [Wil05] presents an approach to derive bounds on the worst-case execution time (WCET) of complex systems with uncertainty. Reineke et al. [RGBW07] analyze the timing behavior of systems with different cache replacement policies. Edwards et al. [EKL<sup>+</sup>09] argue that timing repeatability is more important than timing predictability.

## 1.4 Contributions and Thesis Outline

In this thesis, we present a set of novel approaches to design and analyze resource sharing systems. We consider heterogeneous multiprocessor systems, where resource sharing happens on two levels. First, computational resources are shared among tasks. As a result, the allocation of tasks onto processing elements needs to respect utilization bounds in order to guarantee that tasks behave correctly. Due to the heterogeneity of the processing elements, it is not obvious how to place tasks on the different elements, such that efficiency is optimized. We focus on power efficiency, since many embedded systems are battery powered systems where life-span is an important issue.

Second, communication fabrics represent a shared resource and are requested by concurrently executing tasks and applications. As a result, con-

tention on the shared resource might lead to significantly increased worst-case and average case execution times. We consider systems where an unserved access request causes the issuing task or application to stall until the request has been served. For example, cache requests need to be served before the task and application can continue to execute. This results in stall times, that are related to the contention on the shared resource, i.e., that are independent of the tasks or the applications themselves, but that depend on the behavior of other actors in the system. Therefore, it is not trivial to derive the worst-case delay that results from the interference a task or application might suffer. We propose resource access models and conclude that separating communication and computation is required to achieve timing predictable resource sharing systems.

## Chapter 2: Allocation on heterogeneous MPSoCs

In Chapter 2 we propose a methodology to assign tasks to the processing elements of a heterogeneous multiprocessor platform, i.e., to share computational resources. In the proposed application model, multiple applications execute concurrently on the target platform. Each application is composed of multiple mutually exclusive execution modes, and each mode has an execution probability. As an example, consider a Software-Defined-Radio (SDR) platform where different applications represent different radio standards. For example, one application implements Wireless Local Area Network (WLAN) and another application implements Ultra-Wideband Radio (UWB). Then, each application has execution modes to synchronize to base stations or to send and to receive data. Measurement and profiling allows to derive average case execution frequencies for these modes. In our optimization methodology we consider these frequencies as execution probabilities. As a result, each application operates in a particular execution mode at any point in time. We denote the set of possible execution mode combinations as the set of *scenarios*. As an example, consider a SDR platform with two applications, namely WLAN and UWB. Furthermore, consider each application to be composed of a synchronization and a transmission mode. Then there are four possible scenarios. The transition from one scenario to another is performed by changing the execution mode of at least one application. A number of scenario transition is denoted *scenario sequence*. We consider an application model with a finite set of possible scenario sequences.

A library of processing element types is given, each type exhibiting different advantages and disadvantages. In other words, each processing element type is more suitable for a specific set of tasks, and less suitable for other tasks. As an example, consider a digital signal processor (DSP), that is well suited to perform matrix operations as required in telecommunication

applications. Performing this kind of operation on a general purpose CPU is possible, but results in increased execution time and power consumption. However, in some situations, it might be optimal to execute this task on the less power efficient CPU, since the DSP is already highly utilized or exhibits a large leakage power. This might result in an increased global power consumption or end-to-end execution time, although a local optimal decision has been taken. In this chapter, we present a novel approach to solve the problem of finding a power efficient task to processing element allocation, that respects a given utilization bound of the processing elements. The power model considers leakage and dynamic power consumption, as exhibited by chips manufactured with recent technologies, such as the 32nm manufacturing process. The contributions described in this chapter are as follows:

1. We show that there is no polynomial-time approximation algorithm with constant approximation factor for the task to processing element problem unless  $\mathcal{P} = \mathcal{NP}$ .
2. We propose a multiple-step approach to statically compute a task to processing element allocation.
3. We propose a dynamic mapping process that is described by an offline and an online part. The offline part computes template mappings for each scenario sequence by applying the static multi-step approach.
4. We propose an online manager, that observes mode changes of applications and chooses from a repository of precomputed template mappings at runtime.
5. Adaptivity to changing execution probabilities is introduced. Templates for different execution probabilities are computed and stored on the system.

### Chapter 3: Interference in Resource Sharing MPSoCs

In Chapter 3, we define the problem of interference on shared resources, like shared memory. We propose a task model, where tasks are sequences of superblocks. A superblock is a functional block that has a unique entry and exit point. However, different execution paths inside a superblock are possible. As a result, the sequential order of superblocks remains the same for any execution instance of a particular task. Each superblock is associated with its corresponding worst-case execution time (WCET) and its worst-case number of access requests to a shared resource.

Superblocks can be further specified by phases, where phases can represent implicit communication, computation, or both. As an example, a



superblock might have a dedicated phase to read or write data from the shared memory, where no computation can be performed. These phases are denoted *acquisition* or *replication* phase, respectively. Another phase, the *execution phase*, is reserved for local computation only, i.e., no communication can happen in this phase. Hence, communication and computation are confined to their respective dedicated phases. We denote this model as *dedicated* resource access model. The execution phase is associated with the WCET, while the acquisition and replication phases are denoted with their respective worst-case number of access requests.

Other models proposed in Chapter 3 are the *general* and *hybrid* resource access model. In the *general* model superblocks have no phases, i.e., communication and computation is not separated and can happen at anytime and in any order during the active time of a superblocks. In other words, a superblock modeled according to the general resource access model has a single *general phase*. This phase is associated with the WCET and the worst-case number of access requests. The *hybrid* model represents a trade-off between the two previous models. A superblock that is modeled according to the hybrid resource access model exhibits dedicated communication phases, where no computation can happen, i.e., an acquisition and replication phase. However, there is no dedicated phase to perform local computation only. Hence, these superblocks have a general phase where communication and computation can happen at anytime and in any order.

These models express a different amount of uncertainty. The dedicated model confines accesses to the shared resource to their respective dedicated phases. In the general model, computation and accesses to shared resource can happen anytime and in any order. Furthermore, we propose arrival curves as a metric to represent the access pattern of a set of superblocks on a particular processing element onto the shared resource. An upper arrival curve represents the maximum (worst-case) number of access requests that can happen in a particular time window of length  $\Delta$ . The contributions of this chapter are as follows:

1. We propose models to access the shared resource and models to schedule superblocks on processing elements.
2. We introduce arrival curves to represent the access pattern of superblocks that execute on a particular core.

#### **Chapter 4: Static arbitration on shared resources**

In Chapter 4, we investigate a static arbitration policy on the shared resource. This way, interference among the elements that compete for the shared resource is eliminated. Time division multiple access (TDMA) is a well known policy for systems that require a high degree of timing pre-

dictability. In TDMA, each processing element is assigned a time slot of fixed length, in which the resource is exclusively assigned to it. We consider systems with blocking/non-buffered access requests, i.e., once an access request has been issued, computation on the processing element can only continue as soon as this access request has been served. In this chapter we study the different models proposed in Chapter 3, combined with TDMA arbitration on the shared resource and their effect on the worst-case response time (WCRT). We conclude that the separation of communication and computation is a key element for achieving timing predictable systems. However, we show that time-triggered execution of superblocks, as opposed to sequential execution, decreases the performance of a system with respect to its WCRT.

1. We propose a worst-case analysis framework for shared resources with TDMA arbitration policy.
2. We derive the schedulability relation of the proposed models and recommend the *dedicated* model for resource sharing systems. In this model, computation and communications is separated, while superblocks execute sequentially.

## Chapter 5: Hybrid arbitration on shared resources

Chapter 5 extends the results in Chapter 4 towards hybrid arbitration policies on the shared resource. The arbiter is inspired by the FlexRay protocol, used in the automotive industry. In the FlexRay protocol, a static segment assigns time slots to processing elements and a dynamic segment allows for resource competition. As a result, there is a guaranteed service per arbitration cycle, namely the time slot in the static segment, and a phase with interference, namely the dynamic segment. The arbitration policy in the dynamic segment follows the First-Come-First-Serve (FCFS) policy. Hence, a processing element can be assigned additional time slots during the dynamic segment, depending on the behavior of the other elements in the systems. This arbitration policy is more flexible than the purely static approach presented in Chapter 4, but also incurs a computationally more complex analysis approach, since interference during the dynamic segment has to be taken into account. In this chapter, we present the following contributions:

1. We present an algorithm to derive an upper bound of the worst-case response time (WCRT) for superblocks and tasks, considering the FlexRay arbitration protocol on the shared resource.

2. We present experimental results and show that minor changes in the arbiter might lead to significant increases in the resulting WCRT of superblocks and tasks.

### **Appendix A: Toolbox**

In Appendix A, we present a toolbox that implements the analysis framework for timing predictable systems proposed in this thesis. The toolbox assumes parameters like the maximum and minimum amount of access requests or the required computation time to be given. We describe the integration of the toolbox with a tool to formally analyze the WCET, i.e., aiT from AbsInt. In addition, examples from a realistic automotive application scenario are described. This case study has been defined in cooperation with Bosch<sup>1</sup>.

---

<sup>1</sup>Due to confidentiality reasons, the exact nature of example applications was not revealed to us.



# 2

## Allocation on heterogeneous MPSoCs

Increasing computational demand and reduced time-to-market requirements for industrial embedded systems gave rise to multiprocessor and multicore platforms, such as Multiprocessor System-On-Chips (MPSoCs). An increasing share of these embedded systems are mobile, battery powered systems, that are expected to execute multiple functionalities concurrently. Thus, computational resources have to be shared by multiple, independently executing applications. While responsiveness of concurrent applications is a crucial design issue, power consumption is of equal importance. These issues hold for mobile systems in particular due to their constraint source of energy. In general, industrial embedded systems perform increasingly complex tasks, like control algorithms in automotive and avionic systems, and hence require increased computational resources, while properties like power consumption, space requirements or heat built-up affect the life-time of a system.

In this chapter, we introduce an application model that is capable of representing multiple concurrently executing applications sharing a single hardware platform. Applications have a number of mutually exclusive execution modes, each represented by a set of tasks, and their corresponding execution probabilities. Based on this representation we show how to derive a static allocation of tasks to processing elements, such that the overall power consumption is minimized, taking leakage and dynamic power consumption into consideration.

Embedded systems interact with the environment, by adapting the execution frequency or the execution mode of a particular application. Static

allocation of tasks to processing elements results in unforeseeable system behavior once the anticipated environment changes. Therefore, dynamic adaptation of the task allocation to changing use cases or changing environmental conditions is required to maintain responsiveness and power efficiency of systems. To this end, we introduce an approach that dynamically allocates tasks to processing elements. It takes advantage of structural information about the applications, i.e., their modes of execution, mutual exclusiveness among modes of a single applications and their execution probability. We show that the problem to allocate tasks to processing elements is hard and propose heuristic solutions to the problem. Experimental evaluation shows the competitiveness of our proposed approach in comparison to the optimal solution of the problem.

## 2.1 Introduction

Multiprocessor System-on-Chips (MPSoCs) typically are composed of multiple processors (processing units), memories, and a communication infrastructure. Heterogeneous MPSoCs contain different types of processing units, some specialized for a specific purpose (e.g., Digital Signal Processors (DSP)), others for general purposes. Therefore, system designers can take advantage of their properties when mapping tasks to specific processor types and optimize criteria such as computational performance, cost and energy consumption.

Energy awareness and power management are important design issues for any embedded system in general, and for battery powered and mobile systems in particular. Power consumption not only influences the battery lifetime of mobile devices or the cost of operating a server farm, but also influences the lifespan of systems, due to increased heat build-up. It is caused by a dynamic and a static part [JPG04, CHK06]. In nano-meter manufacturing, leakage current significantly contributes to the static power consumption and cannot be neglected. Dynamic power consumption is related to the utilization of a processing unit.

In multimedia applications, such as mobile phones and Software-Defined-Radio (SDR) systems, it is uncommon to assume a fixed task set, since these applications (a) are usually composed of multiple execution modes and (b) have heavily varying execution patterns over their lifespan. In this chapter, we study how to map applications to processing units, where applications are defined as sets of tasks, and run in one out of a set of modes. An execution probability is assigned to each mode. Concrete activation and active times are not known a priori. *Scenarios* define the possible combinations of modes that execute concurrently. Consequently, a mode change of an application results in a transition to another sce-

nario. As a result, sequences of scenario transitions can be identified and a particular scenario can be reached by a sequence of mode changes. This model is common in various application domains such as multimedia processing, media terminals, mobile phones, and software defined radio (SDR) [LLW<sup>+</sup>06, LKMM07, MVB07, SWvdV08], just to name a few.

For example, a single SDR application consists of a set of tasks and several of these applications are typically executed concurrently. In contrast, modes of a particular application cannot execute concurrently. A single SDR application often represents a signal or media processing algorithm which involves the parallel execution of tasks. Often, the corresponding execution times, power consumption, and rate characteristics have been specified or can be obtained very accurately. The underlying hardware platform is given as a library of available processing unit types. Deriving the actual hardware platform by instantiating processing units is part of the mapping problem.

Given a set of applications and the underlying heterogeneous MPSoC hardware platform as a library of available processing unit types, there is a large degree of freedom in mapping the individual tasks to the allocated processing elements. Adapting to varying resource requirements allows to optimize the average power consumption dynamically. The mapping of tasks to the computational resources admits a fine-grained power management by switching off processing units that are not used or slowing down processing units that are not fully utilized. The mapping process creates a feasible hardware platform from a library of processing unit types by instantiating processing units and determines a task assignment that satisfies the computational demands of the scenarios while minimizing the static and dynamic power consumption.

In this chapter we consider two approaches to compute the task to processing element assignment. First, tasks are assigned to one particular processing element, independently from the actual *scenario*. We denote this problem as the *global static power-aware mapping* problem. This approach requires little memory to store the mapping, but results in increased power consumption once the system diverges from the anticipated usage pattern, i.e., once the modes execution probabilities change. Second, depending on the current scenario, a new task is assigned to different processing elements. We denote this problem as the *dynamic power-aware scenario-mapping* problem. This approach requires memory to store different precomputed template mappings for each scenario, but allows to adapt the task allocation to different usage patterns, i.e., to store different mappings for different execution probabilities. We assume tasks to be resident, i.e., once a task is mapped onto a processing element, it cannot be remapped to any other processing element. As a result, a particular mapping for each sequence of scenario transitions needs to be computed, taking into account the possible future developments of the system.

The objective of both problems is to derive an optimal hardware platform and an optimal task mapping, such that the average power consumption is minimized while satisfying the execution constraints of all possible scenario sequences.

The major contributions of this chapter are:

- We show that there is no polynomial-time approximation algorithm with constant approximation factor for the task to processing element problem unless  $\mathcal{P} = \mathcal{NP}$ .
- We propose a multiple-step approach to solve the global static power-aware mapping problem in polynomial time by (1) computing an initial solution which assigns the tasks to their most effective processing unit types and (2) applying a greedy heuristic algorithm to remap tasks, thereby reducing the expected average power consumption.
- We propose a dynamic mapping process that is described by an offline and an online part. The offline part computes template mappings for each scenario sequence by applying the multi-step approach developed for the global static power-aware mapping problem. In the online part, a manager observes mode changes and chooses an appropriate precomputed template.
- Adaptivity to changing execution probabilities is introduced. Templates for different execution probabilities are computed and stored on the system.
- Experiments show the effectiveness of the proposed algorithm in terms of expected average power consumption and computation time.

The rest of this chapter is organized as follows: Section 2.2 gives an overview of related problems and approaches and Section 2.3 introduces the models and defines the studied problem. Section 2.4 presents the proposed algorithms for the *global static power-aware mapping* problem, while Section 2.5 introduces the adapted algorithms to solve the *dynamic power-aware scenario-mapping*. Section 2.6 provides experiments for performance evaluation using SDR applications and Section 2.7 concludes this chapter.

## 2.2 Related Work

Work highly related to the probabilistic application model is done by Kim et al. [KBDV08] and Schmitz et al. [SAHE05]. Specifically, in [KBDV08] a heuristic algorithm is proposed, that reduces the dynamic energy consumption, which is related to utilization. This is done by adding processing



elements up to an area constraint, thereby reducing the average utilization. Their application model considers probabilities of execution for modes. The power model presented in this chapter considers static and dynamic power as part of the objective to be minimized. In contrast, [KBDV08] only considers dynamic power consumption in the objective and an area constraint that has to be satisfied. Schmitz et al. [SAHE05] consider probabilistic execution of multi-mode applications. They propose a genetic algorithm and four different mutation strategies to reduce the energy dissipation.

Power-aware and energy-efficient scheduling for multiprocessor systems has been explored widely in recent years in both academics and industry, especially for real-time systems, e.g., [CHK06, AEA05, AY03], whereas [CK07b] provides a comprehensive survey. However, only a few results have been developed for power-awareness or energy-efficiency in heterogeneous multiprocessor systems. For example, in [CWSC08, CK06, CSZ<sup>+</sup>05, RFSW06] heuristics and approximation algorithms to minimize the dynamic energy consumption are studied, considering dynamic voltage scaling (DVS) systems.

In nano-meter manufacturing, leakage current contributes significantly to the power consumption of a system e.g., [JPG04]. Xu et al. [XZR<sup>+</sup>05] and Chen et al. [CHK06] explore how to execute tasks and study DVS techniques to turn off processors in homogeneous multi-processor systems. These works focus on developing schedules for a fixed set of tasks, e.g., a single application with known activation and execution time or a set of periodically executing tasks, e.g. [CST09].

There are several research results for energy-efficient and power-aware designs in heterogeneous multiprocessor systems with non-negligible leakage power consumption, see e.g. [RFSW06, CST09]. In these approaches, static usage scenarios are assumed and probabilities of applications are not considered. As a result, all modes could run concurrently, which is not the case in our proposed system model. This not only results in an over-dimensional MPSoC platform but also in a non-optimal task mapping which overestimates the *average power consumption*. It is implausible for the considered application domain, e.g., SDR systems, to assume that all applications are active all the time.

Dynamic mapping methodologies have been studied more recently. These studies basically split in two directions. Some tackle the problem by defining efficient heuristics to assign new arriving tasks onto processing units at runtime, e.g., [MMB07, MMBM05]. Online heuristics cannot guarantee schedulability, e.g., Moreira et al. evaluate their approaches by computing the mapping success rate in [MMB07] and [MMBM05]. Others analyze applications offline and compute schedules and allocations that are then stored on the system, e.g., [BBM08, MCR<sup>+</sup>06, YO09]. In [BBM08] Benini et al. propose to compute system configurations and derive task allocations

and schedules for each of them. At run-time, transitions between allocations are assigned a migration cost. This work assumes that tasks can be migrated from one processing unit to another, once the system configuration changes. The decision whether tasks are migrated or not depends on precomputed migration costs. We assume tasks to be resident, i.e., task migration is prohibited. This increases the complexity of the problem, since we have to consider possible future scenario transitions when we assign a task to a processing unit. Execution probabilities are neglected in [BBM08], which might lead to adverse system configurations. Migration costs might be low compared to the increased dynamic power dissipation that results from not reallocating tasks that execute very frequently.

## 2.3 System Model

In this section, the hardware as well as the application models with the underlying assumptions and terminology is introduced. The hardware model describes processing units selected from a set of processing unit types. Processing units are described by their computational resources and by their power consumption. The application model considers multiple concurrently executing applications, composed of tasks. Applications execute in one mode out of a set of modes. Therefore, for each application, one single mode is executing at a time. Table 2.1 gives an overview of regularly used variables. A formal problem statement and a complexity analysis of the optimal solution are given at the end of the section.

### 2.3.1 Hardware Model

The hardware platform is based on a set ( $\mathcal{P}$ ) of available processing unit (PU) types. A PU type  $p \in \mathcal{P}$  is characterized by its *available computational resources per time unit*  $\lambda$ , e.g., measured in terms of operations or execution cycles per time unit, and its *static and dynamic power consumption*  $\sigma$  and  $\delta$ , respectively. Consider a task with a *computational demand* of  $\gamma$  (measured in operations or executions per time unit) on a specific resource type  $p_j$  with *available computational resources*  $\lambda$ . Over a time interval of length  $\tau$  the computational demand  $\gamma$  and available computational resources  $\lambda$  accumulate to  $\tau\gamma$  and  $\tau\lambda$ , respectively. Under the assumption that a task performs correctly as long as its computational demand is satisfied, the utilization of a resource instance by this task can be formulated as  $\frac{\gamma}{\lambda}$ .

Static power consumption  $\sigma$  describes the leakage power consumed by a processing unit type, independent of whether tasks are executed or not. Dynamic power consumption  $\delta$  describes the additional power consumed by a processing unit type depending on its utilization.

Symbol	Meaning
$\mathcal{P}$	Library of PU types
$p_j$	Processing unit type $j$
$\hat{p}_{j,k}$	instance $k$ of PU type $p_j$
$\sigma_j$	static power consumption of PU type $p_j$
$\delta_j$	dynamic power consumption of PU type $p_j$
$\lambda_j$	computational resources of PU type $p_j$
$\mathcal{S}$	Set of scenarios
$S_m$	scenario $m$
$\chi_m$	Probability of scenario $S_m$
$\mathcal{A}$	Set of concurrent Applications
$A_\ell$	Application $\ell$
$\mathcal{M}_\ell$	Set of modes constituting Application $\ell$
$\mu_n$	Mode $n$ of an application
$\hat{\chi}_n$	Probability of mode $\mu_n$
$Z_{m,j,k}$	binary variable to indicate scenario $S_m$ on $\hat{p}_{j,k}$
$\mathcal{C}$	Set of scenario sequences
$\mathcal{T}$	Set of Tasks
$\gamma_{i,j}$	computational resource demand of task $t_i$ on PU type $p_j$
$u_{i,j}$	utilization of task $t_i$ on PU type $p_j$
$\psi_i$	Probability of task $t_i$
$M_{i,j,k}$	binary variable to indicate mapping of task $t_i$ to $\hat{p}_{j,k}$
$\theta_{i,m}$	binary variable to indicate task $t_i$ to scenario $S_m$ assignment
$c_r$	scenario sequence $r$ , $c_r \in \mathcal{C}$

**Table 2.1:** Overview of regularly used variables.

As an example, suppose that a PU type  $p_j$  has a utilization of  $\alpha$ , i.e., tasks are executing for an  $\alpha$  fraction of the time span  $\tau$ . Then,  $\alpha\delta_j + \sigma_j$  is the *average power consumption* of the processing unit. The corresponding energy consumption for  $\tau$  time units is  $\tau \cdot (\alpha\delta_j + \sigma_j)$ .

Given the available PU Types, a concrete hardware platform is constructed by instantiating processing units with unit types from  $\mathcal{P}$ . Any number  $k \in \mathbb{N}$  of instances is allowed for each PU type. Constructing a feasible hardware platform is part of the mapping problem.

Application	Modes	Tasks	Probability of modes
$A_1$ (DVB-H)	$\mu_1$ ( <b>sync</b> )	FFT <sub>1</sub> phase_shifting <sub>1</sub> noise_detection <sub>1</sub> frequency_shifting <sub>1</sub> ⋮	$\hat{\chi}_1$
	$\mu_2$ ( <b>receive</b> )	FFT <sub>1</sub> phase_shifting <sub>1</sub> payload_decoding <sub>1</sub> video_decoding <sub>1</sub> ⋮	$\hat{\chi}_2$
$A_2$ (WLAN)	$\mu_3$ ( <b>send</b> )	FFT <sub>2</sub> signal_generation encrypting encoding frequency_shifting <sub>2</sub> ⋮	$\hat{\chi}_3$
	$\mu_4$ ( <b>receive</b> )	FFT <sub>2</sub> phase_shifting <sub>2</sub> decrypting payload_decoding <sub>2</sub> frequency_shifting <sub>2</sub> ⋮	$\hat{\chi}_4$

**Table 2.2:** An example for different modes in DVB-H and WLAN.

### 2.3.2 Application and Scenario Specification

An application  $A$  is described as a set of nodes, each node representing a task. A task  $t_i$  in the given task set  $\mathcal{T}$  is described by its computational resource demand  $\gamma_{i,j}$  per time unit on a given PU type  $p_j \in \mathcal{P}$ . The utilization  $u_{i,j}$  of a task  $t_i$  mapped onto PU type  $p_j$  is given as

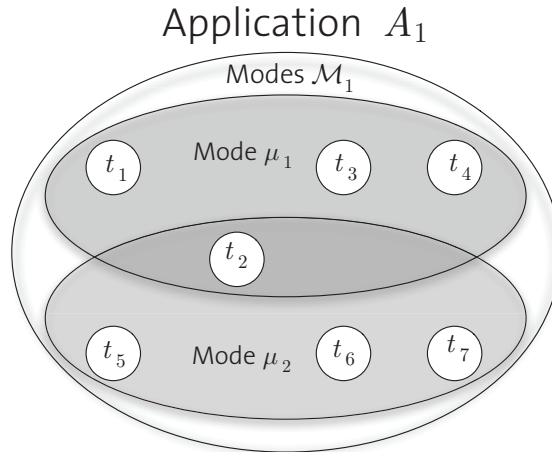
$$u_{i,j} = \frac{\gamma_{i,j}}{\lambda_j}. \quad (2.1)$$

By definition, the total resource demands of tasks mapped on a processing unit cannot exceed the available computational resources. As a result, the total utilization of tasks mapped on a processing unit is constrained not to exceed 100%. For each task  $t_i$ , we assume that there is at least one PU type

$p_j$  with utilization  $u_{i,j} \leq 100\%$ ; otherwise, there is no feasible solution for completing the task in time.

$\mathcal{A}$  represents the set of concurrent applications executing on the hardware platform. An application  $A_\ell \in \mathcal{A}$  is described by the given set of tasks  $\mathcal{T}$ .  $A_\ell$  is constituted by a set  $\mathcal{M}_\ell$  of modes. Each mode defines a use-case of an application, and, hence, has its own functionality. In other words, for an application  $A_\ell \in \mathcal{A}$ , a mode  $\mu_k \in \mathcal{M}_\ell$  defines the active tasks, see Figure 2.1 for an example. Applications have sets of initial and final modes. Initial modes define the start of an application. In Figure 2.2, applications are composed of 3 modes. Modes I1 and I2 are the initial and modes R1 and R2 are the final modes of applications  $A_1$  and  $A_2$  respectively.

Tasks can be active in multiple modes of an application, see task  $t_2$  in Application  $A_1$  in Figure 2.1. Conclusively only one mode of an application can be active at a time. As an example consider an application implementing a radio standard, such as Wireless Local Area Network (WLAN). The application might be described by modes such as *synchronize* or *receive* and there might be a functional dependency. Such an exclusion condition is common in SDR (Software Defined Radio) applications, where tasks such as Fast Fourier Transform (FFT) and decode [SKG<sup>+</sup>07, MVB07] are shared by multiple modes. This situation is depicted for sync/send and receive in Table 2.2, where Task FFT<sub>1</sub> is shared among modes  $\mu_1$  (**sync**) and  $\mu_2$  (**receive**) in Application  $A_1$  (DVB-H) and Task FFT<sub>2</sub> is shared among modes  $\mu_3$  (**send**) and  $\mu_4$  (**receive**) in Application  $A_2$  (WLAN).



**Figure 2.1:** Application (e.g., WLAN) with 2 modes (e.g., send and receive) and 7 tasks.

Modes of an application  $A_\ell \in \mathcal{A}$  and its valid mode transitions are represented by a directed graph  $G_\ell = (E_\ell, V_\ell)$ , where each node  $v_i \in V_\ell$  of the graph describes a mode and each edge  $e_i \in E_\ell$  describes a valid mode change, see Figure 2.2 for example.

Applications can change their modes anytime without affecting the other applications. Thus, there are various combinations of active modes, as each mode of an application can execute concurrently with any other mode of other applications. One combination of active modes for the set of concurrent applications  $\mathcal{A}$  is defined as a *scenario*  $S$  in this chapter. All possible scenarios can be derived by computing the cross product of the graphs representing the applications, i.e.,  $G_\pi = (E_\pi, \mathcal{S}) = \prod_{\forall \ell} G_\ell$ . Therefore,  $\mathcal{S}$  is the set of scenarios and a node  $S_m$  in  $\mathcal{S}$  is a scenario. A directed edge in graph  $G_\pi$  represents a possible transition from one scenario to another.

As an example, suppose that  $\mathcal{A}$  consists of two applications  $A_1$  and  $A_2$  represented by graphs  $G_{A_1} = (E_{A_1}, V_{A_1})$  and  $G_{A_2} = (E_{A_2}, V_{A_2})$  respectively. The resulting cross product  $G_\pi = (E_\pi, \mathcal{S})$  is  $G_{A_1} \times G_{A_2}$  as shown in Figure 2.3. The initial and final modes of applications define the initial and final modes of graph  $G_\pi$ . The number of nodes in graph  $G_\pi$  is  $|\mathcal{S}| = |V_{A_1}| \cdot |V_{A_2}|$ .

A path through the graph  $G_\pi(E_\pi, \mathcal{S})$  is a sequence of transitions, starting at node  $v_i \in \mathcal{S}$  and leading to another node  $v_j \in \mathcal{S} \setminus \{v_i\}$ , following the edges in  $E_\pi$ .

Thus, a path describes a sequence of scenarios and the set of all feasible paths is denoted by  $\mathcal{C}$ . This includes the assumption that mode changes are ordered, i.e., they do not happen at exactly the same time. As an example, consider the cross product in Figure 2.3 and the mode transitions **start 1** and **start 2**. Executing these transitions leads to node (S1, S2) no matter in which order they are executed. However, their order of execution determines the path through the graph  $G_\pi$  in Figure 2.3 that has to be considered.

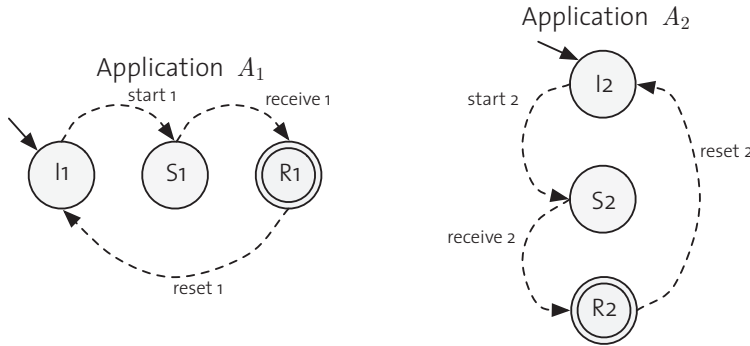
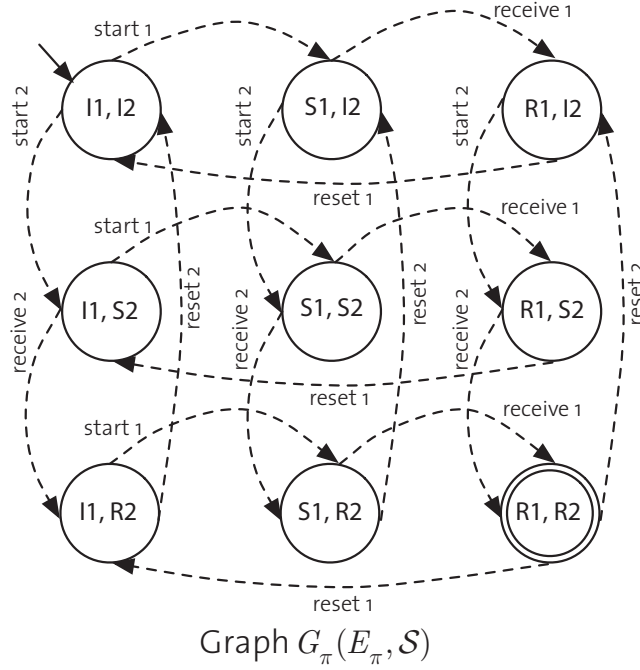


Figure 2.2: Graphs representing applications  $A_1$  and  $A_2$

Modes of an application  $A_\ell$  are denoted  $\mathcal{M}_\ell$  and each mode  $\mu_n \in \mathcal{M}_\ell$  has an execution probability, such that the share of time a mode  $\mu_n$  is active is denoted as  $\hat{\chi}_n$ . Modes of different applications are statistically independent,



**Figure 2.3:** Cross product  $G_\pi = G_{A_1} \times G_{A_2}$  representing all possible scenarios.

and therefore the probability of a scenario  $S_m \in \mathcal{S}$  is derived as the product of its constituting modes:

$$\chi_m = \prod_{\mu_n \in S_m} \hat{\chi}_n. \quad (2.2)$$

Conclusively, the execution probability  $\psi_i$  for a task  $t_i \in \mathcal{T}$  can be computed, where

$$\psi_i = \sum_{S_m: t_i \text{ is active in scenario } S_m} \chi_m. \quad (2.3)$$

### 2.3.3 Problem Definition

The problem explored in this chapter is to find a mapping of tasks in  $\mathcal{T}$  onto a hardware platform which consists of processing units from a given set of PU types  $\mathcal{P}$ . The selection of used processing unit types and the corresponding number of processing units is part of the problem. As described above, applications are characterized by their probabilities of execution. Therefore, the objective is to minimize the *average expected power consumption*. Once the time span of the system execution is known, the expected average energy consumption can be computed.

Besides selecting the optimal number of processing units, the mapping needs to determine the binding of a task  $t_i$  to an allocated processing unit  $\hat{p}_{j,k}$  of PU type  $p_j \in \mathcal{P}$ . A task is mapped onto a PU  $\hat{p}_{j,k}$  while respecting the maximum utilization constraint for all possible scenarios.

We assume that the number of possible instances  $k$  per PU type  $p_j$  is limited to be no more than  $F_j$  and for each task  $t_i$  there is at least one PU type  $p_j$  on which it can be executed, i.e.,  $u_{i,j} \leq 1$ . Hence, there exists a feasible solution to the mapping problem and any feasible solution will at most use  $|\mathcal{T}|$  instances of PU type  $p_j \in \mathcal{P}$ . Therefore, we consider  $F_j \leq |\mathcal{T}|$ .

The binary variables  $M_{i,j,k}$  indicate which processing unit task  $t_i$  is mapped to. Let  $M_{i,j,k} = 1$  if task  $t_i$  is assigned to PU  $\hat{p}_{j,k}$  and  $M_{i,j,k} = 0$  otherwise. Once we have  $M_{i,j,k} = 1$ , task  $t_i$  consumes a portion of the dynamic power  $\delta_j$  on PU  $\hat{p}_{j,k}$ . This portion is related to the task utilization  $u_{i,j}$  and its probability of execution  $\psi_i$ .

Furthermore,  $\theta_{i,m} = 1$  indicates that task  $t_i$  is present in scenario  $S_m$ , and  $\theta_{i,m} = 0$  otherwise. The binary variable  $Z$  indicates which processing units are involved in which scenarios, i.e., whether they need to execute at least one task in a specific scenario. We define  $Z_{m,j,k} = 1$  if there exists a task  $t_i$  mapped onto PU  $\hat{p}_{j,k}$  (i.e.,  $M_{i,j,k} = 1$ ) such that  $\theta_{i,m} = 1$  (i.e., it is present in scenario  $S_m$ ) and  $Z_{m,j,k} = 0$  otherwise. Once we have  $Z_{m,j,k} = 1$ , static power  $\sigma_j$  is consumed on  $\hat{p}_{j,k}$  whenever scenario  $S_m$  is executed, i.e., with probability  $\chi_m$ . The total utilization of the tasks in scenario  $S_m$  mapped onto PU  $\hat{p}_{j,k}$  is constrained to be no more than 100% when  $Z_{m,j,k} = 1$ , or 0% when  $Z_{m,j,k} = 0$ .

As a result, the expected average power consumption for a mapping described by  $M$  and  $Z$  can be determined as in Equation (2.4a), where the first and the second term represents the static and the dynamic power consumptions, respectively. The optimization problem can then be phrased by the following integer linear programming (ILP):

$$\min \sum_{S_m \in \mathcal{S}} \sum_{p_j \in \mathcal{P}} \sum_{k=1}^{F_j} \chi_m \sigma_j Z_{m,j,k} + \sum_{t_i \in \mathcal{T}} \sum_{p_j \in \mathcal{P}} \sum_{k=1}^{F_j} u_{i,j} \delta_j \psi_i M_{i,j,k} \quad (2.4a)$$

s.t.

$$\sum_{t_i \in \mathcal{T}} \theta_{i,m} M_{i,j,k} u_{i,j} \leq Z_{m,j,k}, \quad \forall p_j \in \mathcal{P}, S_m \in \mathcal{S}, \quad \forall k = 1, 2, \dots, F_j \quad (2.4b)$$

$$\sum_{p_j \in \mathcal{P}} \sum_{k=1}^{F_j} M_{i,j,k} = 1, \quad \forall t_i \in \mathcal{T}, \quad (2.4c)$$

$$M_{i,j,k} \in \{0, 1\}, \quad \forall t_i \in \mathcal{T}, p_j \in \mathcal{P}, k = 1, 2, \dots, F_j, \quad (2.4d)$$

$$Z_{m,j,k} \in \{0, 1\}, \quad \forall S_m \in \mathcal{S}, p_j \in \mathcal{P}, k = 1, 2, \dots, F_j, \quad (2.4e)$$



where (2.4b) guarantees that no scenario violates the utilization constraints, and (2.4c) specifies that a task is mapped on exactly one processing unit.

We denote this problem as the *global static power-aware mapping* problem.

This static mapping considers a set of applications with a probability distribution and computes the set of scenarios from that. All tasks that are active in these scenarios are considered for computing a static task to processing unit allocation. The probability of execution for tasks is considered to be known a priori, and taken into account for computing the task allocations. Diverging probability distributions in the actual system might result in significantly increased power dissipation. Since modes of an application can only execute in mutual exclusion, not all tasks of an application can be active at a time. Considering all tasks for the allocation process would limit the degree of freedom. Hence, the performance of the resulting mapping would degrade.

We propose a dynamic approach, which takes advantage of the structure of applications. The scenario sequences  $\mathcal{C}$ , derived in Section 2.5.1, represent possible execution paths of a system. Instead of computing a global static mapping, we compute a static mapping for each feasible sequence of scenarios  $c_r \in \mathcal{C}$ , using the algorithms developed for the global static power-aware mapping problem, called *templates*. Since a scenario sequence represents one possible execution path of a system, not all tasks and scenarios are included, hence the problem size is reduced and the degree of freedom for the mapping algorithms increases. Since probability distributions of applications are typically neither known nor static, we compute different template mappings for a set of representative probability distributions. The mappings are stored in a table on the system, and a manager chooses the appropriate template at run-time. We denote this problem as the *dynamic power-aware scenario-mapping* problem.

A scenario sequence  $c_r \in \mathcal{C}$  contains scenarios  $S_m \in \mathcal{S}$ , which we denote  $\mathcal{S}_{c_r}$  and  $\mathcal{S}_{c_r} \subset \mathcal{S}$ . Therefore, Equation (2.4) has to be adapted to consider only tasks  $\mathcal{T}_{c_r}$  that are active in scenarios  $\mathcal{S}_{c_r}$ . That is,  $\mathcal{S}_m$  in Equations (2.4a), (2.4b) and (2.4e) is replaced by  $\mathcal{S}_{c_r}$  and Equation (2.4) is applied to each scenario sequence  $c_r \in \mathcal{C}$ . As a result, the expected average power consumption for the dynamic power-aware scenario-mapping problem is stated as:

$$\sum_{S_m \in \mathcal{S}_{c_r}} \sum_{p_j \in \mathcal{P}} \sum_{k=1}^{F_j} \chi_m \sigma_j Z_{m,j,k} + \sum_{t_i \in \mathcal{T}_{c_r}} \sum_{p_j \in \mathcal{P}} \sum_{k=1}^{F_j} u_{i,j} \delta_j \psi_i M_{i,j,k}. \quad (2.5)$$

For each sequence  $c_r \in \mathcal{C}$  a template mapping is computed, and thus for each considered execution probability distribution there are  $|\mathcal{C}|$  template mappings.

As an example, consider a radio application. Sometimes synchronization is performed very frequently due to bad signal reception. At times, data transmission is active more often. Thus, the modes probability distributions change and only a subset of tasks is active at a time. Static task allocation can only cover one of the previously shown use-cases and might result in increased power dissipation for the other. Deriving template mappings for all the scenario sequences and the different execution probabilities of modes allows maintaining low power consumption over a systems lifetime and varying usage patterns.

Note that dynamic power management (DPM) is not adopted in the studied problem since the necessary timing information of applications is not contained in the problem definition.

#### 2.3.4 Hardness

It is not difficult to see that the problem is  $\mathcal{NP}$ -hard in a strong sense, since the special case with one scenario and one processing unit type is the bin packing problem. Moreover, when there is a limitation of the allocated units, i.e.,  $F_j < |\mathcal{T}|$ , deriving a feasible solution for Equation (2.4) is a  $\mathcal{NP}$ -complete problem since the bin packing problem is its special case. Therefore, unless  $\mathcal{P} = \mathcal{NP}$ , there is no polynomial-time algorithm for deriving a feasible solution for any input instance that allows for a feasible mapping when  $F_j < |\mathcal{T}|$ . Moreover, even if the architecture can be synthesized without any cost restriction, deriving an optimal solution for Equation (2.4) is still  $\mathcal{NP}$ -hard in a strong sense, and there does not exist any polynomial-time approximation algorithm to have a constant *approximation factor* unless  $\mathcal{P} = \mathcal{NP}$ . An algorithm is said to be with an approximation factor  $\beta$  if the objective function of its derived solution is at most  $\beta$  times the optimal objective solution for any input instance.

**Theorem 1** *Even when there is only one scenario, there does not exist any polynomial-time approximation algorithm with a constant approximation factor for the power-aware scenario-mapping problem, unless  $\mathcal{P} = \mathcal{NP}$ .*

**Proof.** This theorem is proved by an L-reduction [Pap94] from the set cover problem, which does not admit any polynomial-time approximation algorithm with a constant approximation factor unless  $\mathcal{NP} = \mathcal{P}$ . Given a universe  $\mathcal{E} = \{e_1, e_2, \dots, e_m\}$  of  $m$  elements, a collection  $\mathcal{W} = \{\mathcal{W}_1, \mathcal{W}_2, \dots, \mathcal{W}_n\}$  of sub-collections of  $\mathcal{E}$ , and the cost  $C_i > 0$  for each sub-collection  $\mathcal{S}_i$ , the set cover problem is to pickup a minimum-cost sub-collection of  $\mathcal{S}$  that covers all elements in  $\mathcal{E}$ .

The L-reduction is done as follows: For each sub-collection  $\mathcal{W}_j$ , we create a processing unit type  $p_j$  with static power consumption  $\sigma_j$  equal to  $C_j$ . The dynamic power consumption  $\delta_j$  on each processing unit  $p_j$  is a constant  $L$ .

For each element  $e_i$  in  $\mathcal{E}$ , we create a task  $t_i$ . If  $e_i$  is in  $\mathcal{W}_j$ , let  $u_{i,j}$  be  $1/|\mathcal{E}|$ . By restricting to the special case with one scenario, all the constructed tasks are present in one scenario with 100% probability.

For an optimal solution of the set cover problem with cost  $C^*$ , there is a feasible solution of the reduced input instance of the power-aware scenario-mapping problem with  $C^* + L$  expected average power consumption. For a feasible solution with  $C$  expected average power consumption for the reduced input instance of the power-aware scenario-mapping problem, there exists a solution for the set cover problem with cost no more than  $C - L$ . As a result, when  $L \ll C^*$ , if there is a polynomial-time  $\beta$ -approximation algorithm for power-aware scenario-mapping problem, the set cover problem will admit a polynomial-time  $\beta$ -approximation algorithm. The contradiction is reached.

□

According to the  $\mathcal{NP}$ -completeness of the derivation of feasible solutions for the global static power-aware mapping problem when  $F_j$  is less than  $|\mathcal{T}|$ , we focus our study on the case that  $F_j$  is not specified, i.e.,  $F_j = |\mathcal{T}|$ . If  $F_j$  is specified, our remapping algorithm in Section 2.4 can be revised to try to fit the required demand, but there is no feasibility guarantee.

## 2.4 Global static power-aware mapping problem

As it is difficult to derive solutions with worst-case guarantees in polynomial time, this section presents an efficient multi-step heuristics to derive approximative solutions for the global static power-aware mapping problem. We first derive initial solutions based on linear programming relaxation, and then perform task remapping to improve the solutions.

### 2.4.1 Initial Solutions

To derive a feasible initial solution, we can first relax the integral constraints in Equation (2.4d) and Equation (2.4e) so that  $M_{i,j,k}$  and  $Z_{m,j,k}$  can be any fractional variable. That is, constraints (2.4d) and (2.4e) in Equation (2.4) are relaxed and the problem can be formulated as:

$$\min \sum_{S_m \in \mathcal{S}} \sum_{p_j \in \mathcal{P}} \sum_{k=1}^{F_j} \chi_m \sigma_j Z_{m,j,k} + \sum_{t_i \in \mathcal{T}} \sum_{p_j \in \mathcal{P}} \sum_{k=1}^{F_j} u_{i,j} \delta_j \psi_i M_{i,j,k} \quad (2.6a)$$

s.t.

$$\sum_{t_i \in \mathcal{T}} \theta_{i,s} M_{i,j,k} u_{i,j} \leq Z_{m,j,k}, \quad \forall p_j \in \mathcal{P}, S_m \in \mathcal{S}, \quad \forall k = 1, 2, \dots, F_j \quad (2.6b)$$

$$\sum_{p_j \in \mathcal{P}} \sum_{k=1}^{F_j} M_{i,j,k} = 1, \quad \forall t_i \in \mathcal{T}, \quad (2.6c)$$

$$M_{i,j,k} \geq 0, \quad \forall t_i \in \mathcal{T}, p_j \in \mathcal{P}, k = 1, 2, \dots, F_j, \quad (2.6d)$$

$$Z_{m,j,k} \geq 0, \quad \forall S_m \in \mathcal{S}, p_j \in \mathcal{P}, k = 1, 2, \dots, F_j. \quad (2.6e)$$

In contrast to Equation (2.4), the new problem formulation is not upper-bounded anymore, as  $Z_{m,j,k}$  can take any positive value. Therefore, constraint (2.4b) can be removed. Additionally, constraint (2.4c) can be replaced by  $\hat{M}_{i,j} = \sum_{k=1}^{F_j} M_{i,j,k}$ . As a consequence of the unboundedness of  $Z_{m,j,k}$  tasks are assigned to PU types, rather than to instances thereof. Essentially, this means that portions of a task might be mapped onto different processing unit types, but these portions must sum up to 100%. Consequently, the term representing the static power consumption in the objective function is not constraint at all anymore and scenarios and their probabilities of execution are no longer an influential factor in the optimization process.

As a result, the optimal solution for the relaxed problem is equivalent to the following linear program:

$$\min \sum_{p_j \in \mathcal{P}} \sum_{t_i \in \mathcal{T}} \psi_i u_{i,j} (\delta_j + \sigma_j) \hat{M}_{i,j} \quad (2.7a)$$

s.t.

$$\sum_{p_j \in \mathcal{P}} \hat{M}_{i,j} = 1, \quad \forall t_i \in \mathcal{T}, \quad (2.7b)$$

$$\hat{M}_{i,j} \geq 0, \quad \forall t_i \in \mathcal{T}, p_j \in \mathcal{P}, \quad (2.7c)$$

where the variable  $\hat{M}_{i,j}$  indicates the portion of task  $t_i$  that is assigned to execute on PU type  $p_j$ . By applying the extreme point theory [Sch86], it is clear that there exists an optimal solution for Equation (2.7) which maps every task  $t_i \in \mathcal{T}$  to the PU type  $p_j \in \mathcal{P}$  that has the minimum static and dynamic power consumption ( $u_{i,j}(\delta_j + \sigma_j)$ ).

Algorithm 2.1 presents the pseudo-code of the procedures to derive an initial solution for the global static power-aware mapping problem. Step 2.1

**Algorithm 2.1** Initial Solution**Input:**  $\mathcal{T}, \mathcal{P}, \mathcal{S}$ 

- 
- 1:  $\mathcal{T}_j \leftarrow \emptyset, \forall p_j \in \mathcal{P}$ ;
  - 2: for each  $t_i \in \mathcal{T}$ , find  $p_{j^*} \leftarrow \underset{p_j \in \mathcal{P}}{\operatorname{argmin}} u_{i,j}(\delta_j + \sigma_j)$  and  $\mathcal{T}_j \leftarrow \mathcal{T}_j \cup \{t_i\}$ ;
  - 3: **for** each  $\mathcal{T}_j \neq \emptyset$  **do**
  - 4:     order tasks in  $\mathcal{T}_j$ , e.g., decreasing on the utilization;
  - 5:      $U_{m,j,k} \leftarrow 0, \forall S_m \in \mathcal{S}$  and  $k = 1, 2, \dots, |\mathcal{T}|$ ;
  - 6:     **for** each task  $t_i$  in  $\mathcal{T}_j$  **do**
  - 7:         let  $\mathcal{S}_{t_i}$  be the set of scenarios that  $t_i$  is associated with;
  - 8:         **if** there exists an index  $k$  with  $U_{m,j,k} > 0$  for some  $S_m \in \mathcal{S}_{t_i}$  and  $U_{m,j,k} + u_{i,j} \leq 1$  for all  $S_m \in \mathcal{S}_{t_i}$  **then**
  - 9:             assign task  $t_i$  to the  $k$ -th allocated PU of  $p_j$ ;
  - 10:              $U_{m,j,k} \leftarrow U_{m,j,k} + u_{i,j}$  for all  $S_m \in \mathcal{S}_i$ ;
  - 11:         **else**
  - 12:             let  $k^*$  be the smallest  $k$  with  $U_{m,j,k} = 0, \forall S_m \in \mathcal{S}$ ;
  - 13:             allocate the  $k^*$ -th unit of  $p_j$  and assign task  $t_i$  to it;
  - 14:              $U_{m,j,k^*} \leftarrow U_{m,j,k^*} + u_{i,j}$  for all  $S_m \in \mathcal{S}_i$ ;
  - 15:         **end if**
  - 16:     **end for**
  - 17: **end for**
- 

and Step 2 in Algorithm 2.1 derive an assignment of tasks to PU types, where Steps 3 to 17 allocate processing unit instances for tasks as described in the following.

After assigning tasks to PU types, the actual number of instances  $k$  per PU type  $p_j$  has to be computed. Suppose that the set of tasks, that are mapped onto a specific PU type  $p_j$  are denoted as  $\mathcal{T}_j \subseteq \mathcal{T}$ . For each PU type, we order the tasks in  $\mathcal{T}_j$  from high utilization to low utilization. Starting with task  $t_i \in \mathcal{T}_j$  with the highest utilization  $u_{i,j}$ , tasks are assigned to an instance  $k$  of processing unit type  $p_j$ , denoted  $\hat{p}_{j,k}$ . A task assignment is feasible, if the additional utilization  $u_{i,j}$  on target PU  $\hat{p}_{j,k}$  does not violate the utilization constraint for any scenario  $S_m$  that task  $t_i$  is assigned to (compare to constraint (2.4b)). If no feasible assignment exists, an additional instance of the corresponding PU type  $p_j$  is created, and the task is mapped onto this new instance. This process is repeated for each task and each PU type  $p_j$  until all tasks in  $\mathcal{T}$  are assigned to an instance of a processing unit. In case there are already multiple instances of a PU type available, tasks can be assigned to any concrete instance, as long as the utilization constraint is not violated.

Clearly, the derived solution is feasible for the global static power-aware mapping problem. There is at least one available processing unit type  $p_j \in \mathcal{P}$  for each task  $t_i \in \mathcal{T}$ , such that the task utilization  $u_{i,j}$  on processing unit type  $p_j$  is less than 100%. Hence, there is a feasible solution with at most

$F_j = |\mathcal{T}|$  processing units. In Algorithm 2.1, Line 3 has to be executed at most  $|\mathcal{P}|$  times. Line 6 executes at most  $|\mathcal{T}|$  times and the search for an instance  $k$  such that the utilization constraint is satisfied for all scenarios in Line 8 is in  $O(|\mathcal{S}||\mathcal{T}|)$ . Therefore, the time complexity is  $O(|\mathcal{P}||\mathcal{T}|^2|\mathcal{S}|)$ .

## 2.4.2 Task Remapping

In Equation (2.7) tasks are mapped to their most power efficient PU type. This may distribute the tasks over a large amount of PU types and instances thereof, which results in a low utilization of these PUs. According to our power model, a PU consumes static power, once it is switched on. Distributing tasks over a large amount of low utilized PUs leads to a high contribution of static power consumption to the expected average power consumption. Equation (2.7) disregards that fact, and thus might underestimate the objective.

We propose an approach to improve the solution iteratively by applying a multiple-step procedure. Given an initial solution, derived from Algorithm 2.1, we iteratively improve the solution by considering the remapping of tasks. Task remapping is done by considering sets of tasks belonging to a scenario  $\mathcal{S}_m$  on a specific PU  $\hat{p}_{j,k}$ . Let  $\mathcal{T}_{m,j,k}$  be the set of tasks assigned to PU  $\hat{p}_{j,k}$  in scenario  $\mathcal{S}_m$ . The remapping procedure attempts to remap all the tasks in  $\mathcal{T}_{m,j,k}$  to other PUs, in order to reduce the expected average power consumption. To reduce the time complexity for remapping, only PUs are considered as valid target units that already host all scenarios  $\mathcal{S}'_m$  to which the tasks in  $\mathcal{T}_{m,j,k}$  belong to. Tasks in  $\mathcal{T}_{m,j,k}$  can only be mapped onto PU  $\hat{p}_{j',k'}$  if the following condition applies: The sets of tasks  $\mathcal{T}_{m',j',k'}$  belonging to scenarios  $\mathcal{S}_{m'}$ , where  $\theta_{i,m'} = 1 \forall t_i \in \mathcal{T}_{m,j,k}$ , are non-empty sets. Among all task sets  $\mathcal{T}_{m',j',k'} \in \mathcal{T}$  we choose to remap the set of tasks  $\mathcal{T}_{m^*,j^*,k^*}$  first, that yields the highest reduction of expected average power consumption. This remapping step iterates until no further performance gain can be achieved.

The pseudo-code for remapping is presented in Algorithm 2.2, where *the detail of Step 3 will be presented in Algorithm 2.3* later. To reduce the time complexity for remapping, we only consider non-empty sets  $\mathcal{T}_{m,j,k}$ . In Algorithm 2.2, we use  $f_j$  to denote the number of allocated units of PU type  $p_j$  in the initial solution.

We now present how to determine the highest expected average power reduction of task set  $\mathcal{T}_{m,j,k}$ , i.e., the implementation of Step 3 in Algorithm 2.2. Suppose that  $U_{m,j,k}$  is the utilization of tasks in  $\mathcal{T}_{m,j,k}$ , i.e.,  $U_{m,j,k} = \sum_{t_i \in \mathcal{T}_{m,j,k}} u_{i,j}$ . Furthermore, once tasks assigned to scenario  $\mathcal{S}_m$  are mapped onto PU  $\hat{p}_{j,k}$ , we set  $Z_{m,j,k}^\dagger$  as 1, while  $Z_{m,j,k}^\dagger$  is 0 otherwise.

**Algorithm 2.2** Remapping**Input:**  $\mathcal{T}_{m,j,k}, \forall S_m \in \mathcal{S}, p_j \in \mathcal{P}, k = 1, \dots, f_j$ ;

---

```

1: while true do
2:   for each non-empty set  $\mathcal{T}_{m^*,j^*,k^*}$  do
3:     apply Algorithm 3 to remap all the tasks in  $\mathcal{T}_{m^*,j^*,k^*}$ , and let
        $\mathcal{T}_{m,j,k}^{m^*,j^*,k^*}, \forall S_m \in \mathcal{S}, p_j \in \mathcal{P}, k = 1, \dots, f_j$  be the solution;
4:     let  $\Delta_{m^*,j^*,k^*}$  be its reduced expected average power consumption if the
       remapping (in Step 3) is successful;
5:   end for
6:   if there is no successful remapping (in Step 3) then
7:     break;
8:   else
9:     let  $m^\dagger, j^\dagger, k^\dagger$  be the indexes  $m^*, j^*, k^*$  with the minimum  $\Delta_{m^*,j^*,k^*}$ ;
10:     $\mathcal{T}_{m,j,k} \leftarrow \mathcal{T}_{m,j,k}^{m^\dagger, j^\dagger, k^\dagger}, \forall S_m \in \mathcal{S}, p_j \in \mathcal{P}, k = 1, \dots, f_j$ ;
11:  end if
12: end while
13: return the task sets  $\mathcal{T}_{m,j,k}, \forall S_m \in \mathcal{S}, p_j \in \mathcal{P}, k = 1, \dots, f_j$ ;

```

---

A set of tasks  $\mathcal{T}_{m^*,j^*,k^*}$  can only be remapped to a processing unit  $\hat{p}_{j',k'}$  if  $Z_{m^*,j^*,k^*}^\dagger$  is 1. This allows satisfying the previously defined condition, that tasks can only be remapped if all the scenarios they are assigned to are hosted on the target processing unit.

The optimal solution for remapping task set  $\mathcal{T}_{m^*,j^*,k^*}$  can be formulated by the following integer linear programming:

$$\min \sum_{S_m \in \mathcal{S}} \sum_{p_j \in \mathcal{P}} \sum_{k=1}^{F_j} \chi_m \delta_j \left( U_{m,j,k} + \sum_{t_i \in \mathcal{T}_{m^*,j^*,k^*}} M_{i,j,k} u_{i,j} \right) \quad (2.8a)$$

s.t.

$$U_{m,j,k} + \sum_{t_i \in \mathcal{T}_{m^*,j^*,k^*}} M_{i,j,k} u_{i,j} \leq Z_{m,j,k}^\dagger, \quad \forall p_j \in \mathcal{P}, S_m \in \mathcal{S}, \quad \forall k = 1, 2, \dots, F_j \quad (2.8b)$$

$$\sum_{p_j \in \mathcal{P}} \sum_{k=1, k \neq k^*}^{F_j} M_{i,j,k} = 1, \quad \forall t_i \in \mathcal{T}_{m^*,j^*,k^*}, \quad (2.8c)$$

$$M_{i,j,k} \in \{0, 1\}, \quad \forall t_i \in \mathcal{T}_{m^*,j^*,k^*}, p_j \in \mathcal{P}, k = 1, 2, \dots, F_j. \quad (2.8d)$$

Note that  $Z_{m,j,k}^\dagger$  is not a variable in the programming shown in Equation (2.8), but is derived from the initial mapping.

As the number of tasks in task set  $\mathcal{T}_{m^*,j^*,k^*}$  is significantly reduced compared to  $\mathcal{T}$ , it is possible to derive optimal solutions of Equation (2.8). However, the ILP in Equation (2.8) has to be executed many times (in  $O(|\mathcal{P}||\mathcal{S}|)$ ) to determine which remapping is the best so far. Once the resulting task set is remapped, the ILP has to be solved again, to find the

**Algorithm 2.3** Remapping of One Task Set

---

**Input:**  $\mathcal{T}_{m,j,k}, S_m \in \mathcal{S}, p_j \in \mathcal{P}, k = 1, \dots, f_j; \mathcal{T}_{m^*,j^*,k^*}$  for remapping;

- 1: **while**  $\mathcal{T}_{m^*,j^*,k^*} \neq \emptyset$  **do**
- 2:    $t_{i^*} \leftarrow \underset{t_i \in \mathcal{T}_{m^*,j^*,k^*}}{\text{argmin}} u_{i,j^*};$
- 3:   let  $\mathcal{S}_{i^*}$  be the set of the scenarios that task  $t_{i^*}$  is associated with;
- 4:   among the non-empty task sets  $\mathcal{T}_{m,j,k}$  ( $(j,k) \neq (j^*,k^*)$ ) with  $S_m \in \mathcal{S}_{i^*}$  and  $u_{i^*,j} + \sum_{t_i \in \mathcal{T}_{m,j,k}} u_{i,j} \leq 1$  for all  $S_m$  in  $\mathcal{S}_{i^*}$ , let  $(j',k')$  be the indexes with the minimum  $u_{i^*,j'}$  and satisfied constraint  $U_{m,j',k'}$ ;
- 5:   **if** indexes  $(j',k')$  do not exist **then**
- 6:     **return** remapping fails;
- 7:   **else**
- 8:      $\mathcal{T}_{m',j',k'} \leftarrow \mathcal{T}_{m',j',k'} \cup \{t_{i^*}\}, \forall S_{m'} \in \mathcal{S}_{i^*};$
- 9:      $\mathcal{T}_{m',j^*,k^*} \leftarrow \mathcal{T}_{m',j^*,k^*} \setminus \{t_{i^*}\}, \forall S_{m'} \in \mathcal{S}_{i^*};$
- 10:   **end if**
- 11: **end while**
- 12: **return** task sets  $\mathcal{T}_{m,j,k}, S_m \in \mathcal{S}, p_j \in \mathcal{P}, k = 1, \dots, f_j;$

---

next set of tasks for remapping. As a consequence, applying an ILP solver to exactly solve Equation (2.8) with high complexity is impractical. This chapter presents how to perform task remapping by applying a heuristic approach as shown in Algorithm 2.3.

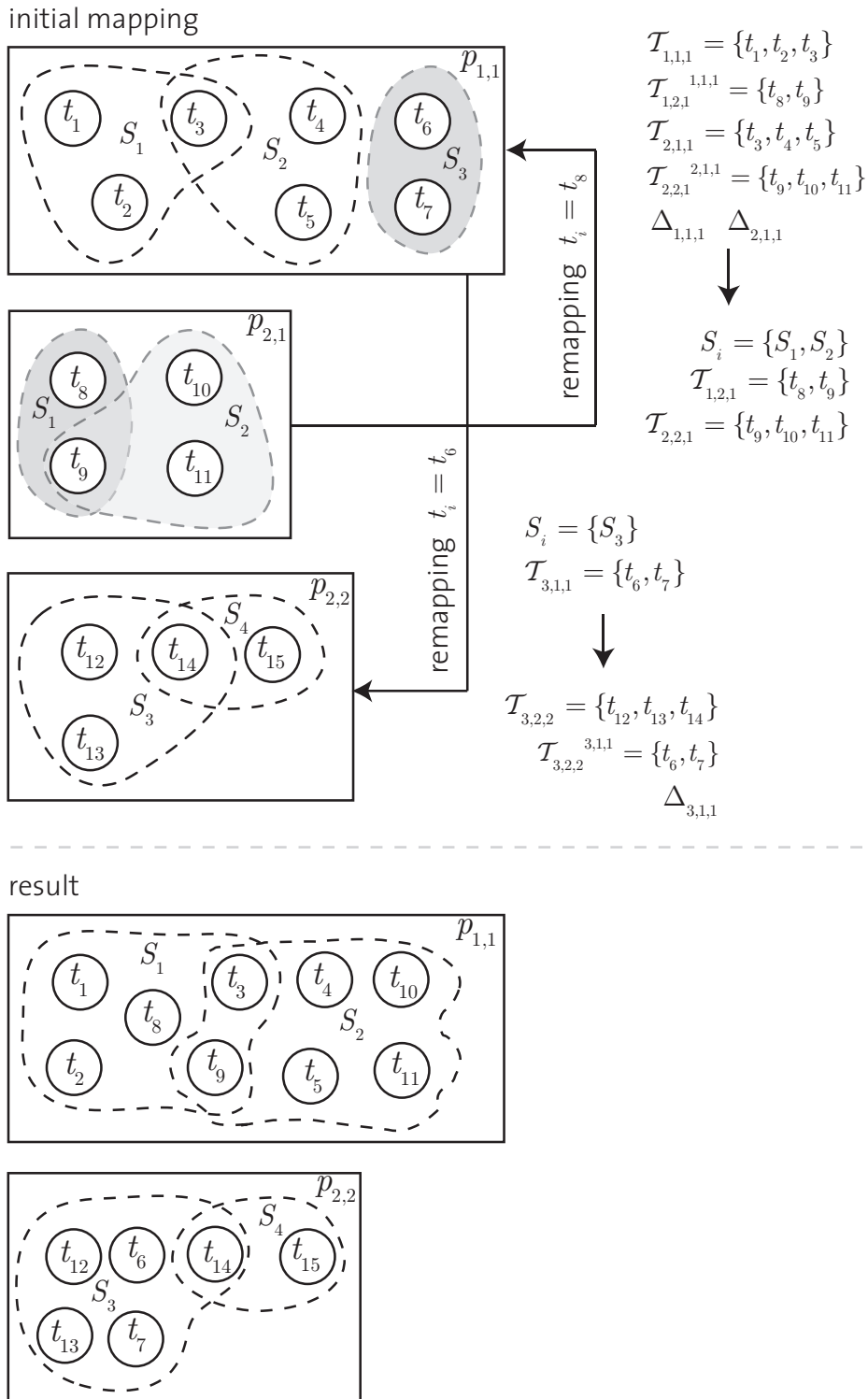
Algorithm 2.3 has as an input the set of tasks  $\mathcal{T}_{m^*,j^*,k^*}$  that shall be remapped. In Step 2, task  $t_{i^*}$  with the lowest utilization  $u_{i^*,j}$  is retrieved. Furthermore  $\mathcal{S}_{i^*}$  is defined to be the scenarios task  $t_{i^*}$  is present in.

The next steps perform the search for a processing unit  $\hat{p}_{j',k'}$  where the utilization  $u_{i^*,j'}$  of task  $t_{i^*}$  is minimized and there exists an instance  $k'$  of PU type  $p'_j$  such that the utilization constraints for all scenarios  $\mathcal{S}_{i^*}$  are satisfied. Furthermore the target and the origin PU cannot be the same unit ( $\hat{p}_{j',k'} \neq \hat{p}_{j,k}$ ). If such a PU cannot be found (Step 5), remapping fails and the algorithm continues with the next task in the input task set. Otherwise (Step 7) the remapping is performed and task  $t_{i^*}$  is removed from the set of tasks to be remapped. This process repeats, as long as there are task in the input task set. The approach aims at reducing the number of low utilized PUs in order to reduce static power consumption. A set of tasks is remapped to another processing element, if the benefit of saving the static power consumption of the current PU outweighs the penalty of increased dynamic power consumption on the target PU.

In Figure 2.4, the initial mapping of tasks to resource instances is shown on the top. Based on that, two examples for the remapping process are presented. Scenario  $S_1, S_2, S_3$  and  $S_4$  are mapped onto resource instances  $p_{1,1}, p_{2,1}$  and  $p_{2,2}$  as shown in Figure 2.4 and the remapping process starts with



task  $t_8$  in scenario  $S_1$  on resource instance  $p_{2,1}$  (e.g., Line 2 in Algorithm 2.3 returns  $t_8$ ). This task belongs only to scenario  $S_1$ , but since all tasks of a scenario are remapped, not just single tasks, the dependent scenarios for all the tasks in  $S_1$  have to be considered as well. It turns out, that task  $t_9$  belongs to scenarios  $S_1$  and  $S_2$ , and therefore, both scenarios are considered for the remapping process. As a result, the task set to be remapped for scenario  $S_1$  is  $\mathcal{T}_{1,2,1}$  and the task set to be remapped for scenario  $S_2$  is  $\mathcal{T}_{2,2,1}$ . In the next step, Step 4 in Algorithm 2.3, a resource instance that could host the task sets without violating its utilization constraints is found. Let this resource instance be  $p_{1,1}$  in Figure 2.4, since  $p_{2,2}$  does not host either  $S_1$  nor  $S_2$  and therefore could not host any tasks belonging to these scenarios. Scenario  $S_1$  on resource instance  $p_{1,1}$  hosts the task set  $\mathcal{T}_{1,1,1}$ , and after a successful mapping would also host the task set  $\mathcal{T}_{1,2,1}^{1,1,1}$ . Similarly for scenario  $S_2$  and task sets  $\mathcal{T}_{2,1,1}$  and  $\mathcal{T}_{2,2,1}^{2,1,1}$ . As a result, resource instance  $p_{2,1}$  does not host scenarios  $S_1$  and  $S_2$  anymore. Therefore, this resource instance can be switched off for a time span, corresponding to the active time of scenarios  $S_1$  and  $S_2$ . Hence static power consumption is reduced. The reduced power consumption  $\Delta_{1,1,1}$  and  $\Delta_{2,1,1}$  for scenarios  $S_1$  and  $S_2$  respectively, is a result of reduced static power consumption on resource instance  $p_{2,1}$ . In case there are multiple resource instances, where task sets  $\mathcal{T}_{1,2,1}$  and  $\mathcal{T}_{2,2,1}$  could be remapped to, the one resource instance that results in the minimum reduced expected average power consumption  $\Delta$  is chosen, see Line 9 in Algorithm 2.2. The second task set in Figure 2.4 that should be remapped belongs to scenario  $S_3$  on resource instance  $p_{2,1}$ . The only other resource instance where scenario  $S_3$  is active, is resource instance  $p_{2,2}$ . Additionally, there are no other scenarios involved, since no task in scenario  $S_3$  on  $p_{2,1}$  is active in any scenario other than  $S_3$ . Similarly to the previous example, the tasks in scenario  $S_3$  on  $p_{2,1}$  are reassigned to resource instance  $p_{2,2}$ . As a result, resource instance  $p_{1,1}$  can be switched off for a time span that corresponds to the active time of scenario  $S_3$ . The resulting task to processing unit allocation is presented in Figure 2.4 on the bottom. In these two examples, we assume that the saving that results from switching off resource instances outweighs the penalty of increased dynamic power consumption that results from executing tasks on other resource instances. However, this is not a requirement of our approach, but a simplification in order not to overload the figures. Remapping of tasks is only performed once the reduced expected power consumption actually results in decreased power consumption; otherwise the tasks remain on their original resource instance.



**Figure 2.4:** Task Remapping Approach - an example for two tasks being remapped.

### 2.4.3 Complexity

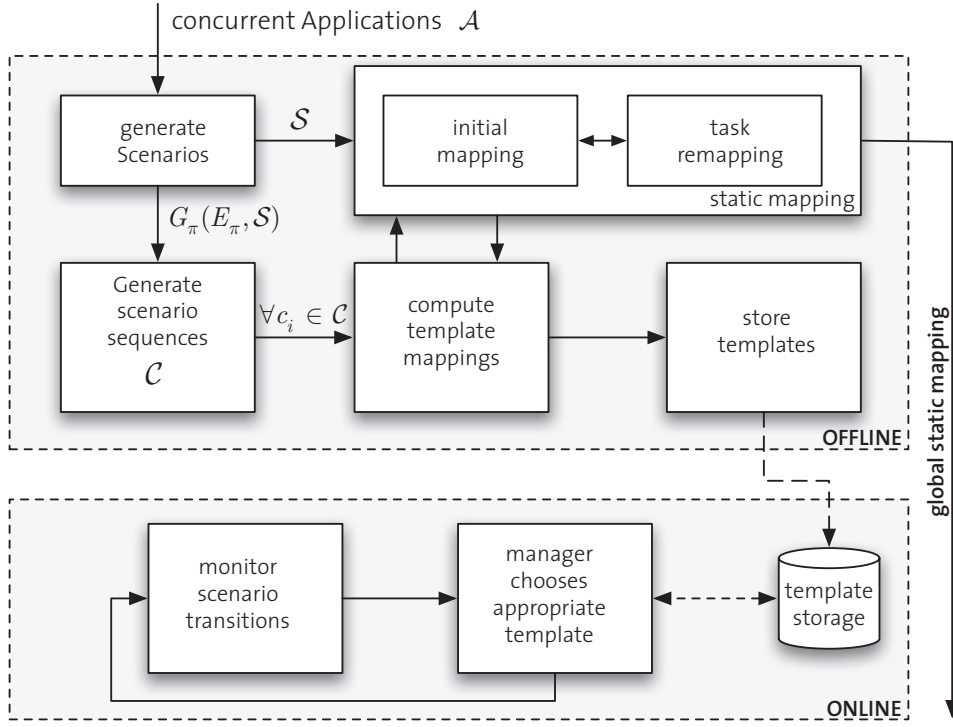
In Algorithm 2.3, the number of tasks in  $\mathcal{T}_{m^*,j^*,k^*}$  at Line 1 is at most  $|\mathcal{T}|$ . In Line 4, for all task sets  $\mathcal{T}_{m,j,k}$  (at most  $|\mathcal{T}|$ ) and all scenarios (at most  $|\mathcal{S}|$ ), the processing element  $p'_j$  with the lowest utilization  $u_{i^*,j'}$  for task  $t_i^*$  has to be found, such that the utilization constraint is satisfied. This search is in  $O(|\mathcal{T}||\mathcal{S}||\mathcal{P}|)$ . As a result, the time complexity of Algorithm 2.3 is in  $O(|\mathcal{S}||\mathcal{P}||\mathcal{T}||\mathcal{T}_{m^*,j^*,k^*}|)$ . Algorithm 2.3 is used in Algorithm 2.2 as a subroutine. The loop in Line 1 is executed at most  $|\mathcal{P}||\mathcal{S}|$  times, since for all possible task set  $\mathcal{T}_{m^*,j^*,k^*}$  the reduced expected average power consumption needs to be computed. While the loop in Line 2 is executed for at most  $|\mathcal{T}|$  times. Since this loop dominates the complexity of Algorithm 2.2, its time complexity is  $O(|\mathcal{T}|^3|\mathcal{P}|^2|\mathcal{S}|^2)$ . Only one of the computed solutions in Algorithm 2.2 is applied, namely the one that yields the lowest expected average power consumption. In order to derive the next set of tasks and their respective remapping possibilities, Algorithm 2.2 has to be executed again. This procedure is repeated once for all scenarios and processing element instances, at most  $|\mathcal{P}||\mathcal{T}||\mathcal{S}|$ . As a result, the remapping process has a time complexity of  $O(|\mathcal{T}|^4|\mathcal{S}|^3|\mathcal{P}|^3)$ .

## 2.5 Dynamic power-aware scenario-mapping problem

This section describes our proposed dynamic approach. Based on the graph  $G_\pi$ , see Figure 2.3, representing the scenarios and their valid transitions, the offline part computes the set of loop free paths through the graph, i.e., the scenario sequences  $\mathcal{C}$ . For each *scenario sequence*, a static mapping, using the algorithms presented to solve the global static power-aware mapping problem, is computed and stored on the system as template. The online part monitors the scenario transitions and, once a new scenario becomes active, chooses an appropriate mapping from the precomputed templates, see Figure 2.5. We show how to derive a finite set of paths through the graph representing the possible scenarios and how to adapt the global static mapping approach for dynamic behavior.

### 2.5.1 Scenario Sequence Generation

Consider the cross product  $G_\pi(E_\pi, \mathcal{S})$  in Figure 2.3 as an example. We construct all possible paths from the initial to the final state. The initial and final modes of applications  $A_1$  and  $A_2$  in Figure 2.2 are known, thus the initial and final states of graph  $G_\pi$  are (I1, I2) and (R1, R2) respectively. Considering loop free paths only, i.e., paths that traverse a state at most once, results in a finite set of paths.



**Figure 2.5:** Dynamic Mapping Approach - Overview of offline and online steps

Deriving the set of paths  $\mathcal{C}$  can be formulated by a recursive algorithm as shown in Alg. 2.4. Consider  $\mathcal{V}_{init}$  as the set of initial states and  $\mathcal{V}_{end}$  as the set of final states. The set of paths from all  $v_{init,i} \in \mathcal{V}_{init}$  to all  $v_{end,j} \in \mathcal{V}_{end}$  is the set of scenario sequences  $\mathcal{C}$  and  $c_r \in \mathcal{C}$  is one unique sequence of scenarios. We apply Algorithm 2.4 to any combination of initial and final states in  $\mathcal{V}_{init}$  and  $\mathcal{V}_{end}$  respectively. In our example in Figure 2.3, there is only one initial and one final state. Therefore, let Algorithm 2.4 be initialized with  $v_{init,1} = (I1, I2)$ , the initial sequence  $c = v_{init,1}$ ,  $v_{end,1} = (R1, R2)$  and the global variable  $\mathcal{C} = \emptyset$ . Each iteration adds the current state  $v_i$  to sequence  $c$  and computes its successor states. Successor states  $v_j$ , that lead to loop free paths are considered for another iteration of the algorithm. Once all the sequences are known, we can compute template mappings for each of them.

### 2.5.2 Deriving templates and the hardware platform

The template mappings for our dynamic approach are computed by applying the global static mapping procedure to each scenario sequence  $c_r \in \mathcal{C}$ . A scenario sequence  $c_r \in \mathcal{C}$  describes the transitions from an initial scenario  $S_{init}$  to another scenario  $S_{end}$ , and  $\mathcal{S}_{c_r}$  denotes the set of scenarios that is

**Algorithm 2.4** compute\_paths**Global:**  $\mathcal{C}$ **Input:**  $G_\pi(\mathcal{S}, E_\pi), v_{init}, v_{end}, v_i, c;$ 


---

```

1: if  $v_i = v_{end}$  then
2:    $\mathcal{C} \leftarrow \mathcal{C} \cup \{c\}$ 
3:   return;
4: end if
5: for each successor  $v_j$  of  $v_i \in \mathcal{S}$  do
6:   if  $v_j \notin c$  then
7:     compute_paths( $G_\pi(\mathcal{S}, E_\pi), v_{init}, v_{end}, v_j, c \cup \{v_j\}$ );
8:   end if
9: end for

```

---

traversed in this scenario sequence. Each scenario of that sequence activates and deactivates some tasks. This results in a set of tasks  $\mathcal{T}_{c_r}$  that has to be considered for allocation. The tasks in  $\mathcal{T}_{c_r}$  all belong to scenarios in the scenario sequence  $c_r$ , but not all scenarios that activate a task  $t_i \in \mathcal{T}_{c_r}$  are traversed by the sequence  $c_r$ . The execution probability  $\psi_i$  of task  $t_i \in \mathcal{T}_{c_r}$  has to be recomputed, such that only those scenarios are considered that are actually traversed by  $c_r$ :

$$\psi_i = \sum_{S_m: t_i \text{ active in } S_m \text{ and } S_m \in \mathcal{S}_{c_r}} \chi_m \quad (2.9)$$

The mapping process is applied and a task allocation for each sequence  $c_r \in \mathcal{C}$  is derived.

The static mapping approach, as formulated in Equation (2.4), is changed such that only those scenarios  $S_m \in \mathcal{S}$  are considered, that are traversed in sequence  $c_r$ . That is, the objective in Equation (2.4a) is changed to:

$$\sum_{S_m \in \mathcal{S}_{c_r}} \sum_{p_j \in \mathcal{P}} \sum_{k=1}^{F_j} \chi_m \sigma_j Z_{m,j,k} + \sum_{t_i \in \mathcal{T}_{c_r}} \sum_{p_j \in \mathcal{P}} \sum_{k=1}^{F_j} u_{i,j} \delta_j \psi_i M_{i,j,k}. \quad (2.10)$$

Note that only those scenarios are considered that are traversed by scenario sequence  $c_r$ . Furthermore, the set of tasks considered for the mapping process is limited to those tasks activated by scenarios  $S_m \in c_r$ , i.e., task set  $\mathcal{T}_{c_r}$ . Based on this formulation, the algorithms presented in Section 2.4 can be applied correspondingly. Optimally solving the dynamic power-aware scenario-mapping problem means applying the ILP solver, as described in Equation(2.4), to each scenario sequence  $c_r \in \mathcal{C}$ . Similarly, solving the problem with a heuristic means applying the heuristic in Algorithms 2.1 and 2.2 to each scenario sequence.

The process of deriving all templates for all scenario sequences can be formulated as in Algorithm 2.5.

---

**Algorithm 2.5** `compute_templates`

---

**Input:**  $\mathcal{P}, G_\pi(\mathcal{S}, E_\pi), \mathcal{T}, \mathcal{V}_{init}, \mathcal{V}_{end}$ ;**Global:**  $\mathcal{C}$ **Output:** templates for  $\mathcal{C}$ 

```

1: for all combinations of  $v_{init,i} \in \mathcal{V}_{init}$  and  $v_{end,j} \in \mathcal{V}_{end}$  do
2:    $c \leftarrow v_{init,i}$ 
3:    $\mathcal{C} = \text{compute\_paths}(G_\pi(\mathcal{S}, E_\pi), v_{init,i}, v_{end,j}, v_{init,i}, c)$ 
4: end for
5: for each  $c_j \in \mathcal{C}$  do
6:   initialize  $Z, M$ 
7:   compute  $\mathcal{T}_{c_j}$  and  $\psi_i$  for each task  $t_i \in \mathcal{T}_{c_j}$ 
8:    $M, Z \leftarrow \text{static\_mapping}(c_j, \mathcal{T}_{c_j}, \mathcal{P})$ 
9:   store mapping in  $M, Z$  as template for  $c_j$ 
10: end for

```

---

Once a scenario is reached, there exists a precomputed template mapping for each sequence of scenario transitions that might have been executed before. As an example, consider a system transitioning to scenario  $S_m \in \mathcal{S}$ . This scenario was considered for template mappings in all the scenario sequences  $\mathcal{C}_{S_m} \subset \mathcal{C}$  that traverse  $S_m$ . Therefore, once a scenario  $S_m$  is reached, the preceding sequence of scenario transitions  $c_{S_m}$  is included in  $\mathcal{C}_{S_m}$ . The tasks activated in scenario  $S_m$  are mapped according to the template computed for scenario sequence  $c_{S_m}$ . Once the system transitions to scenario  $S_{m+1} \notin c_{S_m}$  then the tasks activated in this scenario are mapped according to the template computed for scenario sequence  $c_{S_{m+1}}$ . Note the scenario sequence  $c_{S_m}$  and scenario sequence  $c_{S_{m+1}}$  traverse the same scenarios up until scenario  $S_m$ , i.e., scenario  $S_m \in c_{S_m}$  and  $S_m \in c_{S_{m+1}}$ . As a result, up until scenario  $S_m$  these sequences represent the same set of scenario transitions. Hence, the tasks in scenario  $S_m$  are mapped, such that a transition to all succeeding scenarios, including  $S_{m+1}$  is feasible. See Figure 2.6 for an example. Consider the scenario reached by the scenario sequence (I1, I2) (I1, S2) (S1, S2) (R1, S2). Once arriving in (R1, S2), there are two possibilities to continue. First, transition to (R1, R2), or second, transition to (I1, S2). The path transitioning to (I1, S2) needs not to be considered for the scenario sequences  $\mathcal{C}$ , since this scenario was already traversed by the sequence of scenario transitions so far. Hence, the mapping of the tasks activated by (I1, S2) is already known for this sequence.

At runtime, once a scenario is reached, there exists a mapping for the tasks that are activated by that scenario, for all scenario transitions that can lead to this scenario and for all mapping decisions that have been taken so far. Consider scenario (S1, R2) in the product graph in Figure 2.3 and the possible scenario sequences that could lead to this scenario, as shown in Figure 2.6 (which represents a partial unfolding of the graph in Figure 2.3

only, due to space limitations). Then it can be seen that there are multiple scenario sequences that contain (S1, R2). Now consider a transition from scenario (I1, I2) to (I1, S2) and then to (S1, S2). At each transition to a scenario, a template mapping has to be chosen by the runtime environment, depending on the sequence of preceding scenario transitions. This template was computed offline, such that all tasks that are activated by succeeding scenarios can still be mapped, i.e., the subgraph for which the current scenario represents the root is considered in the mapping process. As a result, no matter which sequence of scenario transitions is executed in the system, there is a template mapping for scenario (S1, S2), but those templates take very different future developments into consideration. As an example, the mapping of tasks activated in scenario (S1, S2) is different for scenario sequence (I1, I2), (I1, S2), (S1, S2) and (I1, I2), (I1, S2), (I1, R2), (S1, R2), (S1, I2), (S1, S2) since in the latter, the number of possible future developments is significantly reduced compared to the first case.

### Upper bound on the number of scenario sequences

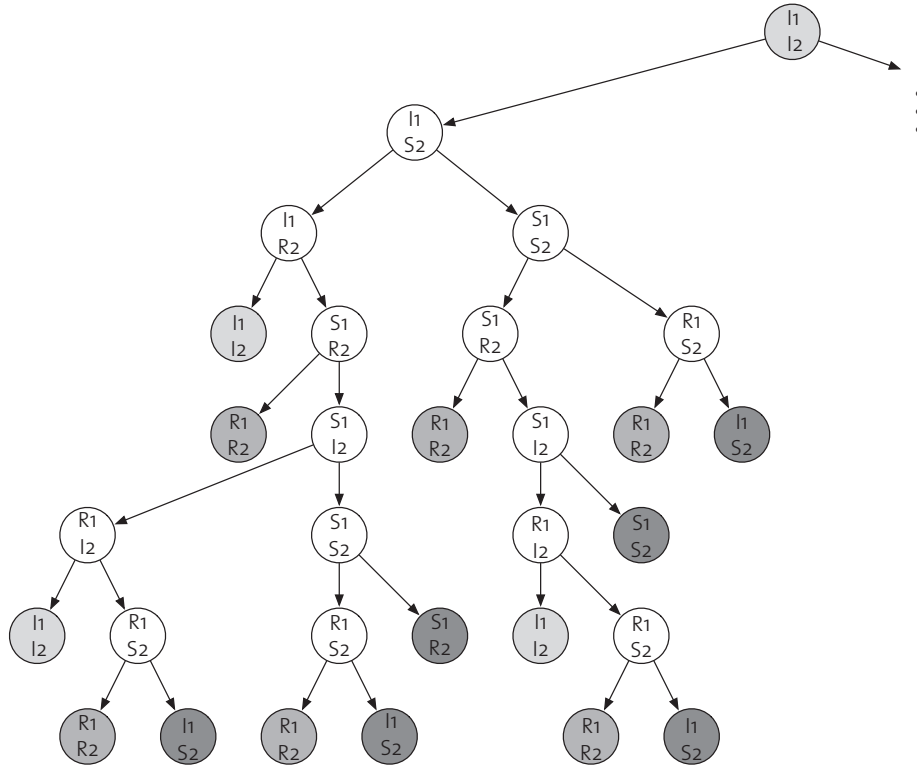
Consider the graphs  $G_{A_1}(V_{A_1}, E_{A_1})$  and  $G_{A_2}(V_{A_2}, E_{A_2})$  representing applications  $A_1$  and  $A_2$  respectively. Furthermore, consider graph  $G_\pi = G_{A_1} \times G_{A_2}$ ,  $k$  as the maximum number of outgoing transitions for each state in  $G_\pi$  and  $n$  as the length of the longest loop-free path to reach the final state. Then, the number of paths to get from an initial state to a final state can be bounded by:

$$|\mathcal{C}| \leq k^n. \quad (2.11)$$

This number is exponential in the number of scenarios, i.e., the number of applications that execute concurrently. For our example in Figure 2.3, each state has two outgoing transitions, therefore the number of paths is bounded by  $|\mathcal{C}| \leq 2^n$ . The actual number of paths is significantly smaller, since in reality we are only considering loop-free paths, but not all paths of length  $n$  are loop free. Conclusively, once a state is traversed that has already been seen on the current path, this path can be omitted.

Tighter bounds on the number of paths can be derived by considering the sub-trees that are omitted due to cycles and the structure of our proposed application model.

In Figure 2.6, we show the unfolding of graph  $G_\pi$  representing the cross product of the example in Figure 2.3. It can be seen that there is a finite number of loop free paths from the initial state to the final state. Light gray nodes in the graph represent the initial state and once the system transitions to that, it is restarted. Dark gray nodes represents a transition to a scenario that has already been traversed, therefore a loop is detected. Medium gray nodes represent the accepting state.



**Figure 2.6:** Unfolding of the cross product  $G_\pi$  of the example system in Figure 2.3. Green nodes represent system restart, red nodes represent cycles that can be skipped and blue nodes represent accepting states.

### Deriving a hardware platform

Each sequence  $c_r \in \mathcal{C}$  results in a distinct template mapping and therefore a distinct number of instances  $k_j$  for each PU type  $p_j$ . The hardware platform, that guarantees feasibility for any scenario sequence  $c_r \in \mathcal{C}$ , is constituted by the maximum number of instances  $k_j$  for each  $p_j \in \mathcal{P}$ , among all the precomputed template mappings.

#### 2.5.3 Online mapping

Templates are stored on the system and a manager observes scenario transitions. Based on this observation a precomputed template is chosen.

There is one mapping for each scenario sequence and the number of such sequences is bound by Equation (2.11). However, in our example in Figure 2.3, the number of templates is 12, representing a much lower number than suggested by Equation (2.11) for  $k = 2$  and  $n = 8$ . Storing template mappings for two applications, constituted by a total of 34 tasks and an assumed hardware platform of 20 processing units, requires 1kb of memory, or 12 times the amount required by a global static mapping.



Consider a scenario sequence  $c_r \in \mathcal{C}$  and a resulting template mapping  $M^r$ . This template mapping assigns the tasks that are active in the scenarios that constitute  $c_r$  to processing elements, such that task  $t_i$  is mapped onto processing element  $p_{j,k}$ , if  $M_{i,j,k}^r = 1$  and  $M_{i,j,k}^r = 0$  otherwise. Furthermore, consider an associative array  $\mathcal{L}$  to store mappings, such that  $\mathcal{L} = \{c_1 \rightarrow M^1, \dots, c_n \rightarrow M^n\}$  and  $n = |\mathcal{C}|$ . Furthermore, consider the observer to know the current system state  $h$ , i.e., the current scenario and the sequence of scenario transitions so far. Then at a transition from a scenario  $S_m$  to another scenario  $S_{m+1}$ , the online manager updates the current system state, such that  $h = h \cup \{S_{m+1}\}$ . In case  $S_{m+1} \in \mathcal{V}_{init}$ , i.e., the system transitions to an initial scenario, system state  $h$  is reset to  $\{S_{m+1}\}$ .

Matching the current system state  $h$  with the index of the associative array  $\mathcal{L}$  allows retrieving the template  $M^r$  that was precomputed for the current scenario sequence. The system state  $h$  as well as every scenario sequence  $c_r \in \mathcal{C}$  start with a scenario  $S_{init} \in \mathcal{V}_{init}$ . As a result, any of the matching scenario sequences can be used to retrieve the mapping of a task by reading their respective  $M_{i,j,k}^r$ . Conclusively, the runtime environment needs to be aware of the current system state. Then template mappings can be retrieved in constant time.

#### 2.5.4 Templates for different probability distributions

For different execution probability distributions optimal template mappings are generated according to the proposed methodology and stored on the system. In addition to monitoring the sequence of scenarios, the online manager also needs to store the execution frequency of each individual scenario, e.g., by incrementing a counter each time a particular scenario is activated. Based on this information, the online manager can then choose the best fitting template. The global static power-aware mapping problem does not allow providing additional mappings and therefore can only guarantee efficient execution for a single probability distribution.

## 2.6 Performance Evaluation

This section provides the simulation results by means of realistic SDR (software defined radio) applications. Specifically, Wireless Local Area Network (WLAN), as described in [MVB07], Digital Video Broadcast - Handhelds (DVB-H), as described in [SKG<sup>+</sup>07], and Ultra Wideband (UWB), as described in [SWvdV08], are adopted in the simulations.

### 2.6.1 Simulation Setup

The simulation setup is characterized in Table 2.3. For each application, a set of tasks is extracted and the required computational resource demand  $\gamma_{i,j}$  of task  $t_i$  on processing unit type  $p_j$  is generated, see Table 2.4.

We simulate systems with two applications (either WLAN and DVB-H or WLAN and UWB), and three modes in each application (init, send/sync and receive). Each application is composed by 3 modes, see Figure 2.2, and parameters are generated as random variables.

Applications	Modes	Number of tasks	Number of shared tasks	$\gamma_{i,j}$	$\hat{\chi}$
$A_1$ (DVB-H)	$\mu_1$ ( <b>sync</b> )	4	0	$0 \leq \gamma_{i,j} \leq 0.5$	determined by profiling
	$\mu_2$ ( <b>receive</b> )	7	0		
$A_2$ (WLAN)	$\mu_3$ ( <b>send</b> )	10	10		
	$\mu_4$ ( <b>receive</b> )	23	10		
$A_3$ (UWB)	$\mu_5$ ( <b>send</b> )	3	1		
	$\mu_6$ ( <b>receive</b> )	5	1		

**Table 2.3:** Simulation setup for WLAN, DVB-H, and UWB applications.

variables	values
$\sigma_j$	random $0 \leq \sigma_j \leq 1.0$
$\delta_j$	random $0 \leq \delta_j \leq \sigma_j$
$\lambda_j$	random $0 \leq \lambda_j \leq 3.0$
$\gamma_{i,j}$	random $0 \leq \gamma_{i,j} \leq 0.5$

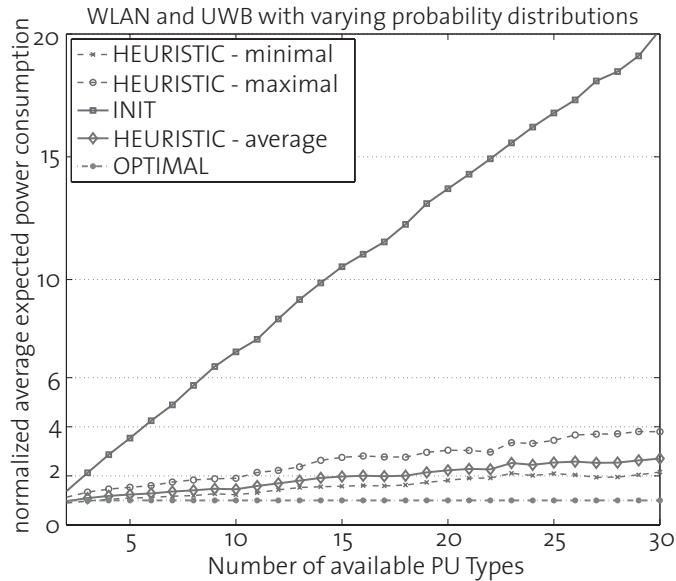
**Table 2.4:** Parameters for generating the processing unit types.

Experiments are executed for different PU type library sizes, reaching from 2 available types to 30. The modes execution probabilities have been varied and experiments were executed for 6 different execution distributions. For each distinct library size we created 1000 system instances. As a result, 6000 realizations per PU type library size were computed. Each instance uses a different set of application parameters and a different library of PU types  $\mathcal{P}$  to construct the hardware platform.

### 2.6.2 Global static power-aware mapping

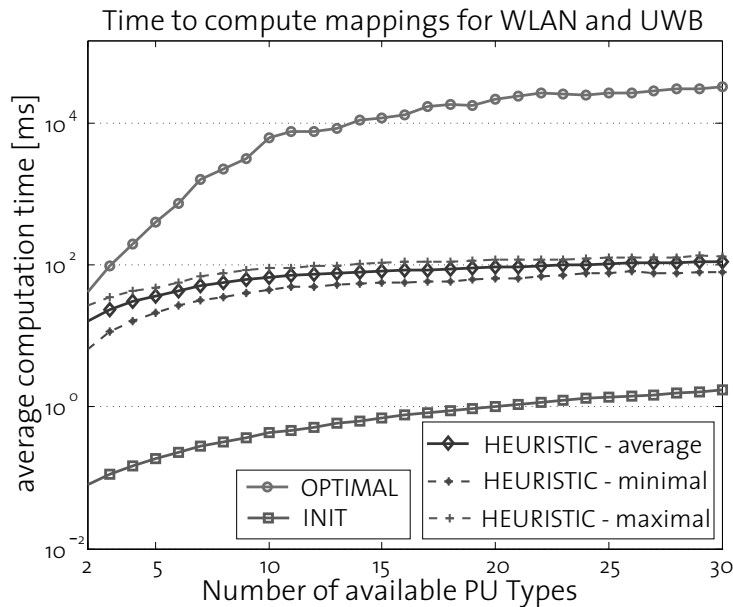
The initial mapping, see Algorithm 2.1, denoted as 'INIT', the heuristic mapping process, see Algorithm 2.2, and the optimal solution, see Equation (2.4), denoted as 'OPTIMAL' are evaluated. The heuristic mapping performance is evaluated as an average, maximum and minimum case. The average case (denoted 'HEURISTIC - average') is the average performance computed from all realizations simulated for a specific PU types library size. The maximum (denoted 'HEURISTIC - maximal') characterizes the worst performance for a specific PU types library size and the minimum (denoted 'HEURISTIC - minimum') characterizes the best performance from all realizations for a given library size.

For each simulation, the time consumed to compute the results is evaluated. For the optimal case, we terminate the ILP after a reasonable amount of time and do not consider those experiments for our results.



**Figure 2.7:** Simulation results for WLAN/UWB.

Figure 2.7 represents the simulation results for systems with WLAN and UWB applications. With increasing PU types library size the performance of the initial mapping is deteriorating. The heuristic performance stays below a factor of 4 even for the maximal case and a large number of PU types. The average performance of our heuristic is below 2 for PU type library size smaller than 15 PU types and below 3 for libraries of less than 30 PU types. The performance degradation is very slow, compared to the initial mapping. This shows the effectiveness of the proposed multi-step heuristic. The initial mapping degrades very fast, due to the fact that tasks

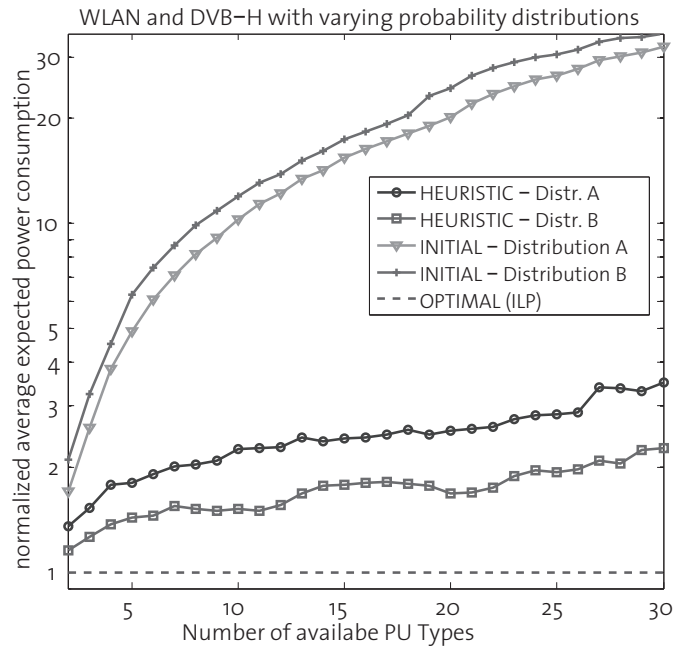


**Figure 2.8:** Time complexity for WLAN/UWB.

are mapped onto processing unit types, that result in minimized dynamic power consumption for that task. As a result, a large number of processing unit instances is created, each hosting only a small number of tasks, i.e., the processing units utilization is small. Conclusively, the overall power consumption increases, due to many low utilized processing unit instances, each consuming static power.

Figure 2.8 represents the timing evaluation of the remapping process for systems with WLAN and UWB applications. Already at 2 available PU types the heuristic mapping process is faster than the optimal. At a library size of 5 PU types, the heuristic process is already one order of magnitude and at a library size of 10 PU types 2 orders of magnitude faster than the optimal algorithm. It can be noted that at a library size of about 25, the time to compute the heuristic remapping process stays constant and the gap between minimal and maximal time elapsed to compute the results for the heuristic approach diminishes.

Increasing the absolute number of tasks makes the problem harder, and thus the performance for systems with WLAN and DVB-H is slightly worse. Figure 2.9 shows the simulation results for systems with WLAN and DVB applications. Here we show results for two different probability distributions. The performance of the initial mapping deteriorates quickly. The heuristic approach still achieves a power performance which is less than 3.5 times worse than the optimal solution. For distribution B, the approximation factor stays below 2.5. Again, the effectiveness of the multi-step

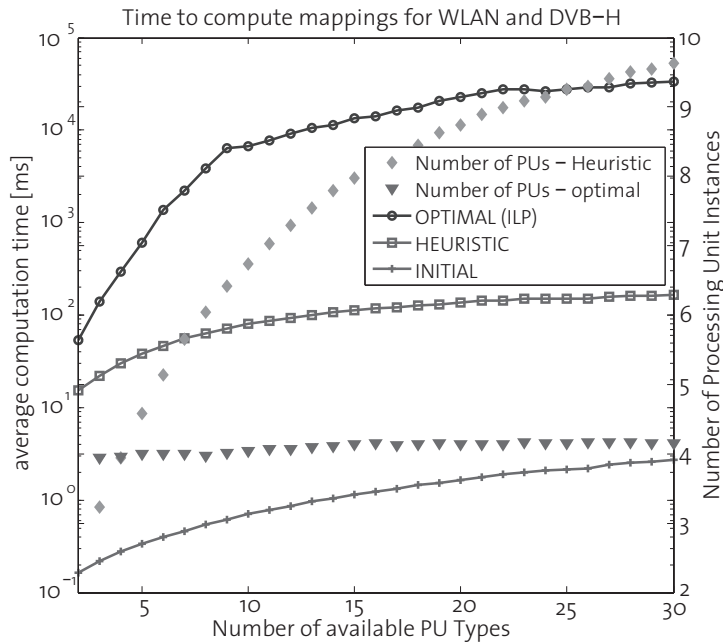


**Figure 2.9:** Simulation results for WLAN/DVB-H for two probability distributions.

heuristics is apparent, despite the fact that no polynomial-time approximation algorithm with a constant *approximation factor* exists.

Figure 2.10 represents the timing evaluation for systems with WLAN and DVB-H applications. The increasing number of tasks, compared to the system with WLAN and UWB presented in Figure 2.8, results in a slightly increased computation time for the heuristic and the optimal approach. As in the previous example, the heuristic approach outperforms the optimal mapping by 2 orders of magnitude in terms of time consumption. The second y-axis in Figure 2.10 corresponds to the number of resource instances that were used in the resulting hardware platform. We denote the number of resource instances that result from the heuristic approach as 'Number of PUs - Heuristic', while the number of resource instances that result from the optimal approach is denoted 'Number of PUs - optimal'. While for the heuristic approach the number of resource instances rises with an increasing number of available PU types, the optimal approach uses a constant amount of resource instances. The heuristic approach suffers from this effect, since an increased number of resource instances results in increased static power consumption.

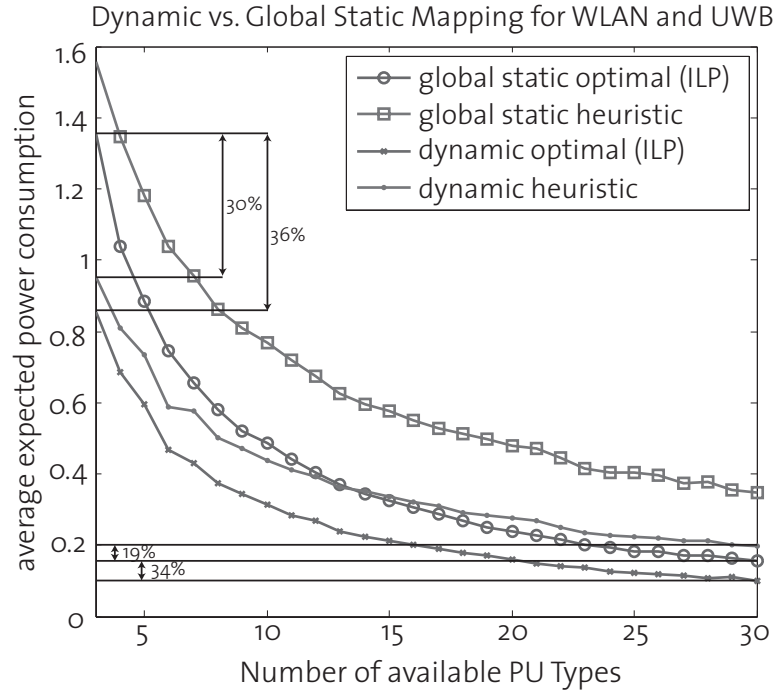
The optimal approach uses an ILP solver to compute the task allocations. The solver is stopped after 20 minutes, in order to limit the total



**Figure 2.10:** Time complexity for WLAN/DVB-H.

amount of time required to perform the experiments. If the computation of the optimal result is terminated prematurely, the result is not included in our experiments. Prematurely terminated computations account for up to 6% for systems with WLAN and DVB-H applications. For PU library sizes below 10 this share is reduced to  $\leq 1\%$  and for libraries with less than 5 available PU types no computation had to be terminated. For systems with WLAN and UWB this performance is enhanced, as the number of tasks is reduced. Prematurely terminated computations for those simulations account for up to 3% and fall below 1% for libraries with less than 10 available PU types.

As shown in the simulation results, the proposed algorithm can derive feasible solutions, and the resulting expected average power consumption of a solution is between 1.1 and 3.5 times of the optimal solution in average cases. The solution stays below twice of the optimal solution for libraries with less than 10 PU types. In terms of computation time a speed up of 2 orders of magnitude is achieved in comparison to the optimal solution for large PU type libraries. Even though we have proved the non-approximability in polynomial-time, the derived solutions are quite promising.

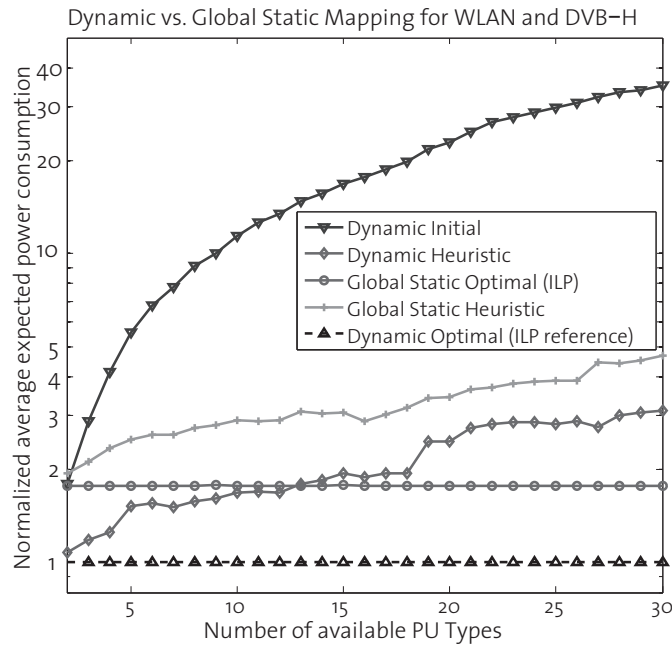


**Figure 2.11:** Global Static and Dynamic Mapping Approach for WLAN and UWB

### 2.6.3 Dynamic power-aware scenario-mapping

In this section we show how the dynamic power-aware scenario-mapping compares to the previously analyzed global static mapping approach. We reuse the previously computed instances of the problem (1000 instances for 6 different probability distributions), and compute a template mapping for each sequence  $c_r \in \mathcal{C}$ , resulting in a corresponding average expected power consumption. For our particular experimental applications, there are 12 scenario sequences. The overall power dissipation can be computed by the sum of the power dissipation of the sequence and their respective probability. Providing templates for different probability distribution increases the computation time and the memory requirement linearly with the number of considered distributions.

In Figure 2.11 we compare the average expected power consumption of the dynamic power-aware scenario mapping approach to the global static mapping approach, assuming a WLAN and an UWB application execute concurrently. We report the average expected power consumption. Where 'global static optimal' represents the expected average power consumption of the global static mapping, solved using Equation (2.4). Similarly, 'global static heuristic' represents the expected average power consumption of the global static mapping, solved using the proposed heuristic, while 'dynamic



**Figure 2.12:** Global Static and Dynamic Mapping vs. Dynamic Optimal Approach for WLAN and DVB-H

optimal (ILP)' and 'dynamic heuristic' represent the results for the optimal approach and heuristic approach applied to each scenario sequence, respectively.

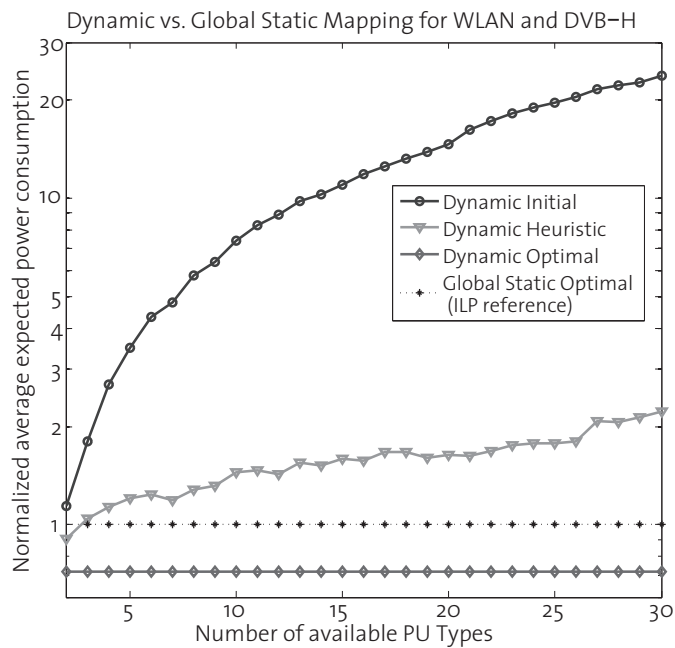
In Figure 2.11, the dynamic optimal approach performs best. The dynamic optimal approach results in a decreased average expected power consumption of about 35% (34% and 36% in Figure 2.11) compared to the global static optimal approach. The dynamic heuristic approach results in an up to 19% increased average expected power consumption in comparison to the global static optimal approach for large problem instances of more than 13 available PU Types. For smaller problem instances the dynamic heuristic approach can reduce the average expected power consumption up to 30% compared to the global static optimal approach. Finally, both the dynamic optimal and the dynamic heuristic approach clearly outperform the global static heuristic approach.

In Figure 2.12, we compare the performance of the global static mapping approaches to the performance of the dynamic approaches, assuming a WLAN and a DVB-H application execute concurrently. The baseline represents the results of the dynamic power-aware scenario-mapping problem, solved using the optimal approach, i.e., applying Equation (2.4) to each scenario sequence and is denoted 'Dynamic Optimal (ILP reference)'. The global static mapping, solved using Equation (2.4), is denoted as 'Global



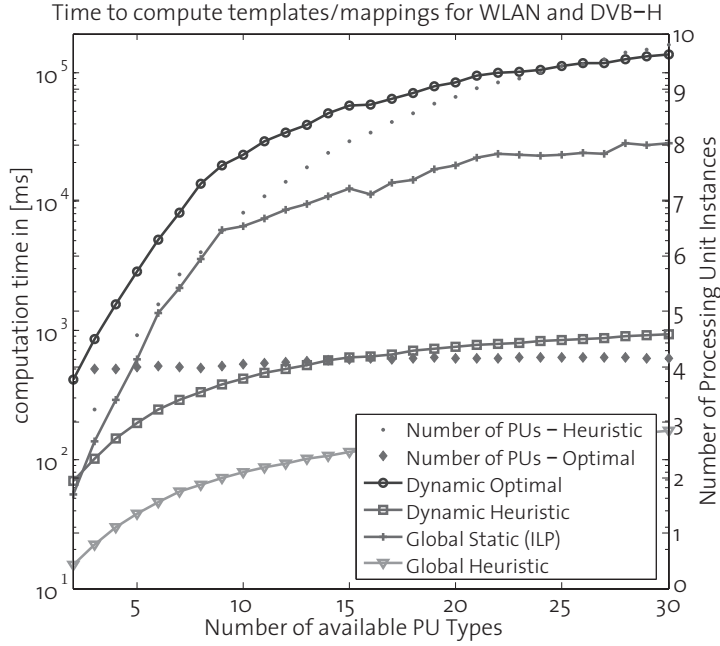
Static Optimal (ILP)', and results in an average expected power consumption that is almost twice the baseline. The heuristic approaches, denoted 'Dynamic Heuristic' and 'Global Static Heuristic' for the dynamic scenario-mapping and the global static mapping approach respectively, show a similar relationship. Until a PU Library size of 13, the dynamic heuristic approach outperforms the global static optimal approach. Finally, the expected average power consumption of the initial mapping for the dynamic approach is reported, denoted as 'Dynamic Initial'.

In Figure 2.13, the baseline represents the results of the global static optimal approach for a single probability distribution. Despite the results in Figure 2.12, the dynamic heuristic approach can only outperform the global static optimal approach for small problem instances. However, the dynamic optimal approach shows to be as efficient as for the problem instance in Figure 2.12, with an improvement in terms of power consumption close to 45%.



**Figure 2.13:** Global Static and Dynamic Mapping vs. Global Static Optimal Approach for WLAN and DVB-H

Large problem instances cannot be computed using the optimal, ILP based, approaches since they fail to compute results in time. Corresponding evaluations are shown in the previous set of experiments for the global static mapping approach. Therefore, heuristics have to be used to compute a mapping and hardware architecture. However, for larger problem instances, heuristic approaches can be used to derive results, and the dy-



**Figure 2.14:** Experimental Results: Time complexity

dynamic heuristic approach performs significantly better than the global static heuristic approach, see Figure 2.12.

Figure 2.14 presents the time required to compute the templates for the dynamic power-aware scenario mapping problem, using the heuristic and the optimal approaches. We compare to their corresponding approaches for the global static mapping problem. Instead of a single global mapping, we have to compute 12 templates. Therefore the required computation time increases compared to the global static mapping approach. However, there are 12 templates to compute but the required computation time is only 9-fold, while the absolute amount of time to compute the templates using the heuristic stays well below 1 second even for large processing unit libraries. In addition, in Figure 2.14 we report the number of processing unit instances in the resulting hardware platform. Similarly to Figure 2.10, once the heuristic approach is applied, the number of processing unit instances rises with the amount of available processing unit types.

Conclusively, computing a template mapping for a single scenario sequence  $c_r \in \mathcal{C}$  is less challenging than computing a global static mapping. In the experiments for the global static mapping approach, it is shown that the optimal global static mapping fails to deliver results in up to 6% due to complexity reasons. The dynamic approach is able to compute optimal templates for all instances and all PU type library sizes in the experiments.

## 2.7 Chapter Summary

This chapter studies the problem how to share the computational resources of a heterogeneous MPSoC among multiple concurrently executing applications. The responsiveness of an application is guaranteed by a utilization bound, while the average expected power consumption of the system is minimized. Applications have multiple modes, each characterized by an execution probability, and are composed of sets of tasks. Multiple concurrently executing multi-mode applications result in a set of scenarios and each scenario can be reached by sequences of mode changes.

- We show that there is no polynomial-time approximation algorithm with a constant approximation factor and provide a polynomial-time heuristic algorithm to solve the allocation problem.
- A multi-step mapping procedure is developed to improve an initial solution that is based on the relaxation of the integer linear programming formulation of the problem.
- A dynamic mapping strategy is proposed, where static mappings for scenario sequences are computed and stored as templates on the system. A manager observes mode changes at runtime and chooses an appropriate precomputed template to assign newly arriving tasks to processing units.

The proposed global static mapping approach derives feasible solutions for the allocation and resource sharing problem. For the set of experiments we performed, the expected average power consumption is between 1.1 and 4 times of the optimal solution. The solution stays below twice of the optimal solution for libraries with less than 10 PU types. In terms of computation time a speed up of 2 orders of magnitude is achieved in comparison to the optimal solution for large PU type libraries. Due to time and memory constraints, the optimal solution fails to derive results for large PU type libraries and large task sets.

For the same set of experiments, the dynamic template based mapping can achieve a reduction of the average expected power consumption of 40 - 45% in comparison to a static global mapping, while keeping the introduced overhead to store the template mappings as low as 1kb. Template mappings for different usage patterns introduce adaptivity to the system and allow maintaining low power consumption over the systems life time.



# 3

## Interference in Resource Sharing MPSoCs

Multiprocessor systems on chip (MPSoCs) and multicore platforms have been widely applied for modern computer systems to reduce production cost and increase computational performance without significantly increasing power consumption. Industrial embedded systems, such as controllers in the avionic and automotive industry, execute increasingly complex tasks, and thus require the computational resources that multicore platforms can offer.

Commercial-Off-The-Shelf (COTS) multicore systems are gaining market share, as the design, evaluation and production of tailor-made systems is getting prohibitively expensive. In the avionic industry, comparatively small numbers of production units would result in a long time-to-market and a large production overhead. In the automotive industry, with a large number of production units, a vast number of third party software vendors require a stable application programming interface (API), such as AutoSAR [Aut].

Multiple processing elements, working collaboratively on a common task, increase the computational power but also increase the need for communication among tasks on different processing elements. Shared resources, such as buses, main memory, and DMA in multicore and MPSoC systems now represent the bottleneck for performance and timing predictability. A single shared main memory, accessed by multiple processing elements, results in contention and may lead to significant delays for tasks.

Multiprocessor and MPSoC systems are typically designed to improve the average-case performance, while worst-case timing guarantees are usually not taken into consideration. However, guarantees on worst-case response/completion times are key requirements for avionic and automotive applications, due to their hard real time constraints.

In this chapter, we introduce resource access and execution models that allow to derive an interference representation, i.e., an arrival curve representation for the access pattern of one processing element on a particular shared resource. Different task models, with varying structural uncertainties, are introduced and the main properties of the assumed hardware platform are presented. The task models, interference representation and assumptions on hardware and shared resources presented here are used in Chapters 4 and 5 to derive analysis methodologies for static and dynamic arbitration policies, respectively.

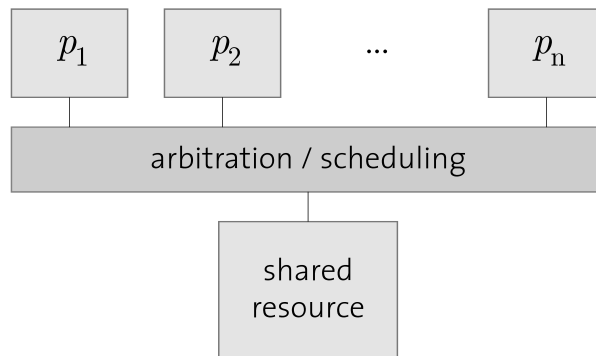
### 3.1 Introduction

Multiprocessor System-On-Chips (MPSoCs) are composed of multiple processing elements, memories and a communication infrastructure. These systems are optimized to increase performance and reduce power consumption in the average-case. However, such systems have recently been considered to be applied in timing critical systems, such as automotive or avionic applications, in which guarantees on worst-case response times are key requirements.

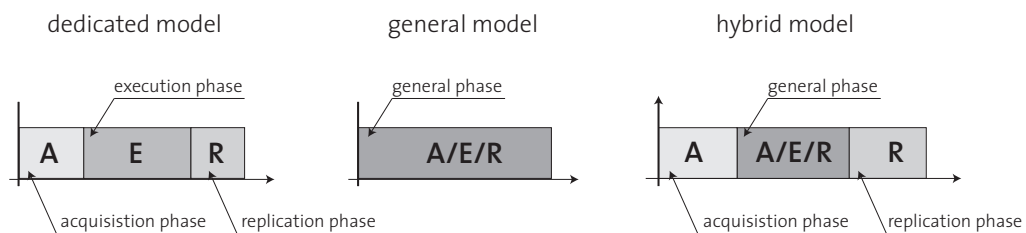
Consider a platform with multiple processing elements and a single shared main memory, see Figure 3.1. Executing a task on a processing element requires fetching of program instructions and acquisition of data from main memory. Moreover, communication among tasks on different processing elements also results in memory accesses. As a result, contention on shared resources in general, and on main memory in particular, significantly delays the completion of tasks.

In this chapter, we consider systems with a single shared resource, that requires a constant amount of time to complete a request. Access to the resource, e.g., a bus in a multicore system or a main memory, is granted for at most one request at a time and results in a corresponding blocking or waiting time for any other request. An ongoing access to a shared resource cannot be preempted.

We consider tasks to be specified by a sequence of non-preemptable superblocks. We propose different models to access shared resources within superblocks: (1) the *dedicated* model, (2) the *general* model and (3) the *hybrid* model, see Figure 3.2.



**Figure 3.1:** Resource Sharing Platform.



**Figure 3.2:** Overview Resource Access Models

In the **dedicated model**, communication and computation are separated from each other. The *acquisition* phase, at the beginning of each superblock, reads data from the shared resource, e.g., acquires information from the shared memory. The acquisition phase performs resource access requests only - no local computation is performed. Upon the start of this phase, computation is required to initiate data transfer. However, computation is negligible in relation to the time required to transfer data from the shared resource. The phase is specified by the maximum and minimum number of access requests and is denoted A in Figure 3.2.

The *execution* phase starts after the acquisition phase has finished. In this phase, computation is performed on local data only. In other words, access requests to the shared resource are not permitted. The phase is specified by the maximum and minimum amount of computation that has to be performed and is denoted E in Figure 3.2.

Finally, the *replication* phase starts after the execution phase has finished. In this phase, data that has been changed or generated during the execution phase is written to the shared resource. Similarly to the acquisition phase, the amount of computation that is required to initiate the data transfer is neglected in our resource access models. The phase is specified by the maximum and minimum number of access requests and is denoted R in Figure 3.2.

phase	resource accesses	computation
acquisition	✓	-
execution	-	✓
replication	✓	-
general	✓	✓

**Table 3.1:** Overview of different phases

In the **general model**, communication and computation are not separated at all. As a result, there is only a single phase, namely the *general* phase. In this phase, access requests to the shared resource and computation can happen at any time and in any order. The phase is specified by the maximum and minimum number of access requests and the maximum and minimum computation. It is denoted A/E/R in Figure 3.2.

The **hybrid model** represents a trade-off between the two previous models. In this model, there is an acquisition phase at the beginning and a replication phase at the end of each superblock. However, there is no execution phase that performs local computation only. In this model, a *general* phase follows the acquisition phase. In the general phase, computation and communication can happen in any order. Acquisition and replication phases are specified as in the dedicated model, while the general phase is specified as in the general model, see Figure 3.2.

Table 3.1 summarizes the behavior of acquisition, replication, execution and general phases.

We consider systems with real time tasks under a given partitioning, in which a set of superblocks is running on a predefined processing element. These sets of superblocks are executed either (1) *sequentially* or (2) *time triggered*. A sequentially executing superblock starts its execution as soon as its preceding superblock has finished. A time triggered superblock starts execution at a predefined time instant. Phases of superblocks are specified by their maximum and minimum computation time and their maximum and minimum number of access requests to a shared resource. This resource grants access according to a particular scheduling policy, e.g., TDMA, First Come First Serve (FCFS), Fixed Priority or any other. The hardware platform is assumed to conform to the the fully timing compositional architecture proposed by Wilhelm et al. [WGR<sup>+</sup>09], i.e., we assume a hardware platform without timing anomalies.

The major contribution of this chapter are:

- We propose models to access the shared resource and models to schedule superblocks on processing elements. The task model is inspired by industrial practice. Extensions are presented that result in increased efficiency and analyzability.



- We introduce arrival curves as representation of the access pattern of the set of superblocks that execute on a particular core. The arrival curve provides an upper bound to the number of access requests that are generated by a processing element in any interval of time.

The rest of this chapter is organized as follows: Section 3.2 gives an overview of related work. Section 3.3 details on the task models and the general model of a shared resource. Section 3.4 proposes arrival curves to represent the access pattern of a particular set of superblocks on a shared resource and shows an efficient algorithm to derive these arrival curves. Section 3.5 concludes the chapter.

## 3.2 Related Work

Systems with shared resources have recently been studied by Pellizzoni et al. [PC07, PBCS08, PSC<sup>+</sup>10], Negrean et al. [NSE09, NSE10], Schliecker et al. [SIE06, SNN<sup>+</sup>08, SN10] and Andersson et al. [AEL10]. In our previous work, [PSC<sup>+</sup>10], we propose a partitioning of tasks into sequentially executing superblocks. Superblocks are specified by their upper bound on access requests to a shared memory and their maximum required computation time. The partitioning of tasks into superblocks is either performed by static analysis of a program or by manually arranging access requests and computation. Different arbitration policies on a shared memory are analyzed and the worst-case delay suffered by a task due to the interference of other tasks on the shared memory is computed.

Schliecker et al. [SIE06, SNN<sup>+</sup>08, SN10] and Negrean et al. [NSE09, NSE10] assume a set of tasks executing on a set of processing elements, all accessing a global shared resource. Accesses to the shared resource are defined as event models, defining the maximum and minimum accesses in a time window. The worst-case interference is then computed in an iterative process. Each transaction takes a certain amount of time to be processed, and therefore the maximal interference that can happen due to higher priority tasks can be derived from the event models. Priorities are assigned statically, and therefore interferences on one task propagate to all lower priority tasks.

In [AEL10], the authors derive a bound on additional execution time due to contention on the memory bus, independent of the actual arbitration policy thereon. As a result, the derived bound is very pessimistic, basically assuming a bus transaction being interfered by all other tasks.

Other works, closely related to interferences on shared resources, focus on interference due to cache accesses, e.g., Guan et al. [GSYY09], Yan and Zhang [YZ08, ZY09], Li et al. [LSL<sup>+</sup>09]. In [GSYY09], the authors propose a scheduling algorithm and a schedulability test based on linear

programming for multiprocessor systems with shared L2 cache, while timing analysis is provided in [LSL<sup>+</sup>09]. Tasks execute following a non-preemptive fixed priority scheme and their worst-case execution time and required cache space sizes are known. The cache is partitioned and a scheduler is proposed such that at no time any two running tasks can have overlapping cache spaces. A schedulability test based on linear programming is presented. In [YZ08], the authors focus on the analysis of the worst-case execution time for a limited hardware model with only a single real-time task. A task executes on a dual-core processor with shared L2 cache and an ILP based approach is presented to derive the worst-case execution time (WCET). In [ZY09], the authors improve the accuracy of the derived WCET, by enhancing the estimation of the inter-thread cache interference.

In [LGY10], Lv et al. propose a technique that uses abstract interpretation to derive the cache pattern of an application. Then, a Timed Automaton is generated that represents the timing information of the accesses to the memory bus. The shared resource is modeled as a Timed Automaton as well, and the UPPAAL model checker is used to find the WCET of the application. The authors present a system with TDMA arbitration and a system with FCFS arbitration on the shared resource as case studies. Using Timed Automata a tight WCET can be found, as all execution traces are considered. However, this approach suffers from scalability issues.

Paolieri et al. [PQnC<sup>+</sup>09] propose a hardware platform that enforces an Upper Bound Delay (UBD). Once this bound is determined, each access request of a hard real-time task (HRT) to a shared resource takes exactly this amount of time. They introduce the WCET Computation Mode. Here, the HRTs execute in isolation, but the platform enforces the UBD for each access request, hence resulting in a safe upper bound on the WCET. This approach allows to analyze HRTs in isolation from each other, since the interference by other tasks is abstracted by the UBD. However, hardware support is required, which is unavailable in many cases, in particular when using COTS systems.

Other approaches also propose designs that eliminate or bound interference. For example, Rosen et al. [RAEP07] and Andrei et al. [AEPR08] use Time Division Multiple Access (TDMA) for bus accesses and a task model, where communication requests are confined to dedicated phases at the beginning and the end of a task.

For non real-time systems, Fedorova et al. [FBZ10] propose a metric to represent contention on shared resources for multicore processors based on the LLC (Last Level Cache) miss rate. They then propose the Distributed Intensity Online (DIO) scheduler and PowerDI (Power Distributed Density) scheduler. In their experiments with the SPEC CPU 2006 benchmark suite, they show that these schedulers result in a significantly improved execution time compared to the standard (Linux) scheduler for most exper-

iments. However, the authors focus on applications that emphasize average case performance rather than timing predictability for hard real-time systems. Conclusively, the LLC miss rate used in this context is based on measurement as opposed to worst-case analysis.

### 3.3 System Model

A system is composed of multiple processing elements  $p_j \in \mathcal{P}$ . The processing elements in  $\mathcal{P}$  execute independently, but share a common resource, e.g., an interconnection fabric (bus) to access a shared memory. Sets of superblocks are scheduled statically on the processing elements and access the shared resource according to a *resource access model*.

#### 3.3.1 Superblock Models

A task is constituted by a sequence of superblocks. Superblocks might have branches and loops, but superblocks constituting a task execute sequentially, i.e., the order of superblocks is the same for every possible execution path of a task. We consider two models for executing superblocks,

1. in the **sequential** model a succeeding superblock is activated as soon as its preceding superblock has finished, and
2. in the **time triggered** model, a superblock starts execution at a predefined time.

Superblocks might be further structured in phases: *acquisition phase*, *execution phase*, *replication phase* and *general phase*. We consider different models to access shared resources within superblocks:

- the *dedicated* model,
- the *general* model and
- the *hybrid* model.

Acquisition and replication phase perform accesses to the shared resource only, i.e., no local computation is performed. The execution phase performs local computation only, without accessing the shared resource. In the general phase, access requests as well as computation can happen. Table 3.2 gives an overview.

phase	resource accesses	computation
acquisition	$\mu^{max,a}, \mu^{min,a}$	0
execution	0	$exec^{max}, exec^{min}$
replication	$\mu^{max,r}, \mu^{min,r}$	0
general	$\mu^{max,e}, \mu^{min,e}$	$exec^{max,e}, exec^{min,e}$

**Table 3.2:** Parameters specifying phases

### Dedicated model

Accesses to the shared resource are limited to the acquisition phase at the beginning of the superblock and to the replication phase at the end of the superblock. After the activation of a superblock, requests to the shared resource are issued, e.g., to receive required data. In the succeeding execution phase, the actual computation takes place, while accesses to the shared resource are prohibited. After results are computed, the replication phase is used to update the corresponding data on the shared resource. Once the replication phase is finished, the superblock completes. See superblock  $s_{3,1}$  in Figure 3.3 for an example, where A, E and R represent the acquisition, execution and replication phase, respectively. Requests to the shared resource, as well as the time required for computation, are specified as upper bounds. The parameters for superblock  $s_{i,j}$  are:

- $\mu_{i,j}^{max,a}, \mu_{i,j}^{min,a}$ : maximum/minimum number of requests in acquisition phase,
- $\mu_{i,j}^{max,r}, \mu_{i,j}^{min,r}$ : maximum/minimum number of requests in replication phase, and
- $exec_{i,j}^{max}, exec_{i,j}^{min}$ : maximum/minimum execution time excluding resource accesses.

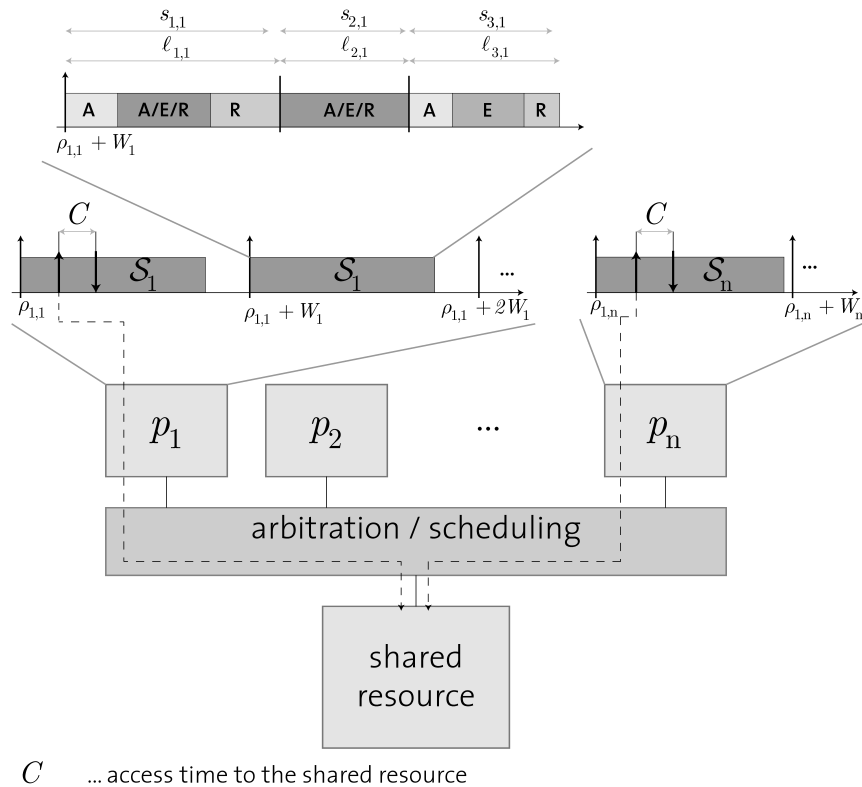
### General model

Accesses to the shared resource are not limited to specific phases and can happen at any time and in any order. Conclusively,  $\mu_{i,j}^{max,a}, \mu_{i,j}^{min,a}, \mu_{i,j}^{max,r}$  and  $\mu_{i,j}^{min,r}$  are 0, while parameters  $\mu_{i,j}^{max,e}$  and  $\mu_{i,j}^{min,e}$  define the maximum number of requests during the active time of the superblock. Superblock  $s_{2,1}$  in Figure 3.3 gives an example, where A/E/R represents the general phase. The execution time is bounded by  $exec_{i,j}^{max,e}$  and  $exec_{i,j}^{min,e}$ . A superblock modeled according to the general model consists of a single general phase.

### Hybrid model

Accesses to the shared resource can happen in the acquisition, the general, and the replication phase. This model allows to access the shared resource outside the dedicated acquisition and replication phases, e.g., reloading altered data during the general phase. Requests to the shared resource during the general phase can happen anytime and are constrained by an upper and lower bound. See superblock  $s_{1,1}$  in Figure 3.3 as an example, where A and R represent the acquisition and replication phase, while A/E/R represents the general phase with possible access requests. As a result, in addition to the parameters specified for the dedicated model, the hybrid model also defines and maximum and minimum amount of access requests during the general phase, i.e., the parameters  $\mu_{i,j}^{max,e}$  and  $\mu_{i,j}^{min,e}$ , respectively.

Without loss of generality, the hybrid access model covers the definitions of the dedicated model and the general model. The dedicated model can be represented by setting the parameters  $\mu_{i,j}^{max,e} = 0$  and  $\mu_{i,j}^{min,e} = 0$ . The general model can be modeled by setting parameters  $\mu_{i,j}^{max,a} = 0$ ,  $\mu_{i,j}^{min,a} = 0$ ,  $\mu_{i,j}^{max,r} = 0$  and  $\mu_{i,j}^{min,r} = 0$ , while  $\mu_{i,j}^{max,e}$  and  $\mu_{i,j}^{min,e}$  represent the corresponding access requests.



**Figure 3.3:** Resource Sharing Platform Overview

### Application of the models

The *dedicated* model requires efforts to confine communication to dedicated phases. This restriction is compensated by increased performance and predictability. For many applications in the domain of control and signal processing, complying to these restrictions does not present a major obstacle, since execution frequencies are typically known in advance. Applications with user-interaction and/or event-triggered behavior cannot comply to this model, and the *general* model has to be applied. The *hybrid* model presents a way to increase the analyzability and performance of a system that is otherwise designed according to the general model.

### Scheduling of superblocks

We consider a (given) repeated schedule of length  $W_j$  time units, denoted as *processing cycle*, on processing element  $p_j$ , in which a superblock  $s_{i,j}$  starts at time  $\rho_{i,j}$ . The first superblock in the first processing cycle starts at time 0, i.e., in the first processing cycle  $\rho_{1,j} = 0, \forall j$ . Sets of superblocks, denoted  $\mathcal{S}_j$  on  $p_j$ , are executed in the repeating time interval  $(\rho_{1,j}, W_j]$ , see Figure 3.3.

Superblocks are indexed by a pre-defined order  $s_{1,j}, s_{2,j}, \dots, s_{|S_j|,j}$ , in which  $s_{i,j}$  precedes  $s_{i+1,j}$ . For superblocks executing according to the sequential execution model, the earliest starting time of superblock  $s_{i,j}$  is  $\rho_{i,j}$  and its *relative deadline* is  $\ell_{i,j}$ . Subsequent superblocks start as soon as their preceding superblocks have finished, see Figure 3.4 for an example of subsequently executing superblocks.

In the time-triggered execution model, each superblock or phase can have a dedicated starting time. For models with time triggered superblocks, but subsequently executing phases within superblocks, each superblock  $s_{i,j}$  has a dedicated starting time  $\rho_{i,j}$ , see Figure 3.5a and 3.5b. Once phases are executed in a time-triggered manner, as in Figure 3.5c, each phase is assigned a dedicated starting time, i.e.,  $\rho_{i,j,[a|e|r]}$  is the starting time of the (*a*)cquisition, (*e*)xecution and (*r*)eplication phase of superblock  $s_{i,j}$ , respectively.

Specifically, for time triggered models, the earliest starting time  $\rho_{i,j}$  of superblock  $s_{i,j}$  must be no less than the deadline of the preceding superblock, i.e.,  $\rho_{i-1,j} + \ell_{i-1,j}$ . That is, to meet the timing constraint, an instance of superblock  $s_{i,j}$  released at time  $gW_j + \rho_{i,j}$  must be finished no later than  $gW_j + \rho_{i,j} + \ell_{i,j}$ , where the completion time minus the release time is the *response time* of the superblock. For simplicity, we assume that the deadline of a superblock released in a processing cycle on  $p_j$  is no more than the end of the processing cycle.

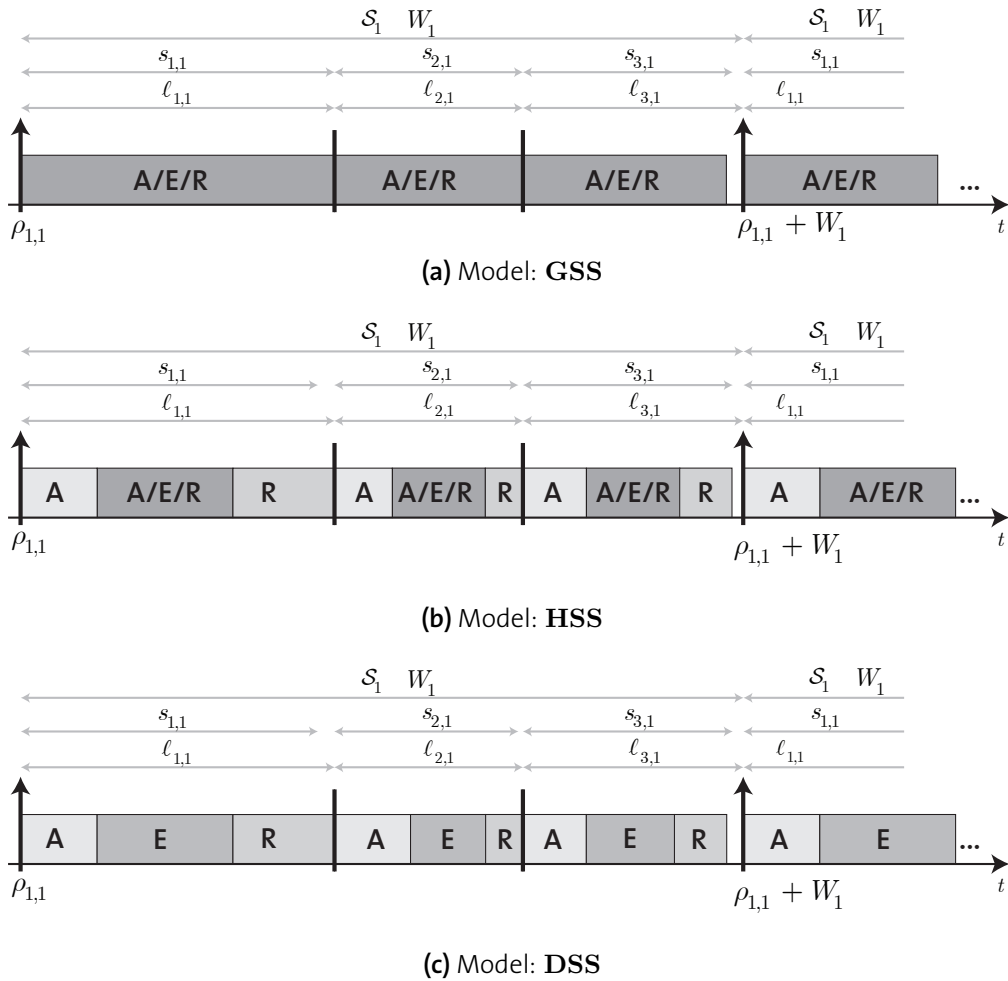


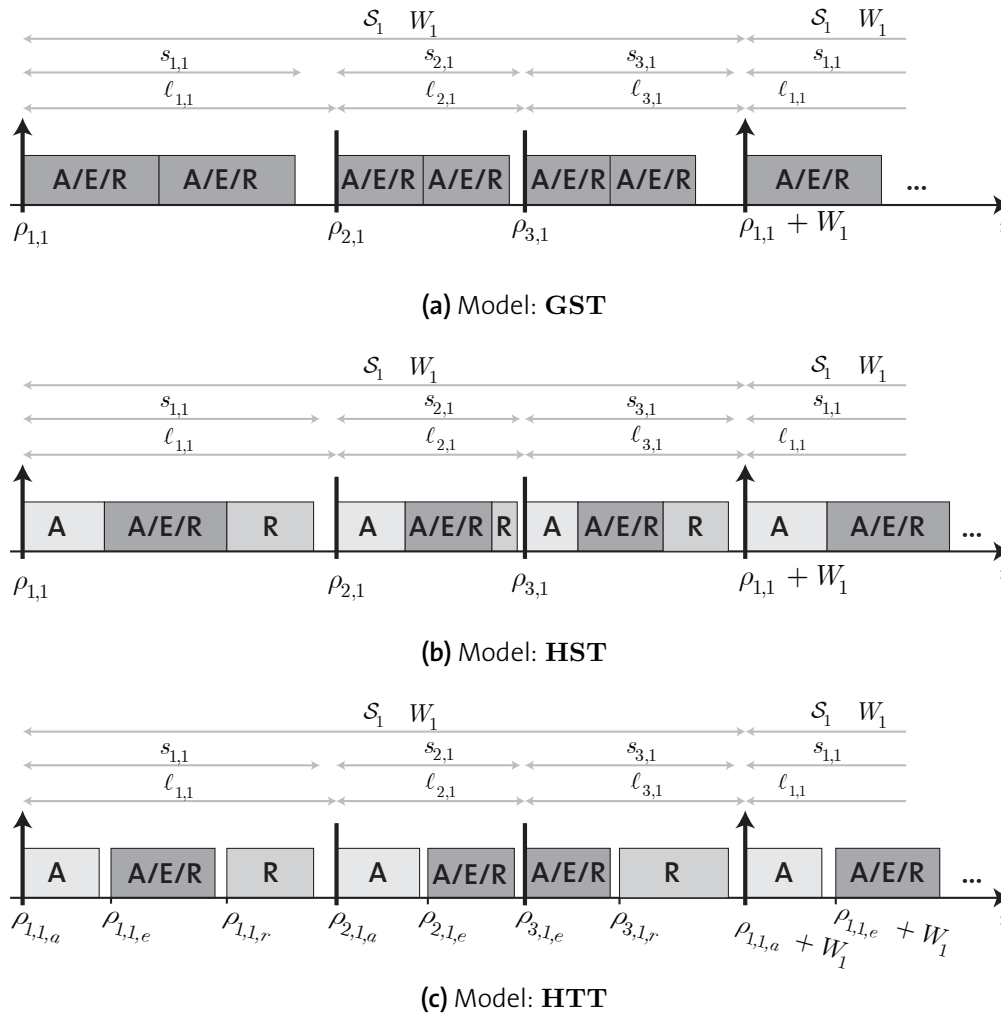
Figure 3.4: Sequential access models.

### 3.3.2 Resource Access Models

Based on these models to trigger the execution of superblocks and to access the shared resource, we derive the following resource access models:

**GSS** - **General Sequential** phases, **Sequential** superblocks. Superblocks execute sequentially and accesses to the shared resource can happen anytime and in any order, see Figure 3.4a.

**HSS** - **Hybrid Sequential** phases, **Sequential** superblocks. Superblocks execute sequentially and accesses to the shared resource are issued in dedicated acquisition and replication phases. Additionally, in the general phase, accesses to the shared resource can be issued at any time, see Figure 3.4b.



**Figure 3.5:** Time-Triggered access models.

**DSS** - **D**edicated **S**equential phases, **S**equential superblocks. Superblocks execute sequentially and accesses to the shared resource are restricted to the acquisition and replication phases, see Figure 3.4c.

**GST** - **G**eneral **S**equential phases, **T**ime triggered superblocks. Superblocks start execution at dedicated points in time and accesses to the shared resource can happen at any time and in any order, see Figure 3.5a.

**HST** - **H**ybrid **S**equential phases, **T**ime triggered superblocks. Superblocks start execution at dedicated points in time and accesses to the shared resource are specified according to the hybrid model, i.e., accesses are issued in the acquisition, general and replication phases, see Figure 3.5b. Phases of a superblock execute sequentially.



**HTT** - **H**ybrid **T**ime triggered phases, **T**ime triggered superblocks. Superblocks are specified according to the hybrid model and each phase starts at a statically defined point in time. Accesses to the shared resource are issued in the acquisition, replication, and general phases, see Figure 3.5c.

### 3.3.3 Model of the Shared Resource

This thesis considers systems with a stateless shared resource. Any request to the shared resource has to wait until it is granted by the resource arbiter. After a request is granted, the shared resource starts to serve the request. At any time, the shared resource can serve at most one request. After an access request is granted access to the shared resource, the accessing time to the shared resource is (bounded by) a *constant*  $C$ . Therefore, if a superblock  $s_{i,j}$  can access the shared resource at any time, the maximal time for executing the superblock  $s_{i,j}$  is  $exec_{i,j}^{max} + (\mu_{i,j}^{max,a} + \mu_{i,j}^{max,e} + \mu_{i,j}^{max,r}) \cdot C$ . The shared resource is assumed non-preemptive, and hence, the resource arbiter only grants access to a pending request when there is currently no other request served by the shared resource. Moreover, resource accesses in this thesis are assumed *non-buffered*, such as shared memory access due to cache misses. This means that a task has to stall until its request to the shared resource is served successfully.

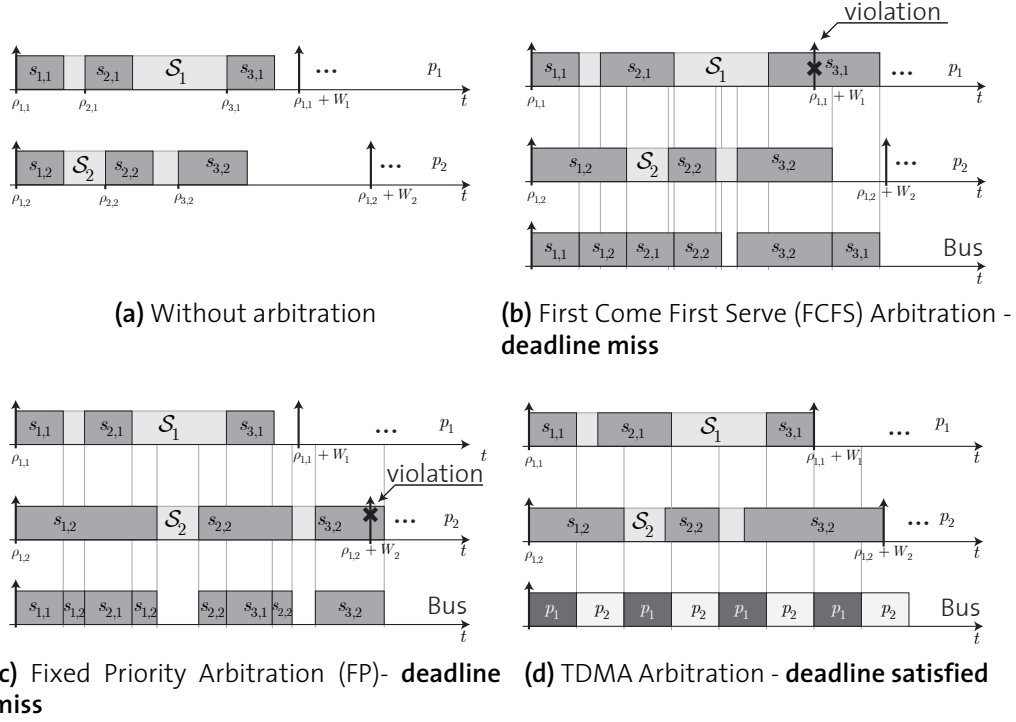
#### Motivational Example for different Arbiters

In Figure 3.6a, the sets of superblocks  $\mathcal{S}_1$  and  $\mathcal{S}_2$  are assigned to processing element  $p_1$  and  $p_2$  respectively. If the superblocks do not access the shared resource, as in Figure 3.6a, then they finish their execution before their deadline (equals period). Once superblocks require access to a shared resource, this resource is arbitrated among the competing processing elements. In Figure 3.6 we present a selection of arbitration policies and their effect on the schedulability of the system. Dark gray areas represent phases that access a shared resource, while light gray areas represent phases that perform computation local to the corresponding processing element.

The system would be unschedulable under First Come First Serve (FCFS), Figure 3.6b, and Fixed Priority (FP), Figure 3.6c, arbitration policies. However, applying a well designed TDMA arbiter results in the system being schedulable, see Figure 3.6d.

## 3.4 Interference on shared resources

As shown in the previous section, concurrent execution of multiple superblocks on multiple processing elements, all accessing a shared resource,



**Figure 3.6:** Two tasks on two processing elements accessing a shared bus with different arbitration policies.

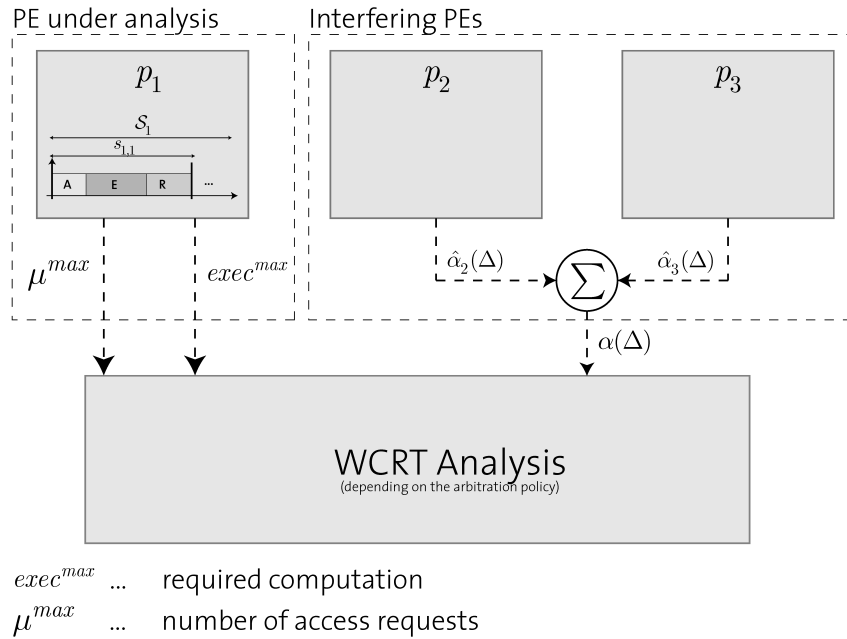
leads to contention on the shared resource. Due to this contention, the execution time can be significantly increased. In this section, we present an approach to represent the resource access pattern of superblocks onto the shared resource. We use arrival curves as described in [TCN00] as data structure to represent this access pattern. The upper arrival curve represents the maximal number of access requests that can happen in any time window of length  $\Delta$ . The worst-case response time (WCRT) analysis approach presented in Chapter 5 uses this representation to account for interference on the shared resource.

### 3.4.1 Analysis Overview

Consider a set of superblocks  $\mathcal{S}_j$  executing on a particular processing element  $p_j$ . In this section, we propose an approach to represent the resource access pattern of such a set of superblocks onto a shared resource as an arrival curve. The set of superblocks is modeled according to the resource access models described in Section 3.3.2. Deriving the arrival curve representation is performed in a multi-step approach. First, we derive an arrival curve representing the resource access pattern for each processing element in a multicore platform. Second, we compute the arrival curve that repre-

sents the joined interference of a set of processing elements as the sum of the arrival curves representing the single processing elements. Then, when the WCRT of the set of superblocks is performed, the corresponding processing element is considered as the processing element under analysis, while all the other processing elements are considered by their joined arrival curve.

In the WCRT analysis approach presented in Chapter 5, one processing element is considered as the processing element under analysis, while the interference by all the other processing elements in the system is represented by a single arrival curve. Figure 3.7 presents an overview. Processing element  $p_1$  is the processing element under analysis. The WCRT analysis considers the resource access pattern of the processing element under analysis, as defined by the set of superblocks  $\mathcal{S}_1$ , and the interference by all other processing elements, denoted  $\alpha(\Delta)$ .



**Figure 3.7:** Arrival Curve  $\alpha(\Delta)$  representing interference of two processing elements to the processing element under analysis.

### Deriving an arrival curve representation for a single processing element

Superblocks in  $\mathcal{S}_j$  are defined with their upper and lower computation time and their upper and lower amount of access requests. Deriving an arrival curve for a set of superblocks involves the following steps:

- From the specification of the superblocks, we derive tuples of the form  $\hat{t} = \langle \gamma, \Delta \rangle$ , where  $\gamma$  describes the amount of resource accesses that

can happen in a time window of length  $\Delta$ . These tuples are computed by (1) deriving the time windows  $\Delta$  that are relevant, see Section 3.4.3.2, and (2) deriving the number of access requests for each time window, see Section 3.4.3.3. Note that for each time window, there are several possible values for  $\gamma$ , i.e., the number of resource accesses. Eventually, only the maximum number of access requests for each time window  $\Delta$  will remain in the final arrival curve representation.

- The set of relevant time windows is computed from the definition of the set of superblocks. The set of superblocks is an ordered set. From this set, we compute all possible ordered subsets. As an example, a set of two superblocks results in four possible ordered subsets. Time windows are computed as a function of the computation and the number of access requests of the superblocks in a particular subset. Each subset defines multiple time windows, see Section 3.4.3.1, as parameters of superblocks are defined as minimum and maximum values rather than exact values. Note that different subsets coincidentally might result in time windows of equal length, however their corresponding numbers of access requests differ. Conclusively, the one subset with the higher number of access requests will contribute to the arrival curve.
- Deriving the tuples for all possible time windows  $\Delta$  allows to derive an arrival curve representation. Upper arrival curve  $\hat{\alpha}^u(\Delta)$  is derived by considering the maximum amount of access requests  $\gamma$ , for a time window of length  $\Delta$ , from all the tuples, see Equation (3.7).
- The derivation of tuples differs for sets of superblocks that are executed sequentially, see Section 3.4.3, or that are executed in a time-triggered manner, see Section 3.4.4. We show the derivation of time windows and tuples for the sequentially executing models. In Section 3.4.4 we show the required changes to the methods to handle the time-triggered case.

As a result, each processing element of a multicore platform is represented by an arrival curve. The arrival curve represents the access pattern of a processing element in isolation, i.e., no interference on the shared resource is considered during the derivation of the arrival curve. When computing the WCRT of the set of superblocks on a particular processing element, the interference by all the other processing elements is considered as the sum of their representative arrival curves. This represents a safe upper bound on the interference that one processing element might suffer.

### Considering different resource access models

The resource access models, proposed in Section 3.3.2, differ by the amount of known structure inside a superblock. Additional structure is represented by phases, such as dedicated phases to access the shared resource. For ease of notation, we translate these models such that each phase is represented as a superblock with corresponding parameters. As an example, consider Figure 3.8 where superblocks with dedicated phases are translated into an equivalent set of superblocks. Superblock  $s_{1,1}$  with acquisition, execution and replication phase translates to superblocks  $s'_{1,1}, s'_{2,1}, s'_{3,1}$ . Superblock  $s'_{1,1}$  represents the acquisition phase with  $exec_{1,1}^{max} = 0, exec_{1,1}^{min} = 0, \mu_{1,1}^{max} = \mu_{1,1}^{max,a}$  and  $\mu_{1,1}^{min} = \mu_{1,1}^{min,a}$ . Superblock  $s'_{2,1}$  represents the execution phase, and thus  $exec_{2,1}^{max} = exec_{1,1}^{max}, exec_{2,1}^{min} = exec_{1,1}^{min}, \mu_{2,1}^{max} = 0$  and  $\mu_{2,1}^{min} = 0$ . The remaining superblocks translate analogously. In the remainder of this chapter, we assume that this translation has been done, i.e., we only consider sets of superblocks where the superblock is the atomic unit and all parameters are related to superblocks.

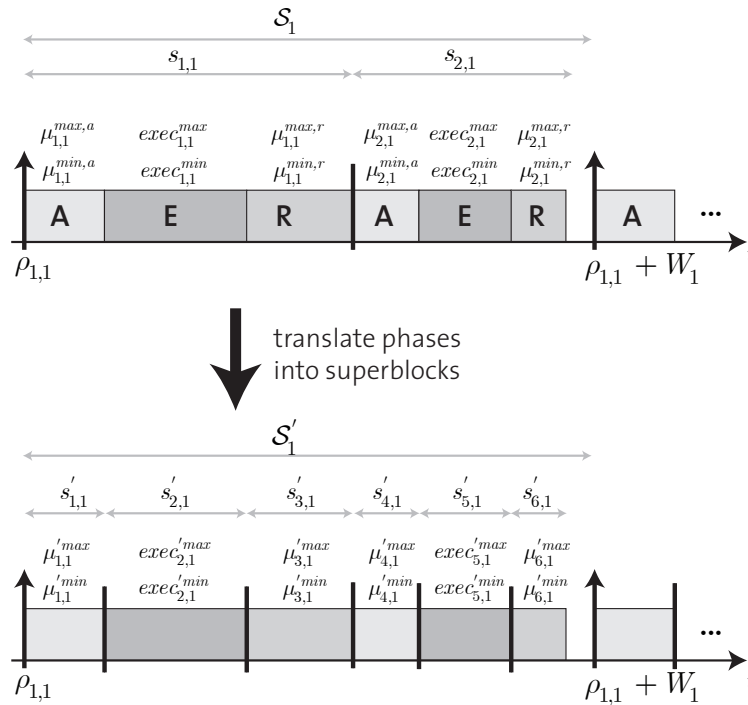


Figure 3.8: Translate phases to superblocks.

#### 3.4.2 Analysis Methodology

A set of superblocks  $\mathcal{S}_j$  executing on processing element  $p_j$ , with processing cycle  $W_j$ , accesses the shared resource according to a pattern. This pat-

tern is specified by the minimum and maximum number of access requests and minimum and maximum computation per superblock. We represent this pattern as an arrival curve [TCN00], specifying the maximum number of access requests for time windows of length  $\Delta$ . Arrival curves are represented by an initial aperiodic part and a periodic part that continues to infinity. This way, an infinite arrival curve can be represented by a finite data structure. The aperiodic part can be derived from two succeeding processing cycles of a set of superblocks. This is required in order to include the behavior at the transition from one instance to the other. The periodic part can be derived from the aperiodic part and the maximum number of access requests that are issued in one period of the set of superblocks that is considered, see Equation (3.8).

An arrival curve represents the access pattern of a set of superblocks on the shared resource, assuming no interference occurs on the shared resource. In other words, the set of superblocks is analyzed in isolation and no other processing elements compete for access to the shared resource. Access requests in a superblock can happen in bursts, i.e., once an access request is issued it is served instantaneously and the subsequent access request can be issued.

### 3.4.3 Sequential execution of superblocks

In this section we introduce our approach to represent accesses to a shared resource as an arrival curve. We assume that superblocks execute sequentially, i.e., a superblock starts as soon as its predecessor has finished, or at time 0, if it is the first superblock.

#### 3.4.3.1 Computing ordered subsets of superblocks

The aim of this approach is to find an upper arrival curve, representing the maximal amount of access requests in a time window of length  $\Delta$ . In this section, we show how to derive subsets from the set of superblocks. These subsets are then used to compute time windows in Section 3.4.3.2 and corresponding tuples in Section 3.4.3.3.

Consider the set of superblocks  $\mathcal{S}_1$  that executes on processing element  $p_1$  to be the ordered set  $\mathcal{S}_1 = \{s_{1,1}, s_{2,1}\}$ . Then the set of all possible ordered subsets  $*\mathcal{S}_1 = \{\emptyset, \{s_{1,1}\}, \{s_{2,1}\}, \{s_{1,1}, s_{2,1}\}\}$ . We refer to elements of  $*\mathcal{S}_1$  as subset  $t'_{m,d}$ , where  $m$  denotes the index of the first constituting superblock of the subset and  $d$  denotes the distance between first and last superblock of the subset.

In order to account for the transition phase between successive processing cycles, we consider two subsequent instances of  $\mathcal{S}_j$ . Therefore we specify  $\mathcal{S}'_j = \{\mathcal{S}_j, \mathcal{S}_j\}$ , such that  $\mathcal{S}'_j = \{s_{1,j} \dots s_{|S_j|,j}, s_{1,j} \dots s_{|S_j|,j}\}$  and  $*\mathcal{S}'_j$  is

the corresponding set of ordered subsets. Subset  $t'_{m,d} \in *S'_j$  now computes as:

$$t'_{m,d} = \{s_{m,j}, \dots, s_{m+d,j}\} \forall d \in [0 \dots |\mathcal{S}_j| - 1], \forall m \in [1 \dots |\mathcal{S}_j|]. \quad (3.1)$$

Since distance  $d \in [0 \dots |\mathcal{S}_j| - 1]$  and offset  $m \in [1 \dots |\mathcal{S}_j|]$ , the set of subsets  $*S'_j$  covers two periods and has  $|\mathcal{S}_j|^2$  elements. Note that distance  $d$  start at 0, i.e., subsets  $t'_{m,0}$  consider only a single superblock. Contrarily, offset  $m$  starts with 1, as it represents an index. Based on the subsets in  $*S'_j$ , time windows and their corresponding number of accesses to the shared resource can be computed in the following sections. The resulting tuples are utilized to eventually derive the arrival curve representation.

### 3.4.3.2 Computing time windows

Based on subset  $t'_{m,d}$ , computed in the previous section, we now derive the corresponding time windows. These time windows are derived for each subset  $t'_{m,d}$  by considering (1) the computation time and (2) the number of resource access of the constituting superblocks. However, superblocks are defined by minimum and maximum computation time and by their minimum and maximum number of access requests to the shared resource. As a result, there are a number of time windows related to each subset. Consider subset  $t'_{m,d}$ , then the corresponding time windows compute as:

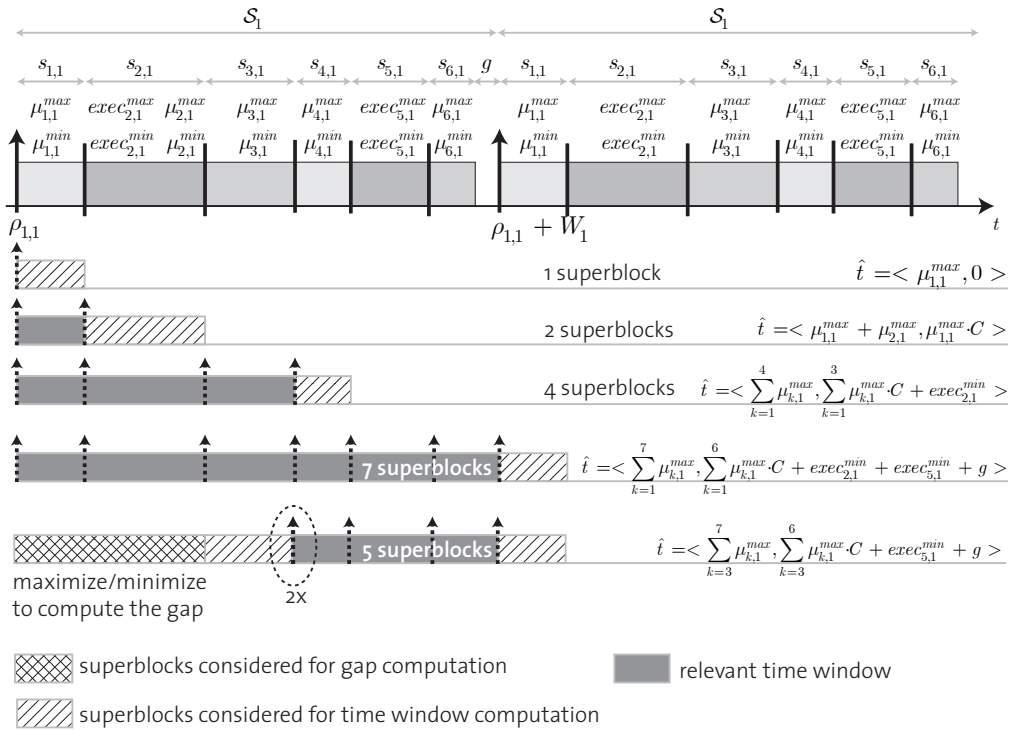
$$\Delta_{m,d}^{e,r} = \sum_{i=m+1}^{m+d-1} exec_{i,j}^e + \sum_{i=m}^{m+d-1} \mu_{i,j}^r \cdot C, \quad (3.2)$$

where the superscripts  $e = max$  or  $e = min$  denote maximum or minimum computation time, respectively, and  $r = max$  or  $r = min$  denote maximum or the minimum number of access requests, respectively. Hence, the time windows related to subset  $t'_{m,d}$ , are  $\Delta_{m,d}^{min,max}$ ,  $\Delta_{m,d}^{min,min}$ ,  $\Delta_{m,d}^{max,max}$  and  $\Delta_{m,d}^{max,min}$ .

In this section, we derive an upper arrival curve representation of the access pattern of a set of superblocks onto the shared resource. Therefore, our aim is to find time windows that are as small as possible, but exhibit as many as possible access requests. In other words, we seek to find the worst-case number of access requests that can happen in a particular time window.

Consequently, time windows for subset  $t'_{m,d}$  are computed by disregarding the computation time of the last and first superblock in  $t'_{m,d}$ , i.e., index  $i$  in the first part of Equation (3.2) starts at  $m+1$  and stops at  $m+d-1$ . Intuitively, in the worst-case, all the access requests in the first superblock are issued at its end in a burst, after computations have been performed. Now consider the successive superblock. Access requests in this superblock are issued at its beginning in a burst, before computations are being performed.

As a result, in a time window that corresponds to the time required to serve the access requests of the first superblock, the total number of access requests that are issued is the sum of access requests in both superblocks. Note that index  $i$  in the second part of Equation (3.2) starts with the first superblock of  $t'_{m,d}$ , i.e.,  $i = m$ , while it ends at  $i = m + d - 1$ . This reflects the possible burst of access requests in the last superblock. In the worst-case, all access requests of the last superblock are issued immediately upon its activation, i.e., no time has to pass and the amount of time required to serve these access request must not be considered for computing time windows.



**Figure 3.9:** Computing time windows for sequences of 1, 2, 4, 5 and 7 superblocks including the gap between periods, for the upper arrival curve. Note that in the circled dotted area, labeled "2x", two superblocks issue their respective access requests, namely  $s_{3,1}$  and  $s_{4,1}$ .

Consider Figure 3.9 for an example how to compute time windows. Note that for each example in Figure 3.9 only one tuple is presented, namely the tuple with the maximum amount of access requests and minimum computation time, i.e., with  $\Delta_{m,d}^{min,max}$ . Tuples representing time windows  $\Delta_{m,d}^{min,min}$ ,  $\Delta_{m,d}^{max,max}$  and  $\Delta_{m,d}^{max,min}$  are not displayed for readability of the figure. In the first example, denoted 1 superblock, the length of the time window computes as zero, meaning that accesses to the shared resource occur concurrently at one instant of time. In example 2 superblocks, the time window



$\Delta$  is computed as the time required to process the first superblocks accesses. However, the corresponding number of access requests is the sum of the issued requests in both superblocks, as explained earlier. Execution times are not considered for the time window, since in the worst case computation is performed before access requests are issued in the first superblock and after access requests are issued in the last superblock. In example 4 superblocks in Figure 3.9, the execution time of superblock  $s_{2,1}$  has to be considered for the time window computation. This superblock is surrounded by other superblocks, and therefore there is no degree of freedom to arrange access requests and computation. In other words, no matter how access requests and computation is arranged in superblock  $s_{2,1}$ , the time windows computed for this example do not change.

### Transition between successive processing cycles

Computing time windows for subsets  $t'_{m,d}$  that span over the period, have to consider the gap  $g$  between the successive processing cycles. Examples 7 superblocks and 5 superblocks in Figure 3.9 illustrate such cases. Gap  $g$  represents the slack time of a set of superblocks, before a subsequent processing cycle starts. Minimizing this gap minimizes the length of the considered time window, but has no influence on the number of access requests that can be issued by the subset. As a result, the worst case time windows for subsets  $t'_{m,d}$  are derived when the gap is minimized.

Minimizing the gap, and deductively the time window, is done by assuming maximum execution time  $exec_{i,j}^{max}$  and maximum number of access requests  $\mu_{i,j}^{max}$  for those superblocks that are not included in  $t'_{m,d}$ . Then, the gaps that have to be considered can be computed by Equation (3.3), for  $r = min$  and  $r = max$ . In Figure 3.9, example 5 superblocks shows such a case. Minimizing the gap equals maximizing the area with diamond hatching. As a result, in order to consider the gap for deriving the upper arrival curve, the two tuples that consider  $g(min)$  and  $g(max)$ , respectively, have to be considered, see Equation 3.6.

$$g(r) = W_j - \sum_{\forall s_{i,j} \in \mathcal{S}_j \setminus (\mathcal{S}_j \cap t'_{m,d})} exec_{i,j}^{max} + \mu_{i,j}^{max} \cdot C \quad (3.3)$$

$$- \sum_{\forall s_{i,j} \in \mathcal{S}_j \cap t'_{m,d}} exec_{i,j}^{min} + \mu_{i,j}^r \cdot C,$$

#### 3.4.3.3 Computing the bound on access requests for time windows

Time windows  $\Delta$  and the corresponding number of access requests to the shared resource  $\gamma$  for an element  $t'_{m,d}$  are computed in Equations (3.4)

and (3.5). Parameters  $e$  and  $r$  represent the maximum and minimum computation and maximum and minimum number of access requests, respectively. All possible combinations of  $e$  and  $r$  need to be considered, in order to guarantee that the worst case behavior is included.

$$\gamma^r = \sum_{i=m}^{m+d} \mu_{i,j}^r \quad (3.4)$$

$$\Delta^{e,r} = \sum_{i=m+1}^{m+d-1} exec_{i,j}^e + \sum_{j=m}^{m+d-1} \mu_{i,j}^r \cdot C \quad (3.5)$$

Based on these values, the tuples for element  $t'_{m,d} \in *S_j$  are computed by Equation (3.6).

$$\hat{t}_{m,d}^{e,r} = \langle \gamma^r ; \Delta^{e,r} + g(r) \rangle \quad (3.6)$$

Computing the upper arrival curve  $\alpha^u(\Delta)$  requires to consider tuples  $\hat{t}_{m,d}^{min,max}$ ,  $\hat{t}_{m,d}^{min,min}$ ,  $\hat{t}_{m,d}^{max,max}$  and  $\hat{t}_{m,d}^{max,min}$ , for all distances  $d \in [0 \dots |\mathcal{S}_j| - 1]$  and offsets  $m \in [1 \dots |\mathcal{S}_j|]$ .

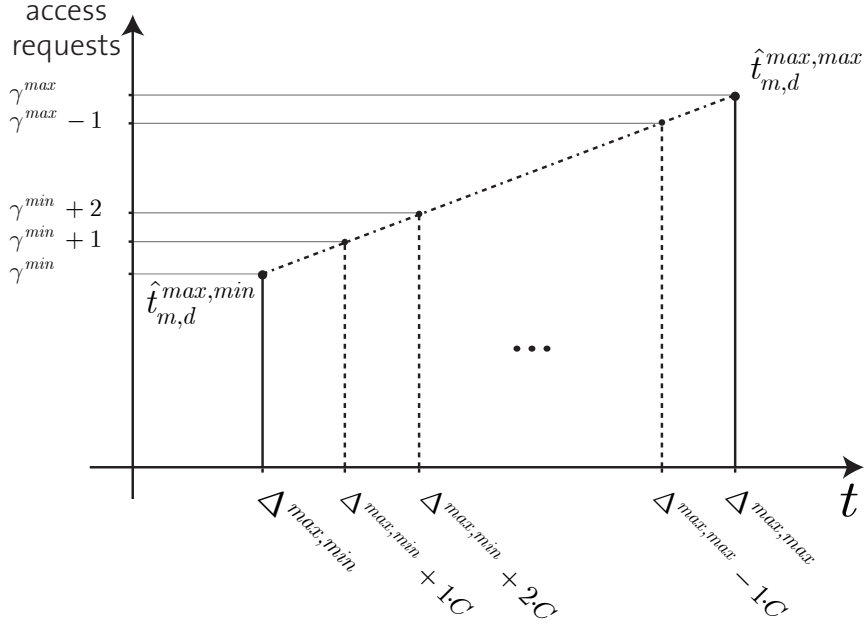
### Linear approximation between tuples

Tuple  $\hat{t}_{m,d}^{max,min}$  can be transformed to tuple  $\hat{t}_{m,d}^{max,max}$  by increasing the amount of access requests for the affected superblocks from the minimum to the maximum number. In other words, they show a linear relation, since the time required to process an access request is constant ( $C$ ). As a result, intermediate tuples can be computed by linear approximation. For any number of access request to the shared resource within the range of these tuples, we can compute a safe upper bound to the number of access requests performed in the corresponding time window by linear approximation, see Figure 3.10. This results in additional tuples that have to be considered when deriving arrival curves.

#### 3.4.3.4 Deriving arrival curves

Retrieving the maximum number of access requests to the shared resource for every time interval  $\Delta = \{0 \dots 2W_j\}$  from the computed tuples allows to compute the arrival curve. Consider the function  $\delta(\hat{t})$  to return the length of the time window and  $\nu(\hat{t})$  to return the number of access requests for each tuple. Then the upper arrival curve  $\tilde{\alpha}_j^u$  can be obtained as:

$$\tilde{\alpha}_j^u(\Delta) = \underset{\forall \hat{t}_{m,d}^{e,r}; \delta(\hat{t}_{m,d}^{e,r}) = \Delta}{\mathbf{argmax}} \nu(\hat{t}_{m,d}^{e,r}). \quad (3.7)$$



**Figure 3.10:** Linear approximation between minimum and maximum number of accesses to the shared resource for a single super-block, disregarding the gap  $g$  for ease of visualization.

We construct the infinite curve  $\hat{\alpha}_j^u$  from an initial aperiodic part, that is represented by  $\tilde{\alpha}_j^u$  and a periodic part which is repeated  $k$ -times for  $k \in \mathbb{N}$ .

$$\hat{\alpha}_j^u(\Delta) = \begin{cases} \tilde{\alpha}_j^u(\Delta) & 0 \leq \Delta \leq W_j \\ \max \left\{ \tilde{\alpha}_j^u(\Delta), \tilde{\alpha}_j^u(\Delta - W_j) + \sum_{\forall i} (\mu_{i,j}^{max}) \right\} & W_j \leq \Delta \leq 2W_j \\ \tilde{\alpha}_j^u(\Delta - k \cdot W_j) + k \sum_{\forall i} (\mu_{i,j}^{max}) & \text{otherwise} \end{cases} \quad (3.8)$$

The computational complexity to obtain the infinite arrival curve is  $O(|\mathcal{S}_j|^2)$ . Following the previous computation we derive Lemma 1.

**Lemma 1** *Deriving arrival curve  $\hat{\alpha}_j^u(\Delta)$  by Equation (3.8) results in the upper bound of access requests to a shared resource by the set of superblocks  $\mathcal{S}_j$  for any time window  $\Delta$ .*

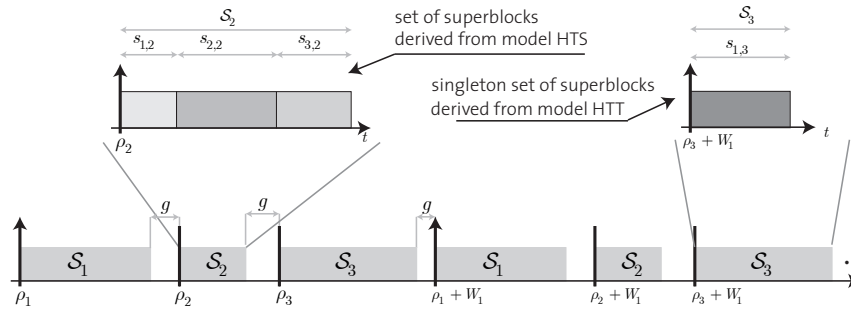
**Proof.** In Equation (3.4) and (3.5), the time windows and number of access requests for tuples in Equation (3.6) are computed. These tuples represent the set of known structure about a set of superblocks  $\mathcal{S}_j$  on processing element  $p_j$  and represent the maximum number of access requests for a given time window. Based on this set of tuples, Equation (3.7) computes the maximum amount of access requests for a given time window  $\Delta$ . Consider, as a contradiction, that there is an execution trace of  $\mathcal{S}_j$ , such that there

are more access requests in time window of length  $\Delta$  than  $\hat{\alpha}_j^u(\Delta)$ . Then, there must exist a tuple  $\hat{t}_{m,d}^{e,r}$ , such that the number of access requests  $\gamma^r$  for the time window  $\Delta$  is larger than  $\hat{\alpha}_j^u(\Delta)$ . Such a tuple can only exist if the constant access time  $C$  is not constant - contradicting the assumption.

□

### 3.4.4 Time-triggered execution of superblocks

In the previous sections we showed how to derive a representation of accesses to a shared resource for sequentially executing superblocks. In this section we extend this approach to cover the time triggered models, defined in Section 3.3.2.



**Figure 3.11:** Example of 3 sets of superblocks, with time-triggered execution

In the time triggered models, superblocks or phases have distinct starting times. As an example, consider model HTS in Section 3.3.2. Superblocks are time triggered, but phases inside each superblock execute sequentially. We can represent such a superblock as a set of superblocks, by translating each of its constituting phases into a superblock, as shown in Figure 3.8.

Figure 3.11 gives an example. Here, the set of superblocks  $\mathcal{S}_2$  is a set of superblocks with starting time  $\rho_2$ . This set of superblocks is derived from a single superblock, modeled according to model HTS. Similarly, the set of superblocks  $\mathcal{S}_3$  is the result of translating a superblock modeled according to model GTS into a set of superblock. In this particular case, the translation results in a set of superblocks that is constituted by a single superblock, i.e., a singleton.

Given this translation, we can now perform the derivation of tuples as in the previous sections. As an input, there are multiple sets of superblocks  $T = \{\mathcal{S}_1 \dots \mathcal{S}_N\}$  scheduled statically, as in Figure 3.11. Each set of superblocks  $\mathcal{S}_n \in T$  represents a set of sequentially executing superblocks,

with a distinct starting time  $\rho_n$ . This translation allows to represent each time triggered model in Section 3.3.2.

Then we can compute a sequence of all superblocks that constitute the sequence of sets in  $T$  as:

$$\sigma = \{s_{1,1} \dots s_{|S_1|,1}, s_{1,2} \dots s_{|S_2|,2} \dots s_{1,N} \dots s_{|S_N|,N}\} \quad (3.9)$$

Based on  $\sigma$  the set of ordered subsets  $*\sigma$  is derived, as shown in Section 3.4.3.1. Similarly to Section 3.4.3.1, we have to consider two subsequent processing cycles to be sure to include the worst case behavior. Therefore, we define  $\sigma' = \{\sigma \sigma\}$  and compute the set of ordered subsets  $*\sigma'$ . Following the approach shown in previous sections,  $t'_{m,d}$  is a subset with  $m$  representing the index of the first superblock in the subset and  $d$  representing the distance between the first and last superblock of the subset.

Time windows are computed for each subset  $t'_{m,d} \in *\sigma'$  following the approach shown in Section 3.4.3.2. Some subsets  $t'_{m,d} \in *\sigma'$  contain superblocks from different statically scheduled sets of superblocks. Therefore the gap of the first set of superblocks in subset  $t'_{m,d}$  has to be accounted for. Similarly to the gap  $g$  for the sequential case, minimizing the gap results in the worst case. Equation (3.3) can be rewritten to compute the gap for a subset, by maximizing all the superblocks that are not considered by subset  $t'_{m,d}$ :

$$\begin{aligned} g(r) = & \max_{\forall S_k \in t'_{m,d}} \rho_k - \min_{\forall S_k \in t'_{m,d}} \rho_k \\ & - \sum_{\forall s_{i,j} \in \sigma \setminus (\sigma \cap t'_{m,d})} exec_{i,j}^{max} + \mu_{i,j}^{max} \cdot C \\ & - \sum_{\forall s_{i,j} \in \sigma \cap t'_{m,d}} exec_{i,j}^{min} + \mu_{i,j}^r \cdot C. \end{aligned} \quad (3.10)$$

Only the minimum and maximum starting time of the sets of superblocks included in  $t'_{m,d}$  have an influence on the gap. This is a direct result of the time triggered execution. The early completion of a set of superblocks has no influence on the starting time of successive sets of superblocks. Following that, the tuples can now be computed as shown in Equation (3.6). From this set of tuples, we can derive the upper arrival curve by Equations (3.7) and (3.8).

### 3.4.5 Resulting arrival curve

Consider a multiprocessor platform as shown in Figure 3.3, with three processing elements, each executing a set of superblocks. Furthermore, consider processing element  $p_1$  to execute  $\mathcal{S}_1$ , composed of three superblocks, with

parameters as shown in Table 3.3, a processing cycle  $W_1 = 70ms$  and a constant access time to the shared resource  $C = 1ms$ .

	$\mu_{i,j}^{min}$	$\mu_{i,j}^{max}$	$exec_{i,j}^{min}$	$exec_{i,j}^{max}$
$s_{1,1}$	6	7	8ms	9ms
$s_{1,2}$	12	14	3ms	6ms
$s_{1,3}$	10	11	4ms	9ms

**Table 3.3:** Parameters for  $\mathcal{S}_1$ .

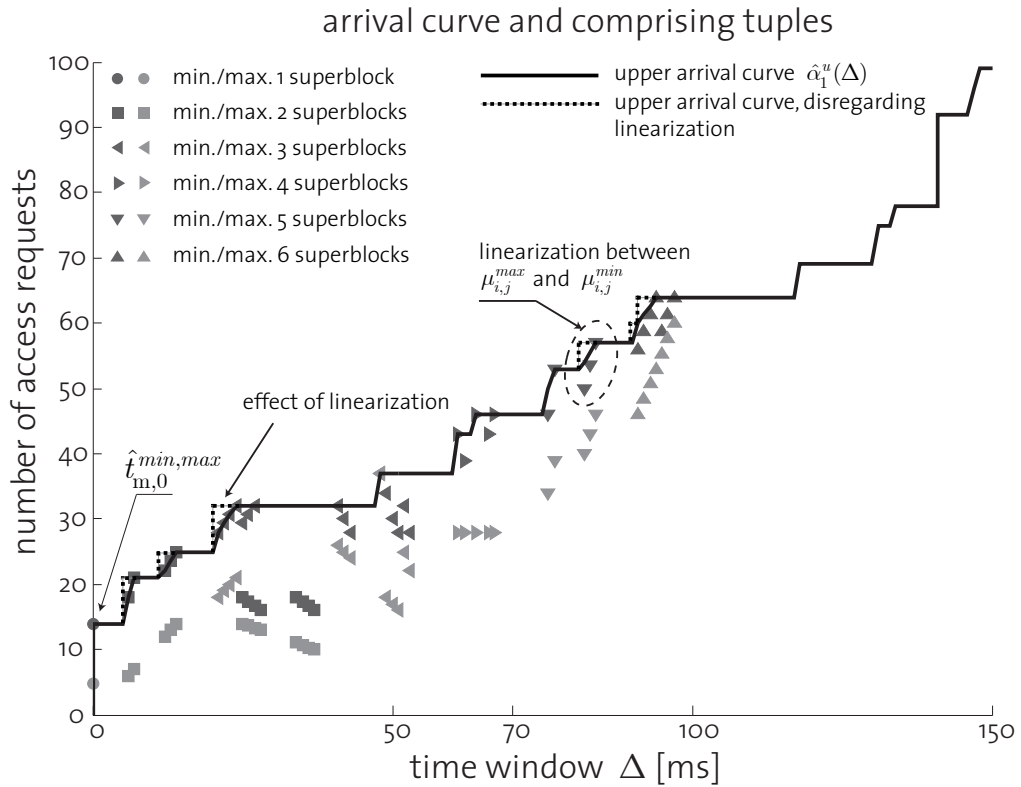
Then the resulting arrival curve, representing the access pattern of the set of superblocks  $\mathcal{S}_1$  to the shared resource is shown in Figure 3.12. The tuples are represented by the marked data points in the figure, i.e., the circle marker represents the tuples computed for a single superblock ( $d = 0$ ) and all possible offsets ( $m = [1 \dots |\mathcal{S}_1|]$ ), and so on. As an example, see tuple  $\hat{t}_{m,0}^{min,max}$  in Figure 3.12. The dark gray and light gray markers correspond to the minimal and maximal computation considered for the tuples, respectively. The effect of the linear approximation between the minimum and the maximum number of access requests, as computed in Section 3.4.3.3, is shown in the dashed circled area in Figure 3.12. The black solid line represents the resulting upper arrival curve  $\hat{\alpha}_1^u(\Delta)$ , as computed by Equation (3.8). The black dotted line shows the arrival curve that would result if intermediate tuples would not be considered.

### 3.5 Chapter Summary

This chapter studies the problem of a shared resource among multiple concurrently executing processing elements. Several concurrent accesses to a shared resource lead to contention on this resource, and thus to significantly increased execution times. We present models of execution and models to access the shared resource.

Tasks are described by superblocks and sets of superblocks execute on processing elements, according to a model of execution (sequential or time triggered). Additional known structure in superblocks is specified by phases. Superblocks and phases are specified by their minimum and maximum computation and by their minimum and maximum number of access requests to a shared resource.

We propose the *dedicated*, *general* and *hybrid* model to access the shared resource. The *dedicated* model defines dedicated phases to access the shared resource and an execution phase to perform local computation. In the *general* model, there is a general phase in which access requests and computation can happen at any time and in any order. The *hybrid* model defines dedicated phases to access a shared resource and a general phase, where



**Figure 3.12:** Upper arrival curve, with constructing tuples.

computation and accesses to the shared resource can happen at any time and in any order.

We show how to represent the access pattern of a set of superblocks to the shared resource as an upper arrival curve, representing the maximum number of access requests that can happen in any time window. This representation can be used to compute the maximum amount of interference a particular processing element can suffer from other processing elements in the system, i.e., to approximate the maximum additional delay tasks and superblocks may suffer due to contention on the shared resource.

The resource access models and models of execution presented in this chapter form the basis of the proposed hardware platforms and worst-case execution time (WCET) estimation approaches presented in Chapter 4 and Chapter 5.

In Chapter 4, we show how interference can be neglected by applying a time division multiple access (TDMA) arbiter on the shared resource and how the different resource access models are related to each other in terms of worst-case execution time (WCET).

In Chapter 5, we apply the FlexRay arbitration protocol on the shared resource. This arbitration protocol increases the adaptivity of the system

by introducing a dynamic arbitration element in the scheduling approach, allowing to reclaim unused access slots to the shared resource by other processing elements in the system. Interference from other processing elements now has to be considered, since processing elements compete for access to the shared resource. We present a worst case completion time (WCCT) estimation approach, that takes advantage of the interference representation proposed in this chapter.



# 4

## Static arbitration on shared resources

Typically, multicore and multiprocessor systems are optimized towards average case performance. If such systems are used in applications with real-time constraints, such as the control or signal processing domain, many challenges arise.

Shared resources are usually connected via a bus that grants access according to a specific (potentially proprietary) arbitration policy. Due to the orientation towards average case performance, these arbitration policies are often dynamic, i.e., they grant access according to the First-Come-First-Serve (FCFS), the Earliest-Deadline-First (EDF) or the Fixed-Priority (FP) policy, to name just a few. Using dynamic arbitration policies makes the derivation of worst-case execution times (WCETs) particularly hard, i.e., these arbitration policies exhibit a large degree of non-determinism. However, in order to guarantee timing predictability, a safe bound on the worst-case execution time (WCET) has to be derived.

Designers of hard real-time systems use a number of techniques to increase predictability, e.g., static schedules, over-approximation of computational resources or special programming techniques to reduce non-determinism. In this chapter, we suppose that the time division multiple access (TDMA) protocol is used for resolving access conflicts on shared resources, such as communication buses or memories. This static arbitration policy allows to analyze the timing properties of a single core or processing element, without the need to consider the access patterns of other processing elements onto the shared resource. We propose a worst-case analysis framework to

derive the worst-case response time (WCRT) for the resource access models presented in Chapter 3 and TDMA arbitration on the shared resource. Furthermore, we give design guidelines for resource sharing real-time systems.

## 4.1 Introduction

Consider a platform with multiple processing elements and a single shared main memory, as presented in Chapter 3, Figure 3.1, with static arbitration on the shared resource. One example for such an arbitration policy is time division multiple access (TDMA), where the shared resource is assigned to a task or processing element for a particular time window. In other words, access to the resource is partitioned over time and only a single actor can acquire it at a time. TDMA arbitration policies are often used in industrial applications, not only to increase timing predictability but also to alleviate schedulability analysis.

We assume a hardware platform where execution time and communication time can be decoupled, e.g., the fully timing compositional architecture proposed by Wilhelm et al. [WGR<sup>+</sup>09]. That is, we implicitly consider a hardware platform that has no timing anomalies. Based on this assumption, we provide an efficient and safe worst-case response time analysis.

This chapter considers systems with a shared resource, where the amount of time required to complete a request can be bounded by a constant. Access to the resource, e.g., a bus in a multicore system, is granted for at most one request at a time, resulting in blocking/waiting time for any other request. An ongoing access to a shared resource cannot be preempted.

Sets of superblocks are assigned to a predefined processing element and executed in a periodically repeating static schedule. Resource accesses within the superblocks are modeled according to the (1) *dedicated* model, (2) *general* model and (3) *hybrid* model.

The contributions of this chapter are as follows:

- For the considered access models and a given TDMA arbiter, we propose an analytical worst-case analysis framework, considering blocking-/non-buffered access to a shared resource.
- For the dedicated access model, our proposed analysis framework is of linear time complexity.
- For a regular TDMA arbiter, in which each processing element has only one time slot in a TDMA cycle, we show the computational complexity of our analysis.
- For TDMA arbiters without constraints, we show that the timing analysis can be done by applying binary search.

- We derive the schedulability relation of the proposed models and propose the *dedicated model*, with sequential execution of superblocks, as the model of choice for time critical resource sharing systems. We show that time triggered superblocks cannot increase the performance in our scenario.

The rest of the chapter is organized as follows: Related Work is presented in Section 4.2. In Section 4.3, we introduce the model of the shared resource. We show the analytical framework to compute the worst-case completion time under a given TDMA arbiter in Section 4.5 for a phase and in Section 4.6 for superblocks and tasks. Section 4.7 derives the schedulability relation among the proposed models. Experimental results using an applications from the automotive domain are shown in Section 4.8. Section 4.9 concludes the paper.

## 4.2 Related Work

Time-triggered models of computation and of accessing resources typically have been used to increase the timing predictability of systems and to ease the prediction of system behavior. In [KG94] Kopetz et al. propose the TTA protocol for safety critical systems, that provides predictable latency for message transport and clock synchronization among other properties. An application of the protocol to systems composed by multiple elements (e.g., COTS systems) is shown in [KESH<sup>+</sup>08]. As opposed to modern multiprocessor systems, e.g., the Cell Broadband Engine [IST06], that is optimized for average case performance and exhibits a large degree of non-determinism in transmission latencies and communication time [AP07, Hüg10], time-triggered architectures provide analyzability and repeatability, and thus timing predictability.

Timing predictability [TW04] is a key issue in the design of safety critical real-time systems system, as seen in the automotive and avionic industry. Following Thiele et al. [TW04], we aim at designing performant systems with sharp upper and lower bounds on their execution times. One of the needs mentioned by Thiele et al. is the need for coordination on the architecture - without which a system would become unpredictable. Time Division Multiple Access (TDMA) satisfies this need for coordination on the shared resource.

TDMA as arbitration policy to either access a shared resource, or to assign computational resources to tasks, is studied in great detail [AEPR08, RAEP07, WT06a, HE05]. Other fields also apply TDMA, e.g., to guarantee bounded communication delays in networking [CF10].

Rosen et al. [RAEP07] and Andrei et al. [AEPR08] use Time Division Multiple Access (TDMA) for bus accesses and a task model, where commu-

nication requests are confined to dedicated phases at the beginning and the end of a task. Execution traces for an application are generated using static analysis, and applied to a given TDMA arbiter to derive the worst-case completion time. Such systems can be analyzed compositionally, i.e., one processing element is analyzed at a time, and the system is feasible, if all its components are feasible. Thus, interference is eliminated through isolation. Contrary to that, we assume that the positions of accesses to the shared resource are not known a priori and neither is their order. Producing all the feasible traces for such a configuration would result in an infinite number of possibilities.

Wandeler et al. [WT06a] present a method to determine the minimal slot length that has to be assigned to a resource, such that its real-time constraints can be met. Based on that, the optimal cycle length for the TDMA arbiter is derived. In contrast to the work presented in this thesis, the model of the shared resource considered by Wandeler et al. allows for events to be buffered. This way, a resource can issue multiple events, without the need to wait for previous events to be served. As a result, there is no blocking due to contention on the shared resource.

Another line of research, that considers dynamic arbitration policies, such as First-Come-First-Server (FCFS), use Model Checking to analyze a system. Gustavsson et al. [GELP10] analyze multi-core systems with private L1 cache and shared L2 cache and main memory using Time Automata [RD94] and the Uppaal Model Checker [LPY97]. In their experiments, for an architecture with 4 cores, the analysis had to be aborted after 3 hours, as the memory requirements exceeded the available 4GB. Since the worst-case execution time is found using a binary search approach, the actual total analysis time is a multiple of this time. The authors estimate a total analysis time of more than 33 hours.

Lv et al. [LGY10] also use timed automata and model checking to analyze the timing behavior of an application, but propose abstract interpretation of the application to reduce the state space. They analyze systems with FCFS or TDMA arbitration policies. The approach is shown to perform well for the presented examples, but eventually suffers from the same complexity issues.

Using TDMA, interference is neglected and the computational complexity of the worst-case execution time (WCET) analysis depends on the properties of the considered application rather than the properties of competing elements in the system.

## 4.3 System Model

In this section, we shortly recapitulate the models to access the shared resource and the models of execution, introduced in the previous chapter. Furthermore, we detail on the model of the shared resources, i.e., the TDMA arbitration policy.

### 4.3.1 Models of Tasks and Processing Elements

We assume a platform with multiple processing elements  $p_j \in \mathcal{P}$ . Sets of superblocks execute independently on a processing element and access the shared resource according to the following models, as described in Section 3.3.1:

- the *dedicated* model,
- the *general* model and
- the *hybrid* model,

with the *sequential* and *time-triggered* models of execution. The dedicated model limits accesses to the shared resource to dedicated phases and computation to the execution phase, while in the general model access requests and computation can happen anytime and in any order. The hybrid model represents a trade-off, with dedicated phases to access the shared resource and a general phase, where computation and resource accesses can happen anytime and in any order.

### 4.3.2 Model of the Shared Resource

As introduced in Section 3.3.3, we consider systems with a stateless shared resource among the processing elements in  $\mathcal{P}$ . At any time, the shared resource can serve at most one request and therefore, if a superblock  $s_{i,j}$  modeled according to the dedicated model can access the shared resource at any time, the maximal time for executing the superblock  $s_{i,j}$  is  $exec_{i,j}^{max} + (\mu_{i,j}^{max,a} + \mu_{i,j}^{max,e} + \mu_{i,j}^{max,r}) \cdot C$ , where  $s_{i,j}$  denotes superblock  $i$  on processing element  $p_j$ . Superblock  $s_{i,j}$  is specified with  $\mu_{i,j}^{max,a}$ ,  $\mu_{i,j}^{max,e}$  and  $\mu_{i,j}^{max,r}$  access requests in its acquisition, general and replication phase, respectively, and  $exec_{i,j}^{max}$  computation to be performed in its general phase. Accesses to the shared resource require  $C$  units of time.

In this chapter, we consider systems with a TDMA arbiter for arbitrating access to the shared resource. A TDMA schedule  $\Theta$  is defined as sequence of time slots, such that  $\sigma_m \in \Theta$  is the starting time of the  $m$ -th time slot (a.k.a. time slot  $m$ ) and its duration is  $\delta_m = \sigma_{m+1} - \sigma_m$ . For notational brevity, we define  $M_\Theta$  as the number of slots in schedule  $\Theta$  and  $L(\Theta)$  as its period. As a result, the TDMA schedule is repeated after every  $L(\Theta)$

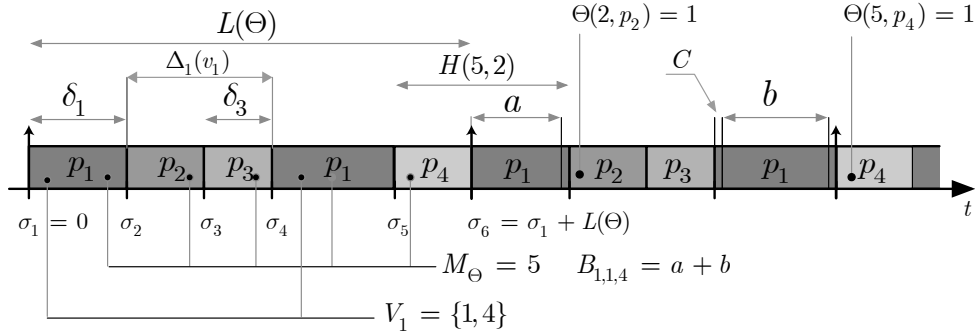


Figure 4.1: TDMA Scheme Overview and Notation

time units. Furthermore, let  $\sigma_1$  be 0 and  $\sigma_{M_\Theta+1}$  be  $L(\Theta)$ . Time slots of the schedule are assigned to processing elements, such that  $\theta(m, p_j) = 1$  if time slot  $m$  is assigned to processing element  $p_j$ , otherwise  $\theta(m, p_j) = 0$ . For example, Figure 4.1 illustrates an example for the TDMA schedule, while undefined variables will be explained later in Section 4.5.1.

Only requests from processing element  $p_j$  with  $\theta(m, p_j)$  equal to 1 are granted access to the shared resource in the  $m$ -th time slot of a TDMA cycle, while requests from other processing elements have to wait. Moreover, if at a time  $t$  the remaining time  $\sigma_{m+1} - t$  of a time slot cannot serve a new request, i.e.,  $\sigma_{m+1} - t \leq C$ , the TDMA arbiter will not grant any requests until the time slot expires. Therefore, in order to provide meaningful results, we assume that slots in the schedule are at least of length  $C$  and there is at least one slot for each processing element  $p_j \in \mathcal{P}$ . Slots with a length less than  $C$  cannot serve any access requests.

We assume that processing elements and the resource arbiter initialize synchronously, such that the first slot on the shared resource and the first superblock on each processing element start at time 0. A TDMA schedule for the shared resource is said to be *schedulable* if all the superblocks/tasks in all processing elements can finish before their respective deadlines, i.e., the response time of a superblock is no more than the relative deadline. An access to the shared resource blocks all other requesting tasks/superblocks, until it is completed, and thus results in significantly increased worst-case response times. The objective of this chapter is to provide efficient analysis methods to compute the worst-case response times for all the tasks on a resource sharing system, assuming TDMA arbitration of the shared resource.

## 4.4 Analysis Overview

In this chapter, we derive the WCRT of a set of superblocks, by determining the worst-case trace of access requests to the shared resource. In other words, we find the concrete execution trace of a superblock that leads to the WCRT.

The worst-case trace is determined by maximizing the stall time of a phase. In other words, we find the placement of access requests and computation, such that the waiting time for accessing the shared resource is maximized. For the TDMA arbitration policy on the shared resource, this is done

1. by issuing access requests at the end of time slots that are assigned to the superblock and
2. by performing computation during active time slots.

Intuitively, we place computation in such a way, that access requests always happen when the shared resource is assigned to other processing elements. This way, the superblock has to stall until the activation of the next time slot. The time required to finish an access request is denoted  $C$ . Conclusively, in order to maximize the stall time for an access request, it should be issued  $C - \epsilon$  units of time before the expiration of the active time slot. Then, the access request cannot be finished in the current time slot, and is served upon activation of the next assigned time slot. The stall time that results from this strategy is related to the time between the expiration of the time slot and the starting time of the next time slot and the time required to serve the access request.

We propose an algorithm to derive this worst-case trace of access requests for single phases in Section 4.5. The analysis for dedicated phases is shown in Section 4.5.2, while the analysis for general phases is shown in Section 4.5.3. In Section 4.6, we show how to compose these analyses approaches to derive the WCRT for a complete superblock and task.

### Dedicated phases

Dedicated phases either perform computation or issue access requests. As a result, the analysis approach needs to compute the amount of arbitration rounds that is required to perform all access requests. In case of a dedicated execution phase, i.e., only local computation is performed, the maximum execution time of the phase corresponds to the WCRT of the phase.

Otherwise, in case of an acquisition or replication phase, the WCRT of the phase depends on the number of access requests that can be served during active time slots, the time between time slots and the number of arbitration cycles that is required until all access requests have been served.

We denote the maximum number of access requests that can be performed by time slots  $m$  to  $n$ , assigned to processing element  $p_j$ , as  $\Lambda_{j,m,n}^{max}$ , see Equation (4.3). The blocking time depends on the distance between two time slots. We denote the distance between the beginning of time slot  $m$  to the beginning of time slot  $n$  as  $H_{m,n}$ , see Equation (4.2). The number of access requests that can be served in each arbitration cycle is denoted  $\Lambda^{max*}$ . Algorithm 4.1 in Section 4.5.2 details on this analysis.

## General phases

In general phases, computation and access requests can happen at anytime and in any order. Therefore, we have to determine the concrete trace that leads to the WCRT. Similarly to dedicated phases, the WCRT for general phases also depends on the number of access requests that can be served in time slots, the time between time slots and the number of arbitration cycles that is required to complete all access requests. However, the amount of computation that is performed during active time slots is indirectly related to the number of access requests that can be served. In other words, the more computation we perform during a time slot, the less access requests can be served in this time slot.

Intuitively, issuing access requests only at the end of active time slots results in maximized stall times. Each time an access request is issued, the time slot expires and the phase has to stall until the next time slot is activated. In general phases, access requests and computation can happen in any order. As a result, we can place computation such that access requests happen at the end of time slot, and thus maximize the stall time. The amount of computation that is required to achieve this particular placement of access requests, for time slots  $m$  to  $n$ , is denoted  $B_{j,m,n}$ , see Equation (4.6). Hence, active time slots are used for performing computation and only a single access request is issued at their end. Therefore, the number of access requests served between time slots  $m$  and  $n$  corresponds to the number of time slots assigned to the phase, denoted  $\psi_{j,m,n}$ , see Equation (4.5). Algorithm 4.2 derives the WCRT for this case.

Algorithm 4.3 considers the case, when the distance between time slots assigned to a particular processing element are not equal. In this case, and if the amount of computation is not sufficient to limit the number of access requests in each time slot to 1, we have to find the optimal distribution of computation. We apply a binary search algorithm to determine the execution trace that leads to the WCRT for the phase under analysis.



## 4.5 Worst-Case Completion Time of A Phase

Given a TDMA schedule  $\Theta$ , this section presents how to analyze the worst-case completion time of an acquisition/replication phase starting at a specific time  $t$ . The delay suffered by the phase depends on its structure, i.e., how accesses to the shared resource are organized. We will first present the required terminologies and notations in Section 4.5.1 with an example schedule presented in Figure 4.1, and then show how to analyze the worst-case completion time of a phase in Sections 4.5.2 and 4.5.3.

### 4.5.1 Terminologies and notations used for analysis

Let  $\pi(t)$  be the time slot of the TDMA schedule at time  $t$  and  $\sigma_{\pi(t)}$  be its starting time in the TDMA cycle, such that

$$\left\lfloor \frac{t}{L(\Theta)} \right\rfloor L(\Theta) + \sigma_{\pi(t)} \leq t < \left\lfloor \frac{t}{L(\Theta)} \right\rfloor L(\Theta) + \sigma_{\pi(t)+1}. \quad (4.1)$$

For brevity, the next time slot  $n$  of the current time slot  $m$  is defined by the closest next time slot  $n$  in the TDMA schedule. That is, if  $n \leq m$ , the next time slot  $n$  is in the next TDMA cycle; otherwise, they are in the same cycle. Therefore, the time distance  $H_{m,n}$  from the beginning of time slot  $m$  to the beginning of the next time slot  $n$  is defined as follows:

$$H_{m,n} = \begin{cases} \sigma_n - \sigma_m & \text{if } n > m \\ \sigma_n + L(\Theta) - \sigma_m & \text{otherwise,} \end{cases} \quad (4.2)$$

where Figure 4.1 provides an example for  $H(5, 2)$ .

The maximal number of completed requests for processing element  $p_j$  in TDMA time slot  $m$  is defined by  $\lambda_{j,m}$ , in which  $\lambda_{j,m}^{max}$  is  $\lfloor \frac{\delta_m}{C} \rfloor \theta(m, p_j)$ , for acquisition/replication phases where no computation can happen. Similarly, for general phases, since we are allowed to perform computation between requests, the number of completed requests for processing element  $p_j$  in a TDMA time slot  $m$  is  $\lambda_{j,m}^{min} = \lfloor \frac{\delta_m - C}{C} \rfloor \theta(m, p_j)$ . As a result, the maximal number of completed requests for processing element  $p_j$  from time slot  $m$  to the next time slot  $n$  is denoted  $\Lambda_{j,m,n}^{max}$  and is computed as:

$$\Lambda_{j,m,n}^{max} = \begin{cases} \sum_{k=m}^n \lambda_{j,k}^{max} & \text{if } n > m \\ \sum_{k=m}^{M_\Theta} \lambda_{j,k}^{max} + \sum_{k=1}^n \lambda_{j,k}^{max} & \text{otherwise.} \end{cases} \quad (4.3)$$

Computing  $\Lambda_{j,m,n}^{min}$  follows suit. We denote  $\sum_{v=1}^{M_\Theta} \lambda_{j,v}^{max}$  by  $\Lambda^{max*}$  and  $\sum_{v=1}^{M_\Theta} \lambda_{j,v}^{min}$  by  $\Lambda^{min*}$ , which is the maximal and minimal number of completed requests in a TDMA cycle  $L(\Theta)$  respectively.

Assume a TDMA schedule  $\Theta$ , with multiple time slots assigned to a processing element  $p_j$ . Then the ordered vector  $\mathcal{V}_j$  contains the time slots

assigned to  $p_j$  and each element  $v_q \in \mathcal{V}_j$  is an integer between 1 and  $M_\Theta$ , such that  $v_1 < v_2 < \dots < v_{|\mathcal{V}_j|}$  and  $\theta(v_q, p_j) = 1$ . Let  $\Delta_j(v_q)$  be the amount of time that processing element  $p_j$  has to wait for the next time slot, after time slot  $v_q$  has expired, then:

$$\Delta_j(v_q) = \begin{cases} \sigma_{v_{(q+1)}} - \sigma_{(v_q+1)} & \text{if } q < |\mathcal{V}_j| \\ \sigma_{v_1} + L(\Theta) - \sigma_{(v_q+1)} & \text{otherwise.} \end{cases} \quad (4.4)$$

See Figure 4.1 for an example to compute  $\Delta_1(v_1)$ . The number of time slots assigned to processing element  $p_j$  from time slot  $m$  to time slot  $n$  (excluding time slot  $n$ ) is defined as follows:

$$\psi_{j,m,n} = \begin{cases} \sum_{k=m}^{n-1} \theta(k, p_j) & \text{if } n > m \\ \sum_{k=m}^{M_\Theta} \theta(k, p_j) + \sum_{k=1}^{n-1} \theta(k, p_j) & \text{otherwise.} \end{cases} \quad (4.5)$$

The number of time slots in a TDMA cycle assigned to processing element  $p_j$  is denoted by  $\Psi_j^*$ , where  $\Psi_j^* = \sum_{m=1}^{M_\Theta} \theta(m, p_j)$ .

Superblocks cannot proceed to perform computation as long as there are unserved requests to the shared resource. As a consequence, if there is an unserved request at the end of a time slot assigned to  $p_j$ , computation stalls until the next time slot becomes active and the request can be served. These stalls can be maximized by issuing accesses to the shared resource at the end of time slots. Hence, the time between consecutive time slots assigned to  $p_j$  is spent stalling, while active time slots are used to compute the unserved access issued at the end of the preceding time slot and to perform computation.

The time used performing computation on  $p_j$  from time slot  $m$  to its next time slot  $n$ , assuming that a single request to the shared resource is issued at the end of each time slot assigned to  $p_j$  is denoted as  $B_{j,m,n}$ . Intuitively, time slot  $m$  is used to perform computation on  $p_j$ , until  $C - \epsilon$  time units before the time slot expires (with  $C > \epsilon > 0$ ), when an access to the shared resource is issued. The request cannot be completed anymore, and  $p_j$  stalls until the next time slot becomes active. Once the next time slot  $n$  becomes active, the unserved access is completed, consuming a constant time  $C$ , and computation continues. Another access request is issued  $C - \epsilon$  units of time before time slot  $n$  expires, such that it could not be served anymore. As a result, in time slot  $n$  computation amounts for  $\delta_n - (2C - \epsilon)$  time units. Therefore, we can derive  $B_{j,m,n}$  as follows:<sup>1</sup>

$$B_{j,m,n} = \theta(m, p_j)(\delta_m - C) + \quad (4.6)$$

<sup>1</sup>If  $z > 0$ , function  $\{z\}^+$  is  $z$ ; otherwise  $\{z\}^+$  is 0. We ignore  $\epsilon$  in (4.6) since the value is meaningful only when it is very close to 0.

$$\begin{cases} \sum_{v=m+1}^{n-1} \theta(v, p_j) \{\delta_v - 2C\}^+ & \text{if } n > m \\ \left\{ \begin{array}{l} \sum_{v=m+1}^{M_\Theta} \theta(v, p_j) \{\delta_v - 2C\}^+ \\ + \sum_{v=1}^{n-1} \theta(v, p_j) \{\delta_v - 2C\}^+ \end{array} \right. & \text{otherwise.} \end{cases}$$

See Figure 4.1 for an example to compute  $B_{1,1,4}$ . Similarly, the maximum time for performing computation, denoted  $B^*$ , in a complete TDMA cycle of length  $L(\Theta)$  considering the previously defined conditions, computes as:

$$B^* = \sum_{v=1}^{M_\Theta} \theta(v, p_j) \{\delta_v - 2C\}^+. \quad (4.7)$$

#### 4.5.2 Worst-case completion time for a dedicated phase

This subsection presents how to analyze the worst-case completion time for an acquisition (replication) phase, starting at time  $t$ . Basically, we compute the time required to complete the access requests and how many time slots are required to do so.

Suppose that  $\mu^{max}$  is the maximum number of requests to the shared resource for a phase. Algorithm 4.1 presents the pseudo-code of the analysis for the worst-case completion time. Parameter  $\Lambda$  is a vector representing either  $\Lambda^{max}$  or  $\Lambda^{min}$ .  $\Lambda_{j,m,n}^{max}$  represents the maximum number of access requests that can be served in time slots  $m$  to  $n$ , that are assigned to  $p_j$ , see Equation (4.3). Then,  $\Lambda^{max}$  represents the vector for all possible combinations of  $m$  and  $n$  for a particular processing element  $p_j$ . Similarly,  $\Lambda^{min}$  represents the maximum number of access requests that can be served by time slots, considering that in each time slot a infinitely small amount of computation can happen, see Equation (4.3). In other words, when considering a dedicated phase, we use  $\Lambda^{max}$  in Algorithm 4.1. Contrarily, consider the analysis of a general phase, where all computation has already been distributed. Then, the remaining access requests are regarded as a dedicated acquisition or replication phase. However, in the worst-case, a small amount of computation might be available to postpone each access request by  $\epsilon$  units of time. As a result, when analyzing general phases, we consider  $\Lambda^{min}$  in Algorithm 4.1.

Let  $t$  be the current time and  $t'$  be the starting time of the next TDMA time slot, i.e.,  $t' = \left\lfloor \frac{t}{L(\Theta)} \right\rfloor L(\Theta) + \sigma_{\pi(t)+1}$ . If the current time slot can serve the required requests  $\mu^{max}$  to the shared resource, the worst-case completion time is simply  $t + C\mu^{max}$  (see Step 2 to 3 in Algorithm 4.1). Otherwise, we consider to issue  $\left\lfloor \frac{t'-t}{C} \right\rfloor \theta(\pi(t), p_j)$  requests in time slot  $\pi(t)$ , which implies that there are  $\mu' = \mu^{max} - \left\lfloor \frac{t'-t}{C} \right\rfloor \theta(\pi(t), p_j)$  unserved requests (Step 5). In each TDMA cycle  $L(\Theta)$ , we can complete at most  $\Lambda^*$  requests. Therefore, we require at least  $\left\lceil \frac{\mu'}{\Lambda^*} \right\rceil$  TDMA cycles to complete the remaining accesses.

**Algorithm 4.1** WCT-AR**Input:**  $\Theta, p_j, \mu^{max}, t, \Lambda;$ **Output:** Worst-case completion time;

- 1: let  $t'$  be the starting time of the next time slot after time  $t$ ;
- 2: **if**  $\left\lfloor \frac{t'-t}{C} \right\rfloor \theta(\pi(t), p_j) \geq \mu^{max}$  **then**
- 3:     **return**  $t + C\mu^{max}$ ;
- 4: **else**
- 5:      $\mu' \leftarrow \mu^{max} - \left\lfloor \frac{t'-t}{C} \right\rfloor \theta(\pi(t), p_j)$ ;
- 6:      $t' \leftarrow t' + \left\lfloor \frac{\mu'}{\Lambda^*} \right\rfloor L(\Theta)$ ;  $\mu' \leftarrow \mu' - \left\lfloor \frac{\mu'}{\Lambda^*} \right\rfloor \Lambda^*$ ;
- 7:     find the *closest next time slot*  $n^*$  such that  $\Lambda_{j,\pi(t'),n^*} \geq \mu'$ ;
- 8:      $t' \leftarrow t' + H_{\pi(t'),n^*} + C \cdot (\mu' - \Lambda_{j,\pi(t'),n^*-1})$ ;
- 9: **end if**

Then time  $t'$  is set to  $t' + \left\lfloor \frac{\mu'}{\Lambda^*} \right\rfloor L(\Theta)$ , while reducing  $\mu'$  to  $\mu' - \left\lfloor \frac{\mu'}{\Lambda^*} \right\rfloor \Lambda^*$  (Step 6). The remaining  $\mu'$  accesses can then be completed in the current or next TDMA cycle. In other words, the algorithm finds slot  $n^*$ , with starting time  $\sigma_{n^*}$ , in  $[\sigma_{\pi(t')}, \dots, \sigma_{M_\Theta}, \sigma_1, \dots, \sigma_{\pi(t')-1}]$ , such that  $\Lambda_{j,\pi(t'),n^*} \geq \mu'$ , i.e., the number of requests that can be served between the slot at  $\pi(t')$  and slot  $n^*$  exceeds the remaining access requests  $\mu'$  (Step 7). Following that, the worst-case completion time is calculated in (Step 8).

We denote Algorithm 4.1 as WCT-AR. The time complexity is  $O(M_\Theta)$ . It can be improved to  $O(\log M_\Theta)$  by applying binary search in Step 7 when all  $\Lambda_{j,m,n}$  for all  $m, n$  are calculated a priori.

**Lemma 2** *For an acquisition or a replication phase on processing element  $p_j$  with at most  $\mu^{max}$  requests to the shared resource, Algorithm WCT-AR by setting  $\Lambda$  as  $\Lambda^{max}$  derives the worst-case completion time.*

**Proof.** This is a result of the definition of the TDMA schedule  $\Theta$  and the resource access model. As there is no interference and no non-determinism with respect to the decision whether to issue an access request or not, the worst-case completion time is the time when the accumulated slot time assigned to processing element  $p_j$  suffices to serve all access requests.

□

### 4.5.3 Worst-case completion time for a general phase

This subsection presents how to analyze the worst-case completion time for a general phase on processing element  $p_j$  when the phase starts at time  $t$ , where  $\mu^{max}$  is the maximum number of requests to the shared resource and  $exec^{max}$  is the maximal amount of computation for the general phase. If  $\mu^{max}$  is 0, the worst-case completion time is  $t + exec^{max}$ .

A request issued  $C - \epsilon$  time units before the expiration of the current time slot cannot be served immediately, for any  $\epsilon > 0$ . For simplicity, we assume a request cannot be served in the current time slot if  $\epsilon \geq 0$ , sacrificing only little tightness.

Given the starting time  $t$  of a phase, we can determine whether or not the current time slot is assigned to processing element  $p_j$ . In case the current time slot is assigned to another processing element, our algorithm for determining the worst-case trace has to consider two possible outcomes. First, issuing an access request immediately at time  $t$ , or second, performing computation. Depending on the distance of time  $t$  to the activation time of the next time slot, both concrete execution traces might lead to the worst-case trace. Therefore, our algorithm considers both cases and returns the worst-case, see Line 9 in Algorithm 4.2. Lemma 3 shows that considering these two cases is sufficient for determining the worst-case.

**Lemma 3** *If  $\theta(\pi(t), p_j) = 0$ , there are two concrete execution traces that might lead to the WCRT. Either (1) performing computation before the earliest next time slot for  $p_j$  or (2) issuing one request immediately, and thus stall until the earliest next time slot for  $p_j$ .*

**Proof.** Consider the other case, that at time  $t$  computation is performed and then a request is issued at a time  $t'$ , such that  $\theta(\pi(t'), p_j) = 0$ , that is, before the activation of the next time slot assigned to  $p_j$ . Then, it is clear that the computation can be postponed such that the completion time is larger. Therefore, one of the two cases in the statement results in the worst-case completion time. □

Algorithm 4.2, denoted WCT-E, presents the pseudo-code of the analysis for the worst-case completion time. If the current time slot is not assigned to processing element  $p_j$ , i.e.,  $\theta(\pi(t), p_j) = 0$ , we can either issue a request, and thereby cause the superblock to stall (Step 6), or perform computation till the next time slot assigned to processing element  $p_j$  (Step 8) becomes active. In the analysis, both cases are examined recursively and the maximum completion time is reported as the worst-case completion time.

Therefore, for the following iterations of Algorithm 4.2, we only have to consider the case that  $\theta(\pi(t), p_j)$  is 1. Again, let  $t'$  be initialized as the starting time of the next TDMA time slot, i.e.,  $t' = \left\lfloor \frac{t}{L(\Theta)} \right\rfloor L(\Theta) + \sigma_{\pi(t)+1}$ . If executing all the required computation can be done before  $t' - C - t$ , the analysis executes computation first and then starts to access the shared resource at time  $t + exec^{max}$ . For this case, the problem is reduced to an acquisition or a replication phase, and hence, we can use Algorithm WCT-AR (Step 16 in Algorithm 4.2). Note that  $\Lambda^{min}$  has to

**Algorithm 4.2** WCT-E**Input:**  $\Theta, p_j, exec^{max}, \mu^{max}, t$ ;**Output:** Worst-case completion time;

---

```

1: if  $\mu^{max} = 0$  then
2:   return  $t + exec^{max}$ ;
3: else
4:   if  $\theta(\pi(t), p_j) = 0$  then
5:     let  $t^*$  be the starting time of the next time slot assigned for  $p_j$  in  $\Theta$ ;
6:      $R_1 \leftarrow$  WCT-E( $\Theta, p_j, exec^{max}, \mu^{max} - 1, t^* + C$ );
7:     if  $exec^{max} - t^* + t > 0$  then
8:        $R_2 \leftarrow$  WCT-E( $\Theta, p_j, exec^{max} - t^* + t, \mu^{max}, t^*$ );
9:       return  $\max\{R_1, R_2\}$ ;
10:    else
11:      return  $R_1$ ;
12:    end if
13:  end if
14:  let  $t'$  be the starting time of the next time slot after time  $t$ ;
15:  if  $(t' - C) - t \geq exec^{max}$  then
16:    return WCT-AR( $\Theta, p_j, \mu^{max}, t + exec^{max} + C, \Lambda^{min}$ );
17:  end if
18:   $e' \leftarrow exec^{max} - \{0, (t' - C) - t\}^+$ ;  $\mu' \leftarrow \mu^{max} - 1$ ;
19:  find  $h$  and  $n^*$  such that  $hB^* + B_{j,\pi(t'),n^*} \geq e' > hB^* + B_{j,\pi(t'),n^*-1}$ ;
20:   $t_c \leftarrow t' + hL(\Theta) + H_{\pi(t'),n^*} + C + (e' - hB^* - B_{j,\pi(t'),n^*-1})$ ;
21:  if  $h\Psi_j^* + n^* < \mu'$  then
22:    if  $t_c$  and  $t_c + C$  are not in the same time slot then
23:      let  $t_c$  be the beginning of the next time slot assigned for  $p_j$  after  $t_c$ ;
24:    end if
25:    return WCT-AR( $\Theta, p_j, \mu' - h\Psi_j^*, t_c + C, \Lambda^{min}$ );
26:  else
27:    return WCT-E-SUB( $\Theta, p_j, exec^{max}, \mu^{max}, t, t_c$ );
28:  end if
29: end if

```

---

be used to represent the number of access requests that can be served in general phases. Otherwise, we perform computation for  $\{(t' - C) - t\}^+$  units of time and then issue one access request (Step 18). As a result, the remaining computation time is  $e' = exec^{max} - \{(t' - C) - t\}^+$  and the remaining requests are  $\mu' = \mu^{max} - 1$ . If we serve *exactly one request* in one time slot assigned to processing element  $p_j$  by injecting *unlimited* requests, we can find the time slot in which the remaining computation time becomes 0. That is, we find integers  $h$  and  $n^*$  such that

$$hB^* + B_{j,\pi(t'),n^*} \geq e' > hB^* + B_{j,\pi(t'),n^*-1}. \quad (4.8)$$

In other words, we compute the number of TDMA cycles ( $h$ ) and the time slot ( $n^*$ ) in the last TDMA cycle that is required to perform all computation, assuming a single access request is issued in each time slot and assuming that the processing element stalls in between subsequent time slots.

**Algorithm 4.3** WCT-E-SUB**Input:**  $\Theta, p_j, exec^{max}, \mu^{max}, t, R_{ub}$ ;**Output:** Worst-case completion time;

- 1: let  $t'$  be the starting time of the next time slot after time  $t$ ;
- 2: find  $h$  and  $n^*$  such that  $h\Psi_j^* + \psi_{j,\pi(t),n^*} = \mu^{max} > h\Psi_j^* + \psi_{j,\pi(t),n^*-1}$ ;
- 3:  $e' \leftarrow exec^{max} - B^* \left\lfloor \frac{\mu'}{\Psi_j^*} \right\rfloor - B_{j,\pi(t'),n^*} - (t' - t - C)$ ;
- 4:  $R_{lb} \leftarrow t' + hL(\Theta) + H_{\pi(t'),n^*} + e'$ ;
- 5: **return** the binary search result  $t^\dagger$  by applying WCT-E-SUB-TEST() by setting  $t^\dagger$  in the range  $[R_{lb}, R_{ub}]$ ;

The estimated completion time  $t_c$  computes as  $t_c \leftarrow t' + hL(\Theta) + H_{\pi(t'),n^*-1} + C + (e' - hB^* - B_{j,\pi(t'),n^*-1})$ . If the number of remaining requests  $\mu'$  is more than there are time slots between  $t'$  and the estimated completion time, i.e.,  $h\Psi_j^* + n^* < \mu'$ , only access requests remain to be served. The completion time for these access requests is analyzed using Algorithm WCT-AR with starting time  $t_c + C$ , since one pending access request is served immediately upon activation of slot  $n^*$  (Step 25).

Steps 16 and 25 represent the cases where all computation is executed and only access requests to the shared resource remain to be finished. The worst-case completion time of the remaining accesses can be computed by Algorithm WCT-AR, considering that in an general phase the issuing time of an access request can be postponed by executing computation for  $\epsilon$  units of time. Therefore, the number of access requests that can be served in any sequence of time slots is represented by  $\Lambda^{min}$ . Furthermore,  $C$  units of time are allocated to serve one open access request that might be pending at the beginning of the next active time slot.

If  $h\Psi_j^* + n^* \geq \mu'$ , the estimated completion time  $t_c$  gives an upper bound of the worst-case completion time. To derive a tighter worst-case completion time, we continue with Algorithm WCT-E-SUB (Algorithm 4.3) by specifying  $R_{ub} = t_c$  as the upper bound of the worst-case completion time. The basic idea in Algorithm WCT-E-SUB is to first find a lower bound of the worst-case completion time by issuing a single request to the shared resource in each time slot from the current time slot. After all requests are served, the remaining computation is performed, which gives a lower bound  $R_{lb}$  (Step 4 in Algorithm 4.3). Now, we have an upper bound and a lower bound of the worst-case completion time, and apply binary search to find  $t^\dagger$  between these bounds and test whether  $t^\dagger$  is feasible or not.

The test, whether  $t^\dagger$  is feasible or not, is shown in Algorithm 4.4. As  $t^\dagger \geq R_{lb}$ , we know that there are at least  $\mu^{max}$  time slots available for processing element  $p_j$  from time  $t$  to time  $t^\dagger$ .

The schedulability test distributes the  $\mu^{max}$  requests among the time slots for processing element  $p_j$  in time interval  $(t, t^\dagger]$ , such that the time spent waiting for grants to the shared resource is maximized. Let  $\mathcal{V}_j$  denote

**Algorithm 4.4** WCT-E-SUB-TEST**Input:**  $\Theta, p_j, exec^{max}, \mu^{max}, t, t^\dagger$ ;**Output:** Feasibility to finish by  $t^\dagger$  for the worst case;

---

```

1: for each  $v_q \in \mathcal{V}_j$  do
2:   derive  $\eta_q$  by Equation (4.9);
3: end for
4:  $\mu' \leftarrow \mu^{max}$ ;  $wait \leftarrow 0$ ;
5: while  $\mu' > 0$  do
6:   find the index  $n^*$  such that  $\Delta_j(v_{n^*})$  is the maximal among those  $v_q \in \mathcal{V}_j$  with
      $\eta_q > 0$ ;
7:   if  $\eta_{n^*} > \mu'$  then
8:      $wait \leftarrow wait + \mu' \Delta_j(v_{n^*})$ ;  $\eta_{n^*} \leftarrow \eta_{n^*} - \mu'$ ;  $\mu' \leftarrow 0$ ;
9:   else
10:     $wait \leftarrow wait + \eta_{n^*} \Delta_j(v_{n^*})$ ;  $\mu' \leftarrow \mu' - \eta_{n^*}$ ;  $\eta_{n^*} \leftarrow 0$ ;
11:   end if
12: end while
13: if  $wait + 2C\mu^{max} + exec^{max} \leq t^\dagger - t$  then
14:   return "feasible";
15: else
16:   return "infeasible";
17: end if

```

---

the ordered vector of time slots assigned to  $p_j$  in time interval  $[t, t^\dagger]$  and  $v_q \in \mathcal{V}_j$ . If we issue a request in a time slot  $v_q$ , we will experience at most  $\Delta_j(v_q) + 2C$  waiting time to finish the request. Furthermore,  $\eta_q$  represents the number of instances of slot  $v_q$  in time interval  $[t, t^\dagger]$  (excluding the the time slot at time  $t^\dagger$ ), since this interval might span over several TDMA cycles. That is,

$$\eta_q = \begin{cases} \lceil \frac{\hat{t}-t}{L(\Theta)} \rceil & \text{if } \pi(t) \leq v_q < \pi(\hat{t}) \text{ or } \pi(\hat{t}) < \pi(t) \leq v_q \\ & \text{or } v_q < \pi(t) < \pi(\hat{t}) \\ \lceil \frac{\hat{t}-t}{L(\Theta)} \rceil - 1 & \text{otherwise,} \end{cases} \quad (4.9)$$

where  $\hat{t} = \sigma_{\pi(t^\dagger)}$  is the starting time of the time slot at time  $t^\dagger$ . As we want to find the largest waiting time among those  $\sum_{v_q \in \mathcal{V}_j} \eta_q$  time slots, we find those  $\mu^{max}$  time slots with the largest waiting time (Step 4 to Step 12 in Algorithm 4.4, where  $wait$  is a variable to indicate the waiting time without considering the time to complete any request). Then, we test whether  $wait + 2C\mu^{max} + exec^{max}$  is less than  $t^\dagger - t$  (Step 13 to Step 17 in Algorithm 4.4) to test the feasibility.

**Correctness and Complexity of Algorithm WCT-E**

The time complexity is  $O(M_\Theta)$  if Algorithm WCT-E-SUB is not required. As  $R_{ub} - R_{lb}$  is bounded by  $exec^{max}$ , the time complexity of Algorithm



WCT-E-SUB is  $O(M_\Theta \log exec^{max})$ , which is also polynomial. Therefore, the overall time complexity is  $O(M_\Theta \log exec^{max})$ .

Given a general phase on processing element  $p_j$  starting at time  $t$  with  $exec^{max}$  computation time and  $\mu^{max}$  requests to the shared resource, consider a schedule  $\mathbf{S}$ , that

- issues an access request  $C - \epsilon$  time units before the end of each upcoming time slot assigned to processing element  $p_j$ . This way, access requests cannot be served in their current time slot anymore and the system stalls until its next time slot becomes active, at which time  $C$  units of time are consumed to execute the access request. (Note: for time slots shorter than  $2C$ , a request is issued immediately after the pending request has completed, resulting in the system to block.)
- executes computation at all times when the system is neither stalling nor executing access requests.

Suppose that, in schedule  $\mathbf{S}$ ,  $t_1$  is the completion time for performing computation and  $t_2$  is the completion time of the last request. Let  $\mu_{t_1}$  be the requests completed before  $t_1$  in schedule  $\mathbf{S}$ .

**Lemma 4** *If  $t_1$  and  $t_1 + C$  are in the same time slot, let  $\bar{t}$  be  $t_1 + C$ ; otherwise, let  $\bar{t}$  be  $C$  plus the starting time of the next time slot for the phase after  $t_1$ . If  $t_2 > t_1$ , the completion time by issuing  $\mu^{max} - \mu_{t_1}$  requests after time  $\bar{t}$  is an upper bound of the completion time for the general phase starting at time  $t$ , assuming  $\Lambda^{min}$  as the number of access requests that can be executed in any sequence of time slots.*

**Proof.** We can prove this lemma by swapping 1 request with computation amounting for  $C$  units of time. Let  $\phi^*$  be a trace with computation and requests located according to schedule  $\mathbf{S}$ . For any different trace  $\phi$ , with an arbitrary access request being swapped with an arbitrary portion of computation amounting for  $C$  units of time, results in (1) a trace that performs computation in between time slots, instead of stalling, and therefore results in a smaller completion time or (2) results in a trace that stalls between assigned time slot and therefore results in the same completion time as the trace produced by schedule  $\mathbf{S}$ . It is not difficult to see that one can always swap the computation and one request without extending the completion time. Therefore, schedule  $\mathbf{S}$  produces the worst-case completion time.

□

**Lemma 5** *If  $t_1 \geq t_2$  and  $|\mathcal{V}_j| = 1$ , the estimated completion time  $t_c$  in Algorithm 4.2 ( $R_{lb}$  in Algorithm 4.4) is an upper bound of the worst-case completion time.*

**Proof.** Similarly, if there is computation left after all access requests are served, the same swapping as in the proof of Lemma 4 can be performed. Again, swapping an access request with an equivalent amount of computation time results in a trace of equal or smaller completion time as schedule  $\mathbf{S}$ .

□

Lemma 5 states that it is not necessary to run a binary feasibility test in Algorithm WCT-E-SUB for a regular TDMA arbiter, in which each processing element has only one access time slot in a TDMA cycle, i.e.,  $|\mathcal{V}_j| = 1$ . The time complexity for such a case becomes  $O(1)$ . However, to the best of our knowledge, for general TDMA arbiters, binary search to test feasibility performs best.

**Lemma 6** *If  $t_1 \geq t_2$  and setting the completion time  $t^\dagger$  as  $t_{\mathbf{S}}$  returns "feasible" by Algorithm WCT-E-SUB-TEST, the completion time of any schedule  $\mathbf{S}'$  for the general phase is no more than  $t^\dagger$ .*

**Proof.** When Algorithm WCT-E-SUB-TEST returns "feasible", it is not difficult to see and prove that Algorithm 4.4 is based on a concrete trace such that the total *blocking* time in which the phase waits for the grants to the shared resource is the *largest*. If a schedule  $\mathbf{S}'$  with completion time larger than  $t^\dagger$  exists, it must create longer blocking times to wait for accessing the shared resource, which contradicts the largest blocking time of the trace considered in Algorithm 4.4.

□

**Lemma 7** *If  $t_1 \geq t_2$  and  $t^\dagger$  is the worst-case completion time, applying Algorithm WCT-E-SUB-TEST by setting the completion time  $t^\dagger - \epsilon$  as  $t_{\mathbf{S}}$  returns "infeasible", for any  $\epsilon > 0$ .*

**Proof.** As the waiting time and the time required to perform computation and to serve the remain access requests is larger than  $t^\dagger - \epsilon - t$ , namely  $t^\dagger - t$ , there must exist a trace, accordingly. Therefore, Algorithm WCT-E-SUB-TEST returns "infeasible".

□

As a result, we conclude the analysis for the general phase by the following theorem.

**Theorem 2** *The worst-case completion time derived from Algorithm WCT-E provides an upper bound of the completion time for any feasible execution bounded by computation time  $exec^{max}$  and requests  $\mu^{max}$  on processing element  $p_j$ .*

**Proof.** Lemma 2 and Lemmas 4, 5, 6 and 7 state that schedule  $\mathbf{S}$  results in the trace with the worst-case completion time. Therefore, assuming a hardware platform without timing anomalies, Algorithm WCT-E derives the worst-case completion time. □

## 4.6 Timing Analysis for Superblocks and Tasks

With the analysis of the completion time of a phase, we can analyze the schedulability of the superblocks on processing element  $p_j$ . Testing the schedulability of tasks on processing element  $p_j$  requires to analyze for a time period equal to the least common multiplier (LCM) of  $p_j$ 's period  $W_j$  and the TDMA cycle  $L(\Theta)$ . This way, all possible offsets between slots and superblocks are considered.

Since there is no interference of other processing elements when we analyze processing element  $p_j$ , it is clear that there is no anomaly. That is, if a superblock completes earlier than its worst-case completion time, this will not increase the worst-case execution time of the remaining superblocks. As a result, we have to consider the worst-case execution time of the superblocks sequentially, see Algorithm 4.5. Algorithm 4.5 represents the analysis for the hybrid resource access model and the subsequent execution of superblocks, i.e., model HSS. The dedicated resource access model with subsequent execution of superblocks, i.e., model DSS, is derived by setting the parameter  $\mu_{i,j}^{max,e} = 0$ . The general resource access model with subsequent execution of superblocks, i.e., model GSS, is derived by setting the parameter  $\mu_{i,j}^{max,a} = 0$  and  $\mu_{i,j}^{max,r} = 0$ . In order to compute the worst-case completion time for the three resource access models, with time-triggered execution of the superblocks, the starting times of the superblocks in Algorithm 4.5, need to be adjusted, see Line 4 where the starting time  $t$  is adjusted to the dedicated starting time  $\rho_{i,j}$  of superblock  $s_{i,j}$ . Furthermore, in order to analyze the model with time-triggered phases, Lines 7, 9 and 11 need to be adjusted to the corresponding earliest starting time  $\rho_{i,j,a}$ ,  $\rho_{i,j,e}$  and  $\rho_{i,j,r}$  for the acquisition, general and replication phase, respectively.

The time complexity of the algorithm is  $O(|\mathcal{S}_j| \frac{LCM(W_j, L(\Theta))}{W_j})$ . To conclude the analysis, the following theorem gives the correctness of Algorithm 4.5.

**Theorem 3** *A TDMA arbiter based on TDMA schedule  $\Theta$  is schedulable for scheduling tasks/superblocks assigned on processing element  $p_j$  if Algorithm 4.5 returns "schedulable".*

**Algorithm 4.5** Schedulability-Test**Input:**  $\Theta, p_j, \mathcal{S}_j, W_j$ ;**Output:** Schedulability of TDMA schedules

---

```

1: for  $g \leftarrow 0$ ;  $g < \frac{LCM(W_j, L(\Theta))}{W_j}$ ;  $g \leftarrow g + 1$  do
2:    $t \leftarrow g \cdot W_j$ ;
3:   for each  $s_{i,j} \in \mathcal{S}_j$  sequentially do
4:     if  $t < g \cdot W_j + \rho_{i,j}$  then ▷ for the time-triggered phases only
5:        $t \leftarrow g \cdot W_j + \rho_{i,j}$ ;
6:     end if
7:      $//t \leftarrow \rho_{i,j,A}$  for time-triggered phases
8:      $t \leftarrow \text{WCT-AR}(\Theta, p_j, \mu_{i,j}^{max,a}, t, \Lambda^{max})$ ;
9:      $//t \leftarrow \rho_{i,j,E}$  for time-triggered phases
10:     $t \leftarrow \text{WCT-E}(\Theta, p_j, \text{exec}_{i,j}^{max,e}, \mu_{i,j}^{max,e}, t)$ ;
11:     $//t \leftarrow \rho_{i,j,R}$  for time-triggered phases
12:     $t \leftarrow \text{WCT-AR}(\Theta, p_j, \mu_{i,j}^{max,r}, t, \Lambda^{max})$ ;
13:    if  $t - (g \cdot W_j + \rho_{i,j}) > \ell_{i,j}$  then
14:      return "might be unschedulable" for  $p_j$ ;
15:    end if
16:  end for
17: end for
18: return "schedulable" for  $p_j$ ;
```

---

**Proof.** This results directly from Lemma 2 and Theorem 2 along with the fact that the early completion of a superblock will not increase the worst-case execution time of the remaining superblocks.

□

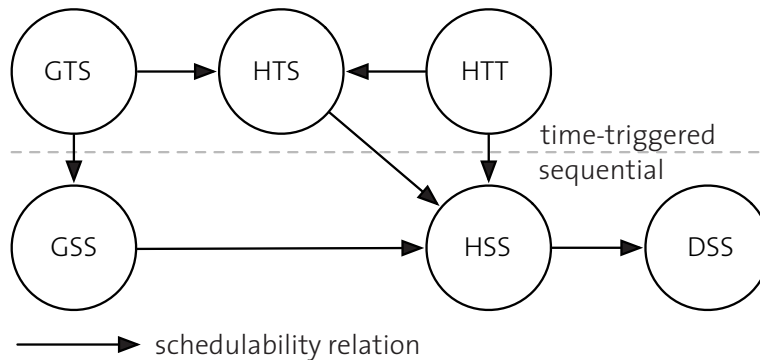
## 4.7 Schedulability among Models

In this section we introduce the relation of schedulability among the proposed models to access a shared resource and to execute the superblocks on the processing element. Consider a set of superblocks, specified according to the models, introduced in Section 3.3.2. Then the schedulability of this set of superblocks can be derived by Algorithm 4.2. Furthermore, assume that (1) the total amount of accesses to the shared resource for each superblock and (2) the amount of computation performed during a superblocks active time are equal for each of the models we consider.

1. If the set of superblocks is schedulable for model GSS, then it is also schedulable for model HSS. Consider a superblock specified according to model HSS and the execution trace that leads to its WCRT. A superblock with equivalent parameters but specified according to model GSS is capable of realizing the same execution trace, since the total amount of accesses and computation time is equal, but their position is confined to dedicated phases in the first model.

2. If the set of superblocks is schedulable for model HSS, then it is also schedulable for the model DSS. Any concrete execution trace that can be realized by a superblock specified according to model DSS, can also be realized by an equivalent superblock specified according to model HSS. Therefore, the execution trace leading to the WCRT for model DSS, is also a feasible execution trace for model HSS, but the trace that leads to the WCRT for model HSS might be infeasible for model DSS.
3. If the set of superblocks is schedulable for the *time triggered superblocks general model* GTS, then it is also schedulable for model HTS. Any execution trace that can be realized by model HTS can also be realized by model GTS for equivalent superblocks and equal start times for the superblocks (equal time triggering), since in the later access requests are restricted to specified phases.
4. If the set of superblocks is schedulable for model HTT, then it is also schedulable for the HTS. In model HTT, the WCRT of the preceding phase of a phase represents the lower bound on its starting time. Therefore, for model HTT, assigning any other starting time for a phases than its preceding phases' WCRT, results in an increased WCRT. Assigning starting times of phases as their respective predecessors WCRT reproduces sequential behavior, as in model HTS.
5. If the set of superblocks is schedulable for model GTS then it is also schedulable for model GSS. The lower bound on the starting time for a superblock in time triggered systems is its predecessors' WCRT. Therefore, the trace leading to the WCRT for a superblock specified according to model GSS, is also a feasible trace for model GTS, but not every possible execution trace for model GTS can be reproduced by model GSS.
6. If the set of superblocks is schedulable for model HTT or for model HTS, then it is also schedulable for model HSS. The starting times of superblocks and phases in model HTS and HTT, respectively are lower bounded by the WCRT of the respective preceding superblocks and phases of an equivalent superblock specified according to model HSS. The WCRT of a superblock specified according model HTS or HSS is at least equal to the WCRT of a superblock specified according to model HSS. The schedulability relation follows directly.

As a conclusion, if any of the six models results as schedulable, model DSS, with dedicated phases to access the shared resource or to perform computation, executed sequentially, is schedulable as well, see Figure 4.2.



**Figure 4.2:** Schedulability Relationship between different models.

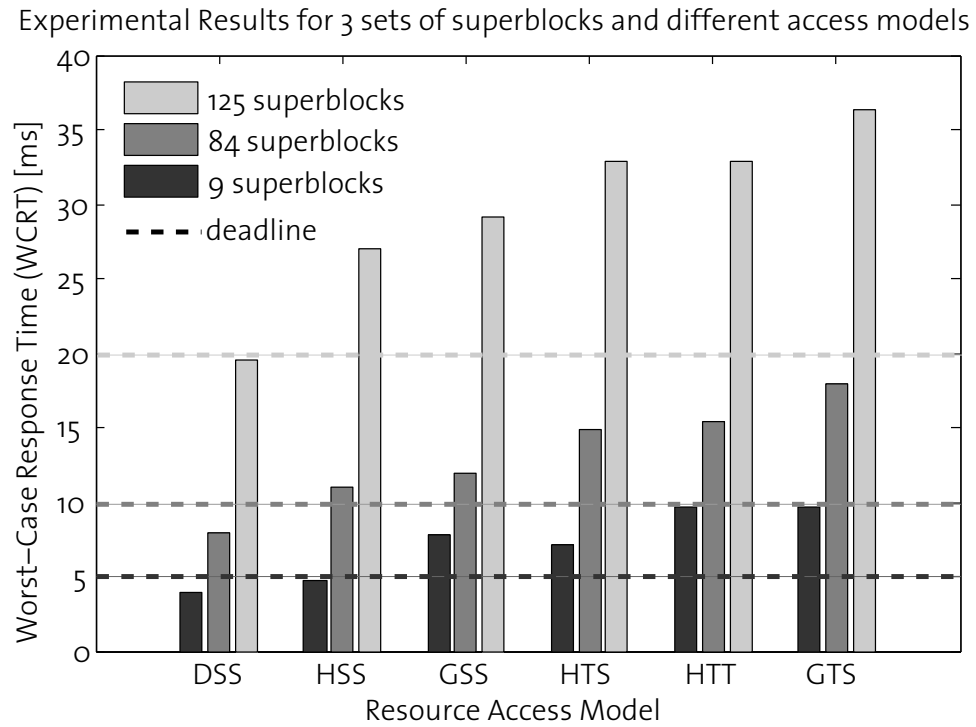
Therefore, this model is the model of choice for multiprocessor, resource sharing systems.

## 4.8 Experimental Results

In this section, we present the results of our proposed worst-case analysis framework, applied to sequences of superblocks of different sizes and under varying arbiters. We also show a concrete execution trace to visualize the process of obtaining the worst-case response time. Our proposed worst-case analysis is applied to three sequences of superblocks. The first sequence represents a small task, with only 9 subsequent superblocks and a processing cycle (period) of  $5ms$ . The second sequence is constituted by 84 superblocks, and a period of  $10ms$ , while the third sequence has 125 superblocks and a period of  $20ms$ . These sequences resemble  $5ms$ ,  $10ms$  and  $20ms$  *runnables* according to the AutoSAR [Aut] specification. The parameters of the superblocks are generated using random number generators, following specifications provided by an industrial partner in the automotive industry.

First a set of superblocks according to the *hybrid model* is generated using a random number generator. Based on this superblock, an equivalent superblock for each resource access model is generated. In other words, the total number of accesses to the shared resource and the maximum required computation is the same for superblocks representing one runnable and different resource access models.

Based on the hybrid model, the general access model is derived by accumulating the accesses of the acquisition and replication phases to a single phase in the general model. The dedicated model is derived by moving a certain share of accesses from the general phase to the acquisition and replication phase, while at the same time, the total number of accesses for a superblock remains the same for each of the three access models.



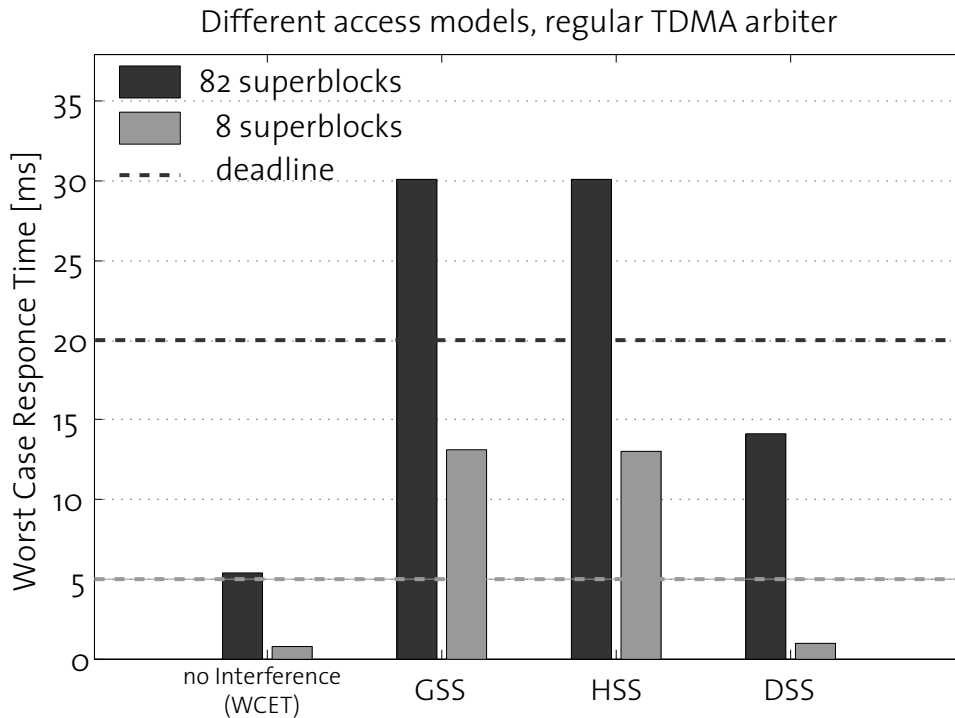
**Figure 4.3:** Experimental results for a regular TDMA arbiter and three resource access model.

Our proposed analysis framework assumes a given TDMA arbiter and therefore, in order to perform experimental evaluations, an arbiter was constructed. Each runnable is assigned to a processing element, and the time slot assigned to this processing element is adjusted, such that the runnable following the DSS model is the only to remain schedulable.

Figure 4.3 presented the resulting WCRT for 9, 84 and 125 superblocks, respectively. The dashed lines represent the corresponding period and deadline. The dedicated access model outperforms the other access models with respect to worst-case response time. This is due to the limited variability inherent to the dedicated access model, as shown in Section 4.7.

Another possibility is to design an arbiter is to create an *irregular TDMA arbiter* with randomly distributed slots, where multiple slots of varying lengths are assigned to a processing element. These arbiters might be able to decrease the WCRT of tasks, due to a better adjustment of the arbiter to the actual resource access pattern, but design methodologies to derive optimal TDMA arbiter are out of the scope of this thesis. Nevertheless, as shown in Theorem 2, the complexity of the analysis approach increases significantly.

The difference among irregular and regular arbiters is presented in Figures 4.4 and 4.5. Two sets of superblocks, resembling  $5ms$  and  $20ms$

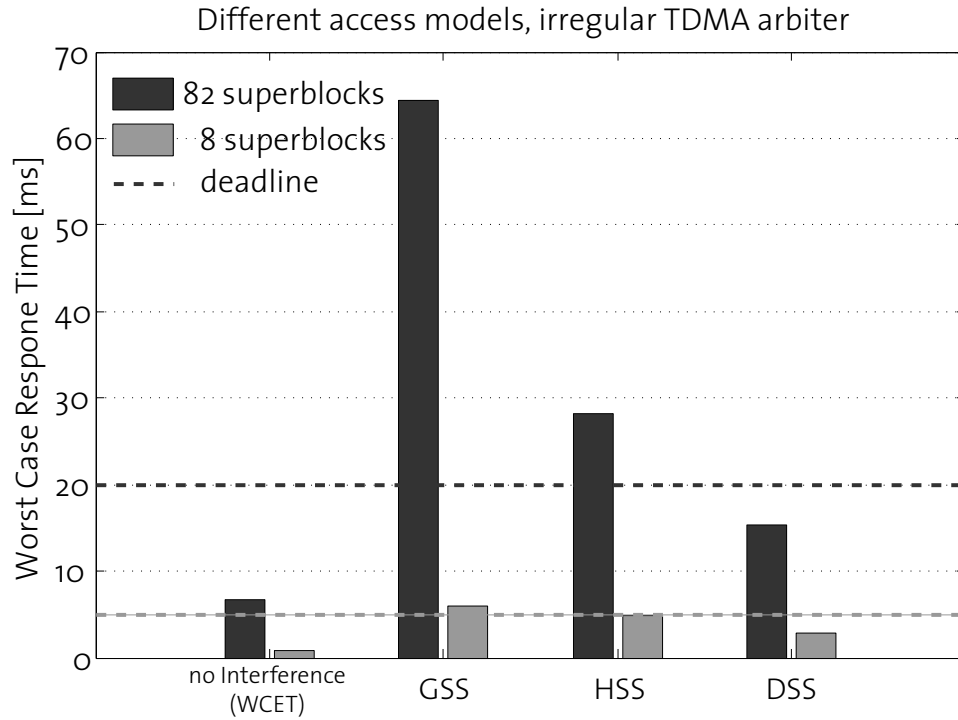


**Figure 4.4:** Experimental results for a regular TDMA arbiter.

runnables with 8 and 82 superblocks respectively, are generated according to the previously presented approach. Two superblock sequences are analyzed, in 4 different access models. The most left pair of bars in Figure 4.4 represents the naive worst-case execution time, considering only a constant amount of time consumed by each access to the shared resource, i.e., when the shared resource is always available. The next three pairs of bars show the results for different memory access models. Since we have shown that time-triggered execution models cannot increase the performance, we omit these models and show only results for subsequent execution models.

In Figure 4.5 the same superblock sequences are analyzed again, but this time with an irregular TDMA arbiter, as described before. Again, the dedicated access model outperforms all the others. This effect is very apparent in Figure 4.5, but also observable in Figure 4.4. For the regular arbiter used in the first set of experiments, the delays are dominated by waiting for the next period of the arbiter, while in the second set of experiments, the distance between the single time slots is the determinant measure. The difference between the WCRT for the irregular and regular arbiter increases with increased non-determinism in the resource access model. The WCRTs for the model GSS diverge significantly for regular and irregular arbiter, while the WCRTs for model DSS and the two arbiters are similar.



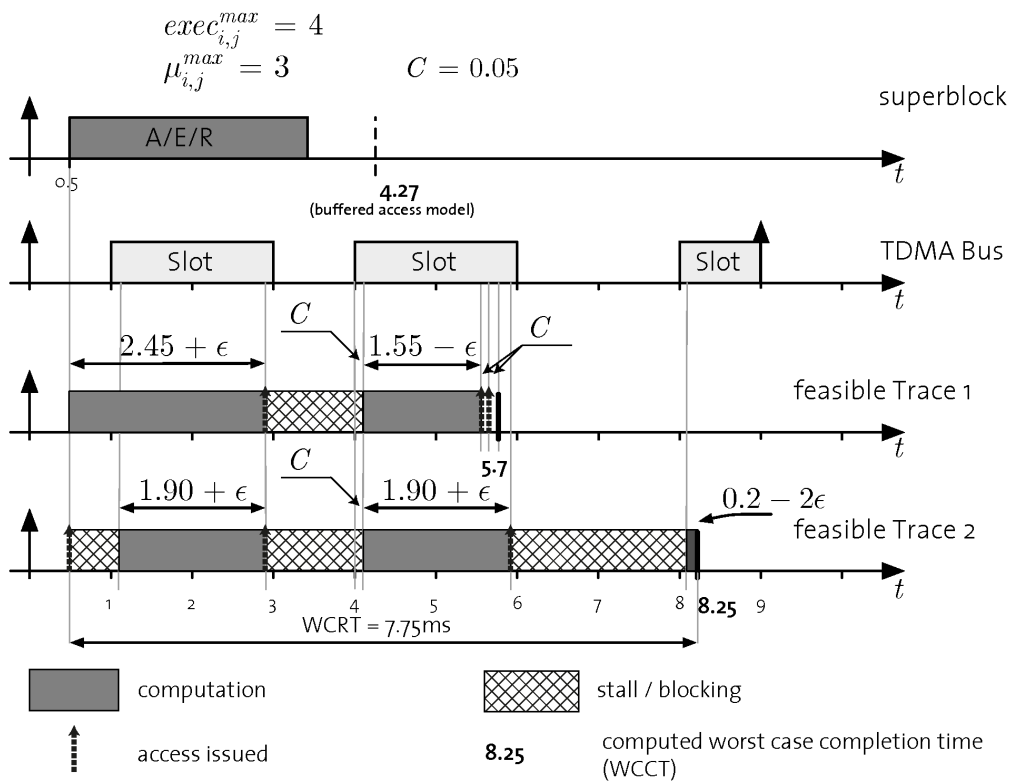


**Figure 4.5:** Experimental results for an irregular TDMA arbiter.

In order to derive the worst-case response time for superblocks, different traces of requests to the shared resource have to be examined. In Figure 4.6, we analyze a sample superblock with earliest release time  $\rho = 0.5ms$ . We show one trace that leads to the worst case response time (WCRT) (feasible Trace 2) and one that does not.

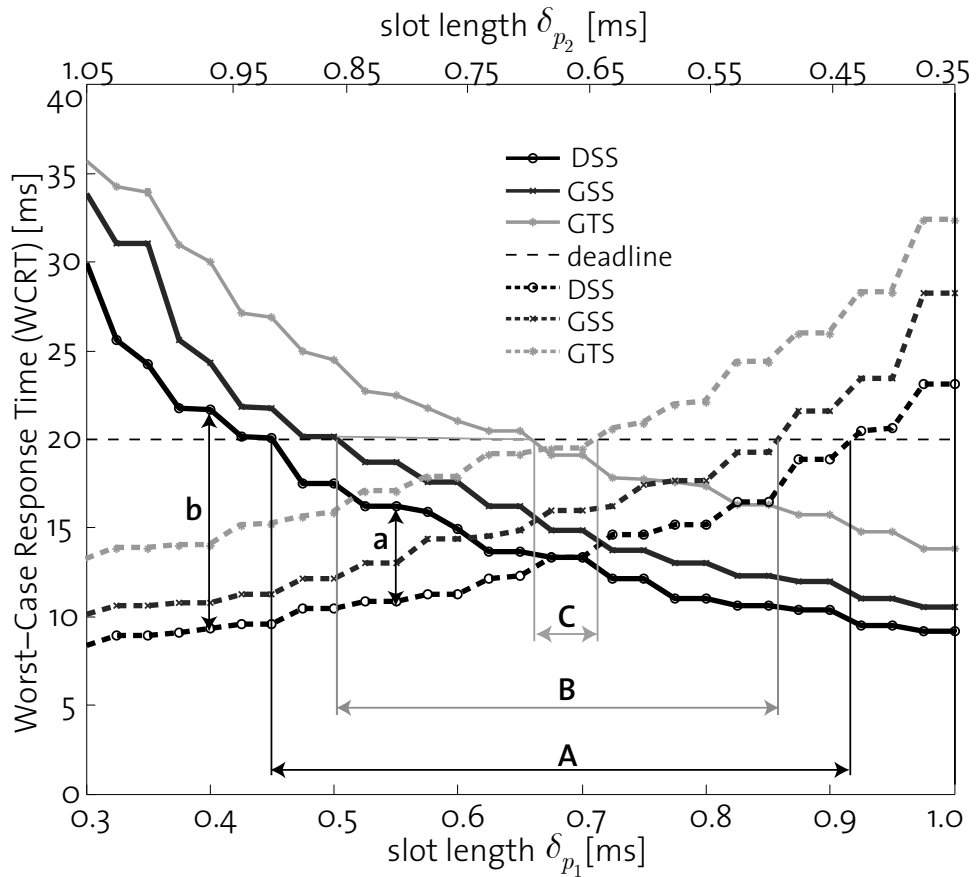
Trace 1 (feasible Trace 1) starts with performing computation at time 0.5 ms while Trace 2 (feasible Trace 2) issues an access to the shared resource and therefore stalls until the next time slot becomes available, at time 2.0ms, see the recursive call in Algorithm 4.2, Line 8. Then, both traces perform computation till the end of the time slot, i.e., till time  $2.95 + \epsilon$ . Eventually, Trace 1 finishes its computation, at time 5.6ms, and the remaining accesses to the shared resource have to be issued. Since the slot is currently active, these accesses are completed and the superblock finishes at 5.7ms. Trace 2, on the other hand, computes until the end of the second slot before issuing another access request. After a long stall block, the system continues to do computation for 0.2ms before finishing at 8.25ms, which results in a worst-case response time (WCRT) of 7.75ms. This shows, that very small deviations can result in large variances on the resulting WCRT.

The relationship between the length of a slot and the schedulability is shown in Figure 4.7. Here we consider two applications, executing on two different processing elements, i.e., they compete for the shared resource.



**Figure 4.6:** Worst-case completion time for a sample superblock and a TDMA arbiter (TDMA Bus).

The application on processing element 1 is represented by the solid lines and the bottom x-axis, labeled  $\delta_{p_1}$ . The application on processing element 2 is represented by the dashed lines and the top x-axis, labeled  $\delta_{p_2}$ . Note that the direction of the two x-axis are antipodal. For clarity of visualization, only three models are shown per application. Furthermore, the length of the schedule  $L(\Theta)$  is constant, i.e.,  $L(\Theta) = \delta_{p_1} + \delta_{p_2} = 1.35ms$ . Since the x-axes are antipodal, the region where both applications are schedulable can be derived. These regions are marked as "A", "B" and "C" in the figure, for model DSS, model GSS and model GTS respectively. Consider example "a" in Figure 4.7, which shows that both applications are schedulable, their slot lengths are  $\delta_{p_1} = 0.55ms$  and  $\delta_{p_2} = 0.7ms$  respectively, and their worst-case completion times are  $16.25ms$  and  $10.82ms$  respectively. Contrarily, example "b" result in a deadline miss for application 1. Conclusively, the region of schedulability for model DSS is larger than for model GSS and GTS, respectively.



**Figure 4.7:** Schedulability for two processing element accessing a shared resource.

## 4.9 Chapter Summary

In this chapter, we introduced a novel methodology to analyze the worst-case response time of real-time tasks/superblocks on systems with shared resources and TDMA arbitration policies. Accesses to the shared resource result in blocking of a real-time task/superblock until the request is completed, and therefore contention on the resource results in significantly increased delays. We present an analysis framework that allows to efficiently compute the worst-case response time and schedulability of tasks/superblocks specified according to any of the proposed access models and for any TDMA arbitration on the shared resource.

The dedicated model with sequential superblocks is shown to be schedulable as soon as any of the other models is schedulable. Experimental results demonstrate the superiority of model DSS with respect to worst-case completion time.

We show that separating computation and accesses to the shared resource (dedicated model) is of crucial importance in the design of resource

sharing systems. Using this model allows to derive tight and efficient algorithms for schedulability analysis and significantly reduces the worst-case response time of real-time tasks.

# 5

## Hybrid arbitration on shared resources

In the previous chapter, Chapter 4, we introduced static arbitration on the shared resource to achieve a predictable system. Static arbitration policies, such as TDMA, provide isolation among the processing elements of a multicore systems. This way, we can analyze the worst-case response time (WCRT) of the superblocks on a processing element, without the need to consider the other processing elements in the system. In fact, we can even add processing elements to the system at runtime, providing that the TDMA arbiter on the shared resource stays unchanged.

However, static arbitration policies need to be designed a priori, and cannot adapt to changing access patterns or execution environments. For many applications in the domain of control and signal processing, this does not represent a major obstacle, since the behavior of the application can be predicted very accurately. Nevertheless, real-time systems are increasingly embedded in all-day life and applications are getting more and more diverse. Therefore, there is an increasing need to accommodate applications that cannot be predicted that easily anymore. Popular examples for such applications are user assistance applications in cars, that are either event-triggered (e.g., collision avoidance or electronic stability program (ESP)) or speed dependent (e.g., fuel injection control).

Industrial efforts in this direction resulted in the FlexRay communication protocol [Fle], that combines static arbitration and dynamic arbitration, and thus represents a hybrid approach to accessing shared resources. In the FlexRay communication protocol, the arbiter is split into two segments. The

first segment represents a static TDMA arbiter, while the second segment assigns mini-slots to processing elements following the First-Come-First-Serve (FCFS) policy. It represents a trade-off between the predictability of static arbiters and the flexibility of the FCFS arbitration policy.

In this chapter, we propose an approach to analyze the WCRT for systems that follow the previously stated models of execution and models of accessing the shared resource, see Chapter 3, and apply the FlexRay communication protocol on the shared resource. We present an efficient algorithm based on dynamic programming and experimental results, that emphasize the need for sophisticated analysis tools for resource sharing systems, particularly when task isolation cannot be achieved and interference has to be taken into consideration.

## 5.1 Introduction

In the multicore era, performance improvement with respect to computation depends on instruction, task and data parallelism. In order to reduce hardware costs, most commercial multicore platforms have shared resources, such as buses, main memory, and DMA in multicore and MPSoC systems. As a result, shared resources have become the bottleneck for performance improvement in multicore systems. Efficient arbitration of shared resources allows to improve performance significantly.

We focused on a static resource sharing strategy in Chapter 4, in order to guarantee predictability. In order to increase the performance in terms of throughput, it is a good strategy to reassign the unused time-slots of a task in a static arbiter to other tasks, with pending access requests. That way, the performance of these tasks can be increased, while the other elements in the system do not suffer from additional delays.

In order to be able to adapt to real-time tasks with early completion or less resource accesses, *adaptive resource arbiters*, such as the FlexRay communication protocol [Fle] in the automotive industry, have been proposed and adopted recently. An adaptive resource arbiter combines dynamic and static arbitration, in which an arbitration interval is divided into a static arbitration segment with a static slot to processing element assignment and a dynamic arbitration segment. As a result, it provides isolation between processing elements in the static arbitration segment, ensuring timing guarantees, and, in the dynamic segment, allows dynamic arbitration to improve the response time.

In safety-critical embedded systems, such as controllers in the Automotive Open System Architecture (AutoSAR) [Aut], timing guarantees are required, and thus, the WCRT has to be determined. Applying a hybrid arbitration policy, such as the FlexRay communication protocol, increases the

complexity of this problem, since the isolation between processing elements and tasks is sacrificed for a potentially increased performance.

For synchronous resource accesses with a TDMA resource arbiter, we can apply the results in Chapter 4, whereas the results in [PSC<sup>+</sup>10] can be adopted for a dynamic arbiter, such as the FCFS or the Round-Robin (RR) strategy. The adaptive resource arbiter presented in this chapter requires the joint considerations of dynamic and static arbitration, and hence, these analysis frameworks cannot be directly applied. To the best of our knowledge, providing timing guarantees for synchronous resource accesses with an adaptive resource arbiter is an open problem.

We consider a given task partitioning, in which a task is allocated on a predefined processing element. Each task is divided into a set of superblocks, which are executed in a fixed sequence, and characterized by their worst-case number of accesses to the shared resource and their worst-case computation time (derived assuming resource accesses take zero time). Interference from other tasks and processing elements has to be considered during the dynamic segment of the arbiter. This interference is represented as an arrival curve, that defines the maximum amount of access requests in a time-window, and is derived in Chapter 3.

In this chapter we study the problem of deriving the worst-case response time (WCRT) of the tasks executing on a processing element  $p_j \in \mathcal{P}$ . We assume an adaptive arbiter  $\Theta$  granting access to the shared resource. We unify the approach to derive a tight bound on the response time for static arbitration, derived in Chapter 4, with the approach for dynamic arbitration policies presented in [PSC<sup>+</sup>10]. Therefore, the proposed approach is a generalization of these studies and allows to derive worst-case response times for adaptive resource arbiters. An adaptive schedule for the shared resource is said to be *schedulable* if all the superblocks on all processing elements can finish before their respective deadlines. The contributions of this chapter are as follows:

- Based on dynamic programming, we develop an algorithm that derives an upper-bound of the worst-case response time (WCRT) for superblocks and tasks, considering the delay caused by accesses to the shared resource.
- Our analysis generalizes the analysis in [PSC<sup>+</sup>10] for dynamic resource arbiters only and in Chapter 4 for static resource arbiters (TDMA).
- We present experimental results for a real-world application, and show that minor suboptimal decisions can result in significantly increase WCRTs.

In Section 5.3, we introduce the proposed task model and model of the shared resource. We give an overview of our proposed algorithm in

Section 5.4 and detail our notation and methodology in Section 5.5. Sections 5.6 and 5.7 describe our WCRT analysis and prove its correctness. Experimental results are shown in Section 5.8. Section 5.9 concludes the chapter.

## 5.2 Related Work

Timing analysis of asynchronous resource accesses for the FlexRay communication protocol has been recently developed by Pop et al. [PPE<sup>+</sup>08, PPEP07], Chokshi et al. [CB10] and Lakshmanan et al. [LBR10].

Pop et al. consider a system with time-triggered tasks and tasks that are scheduled according to the Fixed-Priority (FP) policy, while time-triggered tasks preempt FP tasks. Furthermore, messages produced by tasks are assigned to the static or the dynamic segment of the FlexRay arbiter. During any slot (dynamic or static), only the one message that holds the corresponding frame identifier can be transmitted. This way, bus conflicts can be avoided offline, by allocating one slot to at most one message. The dynamic segment differs from the static segment in a way, that the slot length is assigned differently. In the static segment, the length of a slot is fixed. In contrast to that, in the dynamic segment, a slot has a dynamic number of mini-slots, while the number of mini-slots corresponds to the amount of time that is required to transmit the whole message assigned to it. In our proposed methodology, access requests from a processing element are not assigned to any specific segment, nor to a specific slot in the dynamic segment. This way, an access request is served during a time slot assigned to its processing element in the static segment, or by a mini-slot during the dynamic segment. In the dynamic segment, access requests are served according to their arrival time, i.e., following the FCFS policy. As a result, the number of feasible execution traces is significantly increased, since no slot assignment has been performed that would avoid bus conflicts. While this increases the flexibility of the system, deriving an analysis approach becomes increasingly complex.

Chokshi et al. provide an analysis for the FlexRay communication protocol, assuming buffered (asynchronous) communication using Real-Time Calculus [CKT03a]. In other words, resource access requests can be buffered, while the execution on the processing element may continue. This analysis allows to derive a lower service curve, and by considering event arrivals eventually buffer sizes can be estimated. Lakshmanan et al. [LBR10] consider hierarchical bus structures composed of FlexRay, CAN, etc. and derive End-to-End timing analysis. They derive utilization bounds for the static and dynamic segments of the FlexRay communication protocol, under the assumption that the period of the tasks and the period of the arbiter are



multiples of each other. By experimental evaluation, the authors provide utilization bounds for the CAN bus. Nevertheless, synchronous resource access leads to blocking, which is not considered by these approaches.

Burgio et al. [BRE<sup>+</sup>10], introduce an adaptive TDMA arbiter, to allow for adaptive behavior and to ensure a high utilization of the processors in a multicore environment. Upon a workload change, a slave core requests a new service level at a master. The master collects service requests from all the slaves and generates a new TDMA time wheel, that mediates the overall bandwidth between all the cores. This new time wheel is communicated back to the cores, that adjust their task periods accordingly, such that real-time constraints are guaranteed. The WCET that results from the new time-wheel is stored in a look-up table, since online computation would be infeasible. As a result, the system can handle a discreet set of different TDMA configurations, that have to be computed offline and stored on the system in a table. While this approach conserves the task isolation, that is the key advantage of classical TDMA arbiters, it allows for a limited amount of flexibility, mainly restricted by the amount of a priori computed configurations that can be stored on the system.

Results by Pellizzoni et al. [PSC<sup>+</sup>10] can be adopted for a dynamic arbiter, such as the FCFS or the RR strategy. The adaptive resource arbiter requires the joint considerations of dynamic and static arbitration, and, hence, the analysis frameworks in [PSC<sup>+</sup>10], for dynamic arbitration, and the approach presented in Chapter 4 of this thesis, for static arbitration, cannot be applied directly.

## 5.3 System Model

This section presents the models of the tasks, the processing elements, the schedulers, and the resource arbiter of the shared resource.

### 5.3.1 Models of Tasks and Processing Elements

We consider the same models of tasks and processing elements as in Chapter 4, in which there is a set  $\mathcal{P}$  of processing elements that execute independently. Processing elements share a common resource, for example, an interconnection fabric (bus) to access a shared memory.

Accesses to the shared resource are modeled according to the *dedicated phases* model, the *general* model or the *hybrid* model, while the execution of the superblocks on the processing element is follows the *sequential* or *time-triggered* model.

### 5.3.2 Model of the Shared Resource

Following the model in the previous chapter, Chapter 4, the shared resource is only able to serve at most one access request at any time, and the arbiter decides which request is granted. This chapter considers an adaptive arbiter, that follows the FlexRay protocol, where arbitration is conducted based on a sequence of *arbitration rounds*. Each round comprises a *static segment* followed by a *dynamic segment*. Once access to the shared resource is granted, the arbiter, denoted  $\Theta$ , serves an access request within  $C$  units of time. We assume that processing elements and the resource arbiter initialize synchronously, such that the first arbitration round of  $\Theta$  and the first superblock on each processing element start at time 0.

The static segment comprises a sequence of  $M_\Theta$  (time) slots, indexed from 0 to  $M_\Theta - 1$ . Let  $L_\Theta$  be the length of the static segment, while  $\sigma_m$  is the starting time of slot  $m$  relative to the beginning of the round, with  $\sigma_0 = 0$ . For ease of notation, we define  $\sigma_{M_\Theta} = L_\Theta$ ; the duration of slot  $m$  is then  $\delta_m = \sigma_{m+1} - \sigma_m$ . Each slot serves requests of a single processing element according to function  $\theta(m, p_j)$ , i.e.,  $\theta(m, p_j) = 1$  if slot  $m$  is assigned to  $p_j$  and  $\theta(m, p_j) = 0$  otherwise. Multiple slots can be assigned to the same processing element. A request of  $p_j$  in its assigned slot  $m$  is served only if it can be completed within the slot, i.e. it must arrive at least  $C$  time units before the end of the slot.

The dynamic segment of arbiter  $\Theta$  is defined by its length  $\Delta_\Theta$  and the length  $\ell$  of a *mini slot*. In this thesis, we only consider arbiter where  $\Delta_\Theta > C$ . All processing elements contend for access to the shared resource during the dynamic segment, and again a request is granted only if it can be completed within the dynamic segment. Two arbitration models are possible. Under *constrained* request arbitration, resource accesses are granted in FCFS order at the beginning of each minislot, while *greedy* arbitration grants resource accesses in FCFS order also during minislots. Suppose that the dynamic arbitration segment starts at time  $\tau$ . The dynamic arbitration segment is valid for granting requests in time interval  $[\tau, \tau + \Delta_\Theta - C]$  to guarantee that the granted requests do not interfere the static arbitration segment. For the greedy arbitration in a dynamic segment, the arbiter grants access to pending requests in First-Come, First-Serve (FCFS) order. For the constrained arbitration in a dynamic segment, resource accesses are granted in FCFS order only at the beginning of a mini time slot, i.e.,  $t + k\ell$  for positive integer  $k$  with  $k \leq \lfloor \frac{\Delta_\Theta - C}{\ell} \rfloor$ .

The FlexRay communication protocol is a special case of the above definition when the dynamic arbitration segment is constrained and static slots all have the same duration. An example is shown in Figure 5.1. In this chapter, we focus on constrained request arbitration, as applied in the FlexRay communication protocol, while the results can be easily applied for greedy request arbitration and RR arbitration for the dynamic arbitration segment

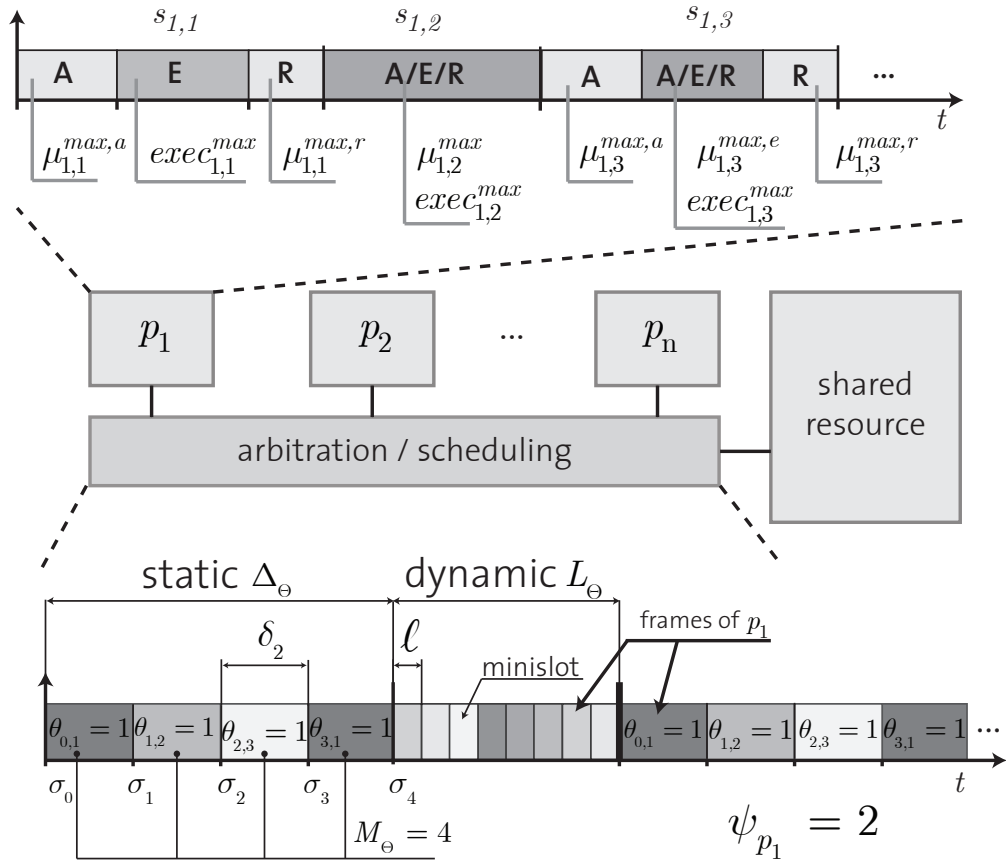


Figure 5.1: An example for the adaptive arbiter.

with minor changes, which are similar to the extension from FCFS to RR in [PSC<sup>+</sup>10].

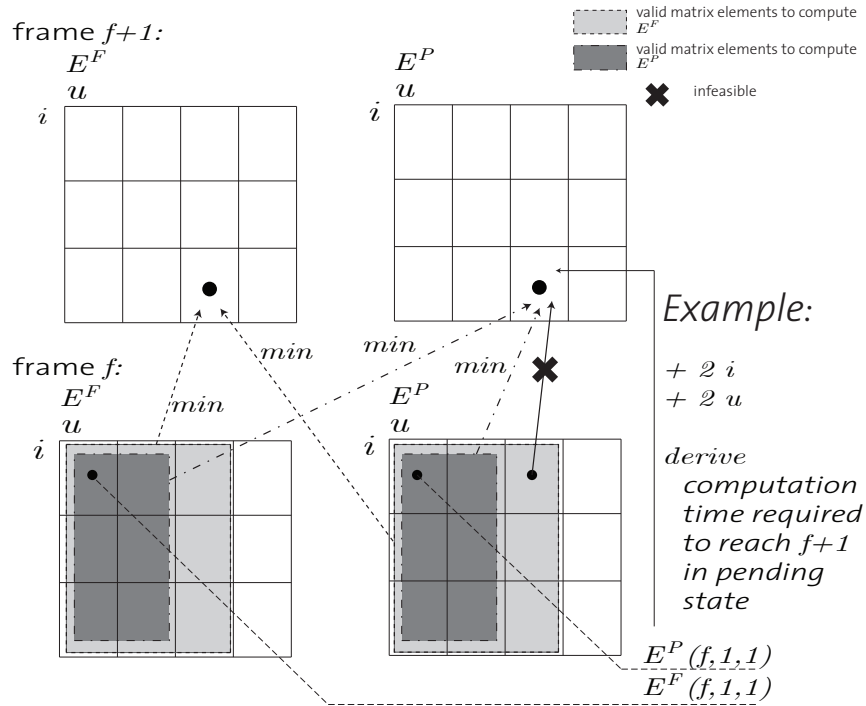
### 5.4 Analysis Overview

In our analysis we propose an algorithm that derives the worst-case execution trace for the sequence of superblocks  $\mathcal{S}_j$  that executes on element  $p_j$ . In other words, we derive the trace that results in the WCRT for the task. The degree of freedom for constructing this worst-case trace is the sequence of access requests and the computation that have to be performed in the individual superblocks. Consider a superblock specified according to the general model. Then the degree of freedom with respect to the worst-case trace, is the points in time when access requests are issued. Since in this model, access requests and computation can happen at any time and in any order, there are many possible execution traces.

Delays due to shared resource contention come from two cases: (1) an access request happens during the static segment, but the current slot is assigned to another processing element or there is not enough time to complete the request; or (2) an access request happens during the dynamic segment of the arbiter, but other access requests are already in the queue or there is not enough time to complete the request. In both cases, the access request has to wait either its turn in the dynamic segment or until the beginning of its next assigned static slot, whichever comes first. Then the access request has to wait its turn. To simplify the discussion, we will refer to time slots assigned to processing element  $p_j$  as "frames of  $p_j$ ". In other words, slot  $m$  is a frame  $f$  of processing element  $p_j$ , iff  $\theta(m, p_j) = 1$ . The dynamic segment is a frame for every processing element  $p_j$  as well, since all elements contend for access to the shared resource during the dynamic segment. Since it is not obvious which pattern of access requests and computation, respectively, creates the worst-case, we propose a dynamic programming algorithm to solve this problem iteratively.

The key idea is as follows. Let  $u$  be the number of access requests performed by the sequence  $\mathcal{S}_j$ , and  $i$  be the number of requests performed by tasks running on other cores that *interfere* (delay through contention) with  $\mathcal{S}_j$  during dynamic segments. Our algorithm derives the *minimum* amount of time that the superblocks in  $\mathcal{S}_j$  must compute to reach the beginning of a particular frame  $f$ , for any feasible value of  $u$  and  $i$ ; note that increasing  $u$  and/or  $i$  can increase the resource access delay suffered by the task, hence less computation time is required to reach a given frame. Based on this information, the algorithm then iteratively derives the minimum amount of computation time required to reach the following frame  $f + 1$ , and so on and so forth until the required computation time to reach frame  $f' + 1$  becomes greater than the worst-case computation time of the task for any value of  $u$  and  $i$ ; this implies that in the worst case the task can only reach up to frame  $f'$ .

Due to the blocking behavior of access requests, a frame  $f$  can be reached in the "free" state, i.e., the processing element that executes  $\mathcal{S}_j$  has no unserved access requests upon activation of frame  $f$ , or in the "pending" state, i.e., the processing element stalls, because there is an unserved access request that has to be served immediately upon activation of frame  $f$ . The minimum amount of computation time is then stored in two tables  $E^F(f, u, i)$  and  $E^P(f, u, i)$  for the free and pending states, respectively. In other words,  $E^F(f, u, i)$  represents the minimum amount of computation that is required to reach frame  $f$ , assuming the state of the processing element is "free" and before the beginning of frame  $f$ ,  $u$  access requests have been served and  $i$  interferences have been suffered.  $E^P(f, u, i)$  represents the analogous for the "pending" state.



**Figure 5.2:** Example how to construct the dynamic programming table.

Consider Figure 5.2 as an example to compute the minimal computation time. We want to determine the minimum amount of computation time  $E^F(f+1, u, i)$  and  $E^P(f+1, u, i)$  that is required to reach frame  $f+1$  in the free and pending state, respectively, for a particular combination of  $u$  access requests and  $i$  interferences. Then, this value depends on the values computed for the previous frame,  $E^F(f, u', i')$  and  $E^P(f, u', i')$ , for all values of  $u', i'$  that are compatible with  $u, i$ : note that  $u - u'$  and  $i - i'$  represent the amount of performed access requests and suffered interferences, respectively, between the beginning of frames  $f$  and  $f+1$ . Clearly,  $i' \leq i$  is a necessary condition. Furthermore,  $u'$  must be strictly smaller than  $u$  to be able to reach frame  $f+1$  in the pending state, since at least one access request needs to be issued after frame  $f$  is reached to enter the pending state. This condition is not required if frame  $f+1$  is reached in the free state. As an example consider Figure 5.2. Computing value  $E^P(f+1, u, i)$  depends on the values  $E^F(f, u', i')$  and  $E^P(f, u', i')$  for the previous frame, for all combinations of  $u < 3$  and  $i \leq 3$  (dark gray field in Figure 5.2); while  $E^F(f+1, u, i)$  depends on the values  $E^F(f, u', i')$  and  $E^P(f, u', i')$  for  $u \leq 3$  and  $i \leq 3$  (light gray field in Figure 5.2). In the remainder of this chapter, we detail on the derivation of the amount of computation time that is required to move from one configuration to the other, i.e., how to compute the minimum amount of computation time that is required to get from  $E^F(f, u', i')$  or

$E^P(f, u', i')$  to  $E^F(f + 1, u, i)$  or  $E^P(f + 1, u, i)$ . Furthermore, we show how to initialize the tables for the first slot and how to compute the final WCRT when the iteration stops.

## 5.5 Analysis Methodology

To simplify the derivation of WCRT bounds, we split our analysis in two parts. Algorithm 5.1, described in Section 5.6, computes the WCRT of a single phase, based on its maximum amount of access requests  $\mu_{i,j}^{max,[a|e|r]}$  to the shared resource (for the acquisition, execution and replication phase, respectively) and its maximum amount of computation time  $exec_{i,j}^{max}$ . Algorithm 5.2, which is detailed in Section 5.7, is then used to compute the WCRT of a complete superblock and task. The start time of a phase  $t^s$  equals the completion time of the preceding phase, in case of sequential execution, or a specified starting time, in case of time-triggered execution. The first phase of the first superblock starts at time 0. Depending on the start time of a phase, the *initialization* stage of Algorithm 5.1 (lines 2-7) initializes the tables for the dynamic programming approach. The second stage, or the *analysis* stage (lines 8-21), performs the dynamic programming analysis described in Section 5.4. This stage iterates until the minimum amount of computation required to reach frame  $f' + 1$  exceeds the maximum amount of computation  $exec_{i,j}^{max}$ . The *finalization* stage (lines 22-23) then derives the WCRT of the phase, based on  $f'$  and the dynamic programming tables  $E^F$  and  $E^P$ .

### Notation used in the analysis for a phase

To simplify notations, we drop the subscripts  $i, j$  from superblock  $s_{i,j}$  and consider a phase under analysis defined by its parameters  $\{t^s, \mu^{max}, exec^{max}\}$ , where  $t^s$  is the starting time of the phase;  $\mu^{max}$  is the maximum number of accesses to the shared resource during the phase; and  $exec^{max}$  is the maximum execution time. For an acquisition or replication phase, we simply set  $exec^{max} = 0$ , and  $\mu^{max} = \mu_{i,j}^{max,a}$ , or  $\mu^{max} = \mu_{i,j}^{max,r}$ , respectively; while for a general phase, we set  $exec^{max} = exec_{i,j}^{max}$ ,  $\mu^{max} = \mu_{i,j}^{max,e}$ .

Consider  $p_j$  to be the processing element on which the phase under analysis executes. Then we define  $\psi$  to be the total number of static slots assigned to processing element  $p_j$ , i.e.,  $\psi = \sum_{0 \leq m < M_\Theta} \theta(m, p_j)$ . Since both assigned static slots and the dynamic segment count as frames of  $p_j$ , it follows that  $p_j$  has  $\psi + 1$  frames in each arbitration round. To simplify the algorithm description, we introduce some notations describing important properties of each frame. We use  $I$  to represent the ordered set of the indexes of frames assigned to  $p_j$ . Static slots have indexes from 0 to  $M_\Theta - 1$  as

detailed in Section 5.3.2, while we assign index  $M_\Theta$  to the dynamic segment, such that  $I = \{I_0, \dots, I_i, \dots, I_\psi\}$ , where  $I_\psi = M_\Theta$  represents the dynamic segment. As an example, consider Figure 5.3, where static slots 1 and 3 are assigned to processing element  $p_2$ , and the dynamic segment has index 5. Then the frame indexes of  $p_2$  are  $I = \{I_0 = 1, I_1 = 3, I_2 = 5\}$ . In other words, frames  $I_0$ ,  $I_1$  and  $I_2$  of PU2 are slots 1, 3 and 5. Finally, let  $H_{I_f}$  (with  $0 \leq f \leq \psi$ ) be the time distance between the end of frame  $I_f$  and the beginning of the following frame. Note that the end of the dynamic segment (frame  $\psi$ ) corresponds to the end of the arbitration round. Then  $H_{I_f}$  can be written as:

$$H_{I_f} = \begin{cases} \sigma_{I_{f+1}} - (\sigma_{I_f} + \delta_{I_f}) & \text{for } 0 \leq f < \psi, \\ \sigma_{I_1} & \text{otherwise.} \end{cases} \quad (5.1)$$

Our proposed algorithm computes the WCRT of a phase, by finding frame  $f'$  that can be reached by the worst-case trace. Depending on the phase under analysis, many arbitration rounds might be necessary until this frame can be found. Therefore, the amount of arbitration rounds that have been performed needs to be considered. Let  $r_0$  be the arbitration round, during which the phase is activated, i.e., the  $w$ -th round such that  $w \cdot (L_\Theta + \Delta_\Theta) \leq t^s < (w+1) \cdot (L_\Theta + \Delta_\Theta)$ , and  $r_i$  be the  $(w+i)$ -th round. Then frame 0 is the first frame of  $p_j$  in round  $r_0$  and frame  $f$  is frame  $(f \bmod (\psi+1))$  of  $p_j$  in round  $r_{\lfloor f/(\psi+1) \rfloor}$ . For notational simplicity, we set  $H_{I_f} = H_{I_{f \bmod (\psi+1)}}$ ; analogously, we translate the starting time  $t^s$  of a phase to the starting time  $\bar{t}^s$  relative to the beginning of the current round.

Interference during the dynamic segment of arbiter  $\Theta$  (e.g., all the elements in a FCFS setting, or only elements executing higher priority superblocks in a FP setting) depends on the access pattern of the other processing elements to the shared resource. We capture this pattern through a set of *arrival curves*: arrival curve  $\alpha_k(\Delta)$  represents the maximum amount of resource access requests that can be produced by tasks running on processing element  $p_k$  during a time window of length  $\Delta$ . In Chapter 3, we introduced a methodology to derive arrival curves for the different superblock models (sequential or time-triggered) that we consider in this work. We translate the  $\alpha_k(\Delta)$  representation, such that  $\alpha_k(f)$  represents the maximum amount of resource accesses by  $p_k$  in a time window of length equal to the interval between the start time of the phase under analysis and the beginning of frame  $f$ . Consider multiple interfering processing element for a phase executing on processing element  $p_j$ , e.g., a set of interferers  $IP$ , such that  $IP \subseteq \mathcal{P} \setminus p_j$ . Then we approximate the overall interference of these interferers to the phase under analysis as the sum of the individual interferences, i.e.,  $\alpha(f) = \sum_{p_k \in IP} \alpha_k(f)$ . This is a pessimistic assumption, but to the best of our knowledge, deriving tight bounds on the combined interference is still an open problem. In fact, obtaining an exact interference pattern,

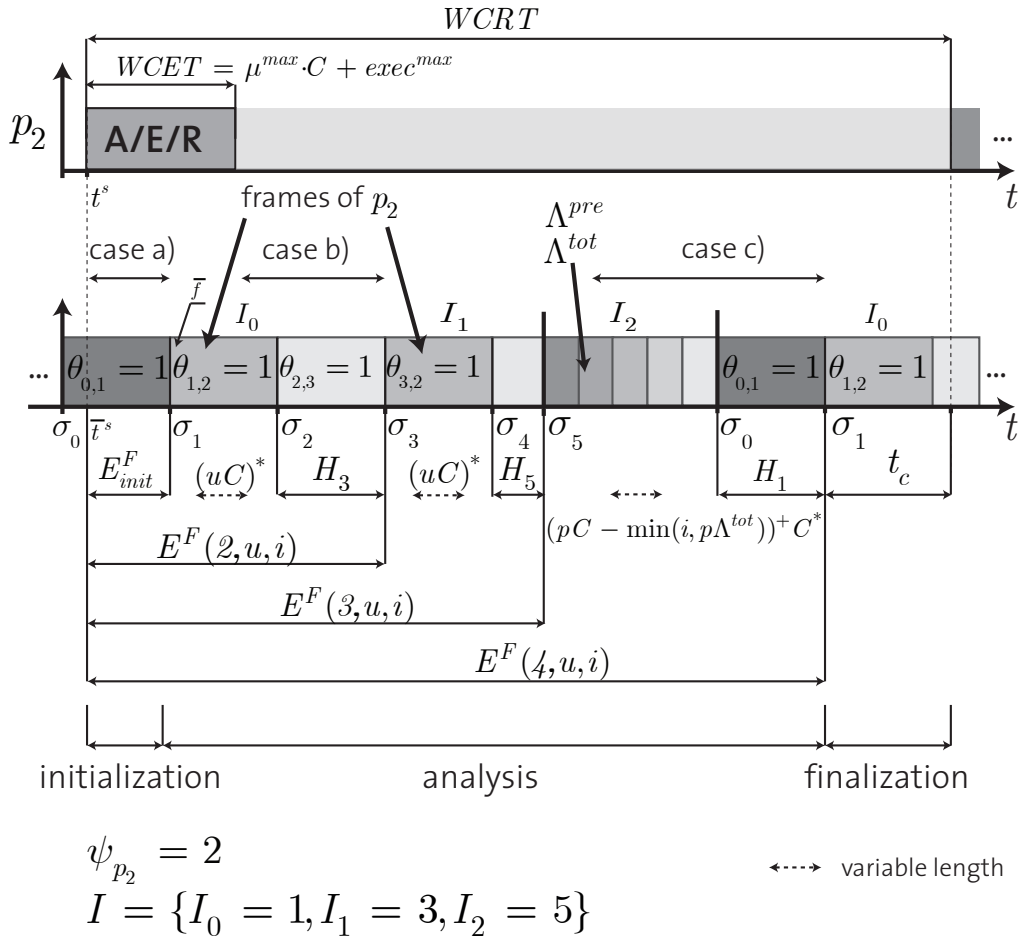


Figure 5.3: Analysis Overview

for multiple competing processing elements, onto the superblock/phase under analysis is exponential in the number of interfering phases [PSC<sup>+</sup>10], which makes it intractable in most practical settings. Finally, interference is affected by two more parameters: (1) the number of processing elements  $\Lambda^{tot}$  that can interfere with the phase under analysis and (2) the number of processing element  $\Lambda^{pre}$ , that can have pending access requests that are serviced before a pending request of  $p_j$  at the beginning of the dynamic segment, i.e., the number of interfering access requests once the dynamic segment becomes active. As we show in Section 5.6, the worst-case pattern in this situation is produced when the phase under analysis transitions to state "pending" in slot  $I_{\psi-1}$ , where slot  $I_{\psi-1}$  is its last assigned slot in the static segment. Since arbitration in the dynamic segment is performed in FCFS order,  $\Lambda^{pre}$  includes all processing elements that can interfere with the phase under analysis and are assigned at least one static slot with index smaller than  $I_{\psi-1}$ .



**Algorithm 5.1** Analyze a single phase

---

```

1: procedure ANALYZEPHASE( $t^s, exec^{max}, \mu^{max}, \Theta, IP$ )
2:    $\forall f, u, i : E^P(f, u, i) := +\infty, E^F(f, u, i) := +\infty$ 
3:    $\bar{t}^s = t^s - \lfloor \frac{t^s}{L_\Theta + \Delta_\Theta} \rfloor \cdot (L_\Theta + \Delta_\Theta)$ 
4:    $\bar{f} = \min\left((\psi + 1), \{f \mid 0 \leq f \leq \psi \wedge \bar{t}^s \leq \sigma_{I_f}\}\right)$ 
5:    $\forall u, i : \text{compute } E^F(\bar{f}, u, i), E^P(\bar{f}, u, i) \text{ by Equation (5.2) and Equation (5.3) resp.}$ 

6:    $reachable = true$ 
7:    $f = \bar{f} - 1$ 
8:   while  $reachable$  do
9:      $reachable = false$ 
10:     $f = f + 1$ 
11:    for  $\forall u \in \{0 \dots \mu_{max}\}$  and  $i \in \{0 \dots \alpha(f + 1)\}$  do
12:      if  $f \bmod \psi + 1 < \psi$  then
13:        compute  $E^P(f + 1, u, i), E^F(f + 1, u, i)$  by Equation (5.4), (5.5)
14:      else
15:        compute  $E^P(f + 1, u, i), E^F(f + 1, u, i)$  by Equation (5.6), (5.7)
16:      end if
17:      if  $\min(E^P(f + 1, u, i), E^F(f + 1, u, i)) \leq exec^{max}$  then
18:         $reachable = true$ 
19:      end if
20:    end for
21:  end while
22:  compute  $t_c$  by Equation (5.8) or (5.9) for  $f$  as static or dynamic frame resp.
23:  return  $WCRT = \lfloor \frac{f}{\psi + 1} \rfloor \cdot (L_\Theta + \Delta_\Theta) + \sigma_{I_f} + t_c - t^s$ 
24: end procedure

```

---

These interference representations are pessimistic, and therefore our derived worst-case response time (WCRT) analysis derives a safe upper bound. However, given a tight interference representation, our proposed algorithm performs a tight (WCRT) analysis.

## 5.6 Analysis for a single phase

In this section, we will introduce the three stages required to derive the WCRT for a single phase. The two data structures  $E^F(f, u, i)$  and  $E^P(f, u, i)$  represent the minimum amount of time required to reach a particular frame  $f$  such that  $u$  access requests have been served and  $i$  interferences have been suffered, for  $u \in \{0 \dots \mu_{max}\}$  and  $i \in \{0 \dots \alpha(f)\}$ . Section 5.6.1 details on the initialization phase, which depends on the starting time  $t^s$  of the phase under analysis. Section 5.6.2 introduces the analysis of a single phase, based on a dynamic programming approach. The algorithm iterates until a frame  $f'$  is reached, such that each entry in  $E^F(f' + 1, u, i)$  and  $E^P(f' + 1, u, i)$ ,  $\forall u, i$ , exceeds  $exec^{max}$ . Section 5.6.3 shows how to derive the final WCRT

for the phase under analysis, based on the previously computed data structures  $E^F$  and  $E^P$ .

### 5.6.1 Initialization Stage

The initialization stage computes the amount of computation that is required to reach the first frame  $\bar{f}$  after the activation of a phase at time  $t^s$ . Depending on this starting time, the phase is activated either (a) within the static segment of arbiter  $\Theta$  but the slot is not assigned to  $p_j$  or (b) within the static segment and within a frame (i.e., a slot assigned to  $p_j$ ) or (c) within the dynamic segment - compare cases "a)", "b)" and "c)" in Figure 5.3. Note that time windows that depend on variable parameters in Figure 5.3 are marked with an asterisk (\*) as superscript - namely time windows that depend on  $u$ ,  $i$  and  $\Lambda^{tot}$ .

First, in "case a)", the amount of time required to reach frame  $\bar{f}$  is only related to the amount of time between the activation time  $t^s$  and the start time of the next frame. Frame  $\bar{f}$  can be reached in the pending state, if an access request can be issued immediately upon the activation of the phase, i.e., no computation is performed and  $E^P(\bar{f}, 1, i) = 0, \forall i$ , see Equation (5.3), case 2.<sup>1</sup> If no access request can be issued, frame  $\bar{f}$  cannot be reached in the pending state, i.e.,  $E^P(\bar{f}, 1, i) = \infty, \forall i$ , see Equation (5.3), case 1. Frame  $\bar{f}$  can be reached in the free state by performing computation from the activation of the phase until the activation of frame  $\bar{f}$ , hence  $E^F(\bar{f}, u, i) = \sigma_{I_{\bar{f}}} - \bar{t}^s, \forall u, i$ , see Equation (5.2), case 1.

Second, "case b)", considers the case, that the current slot is a frame of  $p_j$ , i.e., the phase can immediately issue an access request that will be granted by the arbiter. Therefore,  $E^F(\bar{f}, u, i)$  is a function of the current frames remaining duration, the amount of issued access requests  $u$  and the time between the current frame  $\bar{f} - 1$  and the next frame  $\bar{f}$ , see Equation (5.2), second case.  $E^P(\bar{f}, u, i)$  is computed analogously, except that the last access request is issued  $C - \epsilon$  time units before the frame expires.<sup>2</sup> As a result, this request is not served anymore, the phase has to stall until the activation of frame  $\bar{f}$  and no computation can be performed between the current and the next slot, see Equation (5.3) - third case.

Third, "case c)", considers the phase to start in the dynamic slot. For both cases, reaching slot  $\bar{f}$  in the pending or in the free state, the required amount of computation is related to the distance of the dynamic frames activation time, the amount of access requests and the amount of interference (denoted as  $\mathbf{min}(i, u\Lambda^{tot})C$  - will be explained in more detail in Section 5.6.2). In order to reach frame  $\bar{f}$  in the pending state, computation is performed, such that the last request is issued  $C - \ell$  time units before

<sup>1</sup>If  $z > 0$ , function  $z^+$  is  $z$ ; otherwise  $z^+$  is 0

<sup>2</sup> $\epsilon$  is an arbitrary small value, greater than 0

the expiration of the dynamic segment. As a result, the request cannot be served in the dynamic segment anymore and the processing element has to stall until the activation of frame  $\bar{f}$ , i.e., the element is in pending state upon activation of frame  $\bar{f}$ , see Equation (5.3) - fourth case. In contrast to that, frame  $\bar{f}$  is reached in the free state, if computation is performed during the current frames expiration and the next frames activation, denoted  $H_{I_{\bar{f}-1}}$ , see Equation (5.2) - third case.

$$E_{init}^F(\bar{f}, u, i) = \begin{cases} \sigma_{I_{\bar{f}}} - \bar{t}^s & \text{Condition 1} \\ (\sigma_{I_{\bar{f}-1}} + \delta_{I_{\bar{f}-1}} - \bar{t}^s - u \cdot C)^+ + H_{I_{\bar{f}-1}} & \bar{f} < \psi + 1 \\ (\sigma_{I_{\bar{f}-1}} + \delta_{I_{\bar{f}-1}} - \bar{t}^s - u \cdot C - \\ \quad \mathbf{min}(i, u \cdot \Lambda^{tot}) \cdot C)^+ + H_{I_{\bar{f}-1}} & \text{otherwise} \end{cases} \quad (5.2)$$

for *Condition 1* being  $\bar{t}^s \leq \sigma_{I_0} \vee (\bar{f} > 0 \wedge \bar{t}^s \geq \sigma_{I_{\bar{f}-1}} + \delta_{I_{\bar{f}-1}})$ .

$$E_{init}^P(\bar{f}, u, i) = \begin{cases} +\infty & u = 0 \\ 0 & \text{Condition 2} \\ (\sigma_{I_{\bar{f}-1}} + \delta_{I_{\bar{f}-1}} - \bar{t}^s + \epsilon - u \cdot C)^+ & \bar{f} < \psi + 1 \\ (\sigma_{I_{\bar{f}-1}} + \delta_{I_{\bar{f}-1}} - \bar{t}^s + \ell - u \cdot C - \\ \quad \mathbf{min}(i, u \cdot \Lambda^{tot}) \cdot C)^+ & \text{otherwise} \end{cases} \quad (5.3)$$

for *Condition 2* being  $\bar{t}^s \leq \sigma_{I_0} \vee (\bar{f} > 0 \wedge \bar{t}^s \geq \sigma_{I_{\bar{f}-1}} + \delta_{I_{\bar{f}-1}})$ .

Algorithm 5.1 shows the derivation of the WCRT of a single phase. The input to the algorithm is the activation time  $t^s$ , the maximum amount of computation  $exec^{max}$ , the maximum amount of access requests  $\mu^{max}$ , the arbiter  $\Theta$  and the set of interferers  $IP$  for the phase under analysis. First the data structures  $E^P$  and  $E^F$  are initialized to  $\infty$ . Then the starting time relative to the current arbitration round  $\bar{t}^s$  and the next frame  $\bar{f}$  for the phase under analysis are computed, see Line (3) and (4). Data structures  $E^P(\bar{f}, u, i)$ ,  $E^F(\bar{f}, u, i)$  are initialized for all  $u$  and  $i$ , as described in this section, see Line (5) and the termination condition *reachable* is set to *true*.

### 5.6.2 Analysis Stage

After the initialization is done, the main loop of Algorithm 5.1 starts in Line (8). This loop computes the minimum amount of computation time that is required to reach the next frame for all combinations of  $u$  served access request,  $i$  suffered interferences for states "free" and "pending", see Line (13) in case the frame  $f$  is a static slot and Line (15) in case frame  $f$  is the dynamic segment.

The loop iterates until there is no entry in neither  $E^P$  nor  $E^F$ , that exceeds the maximum amount of computation time  $exec^{max}$  for the phase

under analysis. In other words, there exists no trace such that frame  $f + 1$  can be reached. Then *reachable* is not set to true, and the loop terminates. As a result, the last iteration of the algorithm computes values  $E^P$  and  $E^F$  for the unreachable frame  $f + 1$ . See Figure 5.1 for an example to compute  $E^F(4, u, i)$ , assuming  $E^F(5, u, i)$  cannot be reached for any combination of  $u$  and  $i$ . Therefore, the finalization phase computes the final WCRT based on the previous frame, i.e., frame  $f$ , since the WCRT of the worst-case trace must be between frames  $f$  and  $f + 1$ .

**Lemma 8** *Algorithm 5.1 terminates, for phases with finite values for parameters  $exec^{max}$  and  $\mu^{max}$ , and an arbiter with frame sizes larger than or equal to  $C$ .*

**Proof.** Every iteration of the Algorithm increments the variable  $f$ , that denotes the next frame. For finite parameters  $exec^{max}$  and  $\mu^{max}$ , and an arbitration policy that guarantees at least one access request to be served per cycle, the amount of frames (and arbitration cycles) that is required to finish this phase equals to at most the sum of the amount of access requests  $\mu^{max}$  and the maximum amount of computation  $exec^{max}$  that have to be performed. In order to reach the beginning of frame  $f + 1$  from frame  $f$ , at least one access request has to be issued  $C - \epsilon$  time units before  $f$  expires, i.e.,  $f + 1$  is reached in the pending state. Otherwise, computation amounting for  $H_{I_f}$  is performed between frame  $f$  and  $f + 1$ , i.e.,  $f + 1$  is reached in the free state.

Therefore, there exists a  $f \leq \mu^{max} + \left\lceil \frac{exec^{max}}{\min_{\forall f} \delta_{I_f} + H_{I_f}} \right\rceil$ , such that *reachable* is not set to *true* and the Algorithm 5.1 terminates. □

$E^P$  and  $E^F$  are computed differently, depending on whether the next frame  $f + 1$  is a static slot of the dynamic segment. The following two sections will detail on the differences.

### 5.6.2.1 Static Slot

The minimum amount of computation that is required to reach frame  $f + 1$ , if frame  $f$  is a static frame, is computed in Equation (5.4) and (5.5), for the free and the pending state respectively. Equation (5.4) computes the minimum amount of computation that is required to reach frame  $f + 1$ , by considering  $E^F(f, u - p, i)$  and  $E^P(f, u - p, i)$ , the duration  $\delta_{I_f}$  of frame  $f$ , the time spent serving  $p$  access requests,  $\forall 0 \leq p \leq u$ , and the time between frames  $f$  and  $f + 1$ , denoted  $H_{I_f}$ . As a result,  $E^F(f + 1, u, i)$  computes as the minimum amount of time that is required to reach frame  $f + 1$ , considering all combinations of  $u - p$  access requests served previous to frame  $f$  and  $p$

access requests served in frame  $f$ , for the "free" as well as for the "pending" state. Frame  $f + 1$  is reached in the free state, and therefore Equation (5.4) considers computation to be performed in between frame  $f$  and  $f + 1$ , i.e., by the term  $H_{I_f}$ .

$$E^F(f + 1, u, i) = \min_{0 \leq p \leq u} \begin{cases} E^F(f, u - p, i) + (\delta_{I_f} - pC)^+ + H_{I_f} \\ E^P(f, u - p, i) + (\delta_{I_f} - (p + 1)C)^+ + H_{I_f} \end{cases} \quad (5.4)$$

$$E^P(f + 1, u, i) = \min_{1 \leq p \leq u} \begin{cases} E^F(f, u - p, i) + (\delta_{I_f} + \epsilon - pC)^+ \\ E^P(f, u - p, i) + (\delta_{I_f} + \epsilon - (p + 1)C)^+ \end{cases} \quad (5.5)$$

Analogously, Equation (5.5) computes how to reach frame  $f + 1$  in the pending state. An access request is issued  $C - \epsilon$  units of time before frame  $f$  expires, resulting in this request not being served during frame  $f$  anymore. This way, the stall time of the element is maximized, and thus the amount of computation that is performed during frame  $f$  (at most  $\delta_{I_f} + \epsilon$ ) and between frame  $f$  and  $f + 1$  (equals 0 since  $H_{I_f}$  is neglected) is minimized. Issuing the access request at any later point results in an increased amount of computation that has to be performed, and thus not in the worst-case trace.

Note that in Equation (5.4) and (5.5), the amount of served access requests during frame  $f$  is increased by 1 for the pending states. This is due to the fact that upon the activation of frame  $f$ , an access request will be issued immediately. Furthermore, note that the variable  $p$  for Equation (5.5) starts at 1, since at least a single access request must remain unserved, otherwise frame  $f + 1$  cannot be reached in the pending state.

**Lemma 9** *If frame  $f$  is a static slot, Equation (5.4) and Equation (5.5) compute the minimum amount of computation that have to be performed, such that frame  $f + 1$  can be reached in the free and pending state, respectively, for  $u$  served access requests and  $i$  interfering access requests by other phases.*

**Proof.** For a particular frame  $f + 1$  and number of served access requests  $u$ , Equation (5.4) and Equation (5.5) compute the required amount of computation for all possible values of already served access requests  $u - p$  (before frame  $f$ ) and access requests  $p$  served in frame  $f$ , based on the pending and the free state. The minimum among these values, based on both the pending and the free state, is returned, and therefore Equation (5.4) and (5.5) result in the minimum amount of computation time that is required to reach frame  $f + 1$  in the free state and pending state respectively.

□

### 5.6.2.2 Dynamic Slot

In case frame  $f$  is a dynamic slot, interfering access requests have to be considered. The minimum amount of computation that is required to reach frame  $f + 1$ , is computed in Equation (5.6) and (5.7), for the free and the pending state respectively. The respective amount of computation is derived for all possible remaining access requests  $p$  and remaining interfering access requests  $l$ . The minimum of these values is considered for further computation and stored in  $E^P$  and  $E^F$ , see Lemma 10.

$$E^F(f + 1, u, i) = \min_{\substack{0 \leq p \leq u \\ 0 \leq l \leq i}} \begin{cases} E^F(f, u - p, i - l) + (\Delta_\Theta - pC - \\ \quad \mathbf{min}(l, p\Lambda^{tot})C)^+ + H_{I_f} \\ E^P(f, u - p, i - l) + (\Delta_\Theta - (p + 1)C - \\ \quad \mathbf{min}(l, p\Lambda^{tot} + \Lambda^{pre})C)^+ + H_{I_f} \end{cases} \quad (5.6)$$

$$E^P(f + 1, u, i) = \min_{\substack{1 \leq p \leq u \\ 0 \leq l \leq i}} \begin{cases} E^F(f, u - p, i - l) + (\Delta_\Theta + l - pC - \\ \quad \mathbf{min}(l, p\Lambda^{tot})C)^+ \\ E^P(f, u - p, i - l) + (\Delta_\Theta + l - (p + 1)C - \\ \quad \mathbf{min}(l, p\Lambda^{tot} + \Lambda^{pre})C)^+ \end{cases} \quad (5.7)$$

Equation (5.6) computes the amount of computation that is required to reach frame  $f + 1$  in the free state by considering the amount of computation required to reach the current frame  $f$ , the length of the dynamic segment  $\Delta_\Theta$ , the amount of served access requests  $p$  (or  $p + 1$  for the pending case), the amount of access requests that can be interfered with, and the time between the expiration of the current and the activation of the next frame  $H_{I_f}$ . The term  $\mathbf{min}(l, p\Lambda^{tot})C^+$  in Equation (5.6), for the free state, constrains the maximum amount of interference a phase can suffer during frame  $f$ . Intuitively,  $p\Lambda^{tot}$  represents the property that each access request issued in frame  $f$  can be interfered by every other processing element once, i.e., each access request ends up in the last position of the FIFO queue of the arbiter during the dynamic segment. On the other hand, in case there is less interference, i.e.,  $l$ , then the access requests issued during frame  $f$  cannot suffer more interference. For  $E^F$  based on the pending state, this term changes to  $\mathbf{min}(l, p\Lambda^{tot} + \Lambda^{pre})C^+$ . In other words, upon activation of frame  $f$  there is a pending access request that has to be served and it is interfered by up to  $\Lambda^{pre}$  access requests.

Equation (5.7) computes the amount of computation that is required to reach frame  $f + 1$  in the pending state, similarly to Equation (5.6). However, frame  $f + 1$  shall be reached in the pending state, and therefore, the term  $H_{I_f}$  is not present in Equation (5.7). Instead, the amount of computation during the active frame is increased, such that the last access request cannot

be served anymore. That is, in comparison to Equation (5.6), the amount of computation is increased by one minislot of size  $\ell$ . The last access request cannot be served anymore and the phase stalls until frame  $f + 1$  is activated. As a result, the last access request remains pending and the phase stalls in between the end of the current and the activation of the next frame. If  $p = 0$ ,  $E^P(k, u, i) = \infty$ , i.e., if no access request is issued, the next frame cannot be reached in the pending state.

**Lemma 10** *If frame  $f$  is a dynamic slot, Equation (5.6) and (5.7) compute the minimum amount of computation that have to be performed, such that frame  $f + 1$  can be reached in the free and pending state respectively, for  $u$  served access requests and  $i$  interfering access requests by other phases.*

**Proof.** Assume frame  $f + 1$  should be reached, with  $u$  access requests being served and  $i$  interfering access requests being issued by other processing elements. Then Equation (5.6) and (5.7) compute the amount of computation that is required to reach frame  $f + 1$  for all possible combinations of access request, that are already served at frame  $f$ , denoted  $u - p$ , and access requests that have to be served during frame  $f$ , denoted  $p$ . Similarly, the interferences that have already happened, denoted  $i - l$ , and those that can happen during frame  $f$ , denoted  $l$ , are considered. The rest of the proof is similar to the proof for Lemma 9. □

### 5.6.3 Finalization

Algorithm 5.1 iterates until frame  $f + 1$ , that cannot be reached anymore, is found, i.e., frame  $f + 1$  such that variable *reachable* remains *false* for any  $u, i$ , see Line 17. Then the worst-case completion of the phase under analysis is between frame  $f$  and  $f + 1$ . As a result, the data structures  $E^F(f, u, i)$  and  $E^P(f, u, i)$ ,  $\forall u, i$  are considered to compute the final WCRT.

Equation (5.8) and Equation (5.9) compute the WCRT of a phase, in case frame  $f$  is a static or a dynamic frame, respectively.

$$t_c = \max_{\forall u, i} \begin{cases} (exec^{max} - E^F(f, u, i) + (\mu^{max} - u)C)^+ \\ (exec^{max} - E^P(f, u, i) + (\mu^{max} - u + 1)C)^+ \end{cases} \quad (5.8)$$

$$t_c = \max_{\forall u, i} \begin{cases} (exec^{max} - E^F(f, u, i) + (\mu^{max} - u + \\ \min(\alpha(f + 1) - i, (\mu^{max} - u)\Lambda^{tot}))C)^+ \\ (exec^{max} - E^P(f, u, i) + (\mu^{max} - u + 1 + \\ \min(\alpha(f + 1) - i, (\mu^{max} - u)\Lambda^{tot} + \Lambda^{pre}))C)^+ \end{cases} \quad (5.9)$$

Consider frame  $f$  to be a static frame, then Equation (5.8) computes the response time  $t_c$  of a phase by considering the remaining access requests

$(\mu^{max} - u)$  and the phases computation time ( $exec^{max}$ ). Their difference represents the remaining amount of computation that has to be performed, but proved to be insufficient to reach frame  $f + 1$ .

Similarly, Equation (5.9) computes the response time  $t_c$  of a phase, if frame  $f$  is the dynamic segment. In this case, the interference that might be suffered during that frame has to be considered. As in Equation (5.6) and (5.7), the additional delay due to inference depends on the upper bound of interference ( $\alpha(f + 1) - i$ ), and the number of issued access request that can be interfered with:  $(\mu^{max} - u)\Lambda^{tot}$  in the free state, and  $(\mu^{max} - u)\Lambda^{tot} + \Lambda^{pre}$  in the pending state. The WCRT  $t_c$  is the maximum resulting value for any  $u$  and  $i$ .

**Lemma 11** *Given a constrained adaptive arbiter  $\Theta$ , Algorithm 5.1 computes an upper bound of the worst-case response time for a phase with a defined starting time  $t^s$ , amount of access requests  $\mu_{max}$ , computation time  $exec^{max}$ , and the set of interferers  $IP$ .*

**Proof.**  $E^F(f + 1, u, i)$  and  $E^P(f + 1, u, i)$  are computed as the minimum of  $E^F(f, u - p, i - l)$  and  $E^P(f, u - p, i - l)$ , for  $p$  and  $l$  as defined in Equation (5.4) to (5.7). Instead of considering the minimum value for  $E^F(f + 1, u, i)$  and  $E^P(f + 1, u, i)$ , consider all values that are computed in Equation (5.4) to (5.7). In other words, at each iteration, all possible distributions of (a) already served access requests  $u - p$  and remaining access requests  $u$ , and (b) suffered interferences  $i - l$  and interferences that can be suffered in the current frame  $i$ , and their respective required computation time are stored in a data structure, not only their minimum. Consider  $E_n^F(f + 1, u, i)$  and  $E_n^P(f + 1, u, i)$  to be the amount of computation that is required to reach frame  $f + 1$ , for any distribution of  $u$  and  $i$ , but the one that results as the minimum. In other words,  $\forall n : E_n^F(f + 1, u, i) > E^F(f + 1, u, i)$  and  $\forall n : E_n^P(f + 1, u, i) > E^P(f + 1, u, i)$ . Then the amount of computation that remains after frame  $f + 1$  is reduced, or  $exec^{max} - E_n^F(f + 1, u, i) < exec^{max} - E^F(f + 1, u, i)$  and  $exec^{max} - E_n^P(f + 1, u, i) < exec^{max} - E^P(f + 1, u, i)$ . Since  $E_n^F(f + 1, u, i)$  computes as the sum of  $E_n^F(f, u, i)$  and a term related to the arbitration in frame  $f$ ,  $E_n^F(f + 1, u, i) > E^F(f + 1, u, i)$ . Similarly,  $E_n^P(f + 1, u, i) > E^P(f + 1, u, i)$ .

Eventually the remaining computation time is insufficient to reach the next frame  $f + 1$  and the final response time is computed in Equation (5.8) and (5.9). Since  $E_n^F(f, u, i) > E^F(f, u, i)$  and  $E_n^P(f, u, i) > E^P(f, u, i)$ , the resulting response time  $t_c$  is reduced and so is the WCRT.

Conclusively, considering any other amount of computation to reach frame  $f + 1$ , than the minimum computed in Equation (5.4) to (5.7), leads to a decreased response time, which is upper-bounded by the solution derived from Algorithm 5.1.

□



### 5.6.4 Complexity

The complexity of the proposed methodology depends on the number of access requests and interference. Consider Algorithm 5.1, Line 15, then it is clear, that Equation (5.6) and (5.7) are computed  $f$  times. The complexity of computing Equation (5.6) and Equation (5.7) is  $O((\mu^{max})^2 \alpha(f)^2)$ , since for every combination of  $u$  and  $i$ , the minimum amount of computation has to be computed for all possible remaining access requests  $p$  and interferences  $l$ .

Furthermore,  $f$  is limited by a bound in Lemma 8. As a result, the complexity of the proposed approach is  $O\left((\mu^{max})^3 \alpha(f)^2 + \left\lfloor \frac{exec^{max}}{\min(\delta_{I_f} + H_{I_f})} \right\rfloor\right)$ , i.e., pseudo-polynomial.

## 5.7 Analysis for Superblocks and Tasks

In the previous section we showed how to derive the WCRT of a phase. This section describes how to compute the WCRT of a task, which is composed as a sequence of superblocks. Superblocks are composed of sequences of phases, or are represented by a single phase, depending on the resource access model, see Figure 5.1.

The analysis is performed for each phase, while the worst-case starting time of a subsequent phase is the completion time of its preceding phase (in the subsequent execution model), or a dedicated starting time (in the time-triggered execution model). The starting time of the first phase in superblock  $s_{i,j}$  is the worst-case completion time of its preceding superblock  $s_{i-1,j}$ , or a dedicated starting time in the subsequent and time-triggered execution model, respectively. For the first phase of the first superblock, the starting time is 0. If the deadline  $d_{i,j}$  of a superblock  $s_{i,j}$  is violated, then the task is unschedulable with this arbitration policy.

Consider  $t_{i,j}$  to be the starting time of superblock  $s_{i,j}$  (e.g., in the time-triggered execution mode), then without the need to consider any previous or subsequent superblock, the analysis can be performed accordingly. In case the resulting completion time exceeds deadline  $d_{i,j}$ , the task is unschedulable. The time-triggered models can be easily analyzed by replacing Line 8 in Algorithm 5.2. Instead of setting time  $t$  to the WCRT of the current phase, time  $t$  has to be set to the next phases dedicated starting time.

The pseudo-code for analyzing a task under the sequential model is given in Algorithm 5.2. Consider the set of superblocks  $\mathcal{S}_j$  statically scheduled on processing element  $p_j$  and the set of interfering elements to be in  $IP$ . Furthermore, we assume that the length of an arbitration round  $\Delta_\Theta + L_\Theta$  and the period  $W_j$  of the set of superblocks  $\mathcal{S}_j$  under analysis to be integer

multiples. Otherwise, Algorithm 5.2 would have to iterate over all possible offsets, i.e., the *lcm* of  $\Delta_\Theta + L_\Theta$  and  $W_j$ .

---

**Algorithm 5.2** Analyze Task
 

---

```

1: procedure ANALYZETASK( $\mathcal{S}_j, IP$ )
2:   schedulable = true;
3:    $t = 0$ ;
4:   for each  $s_{i,j} \in \mathcal{S}_j$  do
5:     for each phase in  $s_{i,j}$  do
6:        $exec^{max}$  max. amount of computation
7:        $\mu^{max}$  max. amount of resource accesses
8:        $t = t + \text{AnalyzePhase}(t, exec^{max}, \mu^{max}, \Theta, IP)$ ;
9:     end for
10:    if  $t > d_{i,j}$  then
11:      schedulable = false;
12:    end if
13:  end for
14:  return schedulable,  $t$ 
15: end procedure

```

---

**Theorem 4** *Algorithm 5.2 computes the worst-case response time (WCRT) for a set of superblocks  $\mathcal{S}_j$  executing sequentially on a processing element  $p_j$ .*

**Proof.** The proof is based on Lemma 11 and the assumption of a fully timing compositional architecture, i.e., no timing anomalies. Activating a phase at time  $t_1$  and  $t_2$  results in  $WCRT_1 \leq WCRT_2$  for  $t_1 \leq t_2$ . Therefore a sequence of phases can be analyzed by setting the start time of a phase as its predecessors completion time, or 0 if it is the first phase.

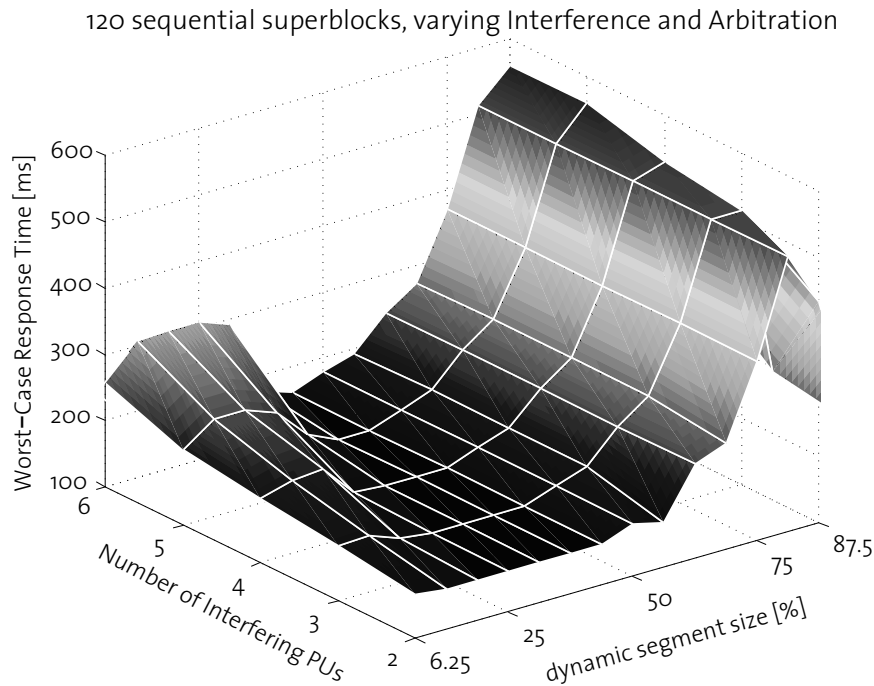
□

## 5.8 Simulations

We present experimental results based on a real-world application and on a generated application. The applications are composed of a set of subsequent superblocks, that are modeled according to the general model. The industrial application, from the automotive domain, is composed of 120 superblocks<sup>3</sup>. The interference on the shared resources is modeled as an arrival curve  $\alpha_j$  for each interfering element, and assumed with a constant slope. The generated application is composed of 209 superblocks, using parameters extracted from the industrial application. It issues 230 access requests and performs computation amounting for about 9 ms. The time required to serve an access request, once access to the resource is granted, is 0.5ms,

---

<sup>3</sup>Due to confidentiality agreements, actual data sample cannot be disclosed.

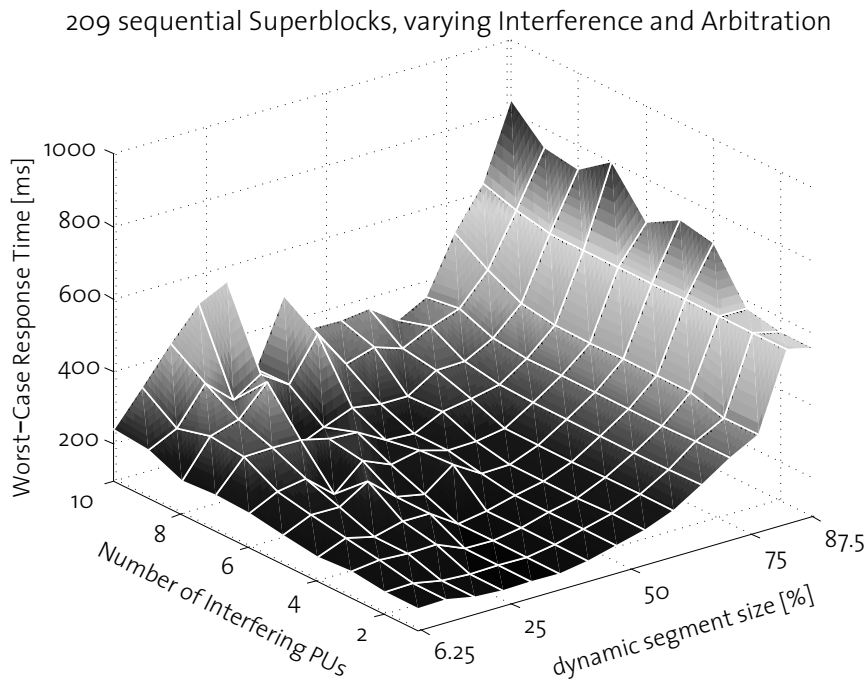


**Figure 5.4:** Results for an automotive application.

for both applications. The configuration of the arbiter is varied, such that 6.25% to 87.5% of an arbitration cycle belongs to the dynamic segment, and conclusively 93.75% to 12.5% belongs to the static segment, respectively. In the static segment, one slot is assigned to the application under analysis, while the remainder of the static segment is assigned to other elements, and thus unavailable. This slot amounts for 62.5% of the static segment length. Varying sizes of the dynamic segment affect the size of the static segment, since a larger dynamic segment implies a smaller static segment.

In Figure 5.4, an automotive application is analyzed. For an arbiter with only a small dynamic segment (6.25%), the WCRT increases from 166ms, for 2 interfering processing elements, to 260ms, for a system with 6 interfering elements (which equals an increase of  $260/166 - 1 = 56.6\%$ ). With increasing share of the dynamic segment in the arbiter, the WCRT rises for systems with a high number of interfering elements. A dynamic segment, amounting for 25% of the arbitration cycle results in a WCRT of 161ms for two interfering elements, but in a WCRT of 309ms for a system with 6 interfering elements. This equals an increase of  $309/161 - 1 = 91.9\%$ .

Further expansion of the dynamic segment, for a large amount of interferers, at first results in a better performance, i.e., the WCRT decreases again. At some point, this effect reverses, and the performance suffers. For the application under consideration, a dynamic segment amounting for 25%



**Figure 5.5:** Results for a generated application.

to 50% of the arbitration cycle results in the lowest WRCT, i.e., the best performance. For larger dynamic segments, the performance suffers from the diminishing share of the static slot. The smaller the static slot, i.e., the guaranteed service, the larger the WCRT, even for systems with only few interfering elements. In Figure 5.4, the increase of the WCRT for dynamic segments of 50% to 75% is apparent. For a system with only two interfering elements the WCRT increases from 161ms to 514ms, respectively (equals an increase of  $514/161 - 1 = 219\%$ .)

Further expansion of the dynamic segment resulted in another reduction of the WCRT. This is a result of (a) only little interference on the shared resource due to the small amount of interfering elements, and (b) the reduced influence of the blocking time in the static segment, between the end of the static slot (aka. frame) assigned to the application under analysis and the activation of the dynamic segment. With the static segment getting smaller and smaller, also this blocking time gets less significant.

Similarly to the automotive application, the generated application in Figure 5.5 illustrates the same effects. An increasing amount of interference also results in an increased WCRT, but the effect is reduced compared to Figure 5.4. The amount of interference that is suffered during the dynamic segment depends on the number of interfering elements. In our experiments,

we assumed the overall amount of interference to be the same as for the automotive application.

Experimental results show that the problem is non-convex and an accurate WCRT is very hard to estimate. Many parameters, such as the size of the respective segments of the arbiter, the amount and pattern of interference and the number of interfering elements have an impact on the WCRT of an application. For that reason, an efficient and tight analysis is even more essential, since small changes in the design might lead to a significant increase of the WCRT.

## 5.9 Chapter Summary

In this chapter, we present an analysis approach to derive a safe (upper) bound of the worst-case response time (WCRT) for tasks in a resource sharing multiprocessor system, with hybrid arbitration. Tasks are composed of superblocks and phases, and are statically scheduled on the processing elements. Arbitration on the shared resource is based on an adaptive protocol, e.g., FlexRay protocol, where a static segment with fixed assigned time slots (TDMA) is succeeded by a dynamic segment with FCFS arbitration. Therefore interference due to contention on the share resource has to be considered for computing the WCRT of a task. Access to the shared resource is non-preemptive and synchronous. Hence, once a resource access is issued, the corresponding task blocks until this resource access has been served.

We propose a computationally efficient analysis approach that accounts for interference due to contention on the shared resource. Interference is represented as an arrival curve, as outlined in Chapter 3. Using a dynamic programming approach, we derive a tight WCRT, for a given tight interference representation. The presented approach can seamlessly be applied to the different resource access models. Experimental results show that the problem cannot be modeled as a convex optimization problem and that minor deviations in the size of the dynamic segment result in significantly increased WCRT (up to 219% in our experiments). With the increasing importance of multiprocessor/multi-core systems in the area of real-time computing, accurate and efficient analysis approach are of crucial importance to designers. The approach presented in this chapter, provides such an approach for an arbitration policy that is widely used in the avionic, automotive and automation industry.



# 6

## Conclusion

This chapter summarizes the contributions presented in this thesis and closes with possible future research directions.

### 6.1 Contribution

In this thesis, we study the effect of resource sharing on the timing predictability of systems. On the one hand, computational resources are a shared resources. Multiple concurrent applications executing on a single-core or multicore processor share the computational resources of this processor. We propose an approach to distribute concurrently executing applications on a heterogeneous MPSoC platform. Our approach satisfies utilization constraints and minimizes the expected power consumption.

On the other hand, communication fabrics, such as buses and main memory, represent a shared resource. Contention on the shared resource has become the bottleneck for performance increase that can be achieved by parallelization. We propose different models to access a shared resource, and provide a worst-case response time (WCRT) analysis framework for static and adaptive arbitration policies on the shared resource. Furthermore, we give design guidelines for resource sharing systems.

#### 6.1.1 Adaptive power-aware multiprocessor design

In Chapter 2, we propose a task model to specify multiple concurrently executing multi-mode applications. An execution probability is associated to

each mode. A *scenario* represents a set of concurrently executing applications in their respective modes, i.e., a global system mode. The hardware platform is given as a library of available processing element types. Each type is defined by its dynamic and static (leakage) power consumption. We propose an algorithm to optimize the mapping of tasks to processing elements.

The problem is formulated as an integer linear program (ILP) and we present a heuristic to efficiently solve it. The proposed methodology results in a mapping that satisfies utilization constraints, while minimizing the expected average power consumption. We propose an adaptive approach, considering different execution probabilities. A set of template mappings is computed offline and stored on the system, where a runtime manager observes scenario transitions and chooses the appropriate template mappings.

### 6.1.2 Task models and interference in resource sharing systems

Executing multiple applications on a multicore or multiprocessor platform results in increased communication among applications and tasks on different processing elements. In multicore platforms, communication often involves shared resources, such as shared memories and buses. As a result, when accessing a shared resource, an application has to wait until the arbiter grants access. The waiting time depends on the arbitration policy and is not a property of the application itself.

In Chapter 3 we present an application model, that specifies periodic tasks as sets of superblocks. Superblocks are specified by their upper and lower bound on execution time and maximum and minimum number of access requests to the shared resource. We propose different models to access the shared resource and to execute superblocks on processing elements. Arrival curves are used as a metric to represent the access pattern of a set of superblocks onto the shared resource. We detail on how to derive this arrival curve representation for the proposed resource access models.

### 6.1.3 Towards timing predictability in resource sharing systems

Accessing shared resources results in delays due to contention. Therefore, in order to guarantee real-time properties on multicore and multiprocessor platforms, appropriate worst-case response time (WCRT) analysis tools are required. In Chapter 4, we propose the time division multiple access (TDMA) protocol as resource arbiter on the shared resource. In this case interference by other processing element in the system can be omitted. In other words, the WCRT of tasks and superblocks on a processing element can be computed in isolation. We present an efficient approach to derive the WCRT, for the resource access models proposed in Chapter 3.



Furthermore, we show that the resource access model that separates communication and computation is the model of choice for resource sharing real-time systems. We present the relation of the proposed resource access models, with respect to their WCRT. The dedicated model with sequentially executing superblocks (DSS) is shown to be the resource access model that is schedule as soon as any of the proposed models is schedulable. In other words, model DSS is the model with the lowest WCRT.

In Chapter 5, we consider systems with the FlexRay arbitration protocol on the shared resource. In FlexRay, the arbitration cycle is constituted by a static and a dynamic segment. In the static segment, static time slots are assigned to processing elements. In the dynamic segment, access to the shared resource is granted according to the First-Come-First-Serve (FCFS) policy. Competing for the resource in the dynamic segment allows for adaptivity. Since previous results on dynamic arbitration [PSC<sup>+</sup>10] and on static arbitration in Chapter 4 cannot be applied, we propose an algorithm based on dynamic programming to derive the WCRT. We show that minor changes in the dynamic segment of the arbiter might result in large deviations in the resulting WCRT.

## 6.2 Outlook

The research presented in this thesis stimulated research in several directions.

### 6.2.1 Reducing Interference or Controlled Interference

In Chapter 3, we propose arrival curves to represent the resource access pattern of a set of superblocks onto the shared resource. Multiple processing element are represented by the sum of their constituting arrival curves. In the WCRT analyses, we use these arrival curves to represent the interference that the processing element under analysis might suffer from the other processing elements in the system. However, this is a very pessimistic estimation of the overall interference.

The correlation between the multiple processing elements is disregarded. Consider a system with  $n$  processing elements accessing a shared resource. Then each of the  $n$  elements suffers from interference, and hence the actual access pattern of these processing elements onto the shared resource changes. As a result, the actual interference a superblock might suffer is smaller than the sum of the arrival curves of all the other processing elements. Reducing the overestimation of interference would result in significantly reduced WCRTs. As a result, more applications could be executed on a platform, while real-time properties could still be guaranteed.

To that end, increasing the amount of structure in applications, e.g., by following the dedicated model, represents one possibility. Another possibility is the usage of exhaustive methods to compute the WCRT, e.g., timed automata. Although, these approaches suffer from scalability issues, abstraction of system properties might lead to feasible solutions.

### 6.2.2 Massive multicore systems

Static and hybrid arbitration is shown to be a feasible way of designing resource sharing real-time systems. These approaches are shown to perform well, for the typically anticipated industrial embedded system with up to 10 processing elements. However, technology advances and future multicore or many-core systems might have a thousand and more cores. For such systems, static arbitration seems not practicable any more, since slot sizes will get very small, and arbitration cycles very large. On the other hand, deriving WCRTs for dynamic arbitration policies with hundreds or possibly thousands of interferers represents a major obstacle. While trivial bounds to compute the interference (e.g., each access request is interfered by all elements in the systems) are of little use, tighter bounds require computationally expensive techniques.

One way of handling the problem could be to introduce phases of controlled interference. In other words, clusters of processors are assigned a specific time slot to access a shared resource. During this time slot the processors in the cluster might interfere each other. After the time slot for a specific cluster has expired the corresponding processors continue with local computation until their next time slot gets activated. This method requires additional coordination on processor cluster level, but might make real-time systems on massive multicore systems feasible in the first place.

### 6.2.3 Multiple shared resources

In this thesis, we considered a single shared resource only. However, a multicore platform might have multiple shared resource. As an example, consider a system with a shared memory for data and a shared instruction flash. In such systems, each instruction needs to be fetched from the shared instruction flash before it can be executed. As a result, domino effects can be experienced, where contention on one shared resource results in additional contention on the other. Additionally, even reduced interference on one shared resource, e.g., due to faster processing, could result in additional interference on the other resource. Again, computational expensive approaches, like exhaustive search, are able to derive the WCRT for small problem instances. However, deriving efficient approaches to compute tight WCRTs is still an open problem.

# References

- [AE10] B. Andersson and A. Easwaran. Provably good multiprocessor scheduling with resource sharing. *Real-Time Systems*, 46:153–159, October 2010.
- [AEA05] T.A. Al Enawy and H. Aydin. Energy-aware task allocation for rate monotonic scheduling. In *Real Time and Embedded Technology and Applications Symposium, 2005. RTAS 2005. 11th IEEE*, pages 213 – 223, 2005.
- [AE10] B. Andersson, A. Easwaran, and J. Lee. Finding an upper bound on the increase in execution time due to contention on the memory bus in cots-based multicore systems. *SIGBED Rev.*, 7:4:1–4:4, January 2010.
- [AEPR08] A. Andrei, P. Eles, Z. Peng, and J. Rosen. Predictable implementation of real-time applications on multiprocessor systems-on-chip. In *Proceedings of the 21st International Conference on VLSI Design, VLSID '08*, pages 103–110, Washington, DC, USA, 2008. IEEE Computer Society.
- [ANG<sup>+</sup>06] R. Atitallah, S. Niar, A. Greiner, S. Meftali, and J. Dekeyser. Estimating energy consumption for an MPSoC architectural exploration. In Werner Grass, Bernhard Sick, and Klaus Waldschmidt, editors, *Architecture of Computing Systems - ARCS 2006*, volume 3894 of *Lecture Notes in Computer Science*, pages 298–310. Springer Berlin / Heidelberg, 2006.
- [AP07] T.W. Ainsworth and T.M. Pinkston. On characterizing performance of the cell broadband engine element interconnect bus. In *Networks-on-Chip, 2007. NOCS 2007. First International Symposium on*, pages 18 –29, May 2007.
- [Aut] AutoSAR. Release 4.0, <http://www.autosar.org>.
- [AY03] H. Aydin and Q. Yang. Energy-aware partitioning for multiprocessor real-time systems. In *Proceedings, International Parallel and Distributed Processing Symposium, 2003.*, page 9 pp., 2003.

- [BBMo8] L. Benini, D. Bertozzi, and M. Milano. Resource management policy handling multiple use-cases in MPSoC platforms using constraint programming. In *Logic Programming*, volume 5366 of *Lecture Notes in Computer Science*, pages 470–484. Springer Berlin / Heidelberg, 2008.
- [BCB<sup>+</sup>08] B.B. Brandenburg, J.M. Calandrino, A. Block, H. Leontyev, and J.H. Anderson. Real-time synchronization on multiprocessors: To block or not to block, to suspend or spin? In *Real-Time and Embedded Technology and Applications Symposium, 2008. RTAS '08. IEEE*, pages 342–353, 2008.
- [BRE<sup>+</sup>10] P. Burgio, M. Ruggiero, F. Esposito, M. Marinoni, G. Buttazzo, and L. Benini. Adaptive TDMA bus allocation and elastic scheduling: a unified approach for enhancing robustness in multi-core RT systems. In *Proceedings of the 28th IEEE International Conference on Computer Design, ICCD '10*, Washington, DC, USA, October 2010. IEEE Computer Society.
- [CB10] D. B. Chokshi and P. Bhaduri. Performance analysis of FlexRay-based systems using real-time calculus, revisited. In *Proceedings of the 2010 ACM Symposium on Applied Computing, SAC '10*, pages 351–356, New York, NY, USA, 2010. ACM.
- [CF10] G. Carvajal and S. Fischmeister. A TDMA ethernet switch for dynamic real-time communication. In *Field-Programmable Custom Computing Machines (FCCM), 2010 18th IEEE Annual International Symposium on*, pages 119–126, May 2010.
- [Cheo8] J.-J. Chen. Expected energy consumption minimization in DVS systems with discrete frequencies. In *Proceedings of the 2008 ACM symposium on Applied computing, SAC '08*, pages 1720–1725, New York, NY, USA, 2008. ACM.
- [CHKo6] J.-J. Chen, H.-R. Hsu, and T.-W. Kuo. Leakage-aware energy-efficient scheduling of real-time tasks in multiprocessor systems. In *Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 408–417, Washington, DC, USA, 2006. IEEE Computer Society.
- [CKo6] J.-J. Chen and T.-W. Kuo. Allocation cost minimization for periodic hard real-time tasks in energy-constrained DVS systems. In *Proceedings of the 2006 IEEE/ACM International Conference on Computer-aided Design, ICCAD '06*, pages 255–260, New York, NY, USA, 2006. ACM.

- [CKO7a] J.-J. Chen and C.-F. Kuo. Energy-efficient scheduling for real-time systems on dynamic voltage scaling (DVS) platforms. In *Proceedings of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA '07*, pages 28–38, Washington, DC, USA, 2007. IEEE Computer Society.
- [CKO7b] J.-J. Chen and C.-F. Kuo. Energy-efficient scheduling for real-time systems on dynamic voltage scaling (DVS) platforms. In *Embedded and Real-Time Computing Systems and Applications, 2007. RTCSA 2007. 13th IEEE International Conference on*, pages 28–38, 2007.
- [CKT03a] S. Chakraborty, S. Künzli, and L. Thiele. A general framework for analysing system properties in platform-based embedded system designs. In *Proceedings of the conference on Design, Automation and Test in Europe - Volume 1, DATE '03*, pages 10190–, Washington, DC, USA, 2003. IEEE Computer Society.
- [CKT03b] Samarjit Chakraborty, Simon Künzli, and Lothar Thiele. A General Framework for Analyzing System Properties in Platform-Based Embedded System Design. In *Proc. Design, Automation and Test in Europe (DATE)*, pages 190–195, Munich, Germany, March 2003.
- [CMCM09] E. Carvalho, C. Marcon, N. Calazans, and F. Moraes. Evaluation of static and dynamic task mapping algorithms in NoC-based MPSoCs. In *Proceedings of the 11th international conference on System-on-chip, SOC'09*, pages 87–90, Piscataway, NJ, USA, 2009. IEEE Press.
- [CRG08] A. K. Coskun, T. S. Rosing, and . C. Gross. Proactive temperature management in MPSoCs. In *Proceeding of the 13th international symposium on Low power electronics and design, ISLPED '08*, pages 165–170, New York, NY, USA, 2008. ACM.
- [CST09] J.-J. Chen, A. Schranzhofer, and L. Thiele. Energy minimization for periodic real-time tasks on heterogeneous processing units. In *Proceedings of the 2009 IEEE International Symposium on Parallel & Distributed Processing*, pages 1–12, Washington, DC, USA, 2009. IEEE Computer Society.
- [CSZ<sup>+</sup>05] Y. Chen, Z. Shao, Q. Zhuge, C. Xue, B. Xiao, and E. H. M. Sha. Minimizing energy via loop scheduling and DVS for multi-core embedded systems. In *Proceedings of the 11th International Con-*

- ference on Parallel and Distributed Systems - Workshops - Volume 02*, ICPADS '05, pages 2–6, Washington, DC, USA, 2005. IEEE Computer Society.
- [CWSCo8] P.-C. Chang, I.-W. Wu, J.-J. Shann, and C.-P. Chung. ETAHM: an energy-aware task allocation algorithm for heterogeneous multiprocessor. In *Proceedings of the 45th annual Design Automation Conference, DAC '08*, pages 776–779, New York, NY, USA, 2008. ACM.
- [EKL<sup>+</sup>09] S.A. Edwards, Sungjun Kim, E.A. Lee, I. Liu, H.D. Patel, and M. Schoeberl. A disruptive computer design idea: Architectures with repeatable timing. In *Computer Design, 2009. ICCD 2009. IEEE International Conference on*, pages 54–59, 2009.
- [FBZ10] A. Fedorova, S. Blagodurov, and S. Zhuravlev. Managing contention for shared resources on multicore processors. *Commun. ACM*, 53:49–57, February 2010.
- [Fle] FlexRay. <http://www.flexray.com/>.
- [fS] ITRS International Roadmap for Semiconductors. <http://www.itrs.net/>.
- [GELP10] A. Gustavsson, A. Ermedahl, B. Lisper, and P. Pettersson. Towards WCET analysis of multicore architectures using UPPAAL. In *Proceedings of the 10th International Workshop on Worst-Case Execution Time Analysis*, pages 103–113. Österreichische Computer Gesellschaft, July 2010.
- [GSYY09] N. Guan, M. Stigge, W. Yi, and G. Yu. Cache-aware scheduling and analysis for multicores. In *Proceedings of the seventh ACM international conference on Embedded software, EMSOFT '09*, pages 245–254, New York, NY, USA, 2009. ACM.
- [HE05] A. Hamann and R. Ernst. TDMA time slot and turn optimization with evolutionary search techniques. In *Proceedings of the conference on Design, Automation and Test in Europe - Volume 1, DATE '05*, pages 312–317, Washington, DC, USA, 2005. IEEE Computer Society.
- [Hüg10] S. Hügi. Predictable communication on multiprocessor platforms. Master's thesis, Swiss Federal Institute of Technology (ETH), Zurich, Switzerland, 2010.
- [Inf] AbsInt Angewandte Informatik. <http://www.absint.com/>.

- [ISTo6] IBM, Sony, and Toshiba. Cell broadband engine architecture, 2006.
- [JLJ<sup>+</sup>10] C. Jalier, D. Lattard, A.A. Jerraya, G. Sassatelli, P. Benoit, and L. Torres. Heterogeneous vs homogeneous MPSoC approaches for a mobile LTE modem. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2010*, pages 184–189, 2010.
- [JLS<sup>+</sup>10] C. Jalier, D. Lattard, G. Sassatelli, P. Benoit, and L. Torres. A homogeneous MPSoC with dynamic task mapping for software defined radio. In *Proceedings of the 2010 IEEE Annual Symposium on VLSI, ISVLSI '10*, pages 345–350, Washington, DC, USA, 2010. IEEE Computer Society.
- [JPGo4] R. Jejurikar, C. Pereira, and R. Gupta. Leakage aware dynamic voltage scaling for real-time embedded systems. In *Proceedings of the 41st annual Design Automation Conference, DAC '04*, pages 275–280, New York, NY, USA, 2004. ACM.
- [KBDVo8] M. Kim, S. Banerjee, N. Dutt, and N. Venkatasubramanian. Energy-aware cosynthesis of real-time multimedia applications on MPSoCs using heterogeneous scheduling policies. *ACM Transactions on Embedded Computing Systems*, 7:9:1–9:19, January 2008.
- [KESH<sup>+</sup>o8] H. Kopetz, C. El Salloum, B. Huber, R. Obermaisser, and C. Paukovits. Composability in the time-triggered system-on-chip architecture. In *SOC Conference, 2008 IEEE International*, pages 87–90, 2008.
- [KG94] H. Kopetz and G. Grunsteidl. TTP-a protocol for fault-tolerant real-time systems. *Computer*, 27(1):14–23, jan 1994.
- [KSMEo5] J.-K. Kim, H. J. Siegel, A. A. Maciejewski, and R. Eigenmann. Dynamic mapping in energy constrained heterogeneous computing systems. In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium - Papers - Volume 01, IPDPS '05*, pages 64.1–, Washington, DC, USA, 2005. IEEE Computer Society.
- [LBR10] K. Lakshmanan, G. Bhatia, and R. Rajkumar. Integrated end-to-end timing analysis of networked AUTOSAR-compliant systems. In *Proceedings of the Conference on Design, Automation and Test in Europe, DATE '10*, pages 331–334, 2010.

- [LESL10] J. Lee, A. Easwaran, I. Shin, and I. Lee. Multiprocessor real-time scheduling considering concurrency and urgency. *SIGBED Review*, 7:5:1–5:5, January 2010.
- [LGY10] M. Lv, N. Guan, W. Yi, and G. Yu. Combining abstract interpretation with model checking for timing analysis of multicore software. In *Proc. IEEE Real-Time Systems Symposium (RTSS 2010)*, volume to appear, Nov. 2010.
- [LKMM07] Y. Lin, M. Kudlur, S. Mahlke, and T. Mudge. Hierarchical coarse-grained stream compilation for software defined radio. In *Proceedings of the 2007 international conference on Compilers, architecture, and synthesis for embedded systems, CASES '07*, pages 115–124, New York, NY, USA, 2007. ACM.
- [LL73] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20:46–61, January 1973.
- [LLW<sup>+</sup>06] Y. Lin, H. Lee, M. Woh, Y. Harel, S. Mahlke, T. Mudge, Ch. Chakrabarti, and K. Flautner. SODA: A low-power architecture for software radio. In *Proceedings of the 33rd annual international symposium on Computer Architecture, ISCA '06*, pages 89–101, Washington, DC, USA, 2006. IEEE Computer Society.
- [LPY97] K.G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a nutshell. *International Journal on Software Tools for Technology Transfer*, 1:134–152, 1997.
- [LSL<sup>+</sup>09] Y. Li, V. Suhendra, Y. Liang, T. Mitra, and A. Roychoudhury. Timing analysis of concurrent programs running on shared cache multi-cores. In *Proceedings of the 2009 30th IEEE Real-Time Systems Symposium, RTSS '09*, pages 57–67, Washington, DC, USA, 2009. IEEE Computer Society.
- [MCR<sup>+</sup>06] S. Murali, M. Coenen, A. Radulescu, K. Goossens, and G. De Micheli. Mapping and configuration methods for multi-use-case networks on chips. In *Proceedings of the 2006 Asia and South Pacific Design Automation Conference, ASP-DAC '06*, pages 146–151, Piscataway, NJ, USA, 2006. IEEE Press.
- [MMBo7] O. Moreira, J.J.-D. Mol, and M. Bekooij. Online resource management in a multiprocessor with a network-on-chip. In *Proceedings of the 2007 ACM symposium on Applied computing, SAC '07*, pages 1557–1564, New York, NY, USA, 2007. ACM.

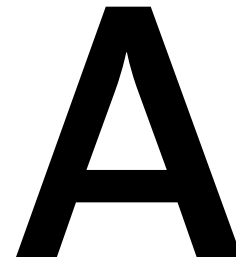


- [MMBM05] O. Moreira, J.-D. Mol, M. Bekooij, and J. v. Meerbergen. Multiprocessor resource allocation for hard-real-time streaming with a dynamic job-mix. In *Proceedings of the 11th IEEE Real Time on Embedded Technology and Applications Symposium*, pages 332–341, Washington, DC, USA, 2005. IEEE Computer Society.
- [MVB07] O. Moreira, F. Valente, and M. Bekooij. Scheduling multiple independent hard-real-time jobs on a heterogeneous multiprocessor. In *Proceedings of the 7th ACM & IEEE international conference on Embedded software*, EMSOFT '07, pages 57–66, New York, NY, USA, 2007. ACM.
- [NSE09] M. Negrean, S. Schliecker, and R. Ernst. Response-time analysis of arbitrarily activated tasks in multiprocessor systems with shared resources. In *Design, Automation Test in Europe Conference Exhibition, 2009. DATE '09.*, pages 524–529, 2009.
- [NSE10] M. Negrean, S. Schliecker, and R. Ernst. Timing implications of sharing resources in multicore real-time automotive systems. In *SAE 2010 World Congress & Exhibition Technical Papers*, volume 3, pages 27–40, Detroit, MI, USA, August 2010.
- [Pap94] Ch. H. Papadimitriou. *Computational Complexity*. Addison-Wesley Publishing Company, 1994.
- [PBCSo8] R. Pellizzoni, B. D. Bui, M. Caccamo, and L. Sha. Coscheduling of CPU and I/O transactions in COTS-based embedded systems. In *Proceedings of the 2008 Real-Time Systems Symposium*, pages 221–231, Washington, DC, USA, 2008. IEEE Computer Society.
- [PCo7] R. Pellizzoni and M. Caccamo. Toward the predictable integration of real-time cots based systems. In *Proceedings of the 28th IEEE International Real-Time Systems Symposium, RTSS '07*, pages 73–82, Washington, DC, USA, 2007. IEEE Computer Society.
- [PPE<sup>+</sup>08] T. Pop, P. Pop, P. Eles, Z. Peng, and A. Andrei. Timing analysis of the FlexRay communication protocol. *Real-Time Systems*, 39:205–235, August 2008.
- [PPEP07] T. Pop, P. Pop, P. Eles, and Z. Peng. Bus access optimisation for FlexRay-based distributed embedded systems. In *Proceedings of the conference on Design, automation and test in Europe, DATE '07*, pages 51–56, San Jose, CA, USA, 2007. EDA Consortium.

- [PQnC<sup>+</sup>09] M. Paolieri, E. Quiñones, F. J. Cazorla, G. Bernat, and M. Valero. Hardware support for wcet analysis of hard real-time multicore systems. In *Proceedings of the 36th annual international symposium on Computer architecture*, ISCA '09, pages 57–68, New York, NY, USA, 2009. ACM.
- [Pre] European Union 7<sup>th</sup> Framework Programme Project Predator. <http://www.predator-project.eu/>.
- [PSC<sup>+</sup>10] R. Pellizzoni, A. Schranzhofer, J.-J. Chen, M. Caccamo, and L. Thiele. Worst case delay analysis for memory interference in multicore systems. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '10, pages 741–746, 2010.
- [RAEP07] J. Rosen, A. Andrei, P. Eles, and Z. Peng. Bus access optimization for predictable implementation of real-time applications on multiprocessor systems-on-chip. In *Proceedings of the 28th IEEE International Real-Time Systems Symposium*, RTSS '07, pages 49–60, Washington, DC, USA, 2007. IEEE Computer Society.
- [Raj91] R. Rajkumar. *Synchronization in Real-Time Systems: A Priority Inheritance Approach*. Kluwer Academic Publishers, Norwell, MA, USA, 1991.
- [RD94] A. Rajeev and David L. D.L. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, April 1994.
- [RFSWo6] C. Rusu, A. Ferreira, C. Scordino, and A. Watson. Energy-efficient real-time heterogeneous server clusters. In *Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 418–428, Washington, DC, USA, 2006. IEEE Computer Society.
- [RGBWo7] J. Reineke, D. Grund, C. Berg, and R. Wilhelm. Timing predictability of cache replacement policies. *Real-Time Systems*, 37:99–122, November 2007.
- [SAHE05] M.T. Schmitz, B.M. Al-Hashimi, and P. Eles. Cosynthesis of energy-efficient multimode embedded systems with consideration of mode-execution probabilities. *Computer-Aided Design of Integrated Circuits and Systems*, *IEEE Transactions on*, 24(2):153–169, 2005.
- [Sch86] A. Schrijver. *Theory of Linear and Integer Programming*. John Wiley & Sons, Chichester, 1986.

- [SIEo6] S. Schliecker, M. Ivers, and R. Ernst. Integrated analysis of communicating tasks in MPSoCs. In *Proceedings of the 4th international conference on Hardware/software codesign and system synthesis*, CODES+ISSS '06, pages 288–293, New York, NY, USA, 2006. ACM.
- [SKG<sup>+</sup>07] L. Schwoerer, A. Kaufmann, D. Guevorkian, J. Westmeijer, and Y. Xu. DVB (synchronized and receiving mode) summary for SDR scheduler modeling workshop: timing corresponds to 8k case. Technical report, Radio Communications CTC, NOKIA Research Center, Helsinki, June 2007.
- [SN10] S. Schliecker and R. Negrean, M. and Ernst. Bounding the shared resource load for the performance analysis of multiprocessor systems. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '10, pages 759–764, 2010.
- [SNN<sup>+</sup>08] S. Schliecker, M. Negrean, G. Nicolescu, P. Paulin, and R. Ernst. Reliable performance analysis of a multicore multithreaded system-on-chip. In *Proceedings of the 6th IEEE/ACM/IFIP international conference on Hardware/Software codesign and system synthesis*, CODES+ISSS '08, pages 161–166, New York, NY, USA, 2008. ACM.
- [SWvdVo8] A. Schranzhofer, Yiyin Wang, and A.-J. van der Veen. Acquisition for a transmitted reference UWB receiver. In *Ultra-Wideband, 2008. ICUWB 2008. IEEE International Conference on*, volume 2, pages 149–152, 2008.
- [TCN00] L. Thiele, S. Chakraborty, and M. Naedele. Real-time calculus for scheduling hard real-time systems. In *Circuits and Systems, 2000. Proceedings. ISCAS 2000. The IEEE International Symposium on*, 2000.
- [TW04] L. Thiele and R. Wilhelm. Design for timing predictability. *Real-Time Systems*, 28:157–177, November 2004.
- [WGR<sup>+</sup>09] R. Wilhelm, D. Grund, J. Reineke, M. Schlickling, M. Pister, and Ch. Ferdinand. Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems. *Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 28:966–978, July 2009.
- [Wil05] R. Wilhelm. Timing analysis and timing predictability. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and

- Willem-Paul de Roever, editors, *Formal Methods for Components and Objects*, volume 3657 of *Lecture Notes in Computer Science*, pages 317–323. Springer Berlin / Heidelberg, 2005. 10.1007/1156116314.
- [WJMo8] W. Wolf, A.A. Jerraya, and G. Martin. Multiprocessor system-on-chip (MPSoC) technology. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 27(10):1701–1713, 2008.
- [WTo6a] E. Wandeler and L. Thiele. Optimal TDMA time slot and cycle length allocation for hard real-time systems. In *Proceedings of the 2006 Asia and South Pacific Design Automation Conference, ASP-DAC '06*, pages 479–484, Piscataway, NJ, USA, 2006. IEEE Press.
- [WTo6b] Ernesto Wandeler and Lothar Thiele. Real-Time Calculus (RTC) Toolbox. <http://www.mpa.ethz.ch/rtctoolbox>, 2006.
- [XZR<sup>+</sup>05] R. Xu, D. Zhu, C. Rusu, R. Melhem, and D. Mossé. Energy-efficient policies for embedded clusters. In *Proceedings of the 2005 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems, LCTES '05*, pages 1–10, New York, NY, USA, 2005. ACM.
- [YO09] Ch. Yang and A. Orailoglu. Towards no-cost adaptive MP-SoC static schedules through exploitation of logical-to-physical core mapping latitude. In *Proceedings of the Conference on Design, Automation and Test in Europe, DATE '09*, pages 63–68, 2009.
- [YZo8] J. Yan and W. Zhang. WCET analysis for multi-core processors with shared L2 instruction caches. In *Proceedings of the 2008 IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS*, pages 80–89, Washington, DC, USA, 2008. IEEE Computer Society.
- [ZY09] W. Zhang and J. Yan. Accurately estimating worst-case execution time for multi-core processors with shared direct-mapped instruction caches. In *Embedded and Real-Time Computing Systems and Applications, 2009. RTCSA '09. 15th IEEE International Conference on*, pages 455–463, 2009.



## Toolbox

In this chapter, we outline the analysis toolbox that has been developed in the course of this thesis. The toolbox presented in this chapter is included in the RTC toolbox [WT06b] and takes advantage of RTC data structures, such as arrival curves [CKT03b]. In Section A.1 we present the representation of interference as arrival curve, as described in Chapter 3. Section A.2 provides the implementation of the WCRT analysis for static arbitration policies on the shared resource, as presented in Chapter 4. The WCRT analysis for hybrid arbitration policies, i.e., the FlexRay protocol presented in Chapter 5, is given in Section A.3. Section A.4 outlines the integration of this thesis into the Predator project [Pre] and input provided by project partners.

### A.1 Representing Interference

We implemented a MATLAB toolbox to derive the access pattern of a set of superblocks, that executes periodically, onto a shared resource. The definition of the set of superblocks is done in an ASCII file, which is the input to the analysis. The derivation of this access pattern splits into two cases: (1) superblocks execute sequentially or (2) superblocks or phases execute in a time-triggered manner.

### A.1.1 Sequential execution of superblocks

For the sequentially executing models, the starting time of the superblocks does not need to be considered, since superblocks execute as soon as their predecessor has finished. The set of superblocks are specified in a file, which is denoted as *configuration* file. This file is the input to our analysis methodology, which is described in Section 3.4.3, and is formatted according to Listing A.1.

**Listing A.1:** File specifying a set of 10 subsequently executing superblocks, with a period of 285ms.

```

1 %number of superblocks
2 %period
3 %minimum number of access requests
4 %maximum number of access requests
5 %communication time per access request
6 %minimum computation time
7 %maximum computation time
8 10
9 285
10 7 2 9 3 11 2 5 5 12 13
11 10 12 9 10 14 4 6 12 13 18
12 2
13 8 9 5 4 7 6 3 1 2 5
14 10 30 9 10 15 9 4 2 4 5

```

The derivation of the arrival curves, as shown in Chapter 3, Equations (3.8) and (??), is then performed by calling the function:

```

interference_single_periodic_task(config_file, ...
    output, delta, debug).

```

The first argument to this function is `config_file`, representing the MATLAB file handle to the configuration file specified in Listing A.1. The argument `output`, defines whether a plot representing the curves should be plotted `output=1`, or not `output=0`, while argument `delta` specifies the maximum value on the x-axis, in case a plot is produced. Argument `debug=1` allows to produce a plot with all the tuples, see Section 3.4.3.3, or `debug=0` otherwise. The function returns the parameters `RTCcurve_upper` representing the resulting arrival curves  $\hat{\alpha}_j^u(\Delta)$  see Listing A.2.

**Listing A.2:** Derive the interference representation for a subsequently executing set of superblocks.

```

1 %load the configuration file
2 config_file = ('\path\to\configuration\file ');
3 %produce a plot of the resulting arrival curves
4 output = 1;
5 %specify the length of the x-axis
6 delta = 100;

```

```

7 %do not plot the tuples
8 debug = 0;
9
10 [RTCcurve_upper] = ...
11             interference_single_periodic_task( ...
12             config_file , output , delta , debug);

```

### A.1.2 Time-triggered execution of superblocks

This section describes the extension of the toolbox to derive the access pattern, in case superblocks or phases are executed in a time-triggered manner. The analysis of these models is given in Section 3.4.4. As opposed to sequentially executing superblocks, analysis of the time-triggered case requires to consider the dedicated starting times. As a result, the configuration file needs to be extended to account for that. See Listing A.3 for an example configuration file, specifying two sets of superblocks - in the first line. Following that, the time window for each set of superblocks is specified, i.e., the first set of superblocks starts execution at time 0 and might continue to do so until time 22, while the second set starts execution at time 22 and then might continue to execute for 12 units of time. Conclusively, the period of the considered set of superblocks is  $34ms$ . Then follows the specification of the number of superblocks in the respective sets of superblocks and 4 lines to specify each set of superblocks.

**Listing A.3:** File specifying 2 sets of superblocks executed in a time-triggered manner, with a period of  $34ms$ .

```

1 %Number of sets of superblocks
2 %time windows for the set of superblocks
3 %the number of superblocks in each set
4 %for each set of superblocks 4 lines:
5     %min. computational time for each superblock
6     %max. computational time for each superblock
7     %min. access requests for each superblock
8     %max. access requests for each superblock
9 2
10 22 12
11 2 1
12 3 2
13 5 3
14 1 1
15 2 2
16 4
17 7
18 1
19 2

```

Then, the arrival curve is generated by calling function

```
RTCupperCurve = computeMultiTaskInterference( ...
    config_file, output, delta),
```

where the arguments `config_file`, `output` and `delta` are defined as in Section A.1.1, i.e., the MATLAB handle to the configuration file, a binary value specifying whether or not to produce a plot and the length of the x-axis in case a plot is generated, respectively. The function returns parameter `RTCupperCurve` representing the resulting upper arrival curve, as specified in Section 3.4.4.

## A.2 Worst-Case Analysis for static arbitration

In Chapter 4, we present an approach to find the worst-case completion time (WCCT) of a set of superblocks, given a TDMA arbitration policy on the shared resource, for the different resource access models, as proposed in Section 3.3.2.

We specify the system by an array of tasks, i.e., `tasks[]`. The elements of this array are arrays of superblocks, i.e., for a task  $i$ , the superblocks are `tasks(i).superblocks[]`. A superblock has two parameters (1) the maximum amount of computation that has to be performed and (2) the maximum number of access requests in the phases of the superblock. Consider a superblock  $j$  of task  $i$ , then the maximum amount of computations is denoted `tasks(i).superblocks(j).exec_time_u`, while the maximum number of access requests for the three different types of phases (acquisition/general/replication) is stored in an array `accesses_upper[]` as field of the structure `tasks(i).superblocks(j)`. This array has three elements, where the first, second and third element represent the maximum number of access requests in the acquisition, general and replication phase, respectively. Following that, a superblock that follows the dedicated phases model stores the corresponding values in `accesses_upper(1)` and `accesses_upper(3)` for the acquisition and replication phase, respectively, while no access requests can happen during its execution phase, i.e., `accesses_upper(2) = 0`. The hybrid model is specified similarly, except that there might be access requests in the general phase, i.e., `accesses_upper(2)` has a value. In order to model the general model, the sum of all three phases is considered as the number of access requests that might happen anytime and in any order.

Since tasks represent an array of superblocks, i.e., a set of superblocks, we assume one periodic task to execute on each processing element of a multicore system. As a result, the analysis is done for one task at a time, with each task being assigned a time slot in the TDMA arbiter. The TDMA arbiter is defined by three parameters: (1) the TDMA cycle, (2) that starting time of the slot assigned to the task under analysis and (3) the duration of the time slot assigned to the task under analysis.



The WCCT of such a system, as discussed in Chapter 4, can be computed as shown in Listing A.4. Note that in this skript, we assume a task under analysis, and compute the WCCT of this task for all different resource access models. Function `SEQ_WCCT` computes the WCCT of the task for the three sequential models and additionally derives the earliest starting time for the time-triggered models, i.e., the dedicated starting times for superblocks in model HTS and GTS, and the dedicated starting times for phases in model HTT. As arguments, this function expects the task structure as specified earlier, `tasks`, the arbitration cycle of the TDMA arbiter, `length`, the duration and starting time of the slot assigned to the task under analysis, `duration` and `start_time`, the time required to complete an access request, `C`, the index of the task under analysis, `considered_task`, as well as the period of the task, `period`. Note that tasks are either generated by a random number generator or according to specifications given by industrial partners, where the argument to function `createProblemInstance` specifies in which fields of the data structure `tasks[].superblocks[].access_upper[]` the access requests are written.

Functions `HTS_WCCT`, `GTS_WCCT` and `HTT_WCCT` compute the WCCT for models HTS, GTS and HTT, respectively. These functions have additional arguments to specify the starting times of superblocks (models HTS and GTS) and phases (model HTT). In our experiments, we used the earliest possible starting times, as derived by function `SEQ_WCCT` for these models, in order to provide a fair comparison. The starting times are stored in an array, each element representing a starting time. For function `HTS_WCCT` and `GTS_WCCT`, the starting times are stored in `start_hs_sb_max` and `start_gs_sb_max`, respectively. Function `HTT_WCCT` has three additional arguments, where `start_ht_acq_max`, `start_ht_exec_max` and `start_ht_rep_max` represent the starting times of the acquisition, general and replication phases, respectively.

**Listing A.4:** Derive the WCCT of a task, for different resource access models and a given TDMA arbiter.

```

1  %generate a task according to a pattern
2  %either by a random number generator
3  %or by data from an industrial partner
4  tasks = createProblemInstance('hybrid');
5
6  %choose the task to analyze
7  considered_task = 1;
8
9  %the arbitration cycle
10 period = tasks(considered_task).period;
11
12 %starting time of the slot assigned to the task under analysis
13 start_time = 0;
14
```

```

15 %duration of that time slot
16 duration = 5;
17
18 %communication time per access request
19 C = 2;
20
21 %the length of the TDMA arbitration cycle
22 length = 10;
23
24 %compute the WCRT for the sequential models
25 %and the earliest possible starting times for
26 %the time-triggered models
27 %the output variables:
28 % wert_GSS ... WCRT for model GSS
29 % wert_HSS ... WCRT for model HSS
30 % wert_DSS ... WCRT for model DSS
31 % start_ds_sb_max ... starting times for superblocks
32 % start_hs_sb_max ... starting times for superblocks, model HTS
33 % start_gs_sb_max ... starting times for superblocks, model GTS
34 % start_ht_acq_max ... starting times for phases, model HTT
35 % start_ht_exec_max ... starting times for phases, model HTT
36 % start_ht_rep_maxxx ... starting times for phases, model HTT
37
38 [ wert_GSS ...
39   wert_HSS ...
40   wert_DSS ...
41   start_ds_sb_max ...
42   start_hs_sb_max ...
43   start_gs_sb_max ...
44   start_ht_acq_max ...
45   start_ht_exec_max ...
46   start_ht_rep_max ] = ...
47   SEQ_WCCT(tasks, length, duration, start_time, C, ...
48             considered_task, period);
49
50 %compute the WCRT for model HTS
51 [wert_HTS] = ...
52   HTS_WCCT(tasks, length, duration, start_time, C, ...
53            start_hs_sb_max, considered_task, period);
54
55 %compute the WCRT for model GTS
56 [wert_GTS] = ...
57   GTS_WCCT(tasks, length, duration, start_time, C, ...
58            start_gs_sb_max, considered_task, period);
59
60 %compute the WCRT for model HTT
61 [wert_HTT] = ...
62   HTT_WCCT(tasks, length, duration, start_time, C, ...
63            start_ht_acq_max, start_ht_exec_max, ...
64            start_ht_rep_max, considered_task, period);
65
66 results = [wert_DSS ...

```

```

67         wcr_t_HSS ...
68         wcr_t_GSS ...
69         wcr_t_HTS ...
70         wcr_t_HTT ...
71         wcr_t_GTS ];

```

### A.3 Worst-Case Analysis for hybrid arbitration

As opposed to static arbitration, we analyze systems with hybrid arbitration in Chapter 5. Therefore, resource interference, as described in Chapter 3 has to be considered. In this section, we introduce our analysis framework, that allows to derive the WCRT of a set of superblocks, given an interference representation and a FlexRay arbiter on the shared resource. The interference representation can be computed by the functions presented in Section A.1 of this chapter.

In Listing A.5, we show how to compute the WCCT of a set of superblocks. First, the problem instance is generated, as in Section A.2. Then the superblock data structure is generated. The set of superblocks, executing on a processing element, is represented by the data structure  $\mathbf{S}$ , which is a  $N \times 5$  matrix, where  $N$  represents the number of superblocks in the set. The parameters stored for each superblock are (1) the maximum computation time, (2) the maximum number of access requests, (3) the assignment to a time slot in the static segment, (4) the starting time of the superblock, or 0 if the superblock should be executed sequentially and (5) the deadline and period of the superblock. Then, the communication time  $C$  and the arbiter is specified. Note that deriving a TDMA or FlexRay arbiter is out of scope of this toolbox.

Then, after deriving an arrival curve representation of the other sets of superblocks in the system, the WCCT can be derived by `analyzeTask`, with the arguments  $\mathbf{S}$ , as described earlier, `alpha_upper` as the interference by other elements, `theta` as the arbiter, `Ltot` as the number of interfering elements, `C` as the communication time, `m1` as the length of a minislot and `Lpre` as the maximum number of pending access requests at the beginning of the dynamic segment. The function returns a binary variable `schedulable` and the actual WCCT of the set of superblocks, as variable `wcct`.

**Listing A.5:** Derive the WCCT of a task, for different resource access models and a given FlexRay arbiter.

```

1 %generate a task according to a pattern
2 %either by a random number generator
3 %or by data from an industrial partner
4 tasks = createProblemInstance('hybrid');
5
6 %choose the task to analyze

```

```

7  considered_task = 1;
8
9  %load the configuration file
10 config_file = ('\path\to\configuration\file ');
11
12 %do not plot a figure for the interference
13 output = 0;
14 delta = 1;
15 debug = 0;
16
17 %choose the task
18 task = tasks(considerd_task);
19 number_of_blocks = task.number_of_superblocks;
20
21 %initialize the superblock datastructure
22 S = zeros(number_of_blocks, 5);
23 for j=1:number_of_blocks
24     S(j, 1) = task.superblocks(j).execution_time_upper;
25     S(j, 2) = sum(task.superblocks(j).accesses_upper);
26     %the superblock to time slot assignment,
27     %note that all superblock in the set are assigned
28     %to the same slot.
29     S(j, 3) = 1;
30     %the starting time of the superblock
31     %0 if sequential execution
32     S(j, 4) = 0;
33     %the deadline of the superblock = the period
34     S(j, 5) = 100;
35 end
36
37 %communication time per access request
38 C = 0.5;
39
40 %the arbiter
41 %Superblock 2 assigned to slot 1 in
42 %the static segment
43 theta.assignments = [ 2 1 3];
44 %the starting times of the static slots
45 theta.starting_times = [0 11 15];
46 %the length of the static segment
47 theta.L = 17
48 %the length of the dynamic segment
49 theta.dyn_length = 7;
50
51 %the worst-case number of interferers and
52 %pending access requests at the beginning
53 %of the dynamic segment
54 Ltot = 2
55 Lpre = 2;
56
57 %the length of a minislot in the dynamic
58 %segment

```

```

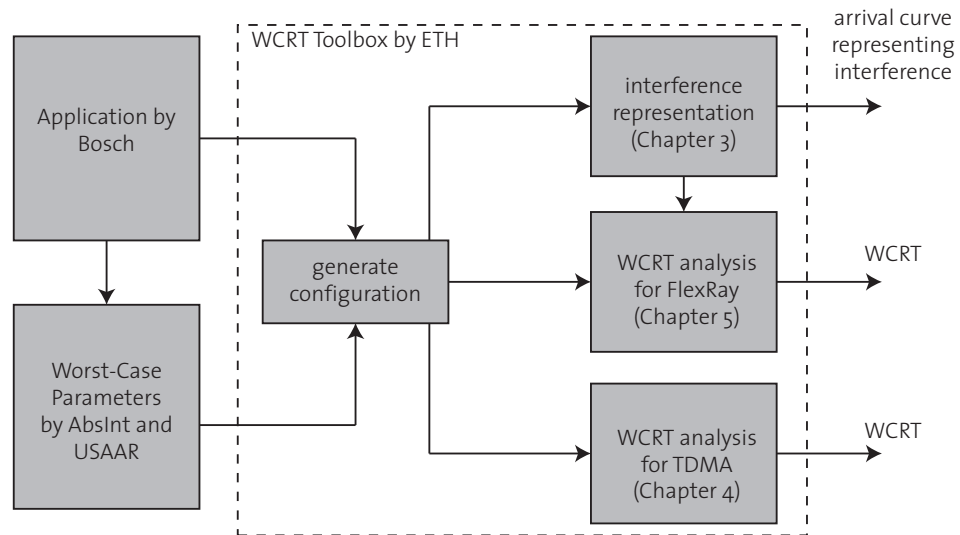
59 ml = 0.5;
60
61 [alpha_upper] = interference_single_periodic_task( ...
62               config_file , output , delta , debug);
63
64 [schedulable , wcct] = analyzeTask(S, alpha_upper , theta , ...
65               Ltot , C, ml, Lpre);

```

## A.4 Integration with Project Partners

In this thesis, we rely on the availability of high-level application specifications, like the worst-case execution time of a task or a superblock, and the maximum and minimum number of access requests of a task or superblock to a shared resource.

In the Predator project, we received data for example applications in the automotive domain from Bosch. These applications were analyzed with an adapted version of the aiT analysis tool by AbsInt [Inf] and the University of Saarland (abbreviated: USAAR). The aiT tool analyzes the worst-case execution time (WCET) of a task or superblocks. The tool was extended to analyze the maximum number of access requests to a shared resources. Deriving the minimum number of access requests is not yet implemented in the current version of aiT. See Figure A.1 for an overview of project partners, contributing to the analysis framework.



**Figure A.1:** WCRT Analysis Toolbox for resource sharing systems.



# List of Publications

The following list presents the publications that form the basis of this thesis. The according chapters are given in brackets.

Andreas Schranzhofer, Jian-Jia Chen and Lothar Thiele, Power-Aware Mapping of Probabilistic Applications onto Heterogeneous MPSoC Platforms. In *Real-Time and Embedded Technology and Applications Symposium (RTAS 09)*, San Francisco, CA, April 2009, (Chapter 2)

Andreas Schranzhofer, Jian-Jia Chen and Lothar Thiele, Timing Predictability on Multi-Processor Systems with Shared Resources. In *Workshop on Reconciling Performance with Predictability (RePP 09)*, co-located with ESWEEK, Grenoble, France, October 2009 (Chapter 3 and 4)

Andreas Schranzhofer, Jian-Jia Chen, Luca Santinelli and Lothar Thiele, Dynamic and Adaptive Allocation of Applications on MPSoC Platforms. In *Asia and South Pacific Design Automation Conference (ASPDAC 10)*, Taipei, Taiwan, January 2010, (Chapter 2)

Andreas Schranzhofer, Jian-Jia Chen and Lothar Thiele, Dynamic Power-Aware Mapping of Applications onto Heterogeneous MPSoC Platforms. In *IEEE Transactions on Industrial Informatics (TII)*, November 2010, Volume 6, Nr. 4, Pages 692 - 707 (Chapter 2)

Rodolfo Pellizzoni, Andreas Schranzhofer, Jian-Jia Chen, Marco Caccamo and Lothar Thiele, Worst Case Delay Analysis for Memory Interference in Multicore Systems, in *Conference of Design, Automation, and Test in Europe (DATE 10)*, Dresden, Germany, March 2010 (Chapter 3)

Andreas Schranzhofer, Jian-Jia Chen and Lothar Thiele, Timing Analysis for TDMA Arbitration in Resource Sharing Systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS 10)*, Stockholm, Sweden, April 2010  
(Chapter 4)

Andreas Schranzhofer, Rodolfo Pellizzoni, Jian-Jia Chen, Lothar Thiele and Marco Caccamo, Worst-Case Response Time Analysis of Resource Access Models in Multi-Core Systems. In *Design Automation Conference (DAC 10)*, Anaheim, CA, June 2010  
(Chapter 4)

Andreas Schranzhofer, Rodolfo Pellizzoni, Jian-Jia Chen, Lothar Thiele and Marco Caccamo, Timing Analysis for Resource Access Interference on Adaptive Resource Arbiters. to appear in *Real-Time and Embedded Technology and Applications Symposium (RTAS 2011)*, Chicago, IL, April 2011.  
(Chapter 5)

The following publications are not part of this thesis.

Jian-Jia Chen, Andreas Schranzhofer and Lothar Thiele Energy minimization for periodic real-time tasks on heterogeneous processing units. In *International Symposium on Parallel. Distributed Processing (IPDPS 09)*, Rome, Italy, May 2009

Jian-Jia Chen, Andreas Schranzhofer and Lothar Thiele Energy-Aware Task Partitioning and Processing Unit Allocation for Periodic Real-Time Tasks on Systems with Heterogeneous Processing Units. In *Work-in-Progress in EuroMicro Conference on Real-Time Systems (ECRTS 08)*, Prague, Czech Republic, July 2008



# Curriculum Vitae

## Personal Data

Name: Andreas Schranzhofer  
Date of birth: December 23, 1980  
Place of birth: Lienz, Austria  
Citizenship: Austria

## Education

2007 - 03/2011 **PhD** in Computer Engineering, ETH Zurich, Switzerland  
with Prof. Dr. Lothar Thiele  
Thesis: *Efficiency and predictability in resource sharing Multi-Core Systems*

2005 - 2007 EU Project: Predator - Design for predictability and efficiency  
**M.Sc.** in Telematics, TU Graz, Austria  
Graduation with distinction  
Thesis: *Acquisition for a Transmitted Reference UWB Receiver*, at Circuits and Systems Group, TU Delft, The Netherlands  
Supervisors: prof. dr. ir. Alle-Jan v. d. Veen (TU Delft) and Dr. Dipl. Ing. Klaus Witrissal (TU Graz)

2004 - 2005 Erasmus, TU Tampere, Finland  
Department of Information Technology

2001 - 2005 **B.Sc.** in Telematics, TU Graz, Austria

1995 - 2000 Technical High School (HTL), Jenbach, Austria  
focus on Engineering, Mathematics and Physics

1991 - 1995 Grammar school (Gymnasium) in Lienz, Austria

1987 - 1991 Elementary School (Volksschule) in Sillian, Austria

## Professional Experience

2007 - 2011 Teaching Assistant at Computer Engineering and Networks Laboratory (TIK), ETH Zurich

2007 - 2011 Supervision of Master Theses in the field of embedded systems

09/2005 - 11/2006 J. Christof Group, Graz, Austria, [www.christof-group.at](http://www.christof-group.at)  
Development of an intranet based management application

07/2002 - 08/2004 Laufer & Partner OEG, Graz, Austria, [www.laufer.at](http://www.laufer.at)  
Programming in PHP, Perl, C, MySQL, Apache, Unix/Linux

01/2001- 08/2001 Military Service, Austrian Armed Forces