Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für
Technische Informatik und
Kommunikationsnetze

Master's Thesis at the
Department of Information Technology and
Electrical Engineering

for

Simon Hügi

Predictable Communication on Multiprocessor Platforms

Advisors: Andreas Schranzhofer
Dr. Wolfgang Haid

Professor: Prof. Dr. Lothar Thiele

1. December 2010

Abstract

The on-going trend towards multiprocessor platforms and multiprocessor systems on chip (MPSoCs) for embedded systems fulfills the need of increased computation performance. MPSoCs normally contain shared resources in order to satisfy cost constraints. While this improves the average performance, this development is at the cost of timing predictability, which is required for real-time systems: If real-time tasks access shared resources concurrently, they can get delayed due to contention. Hence, it is necessary to estimate upper bounds of these delays.

This master's thesis focuses on the problem of implementing accesses to these shared resources in a way that increases the timing predictability of the system. First, a Time Division Multiple Access (TDMA) scheduler granting access to the shared data memory is implemented on the Cell Broadband Engine. Experiments on this MPSoC show the influence of the scheduler and the hardware behavior on the completion time of example tasks. Second, a multiprocessor system with a shared data memory and a shared instruction flash is analyzed for the dynamic First-Come, First-Served (FCFS) and and the static TDMA arbitration. We present approaches to determine the worst-case completion time of tasks and the difficulty to find methods for the dynamic FCFS. These two parts emphasize the importance of applying methods to improve the analyzability and consequently, the timing predictability of a multiprocessor system. These methods include the introduction of structure for the resource arbiter (TDMA) and for the tasks (separation of data accesses and execution).

Acknowledgement

I owe my deepest gratitude to my advisors, Andreas Schranzhofer and Wolfgang Haid, whose encouragement, guidance and support enabled me to write this master's thesis. I would like to thank them for giving me constant feedbacks during the completion of this project and taking time for valuable discussions. I had the pleasure to work with two kind and humorous guys.

Last but not least, I offer my blessings to my family and my friends who supported me during my studies.

Simon Hügi, Zürich, December, 2010

Contents

List of Figures

List of Tables

1 Introduction

1.1 Motivation

The usage of multiprocessor platforms for embedded systems is necessary due to increased computational performance requirements. Multiprocessor systems on chip (MPSoCs) are often applied for this purpose, thereby satisfying cost constraints and reducing the energy consumption. Such systems contain resources that are shared among the processing cores in order to increase the average performance and to reduce the production costs. However, these shared resources such as interconnect buses, main memory, etc. have become the main bottleneck concerning the timing predictability. This is especially a problem for real-time systems, where timing constraints have to be guaranteed.

Therefore, recent research in the area of real-time systems has focused on the problem of analyzing the performance of such systems. If tasks execute on processing elements access the shared resource simultaneously, each request can be delayed due to contention. This increases the worst-case access time compared to single processor architectures and consequently, the worst-case completion time (WCCT) of each task. It is important to determine this WCCT in order to guarantee the correct functionality of real-time systems by avoiding deadline violations. This affects embedded systems in general and hard real-time systems in particular – where the completion of an operation after its deadline can result in a critical failure. Examples for such systems are control systems for the automotive or avionic industry.

1.2 Related Work

The work of Thiele et al. [1] addresses the necessity of timing predictability in safety-critical embedded systems. They define this term as follows:

> "The timing predictability of a system is related to the differences between best case and lower bound on the one hand and upper bound and worst-case on the other." ... "Bad predictability is caused by interference and limited analyzability of the behavior."

In other words, the timing behavior of a system should be analyzable on the one part and the methods to estimate the WCCT should be precise on the other part. For instance, First-Come, First-Served (FCFS) is a common arbitration policy in many hardware systems. This dynamic policy is difficult to analyze because it depends on all possible interferers as can be seen later. Hence, a possible solution to increase the timing predictability is to use static arbitration instead, e.g., Time Division Multiple Access (TDMA). This static policy simplifies the performance analysis by explicitly removing the memory interference problem: Each processing element is allowed to access the shared resources during predefined time slots. As a result, finding a WCCT only depends on one task under analysis.

Another mechanism for improving timing predictability is the Time-Triggered Protocol (TTP) proposed by Kopetz et al. [2]. They present a communication protocol for time-triggered architectures, where system activities are activated at predetermined time instances. This represents another technique to simplify the analyzability of a system. Wilhelm et al. [3] discuss the influence of the architecture on the performance analysis and recommend features for future architectures that improve the analyzability as well. Schliecker et al. [4] investigate the timing implications that shared resources cause in multiprocessor platforms. They outline a method to bound the memory interference delay by considering alternating tasks. Guan et al. [5] propose the usage of cache space isolation to avoid contention on shared L2 cache components and present a possible scheduling strategy.

1.3 Problem Statement

Based on this background, this thesis focuses on the topic of analyzing the timing behavior of multiprocessor systems containing one and two shared resources, respectively. It discusses possibilities to implement the communication to these resources in a way that enables precise methods to determine the WCCT.

The efforts needed to implement a predictable TDMA schedule on an unpredictable hardware system are outlined, namely the Cell Broadband Engine. In order to apply the considered theoretical model containing one shared resource, it is necessary to estimate the system parameters by measurement. Adding a second shared resource to the system has several consequences that reduce the analyzability of the system and accordingly, the timing predictability. As a result, finding a WCCT is complicated and the arising problems are investigated. Then, different arbitration policies are compared for this adapted system.

1.4 Thesis Outline and Contributions

This thesis is divided into these chapters, which are shortly summarized in the following:

- **Chapter 2**: An overview of the used multiprocessor hardware system is provided. This includes the communication architecture as well as the relevant functionalities of the Cell Broadband Engine. The second part introduces a theoretical model, which is the basic model for the remaining thesis. Its assumptions as well as the mathematical notation are described.
- **Chapter 3:** This chapter shows one possibility how the introduced theoretical model can be used for the design of real-time embedded systems composed of commercial systems and components-of-the-shelf (COTS). For this, a TDMA scheduler is implemented on top of the the Cell Broadband Engine. As this multiprocessor system does not contain a global time, an approach is presented in order to synchronize the processing elements. The resulting synchronization overhead and the communication time needed for the performance analysis is estimated by measurements. The experiments analyze the behavior of the implemented system on the Cell Broadband Engine. In particular, the predicted worst-case completion times of example tasks are compared to the measured completion times. Finally, the differences to the considered theoretical model are outlined.
- **Chapter 4:** The timing behavior of a system containing two shared resources – an instruction and a data cache – is analyzed. The theoretical model needs to be adapted in order to take the new behavior

into account. The difficulties for finding a tight worst-case completion time are discussed and possible approaches for FCFS arbitration are presented. An equivalent architecture with TDMA arbitration is motivated to simplify the performance analysis and thereby, to increase the timing predictability. In the end, the experiments compare these two arbitration policies.

- **Chapter 5:** The last chapter concludes the thesis and summarized the achieved results. An outlook proposes some interesting issues for future work.

2 Background

2.1 Overview of the Cell Broadband Engine

The Cell Broadband Engine [6] of Sony/Toshiba/IBM is used as a multiprocessor platform for this thesis. Even though not designed for hard real-time applications, it is a platform that satisfies the requirements for a testing platform in the context of this thesis: First of all, it is a multiprocessor system with a shared resource – the main memory. Furthermore, software development is facilitated by a mature software development kit offering debugging possibilities and many runtime libraries that can easily be extended. Hence, it is possible to set up a configurable framework on top of this system, to implement monitoring functionalities for experiments and to measure performance metrics, in particular, the worst-case completion time of tasks executed on the available processing elements.

The Cell Broadband Engine contains one main processor, the *Power Processor Element* (PPE), and up to eight *Synergistic Processor Elements* (SPE). An overview of the architecture can be found in Fig. 2.1. The PPE is an extension of the 64-bit PowerPC Architecture [7] and its main task is to coordinate the SPEs, which are RISC processors optimized for media and streaming workloads. The *Element Interconnect Bus* (EIB) has a ring topology and connects the processor cores with each other and allows access to the off-chip main memory over the *Memory Interface Controller* (MIC). This multiprocessor system on chip is used in the Sony PlayStation 3, where only six SPEs are available to the Linux operating system.

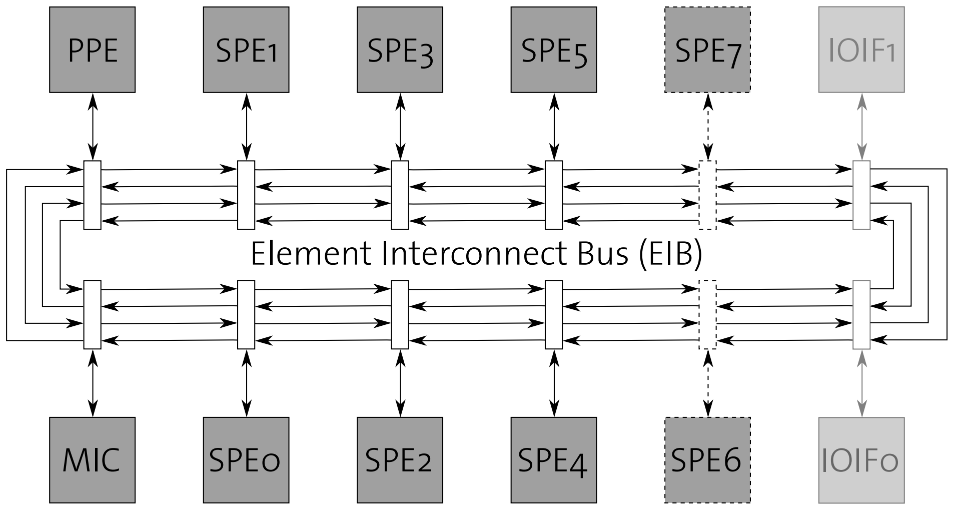


Figure 2.1: Systematic architecture overview of the Cell Broadband Engine.

2.1.1 Communication Architecture

Figure 2.2 shows an overview of the communication architecture of the Cell Broadband Engine. The *Memory Flow Controller* (MFC) is the interface between the SPE and the Element Interconnect Bus. Its main functionality is to support direct memory access (DMA) transfers between the main memory and the *Local Store* (LS) [8]. This 256 KB storage is used to accumulate instructions and data during the runtime of an SPE and can be described as scratchpad memory (software-controlled cache) [9].

In addition, the MFC allows an SPE to communicate with other processor elements using mailboxes. A mailbox is a unidirectional message-passing interface that supports exchanging 32-bit messages [9]. Each SPU uses its SPU Channel and DMA Unit to interact with the DMA controller (DMAC). There is an SPE channel assignment as well as a corresponding *Memory-Mapped I/O* (MMIO) register assignment for each mailbox.

Each SPE has three different mailboxes of two different types:

- Two 1-entry mailboxes for sending messages: *SPU Write Outbound Mailbox* and *SPU Write Outbound Interrupt Mailbox*.
- One 4-entry mailbox for receiving messages: *SPU Read Inbound Mailbox*.

The access to these mailboxes from the SPE and the PPE can be implemented in two different ways:

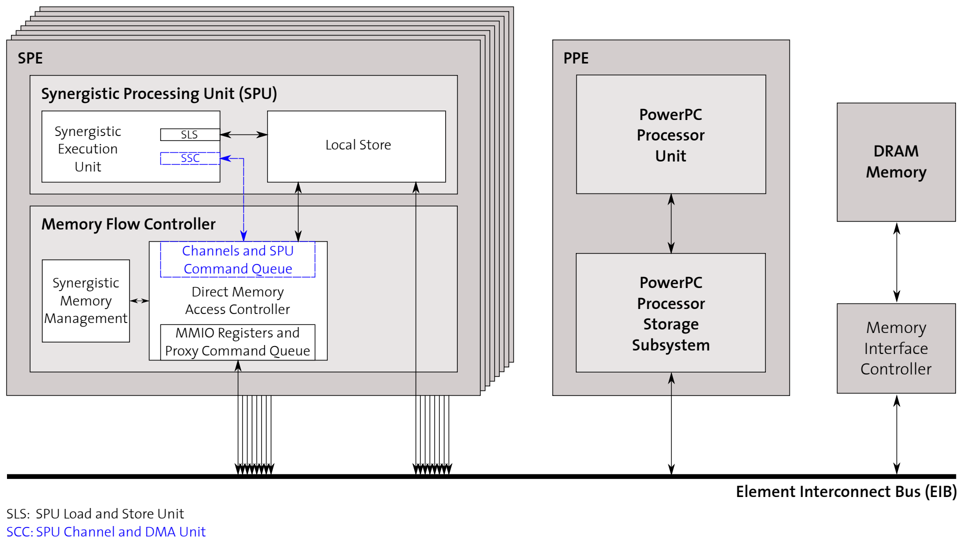

SPE
Synergistic Processing Unit (SPU)
Synergistic Execution Unit
SLS
SSC
Local Store
Memory Flow Controller
Synergistic Memory Management
Channels and SPU Command Queue
Direct Memory Access Controller
MMIO Registers and Proxy Command Queue
PPE
PowerPC Processor Unit
PowerPC Processor Storage Subsystem
DRAM Memory
Memory Interface Controller
Element Interconnect Bus (EIB)
SLS: SPU Load and Store Unit
SCC: SPU Channel and DMA Unit

Figure 2.2: Overview of the communication architecture.

- *Blocking:* A read of an empty / write to a full mailbox stalls the execution until a message is available.
- *Non-Blocking:* A read of / a write to a mailbox immediately returns. The return value indicates whether a message has been read / written.

2.2 System Model

Schranzhofer et al. [10] investigated the timing behavior of real-time tasks in multiprocessor systems accessing a shared resource such as buses, main memory or DMA controllers. They presented a way to analytically determine the *worst-case completion time* (WCCT) of tasks consisting of superblocks if the access policy to the shared resource is *time division multiple access* (TDMA). This section summaries the underlying system model and the calculation of the WCCT is summarized in the Section 2.3.

2.2.1 Architecture

A multiprocessor system contains several processing elements $p_j \in \mathcal{P}$ which might access a shared resource simultaneously, for instance an interconnect bus. Consider Fig. 2.3 for a schematic overview. A processing element executes a set of tasks independently from the other elements. These tasks are modeled according to Section 2.2.2 and the shared resource based on

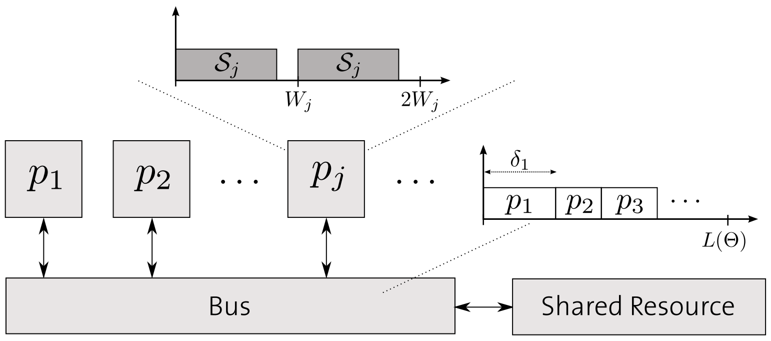


Figure 2.3: Overview of the presented system model [10].

Section 2.2.3. The resource arbitration is TDMA and configured with a specific schedule.

2.2.2 Task Model

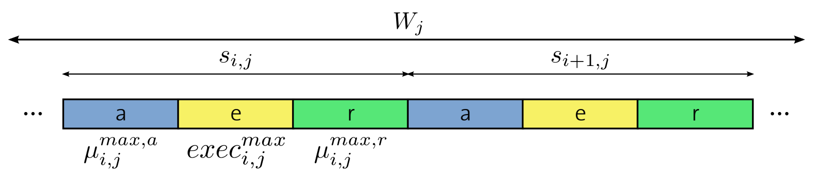
Like in [10], it is assumed that on each processing element p_j, a task τ_j is executed repeatedly with period W_j, called *processing cycle*. A task consists of non-preemptable superblocks $s_{i,j} \in \mathcal{S}_j$ which are defined as sequences of basic blocks[1].

Each sequence of superblocks is executed either *event-triggered* or *time-triggered*. Event-triggered or sequential execution activates the next superblock as soon as the previous has finished, whereas in the time-triggered case, a superblock is activated on predefined time instants. For TDMA arbitration, only sequentially executed superblocks are considered. The reason for this choice is that the experimental results in [11] showed that time-triggered execution of superblocks has no benefits regarding the predictability, and even worse, increases the WCCT.

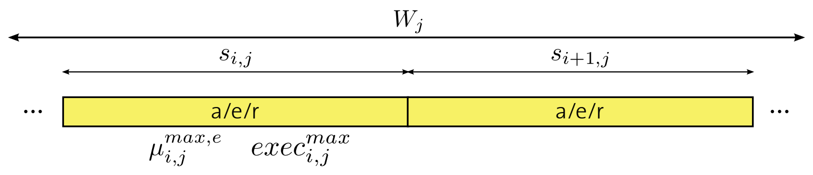
2.2.2.1 Access Models

Each superblock has an upper bound for the number of accesses to the shared resource and an upper bound for the computation time. A superblock can be divided into the three successive phases *acquisition*, *execution* and *replication*. Usually, a task reads data from the main memory within the acquisition phase, performs computation operations during the execution phase and transfers the calculated data back to the main memory in the replication phase.

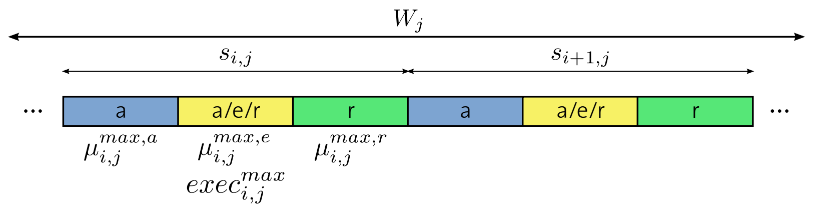
[1]A sequence of instructions that has exactly one entry and one exit point.



(a) *Dedicated access model* with parameters $\mu_{i,j}^{max,a}$, $\mu_{i,j}^{max,r}$ and $exec_{i,j}^{max}$



(b) *General access model* with parameters $\mu_{i,j}^{max,e}$ and $exec_{i,j}^{max}$



(c) *Hybrid access model* with parameters $\mu_{i,j}^{max,a}$, $\mu_{i,j}^{max,e}$, $\mu_{i,j}^{max,r}$ and $exec_{i,j}^{max}$

Figure 2.4: The three access models to the shared resource [10].

Access Model	$\mu^{max,a}$	$\mu^{max,e}$	$\mu^{max,r}$
Dedicated	≥ 0	$= 0$	≥ 0
General	$= 0$	> 0	$= 0$
Hybrid	≥ 0	> 0	≥ 0

Table 2.1: Relation between access parameters and the three access models.

The three different access models illustrated in Fig. 2.4 – *dedicated*, *general* and *hybrid* – describe the access pattern of a task to the shared resource. In the dedicated model, requests to the shared resource only happen in the dedicated acquisition and replication phase. The upper bound for the number of requests during a superblock $s_{i,j}$ of task τ_j is denoted as $\mu_{i,j}^{max,a}$ for the acquisition and $\mu_{i,j}^{max,r}$ for the replication phase. The general model allows accesses at any time and hence, the dedicated phases are omitted. The number of accesses is bounded by $\mu_{i,j}^{max,e}$. Finally, the hybrid model combines the general and the dedicated model.

For each of these models, $exec_{i,j}^{max}$ describes the maximal execution time of the computation operations. This upper bound excludes any communication time. The different models are distinguished based on the choice of the access parameters. This relation is outlined in Tab. 2.1. These parameters are usually referred to as cache profile [12]. For this thesis, the actual values are assumed to be determined, for instance by measurement.

2.2.3 Model of the Shared Resource

The shared resource is assumed to be blocking during the access operation and this results in stalling other processors. The time needed for a resource access can be described as *constant communication time* C. A frequently used arbitration policy in hardware systems is FCFS. One reason for introducing a static resource arbitration policy such as TDMA is to reduce the mutual interdependence of processing cores. This simplifies the performance analysis and therefore, increases system predictability.

For the TDMA policy, a schedule Θ has time slots assigned to processing elements. During its active slot with relative start time σ_m and length δ_m, a processing element p_j has the exclusive access right, and any other element that wants to operate on the shared resource has to wait until its assigned slot becomes active. If a processing element starts a resource request at a time instant where the remaining time of its slot is insufficient to serve the access operation, it has to wait for the next active slot. Thus, in order to process any resource request at all, a time slot must be at least of length C.

A scheduler that assigns exactly one slot to each processing element is called *regular* [11], and consequently, its total length $L(\Theta)$ can be expressed as $\sum\limits_{\forall j} \delta_{p_j}$. After this TDMA cycle, the schedule is repeated.

2.2.4 Worst-Case Execution Time

Based on the presented parameters in this section, the resulting worst-case execution time (WCET) of a task τ_j in isolation can be expressed as follows:

$$wcet_j = \sum_{i=1}^{|\mathcal{S}_j|} \left(exec_{i,j}^{max} + (\mu_{i,j}^{max,a} + \mu_{i,j}^{max,e} + \mu_{i,j}^{max,r}) \cdot C \right) \tag{2.1}$$

This bound would be a valid upper bound for a task where no interference on the shared resource can happen, e.g. if it is executed on a single processor. However, for more than one processing elements accessing the shared resource, the possible delays due to contention have to be considered. The steps needed to obtain the WCCT of a task for a multiprocessor system sharing a common resource is shown in Section 2.3.

2.3 Worst-Case Completion Time for TDMA

This section shows the basic ideas of how to compute the WCCT of a superblock. The detailed analysis and the exact algorithms can be found in [10]. The WCCT of a task consisting of superblocks is simply the sum of WCCTs for each superblock. The calculation of the acquisition / replication phase differs from the calculation of the execution phase. For the analysis, it is sufficient to consider one task under analysis and the TDMA schedule. The essential parameters are as follows:

- The TDMA schedule Θ including the total length $L(\Theta)$.
- The total number of resource requests μ^{max} during a phase for each superblock.
- The maximal execution time $exec^{max}$ without resource accesses for each superblock.
- The communication time C to the shared resource.
- The starting time t of a superblock.

2.3.1 WCCT for an Acquisition / Replication Phase

For the dedicated phases, there is no execution and the WCCT is the time needed to perform all μ^{max} requests to the shared resource. These requests are processed during the assigned time slots of the processing element. Based on the current time t, the WCCT is obtained by incrementing the time until the shared resource is available, invoke as many requests as can be served during the active time of a slot and repeat this procedure until there are no more remaining requests. The resulting ending time is the WCCT.

2.3.2 WCCT for an Execution Phase

Analyzing the WCCT of a general phase is more involved, because both communication and execution are possible. As a result, if there are pending requests for both operations, the question is whether to invoke an access request or to perform executions. Therefore, the main idea for calculating the WCCT is to find the trace of operations that maximizes the WCCT. Basically, the strategy is to waste the active slot with execution instead of resource accesses and to stall the execution of the processing element by initiating a resource request once the availability of the shared resource has expired. The algorithm in [10] is recursive. At any time, there are two cases that have to be handled differently.

(1) If the shared resource is not available, the completion times for invoking a request as well as performing computations until the next assigned slot becomes active are calculated. The larger of these two values leads to the worst case.

(2) On the other hand, if access to the shared resource is currently possible, the time is set to the beginning of the next time slot.

- If all computations and one resource request can be performed until the next active slot, the problem can be reduced by setting the current time to $t + exec^{max}$ and calculating the WCCT for an acquisition / replication phase for the remaining requests.
- Otherwise, the situation is considered, where exactly one request can be served per time slot, i.e., computations are performed until the latest possible time instant for issuing one access request. The completion time can then be derived by finding the last possible time slot when the remaining computation time becomes zero. Afterwards, the remaining requests can be handled again by calculating the WCCT for an acquisition / replication phase.

  For an irregular TDMA schedule, a processing element can have more than one assigned slots. As a result, it is not directly possible to retrieve the WCCT. For this case, the described completion time is an upper bound and a tighter WCCT can by derived by first finding a corresponding lower bound and then applying binary search to find a valid completion time between these two bounds.

3 TDMA Scheduling on the Cell Broadband Engine

3.1 Introduction

The on-going trend towards multiprocessor systems not only holds for general computing systems but includes real-time systems as well. The challenge for systems with strict timing constraints is to find upper bounds for the completion time of tasks, usually referred to as the worst-case completion time. This is necessary because in such systems, in time termination of tasks is as important as their actual objective.

One of the biggest contributions to the WCCT in multicore platforms are the interferences that happen if several processors access shared resources, such as shared memories, buses and DMA controllers. An access to a shared resource occupied by a processor may block the execution of any other processor until it becomes available again. Schranzhofer et al. [10] developed an analytical method to determine the WCCT of periodic real-time tasks that are modeled as sequences of superblocks (introduced in Section 2.2), where the arbitration policy to the shared resource is TDMA.

The main question that is attempted to be answered in this chapter is how well these analytical methods can be used for commercial systems and components-of-the-shelf (COTS), such as the Cell Broadband Engine [6]. Such platforms are normally optimized for good average performance. In

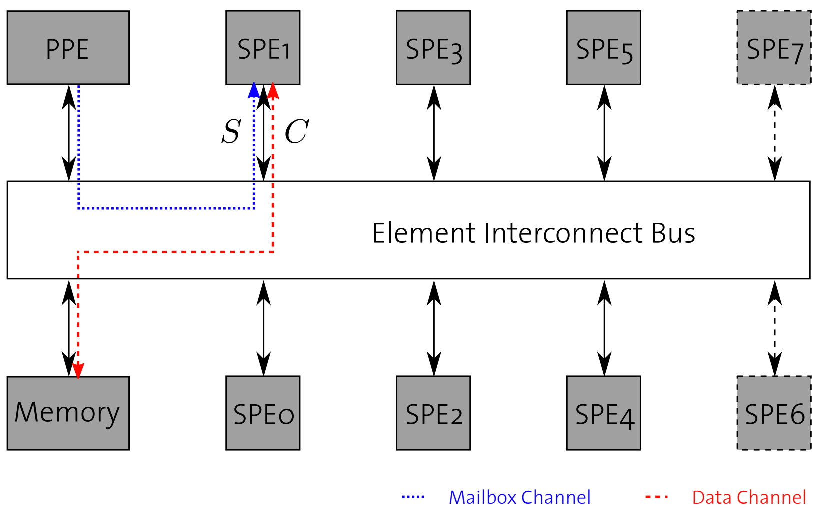


Figure 3.1: The communication channels of the Cell Broadband Engine. The time for sending two messages is bounded by S, the completion of a DMA transfer by C.

contrast, timing predictability is required for hard real-time systems. The behavior of TDMA scheduling implemented on the Cell Broadband Engine is investigated, and values obtained by measurement are compared with results obtained from the analytical model.

3.2 Approach

First, this section shows how the system model presented in Section 2.2 can be applied to the Cell Broadband Engine and which extensions are required. Consider Fig. 3.1: The PPE is used as scheduler allowing each SPE to access the main memory based on the configured schedule Θ. Therefore, a way to synchronize the processing elements with the resource arbiter is necessary and presented in Section 3.3.

Second, the synchronization of the processing elements as well as their communication to the shared resource involve the Element Interconnect Bus. Hence, there is a *synchronization overhead* for sending messages to an SPE, which influences each slot length. The upper bound for sending two synchronization messages is denoted as S. The time needed for a memory access can be bounded by the *communication time* C. The determination of S and C is shown in Section 3.4.

Finally, in Section 3.5, the experimental setup is explained and the *maximal observed completion times* of sample tasks are compared with the *predicted* *worst-case completion times*. Conclusions summarize this chapter.

3.2.1 Restrictions of the System Model

The system model in [10] assumes that only one request can be served by the shared resource at any time. This is apparently not true for the Element Interconnect Bus – it contains four 16-byte-wide data rings allowing parallel communication over this topology. Each SPE can initiate several DMA transfers simultaneously to other SPEs. So the assumption of blocking behavior is implemented by the resource arbiter: An SPE initiating a DMA transfer has to wait until its assigned slot becomes active. Furthermore, during the completion of a DMA request, an SPE is not allowed to perform computations. This is important because the decoupling of execution and communication is one of the requirements of [10].

Moreover, the communication time is clearly not constant on the available hardware system – the time needed for a DMA transfer varies for each request, which is further discussed in Section 3.4. In this context, the constant communication time C is an upper bound and the average time consumption of a DMA transfer lies below C resulting in a pessimistic prediction of the worst-case completion time.

3.2.2 Extensions of the System Model

The PPE uses the mailbox interface to communicate with the SPEs, mainly to inform them about the availability of the main memory. Therefore, the synchronization overhead S between the arbiter and any processing element has to be taken into account. It is two times the upper bound of the sending delay between PPE and SPE:

$$S = 2 \cdot Ub_{send} \tag{3.1}$$

This issue will be discussed in detail in Section 3.4. One delay occurs at the beginning of a slot for sending a *Slot Start* message and one at the ending for sending a *Slot End* message.

To avoid interferences with other SPEs during a DMA transfer, the PPE sends the *Slot End* message $\frac{S}{2} + C$ time units before the actual ending of the current slot. This guarantees that an SPE invoking a DMA transfer at the latest possible moment of its active slot is not affected by the next SPE in the TDMA cycle.

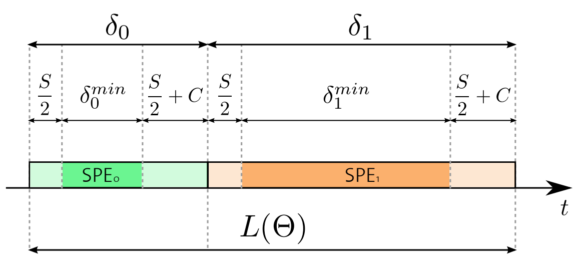


Figure 3.2: Example of the minimum slot lengths of two SPEs.

Since the time needed for sending the synchronization messages varies for each slot between $(0, S]$, the effective slot length of the SPE is different each time. In the worst case, it is assumed that sending the *Slot Start* message takes the full $\frac{S}{2}$ units of time, whereas the SPE receives the *Slot End* message immediately. Therefore, the minimum slot length available to the SPE can be expressed as:

$$\delta_m^{min} = \delta_m - C - S > 0 \tag{3.2}$$

$$\delta_m > C + S \tag{3.3}$$

Additionally, one of the requirements in [10] is that during an active time slot, the processing element should have access to the shared resource at least for one time. This is ensured by the PPE by configuring each slot length, such that it should endure at least $C + S$ time units. The length δ_m^{min} is taken as slot length during the worst-case analysis, while the total scheduling length $L(\Theta)$ remains unchanged. This relation is outlined in Fig. 3.2.

3.3 Framework

An overview of the scheduling framework implemented on the Cell Broadband Engine is illustrated in Fig. 3.3. The PPE is responsible for granting access to the main memory using a TDMA schedule Θ. It uses mailbox messages to communicate the availability of this shared resource to the SPEs. As soon as an SPE has received a *Slot Start* synchronization message **(1)**, it is allowed to invoke a DMA transfer **(2)**. This exclusive access right is valid until the reception of a *Slot End* synchronization message **(3)**.

Therefore, one task of the PPE is to send these messages at the correct time instants. The clock domain of the PPE is used for this purpose (see

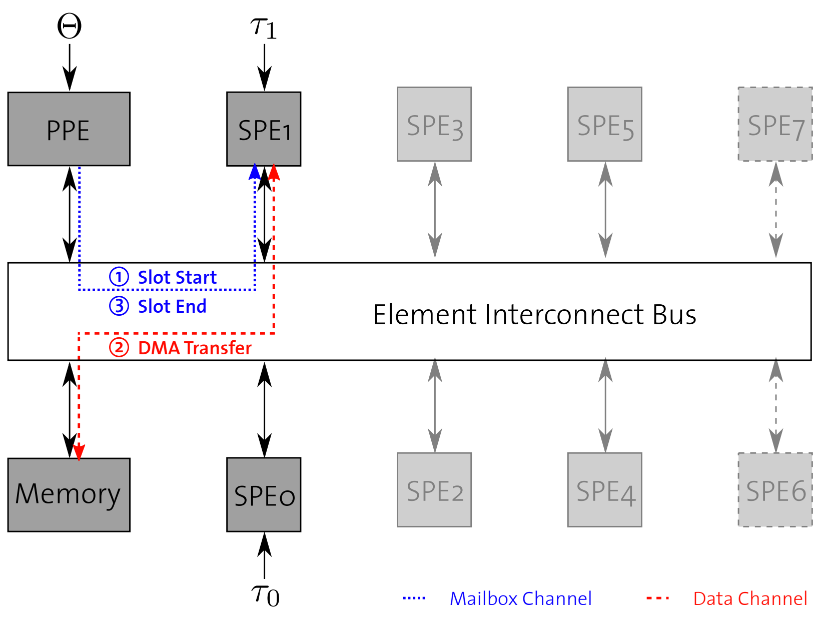
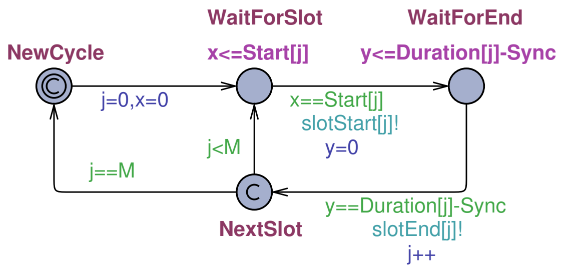


Figure 3.3: Overview of the scheduling framework on the Cell Broadband Engine. In this example, the PPE is configured with schedule Θ, and SPEs 0 and 1 are executing task τ_0 and τ_1 respectively.

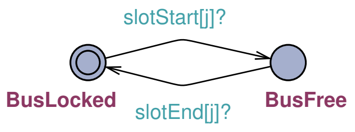
Appendix A, Section A.2): There is a single controller with a single clock, thus no global time needs to be maintained. The synchronization protocol is shown in Fig. 3.4 represented using timed automata for the PPE and the SPE with clock variables `x` and `y`. The value `Sync` is set to $\frac{S}{2} + C$ and `M` to $M(\Theta)$.

The *Slot Start* message is sent right at the beginning of the next active slot (`WaitForStart`), whereas the *Slot End* message $\frac{S}{2} + C$ time units before its ending (`WaitForEnd`). The latter guarantees that a resource request invoked on SPE p_j immediately before the reception of the *Slot End* message does not interfere SPE p_{j+1}, because the transfer is finished at worst after C time units.

An SPE is used as processing element p_j that periodically executes a preconfigured task τ_j consisting of a set of superblocks. A superblock may contain memory accesses as well as computations. If a resource request is started, the SPE has to wait until the access is allowed (`BusFree`). This results in stalling the execution. The time needed for the whole task is measured on the SPE and the resulting maximal observed completion time can be compared with the predicted WCCT of τ_j.



(a) TA for the PPE



(b) TA for the SPE

Figure 3.4: Timed automata for the scheduler (PPE) and for the processing element p_j (SPE).

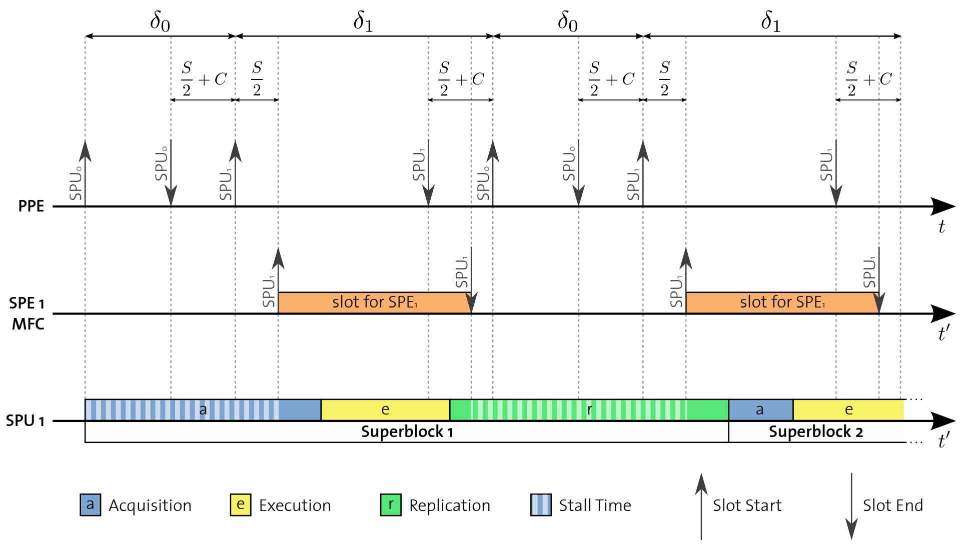


Figure 3.5: Example of TDMA scheduling of two SPEs. The superblocks of SPE 1 access the shared resource within the dedicated phases.

3.3.1 Execution of the Scheduler on the PPE

The PPE is used as the resource arbiter granting access to the main memory. Each SPE is only allowed to access the memory during its assigned slot. In addition to the scheduling, the PPE controls and configures all SPEs. The scheduler is fully configurable through an XML file, which is processed during the initialization of the framework. In particular, the important parameters for the PPE are the synchronization overhead S, the communication time C, the TDMA schedule Θ and the total runtime.

After the start-up and synchronization phase, the actual scheduling takes place. An example with two SPEs can be found in Fig. 3.5. The top axis shows the time instants, when the synchronization messages are sent from the PPE to the available SPEs. The middle one illustrates the arrival of messages intended for SPE 1 in its inbound mailbox, which is located within the Memory Flow Controller (MFC). The processing of superblocks is sketched on the bottom axis, and in addition, the inactive time of an SPE, called *stall time*, is highlighted.

The time-triggered sending of *Slot Start* and *Slot End* messages is implemented by using absolute clock values (mentioned in Appendix A, Section A.2). The next clock value for sending a message is calculated during the idle time, i.e., during the active slot, and the PPE busy-waits until this time instance.

3.3.2 Execution of Superblocks on the SPE

An SPE acts as the processing element p_j and regularly repeats superblocks with period W_j. The time-triggered activation of this period is implemented in the same way as the time-triggered mechanism of the PPE.

During the runtime of an SPE, the reception of the synchronization messages is isolated from the execution with an interrupt handler. This event-triggered approach minimized the synchronization overhead (see Appendix A, Section A.3).

For each SPE, a sequence of superblocks $\mathcal{S}_j$ constitutes the task τ_j. A superblock $s_{i,j}$ can be defined as:

$$s_{i,j} = \Big(\{\mu_{i,j}^{max,a}\},\ \{\mu_{i,j}^{max,e},\ \lambda_{i,j}^{exec}\},\ \{\mu_{i,j}^{max,r}\}\Big) \tag{3.4}$$

$\mu_{i,j}^{max}$ denotes the maximal number of DMA requests during a phase. However, the execution time assumed in the theoretical model cannot directly applied to the SPE. Computer systems are of discrete nature and consequently, operations require discrete computation times. We model this as discretization by executing a basic calculation with *constant execution time* $E_{i,j}^{calc}$. The total number of calculations is denoted $\lambda_{i,j}^{exec}$. The required value $exec_{i,j}^{max}$ for the performance analysis can be retrieved as follows:

$$exec_{i,j}^{max} = \lambda_{i,j}^{exec} \cdot E_{i,j}^{calc} \tag{3.5}$$

The PPE configures each SPE according to the user-specified XML file. A user-defined number of slots with specified length δ_{p_j} can be assigned to each SPE p_j, as well as the processing cycle W_j and the sequence of superblocks $\mathcal{S}_j$.

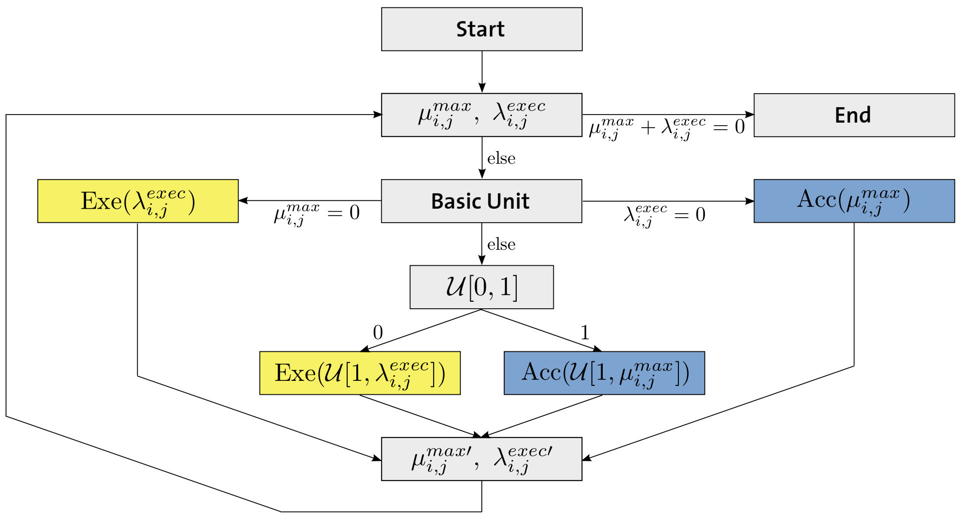

Start
$\mu_{i,j}^{max}$, $\lambda_{i,j}^{exec}$
$\mu_{i,j}^{max} + \lambda_{i,j}^{exec} = 0$
End
else
Exe($\lambda_{i,j}^{exec}$)
$\mu_{i,j}^{max} = 0$
Basic Unit
$\lambda_{i,j}^{exec} = 0$
Acc($\mu_{i,j}^{max}$)
else
$\mathcal{U}[0,1]$
0
1
Exe($\mathcal{U}[1, \lambda_{i,j}^{exec}]$)
Acc($\mathcal{U}[1, \mu_{i,j}^{max}]$)
$\mu_{i,j}^{max\prime}$, $\lambda_{i,j}^{exec\prime}$

Figure 3.6: Generation of superblocks by iteratively executing *Basic Unit*, $\mathcal{U}[a, b]$ denotes an independent identically distributed (i.i.d.) number between a and b, Exe(λ) the execution of λ basic calculations, Acq(μ) and Req(μ) the execution of μ memory requests.

3.3.2.1 Generation of Superblocks

A superblock $s_{i,j}$ is generated on the SPE during runtime and is the outcome of several random decisions. Consider the diagram in Fig. 3.6: Based on the numbers $\mu_{i,j}^{max}$ and $\lambda_{i,j}^{exec}$, a *Basic Unit* decides randomly between execution and memory transfers if there are pending requests for both operations. The effective number of operations that are performed in the current run is again a random variable with discrete uniform distribution $\mathcal{U}$ between 1 and the number of unserved requests.

Otherwise, all calculations and memory transfers are processed, respectively. After the execution of a Basic Unit, the remaining numbers $\mu_{i,j}^{max\prime}$ and $\lambda_{i,j}^{exec\prime}$ are used as input parameters for the next iteration. This process is repeated until all requests have been executed.

Since the same communication overhead C is taken for DMA put and get requests, they do not have to be treated separately for the calculation of the WCCT. Consequently, it would be possible to use either get or put operations. Since both operations happen in real-world applications, the choice is made randomly each time.

For each processing cycle, the time needed for executing the whole task is measured within the clock domain of the SPE (see Appendix A, Section A.2). This is possible since only relative delays are considered. The maximal

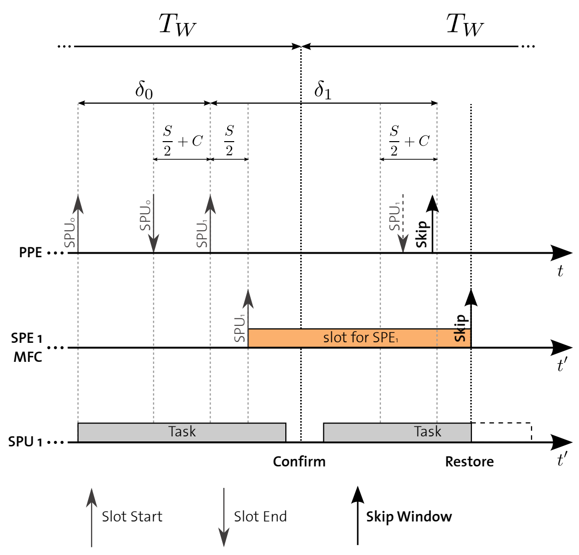


Figure 3.7: Visualization of the time window with period T_W.

observed completion time is stored and after finishing the experiment, each SPE transfers its maximal observed completion time to the PPE. If the task τ_j executed on the SPE p_j exceeds the processing cycle W_j, the execution on the particular SPE is aborted.

3.3.3 Time Window

Scarcely, the process on the PPE executing the scheduling seems to be suspended for an unpredictable amount of time. Real-time process scheduling [13] may reduce this occurrence, but nevertheless, it is not possible to avoid it completely. As a consequence, correct synchronization messages are sent too late. If this happens, the time measurements on all the SPEs are influenced. In an unfavorable case, this can lead to the maximal observed completion time, which does not really represent the worst case observed under normal circumstances.

In order to overcome this inaccuracy, a *time window* with period T_w has been introduced – an additional, adjustable parameter. It is started at the beginning of the execution both on the SPEs and the PPE. After the successful expiration of this duration, all measured values on any SPE are confirmed and the period T_w is repeated. If, however, the scheduling process on the PPE is suspended too long, it immediately informs all SPEs about this incident. The observed values of the current time window are omitted and the execution is stalled until the beginning of the next time window. An example is sketched in Fig. 3.7: The first period T_W is finished successfully, whereas during the second T_W, the *Slot End* message would be sent too late. Instead, a *Skip Window* messages informs all SPE to restore their measured values and to stop the current task.

This work-around is a restriction and used in order to retrieve meaningful completion times. However, the chosen approach is just appropriate for this uncritical test scenario. In a critical real-time system, a deadline violation must be prevented.

3.4 Timing Behavior of the Element Interconnect Bus

To implement a time-triggered protocol on top of a multiprocessor, the individual processors need to be synchronized. Consider the case where the PPE is used as resource arbiter: It needs to communicate the availability of the shared resource on determined time instances. Unfortunately, the processors on the Cell Broadband Engine do not have a global time register that could be used for this purpose. Therefore, explicit software is required to synchronize the SPEs. This potentially consumes a significant amount of processing resources that are otherwise available to user applications. Consequently, this run-time overhead, denoted as *synchronization overhead S*, needs to be taken into account during the design and performance analysis of multiprocessors.

Focusing on this issue, this section first presents a possibility for implementing the synchronization between the PPE and the SPEs by exchanging synchronization messages. Afterwards, this implementation is characterized by determining bounds on the round-trip time (RTT) of these messages. More precisely, we are interested in the worst-case sending delay between the PPE and the SPE. Since two messages are required per TDMA slot, the resulting synchronization overhead S is two times the upper bound of this sending delay (see Eq. (3.1)).

Another important parameter concerning the Element Interconnect Bus is the *communication time C*, i.e., the time needed for serving a DMA request. For this thesis, this operation is implemented with blocking-behavior because the system model in Section 2.2 assumes only one memory access per processing element at a given time. Additionally, the SPE is stalled until the DMA transfer has been completed.

Both bounds have to be determined by experimental measurements. Several papers have been published that include performance analysis for the Element Interconnect Bus of the Cell Broadband Engine, which connects the different processors with each other and with the main memory. The work of Kistler et al. [7] was one of the first ones that describes the underlying architecture and presents experimental results mainly for direct memory transfers executed on early hardware prototypes. A more recent paper of Jos L. Abellán et al. [8] provides an evaluation tool for characterizing the synchronization and communication functionalities of the Cell Broadband Engine. Both papers concentrate on the average performance of the communication network, which differs significantly from the maximal observed values as shown in this section.

3.4.1 Measuring the Round-Trip Time

The challenge of measuring the latencies of the mailbox functions is that two different clock domains are involved – the clocks at the source and the destination processing element. It is not possible to obtain the exact time for sending or receiving a message from one to another processing element: Returning from the sending function does not guarantee that the message has received its destination nor that it has already been processed. Therefore, the round-trip time needed to send a message from the PPE to the SPE's Inbound Mailbox and receiving the acknowledgement from the SPE's Outbound Mailbox is measured.

The SPE waits for a new message and upon reception, the acknowledgement is directly sent back. This step is called *iteration* and it is repeated until the *number of iterations* has been reached. The whole protocol is illustrated in Fig. 3.8. The initial step is only necessary in order to synchronize the SPE with the PPE, i.e., to avoid measuring the wrong delay for the first iteration. Without this synchronization, the SPE would start measuring the receiving delay before the PPE is ready to send the first message. Thereby, the number of iterations is communicated to the SPE. This number determines how often the SPE reads the Inbound Mailbox and acknowledges the reception of the message.

The time needed for the mailbox operations on the PPE is measured using the time-base register (see Appendix A, Section A.2). The value read from this register is converted from clock ticks to time units. Based on this measurement, the RTT and upper bound of the sending delay is obtained. All the involved mailbox functions for receiving and for sending messages are called with blocking behavior. This means that reading an empty mailbox or writing to a full mailbox causes the SPE to stall until the operation successfully completes.

A test application has been implemented which measures the sending and the receiving delays needed by the mailbox functions. The usage of the

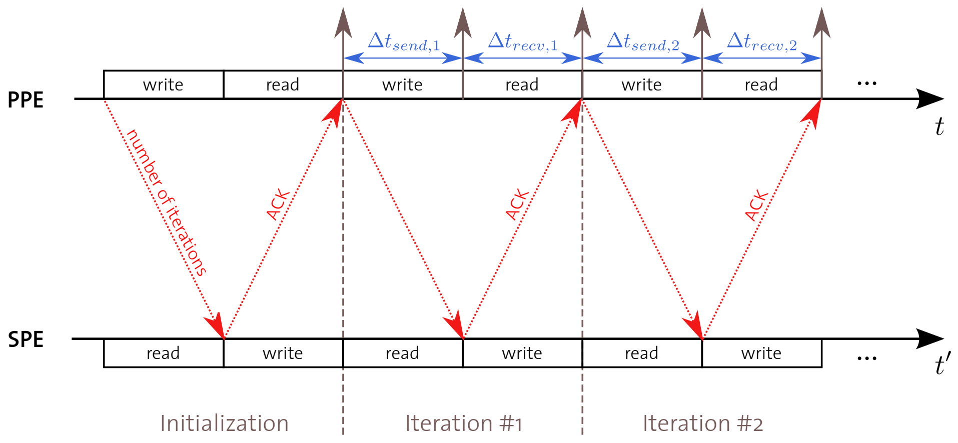


Figure 3.8: Time diagram for measuring the round-trip time.

mailboxes is illustrated in Appendix A, Section A.1. Basically, two different ways of accessing the mailboxes are distinguished: over SPU channels (*LIB SPE*) or over memory-mapped I/O (*CBE MFC*). The time measurement is implemented according to Appendix A, Section A.2.

3.4.2 Measuring the Communication Time

Each SPE can initiate a DMA request to transfer data from its local store to the main memory (PUT) or vice versa (GET). The data is asynchronously transfered between the SPE and the main memory, i.e., the execution of the SPE is stalled until the data is available. The timing analysis is easier compared to the previous one because only one clock domain is involved. The results from [7] and [8] indicate that the DMA operation latency depends on the number of requesting SPEs. Since the TDMA scheduler only allows access to one SPE, we are interested in the transfer time for serving one DMA request from one particular SPE while no other SPE is using the EIB.

Furthermore, the DMA latency depends on the number of transfered bytes (1, 2, 4, 8 and multiple of 16 bytes). Peak performance is possible with 128-byte aligned data and transfer sizes that are multiple of 128 bytes [9]. In this thesis, the transfer size is fixed to 128 bytes (which is equivalent to 16 double-precision floating-point values on the Cell Broadband Engine).

3.4.3 Evaluation

The experiments for measuring the round-trip time and the DMA latencies are executed on the PlayStation 3 and the IBM Full-System Simulator. The operating system of the PlayStation 3 is Yellow Dog Linux Release 6.0 with kernel version 2.6.23-9 and the Simulator runs with a Fedora 9 with kernel version 2.6.25.14-108. Since an exact time measurement is required, the Simulator has to run in cycle-accurate mode. Hence its runtime is many times larger compared to the PlayStation 3 and practically not feasible for a large number of iterations.

The usage of the measurement tools is summarized in Appendix B, Section B.2. Basically, they offer two different output modes: Running the applications in *output* mode collects the observed delays for each iteration and calculates the maximum, the average and the estimated variance values[1]. In *silent* mode, no iteration is saved and only the calculation is done. The *silent* mode is intended for a large number of iterations, whereas the resulting files for the *output* mode can be further processed.

3.4.3.1 Average and Worst-Case RTT

An experiment with 10^{10} iterations in *silent* mode executed on the PlayStation 3 showed that the average performance is reasonable: The observed average round-trip time is $8.12\,\mu s$ for accessing mailboxes through SPU channels (SPE library calls), which is in the same order of magnitude than the $6\,\mu s$ found out in [8]. For the direct access through Memory-Mapped I/O registers (CBE MFC library calls), the value is $0.36\,\mu s$, so this method is approximately 22 times faster. Another advantage of the direct approach is the reduced variance of the delay times.

The comparison between the performance of the mailbox functions for the SPE library and of the CBE MFC library are presented in Tab. 3.1. The experiments have been executed on several SPEs and this variation did not change the outcome of the measurement.

The time consumed by invoking a mailbox function should only vary within a small range if it should be used for the purpose of synchronization. Unfortunately, the mailbox functionality of the Cell Broadband Engine is designed for good average results [8], whereas the maximal delay time is more than 10^5 larger than the average.

3.4.3.2 Statistical Evaluation of the RTT

Running an experiment in *output* mode gives a deeper insight how the measured delays are distributed. The resulting plots of an experiment with

[1]The retrieved values are directly written to a file or standard output and therefore, an on-line algorithm is used to estimate the variance.

Sending	CBE MFC	LIB SPE
Average	0.14 μs	4.39 μs
Maximum	853 μs	27680 μs
Variance	$8.55{\cdot}10^{-3}$	1.88

Receiving	CBE MFC	LIB SPE
Average	0.22 μs	3.73 μs
Maximum	6248 μs	53722 μs
Variance	$21.81{\cdot}10^{-3}$	1.98

RTT	CBE MFC	LIB SPE
Average	0.36 μs	8.12 μs
Maximum	6248 μs	53729 μs
Variance	$30.39{\cdot}10^{-3}$	3.93

Table 3.1: Measured delays for sending and receiving mails on the PPE as well as the resulting round-trip time.

10^7 iterations can be found in Fig. 3.9 for the CBE MFC library and in Fig. 3.11 for the SPE library. The dashed lines mark the minimum and the maximal value, the dot-dashed line the 99.999 % bound, i.e., the range where 99.999 %of the measured delays are located.

It is evident that the major part of the delays is around the average value and only a very small number is significantly larger. The observed values are distributed within the range of 0 μs up to 250 μs. And the previous section showed that this range is even larger if the number of iterations is increased.

The bound between the lower 99.999 % and the upper 0.001 % part of the distribution has a practical reason: Fig. 3.10 and Fig. 3.12 illustrate the trade-off between performance (in terms of worst-case execution time) and reliability. In this context, 100 % reliability means that all delays are less or equal to the maximal value measured in the same experiment, although larger values will occur in different experiments. The choice of the highest measured delay as upper bound is obviously a suboptimal solution regarding the performance.

Moreover, it is actually not possible to obtain an upper bound by experimental measurements. On the other hand, taking the WCET of 99.999 % reliability is a good compromise. This value is referred to as high-availability of a computer system [14]. This term has been introduced as with the growing size of computer systems, every aspect of their environment became more complex, too. This led to the situation, where such systems were less likely to be highly available.

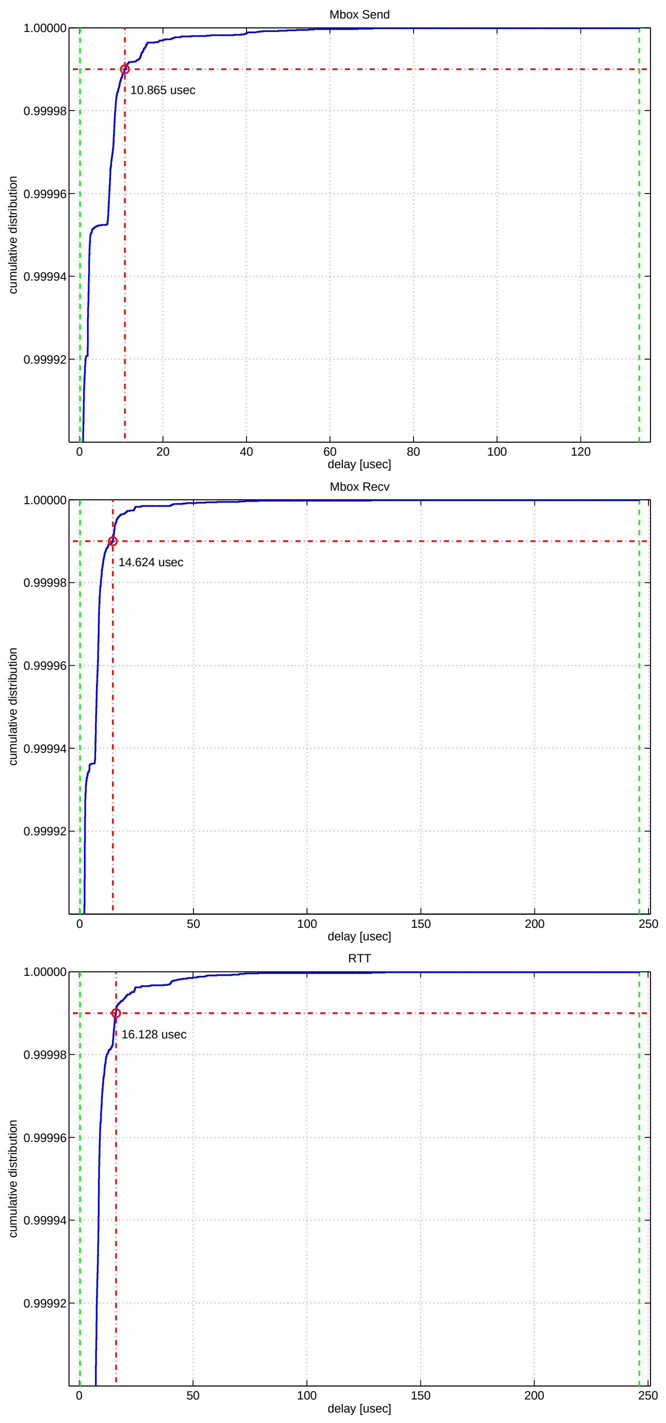


Figure 3.9: Distribution of the measured delays using the CBE MFC library.

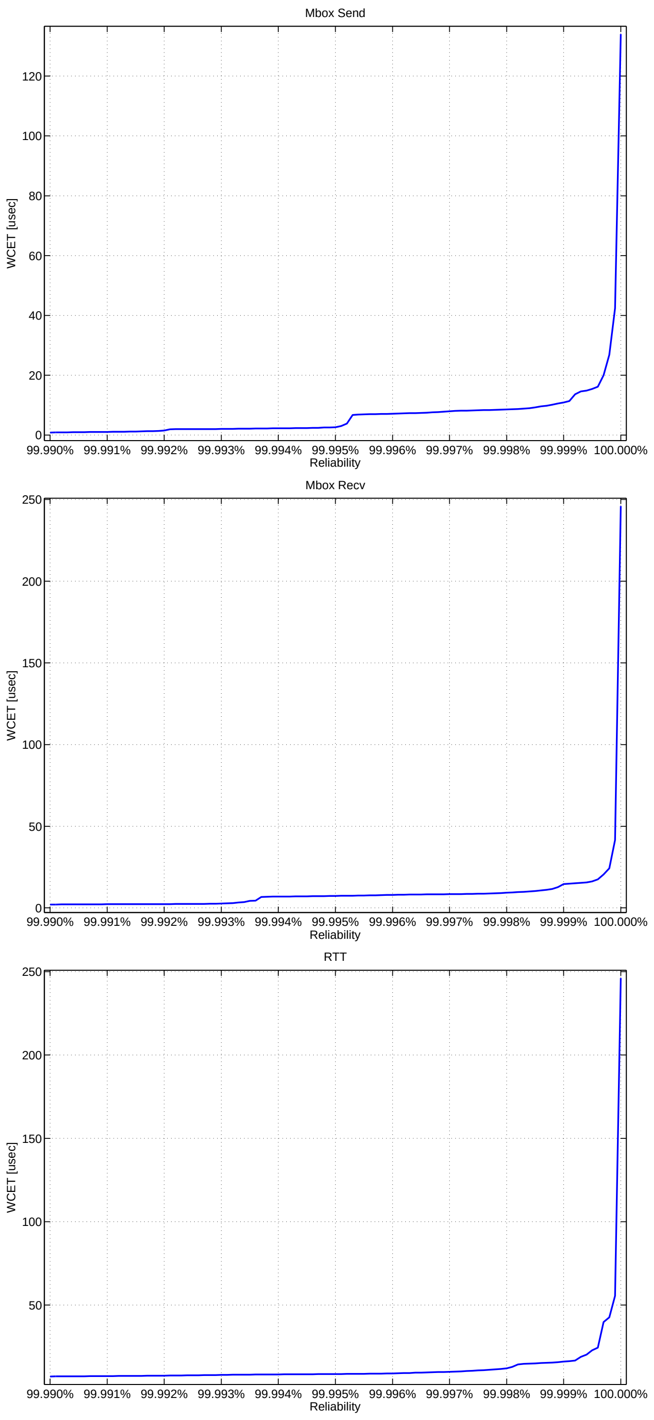


Figure 3.10: Trade-off between performance (WCET) and reliability for the CBE MFC library.

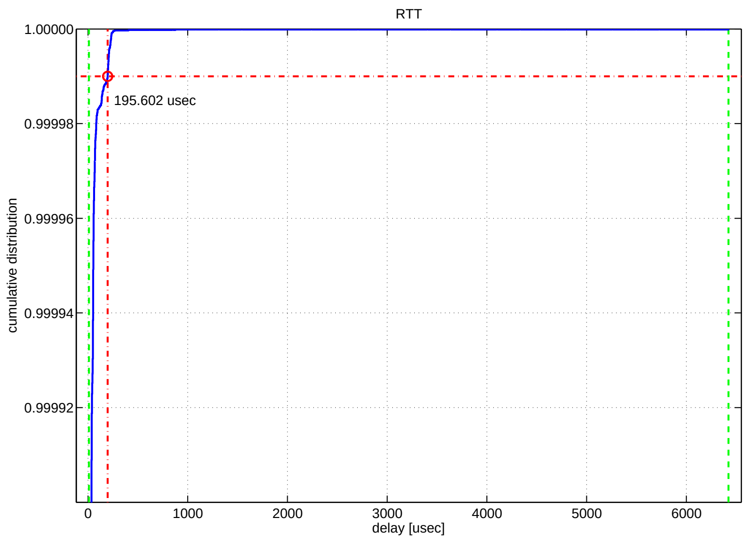


Figure 3.11: Distribution of the measured round-trip time using the SPE library.

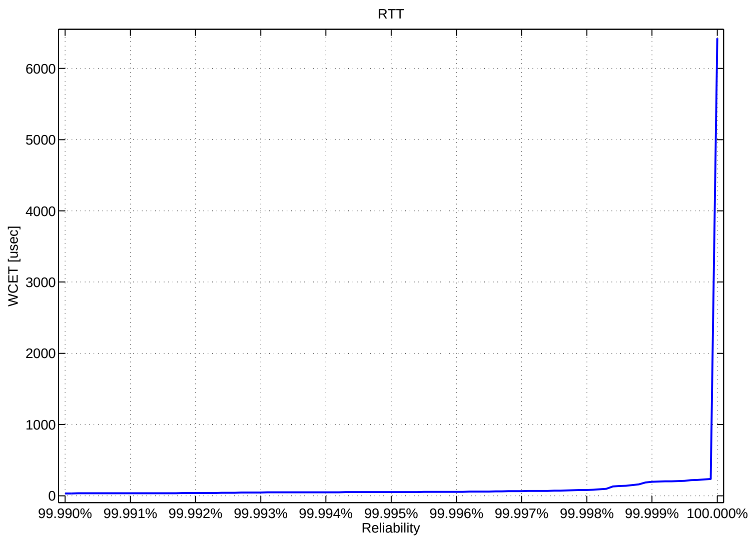


Figure 3.12: Trade-Off Between Performance (WCET) and Reliability for the SPE Library.

	DMA GET	**DMA PUT**
Average	0.199 μs	0.125 μs
Maximum	9.198 μs	0.163 μs
Variance	0	0

Table 3.2: Measured DMA transfer latencies for GET and PUT operations.

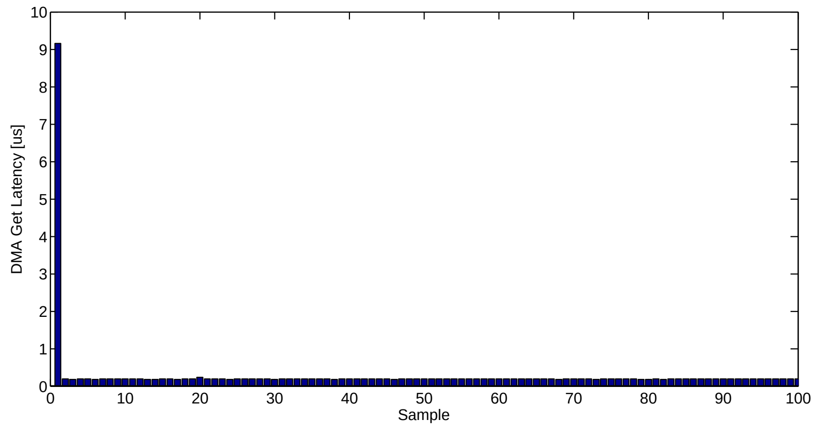


Figure 3.13: Measured DMA transfer latencies for 100 GET operations.

3.4.3.3 DMA Transfers

The experiments for measuring the DMA latencies between an SPE and the main memory were executed on the PlayStation 3 for GET and PUT operations. Table 3.2 summarizes the results for 10^7 iterations in *silent* mode. Another experiment in *output* mode was performed in order to plot the distribution of the measured values. The measured latencies for the first 100 GET operations are shown in Fig. 3.13. It is remarkable that the variance is actually zero and only the first performed DMA request consumes the maximal measured value. The result would be similar for the PUT operation if performed before the GET operation since it accesses exactly the same memory addresses.

The reason for the reduces transfer times for the subsequent DMA transfers is that the Memory-Flow Controller (MFC) of the SPE contains a translation look-aside buffer (TLB) [15]. The main memory is virtually structured with pages. Consequently, it is necessary to translate the virtual addresses to real addresses. This TLB was designed in order to accumulate this address translation by caching recently accessed page table entries. Nevertheless, the upper bound of the communication time has to be larger than the maximal measured value because the TLB cache could be invalidated at any time.

3.4.4 Conclusions

The mailbox functionality of the Cell Broadband Engine provides an easy-to-use possibility for synchronizing several SPEs. It is designed for a good average performance, but the experiments showed that the delay drastically increases in the worst case compared to the average case.

The straightforward approach of accessing the mailboxes from the PPE is to use the runtime management library (LIB SPE). But the experiments clearly showed that the required runtime can be reduced by directly accessing the memory-mapped mailbox registers with the Memory Flow Controller library (CBE MFC).

The measured delays between PPE and SPE by using mailbox operations are distributed over a wide range. Therefore, finding an upper bound S for the synchronization overhead is hard. Moreover, taking the maximal observed value as upper bound increases the overhead, because for each message, this time consumption is assumed. This is why a value of at least $2 \cdot 10.865\,\mu s \approx 22\,\mu s$ should be chosen which offers 99.999 % reliability and reduces the resulting overhead.

Apart from that, finding the upper bound C for the communication time is less difficult. The variance of the measured latencies is almost zero and the values are distributed over a smaller range compared to the synchronization overhead. The efforts that were needed for S are not necessary for C. All in all, a minimal value of $10\,\mu s$ as upper bound is reasonable.

3.5 Experiments

3.5.1 Experimental Setup

The tasks for the experiments are taken from [11], where sets of superblocks are generated with random access numbers for all phases. These tasks representing the *hybrid access model* are transformed to the other access models by moving the accesses from the acquisition / replication phase to the execution phase for the *general access model* and vice versa for the *dedicated access model.*

For these experiments, the number of basic calculations λ^{exec} for each superblock is chosen in the way that $exec^{max} = \lambda^{exec} \cdot E^{calc}$ is not larger than the maximal execution time of the given superblocks from [11]. Because the total number of accesses and the total number of calculations is the same for

Task	Accesses	W_0	δ_0	WCET$_0$
(a) 9 superblocks	11	$5\,ms$	$0.24\,ms$	$0.566\,ms$
(b) 125 superblocks	172	$20\,ms$	$0.66\,ms$	$9.032\,ms$

Table 3.3: Setup of the two tasks (a) and (b).

each transformed task, the worst-case execution time WCET in Eq. (2.1) is the same for each access model.

There are two processing elements in this experimental setup. The tasks shown in Tab. 3.3 are executed successively on SPE 0, which has δ_0 as slot length. SPE 1 is configured to consume the remaining TDMA cycle of $1.35\,ms$, i.e., $\delta_1 = L(\Theta) - \delta_0$. The slot length δ_{min} is chosen such that only the tasks for the dedicated access model are schedulable, which is the case if the predicted WCCT lies below the processing cycle W_0. This relationship is illustrated in Fig. 3.14. The slot length of SPE 0 is configured with the additional overhead of Eq. (3.2), i.e., $\delta_0 = \delta_{min} + C + S = \delta_{min} + 0.02\,ms + 0.04\,ms$.

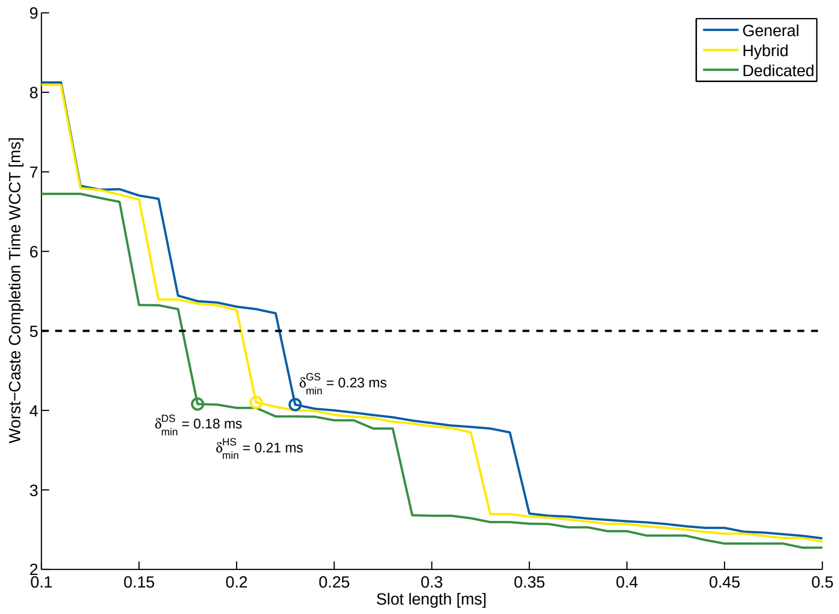
3.5.2 Experimental Results

Figure 3.15 shows the results for two SPEs with interference as well as for one SPE with fully available bus. Although the slot length is configured in the way that the general and the hybrid tasks should not be schedulable, there is no deadline violation during the experiments. Table 3.4 summarizes the gaps between predicted worst-case and maximal observed completion and execution times for the two tasks (a) and (b)[2]. The relative difference is the absolute difference compared with the predicted value. For instance, this is $100\,\% \cdot \frac{|5.3728-3.8049|}{5.3728} \approx 29.18\,\%$ for the general task (a). There are several reasons for this discrepancy, that are described in hereafter:

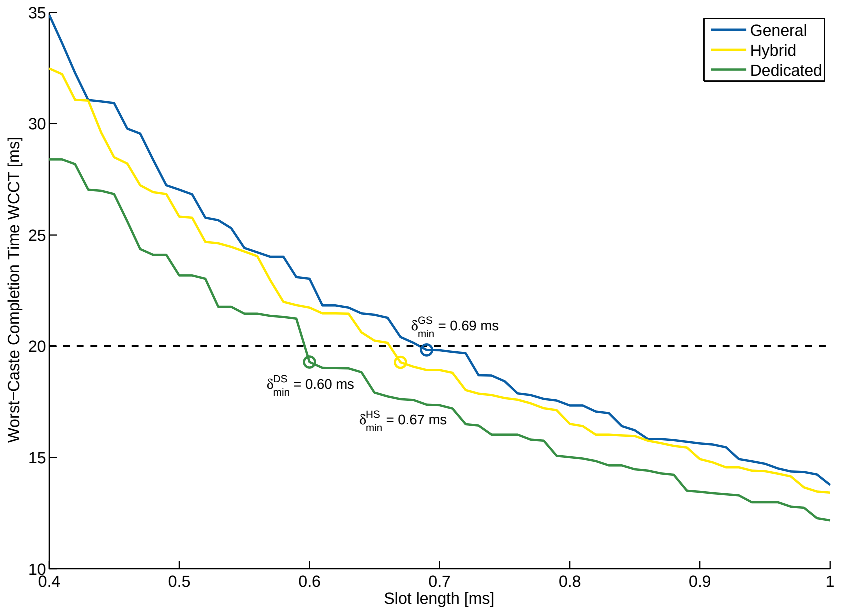
(1) The communication time C is an upper bound for the time needed to access the memory that seldom occurs on the Cell Broadband Engine as it has been explained in Section 3.4. This leads to an overestimation of the WCET, which becomes larger with increasing number of memory requests. The difference between the predicted worst-case and the maximal observed execution times in Tab. 3.4 and Fig. 3.15 directly shows this influence.

(2) The performance analysis in [10] assumes a constant C for finding the worst-case trace. This is mainly significant for the general and

[2]Note that in this thesis, the execution time is used for a task that does not suffer from interference, while the completion time also includes the delay due to memory interference.



(a) 9 Superblocks, 5 ms processing cycle



(b) 125 Superblocks, 20 ms processing cycle

Figure 3.14: Relation between the slot length and the WCCT. δ_{min} is the minimum slot length for which the task with the specific access model is schedulable.

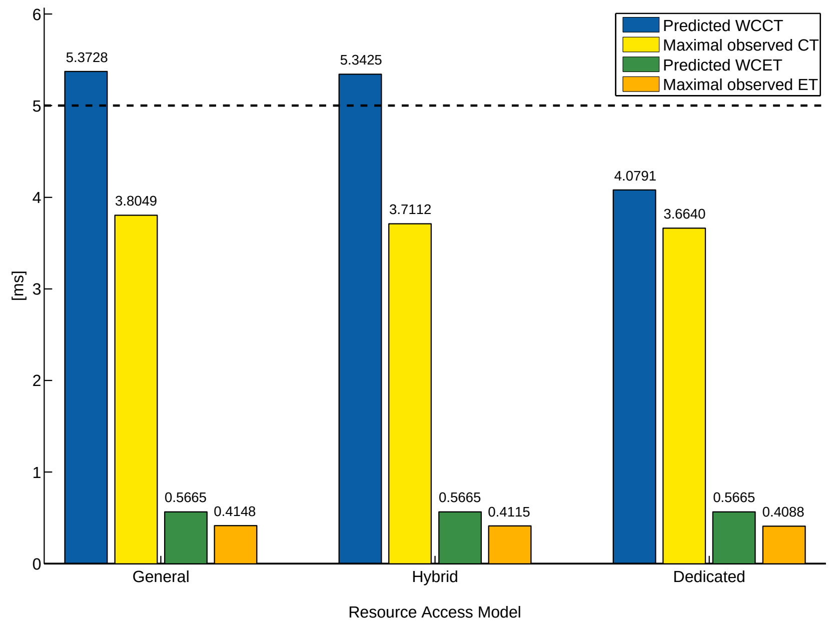
	Task	ΔCT		ΔET	
(a)	General	1.568 ms	29.18 %	0.152 ms	26.78 %
(a)	Hybrid	1.631 ms	30.53 %	0.155 ms	27.35 %
(a)	Dedicated	0.415 ms	10.18 %	0.158 ms	27.83 %
(b)	General	8.690 ms	37.72 %	2.482 ms	27.48 %
(b)	Hybrid	8.047 ms	37.04 %	2.508 ms	27.76 %
(b)	Dedicated	5.671 ms	29.40 %	2.567 ms	28.42 %

Table 3.4: Absolute and relative difference between predicted worst-case and the maximal observed completion (CT) and execution times (ET).

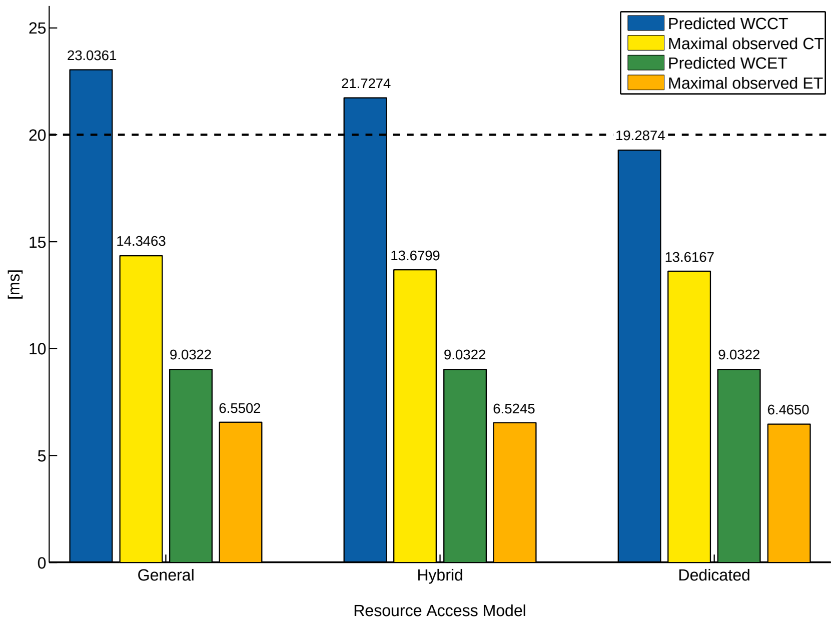
the hybrid access models as it is possible to decide between execution and communication (Section 2.3). Because of the variation of the communication time on the Cell Broadband Engine, the worst-case trace might not be reproducible during the experiments. This is why the prediction of the WCCT for the dedicated access model is the most accurate one of all access models.

(3) Finding the worst-case trace is not guaranteed by the generation of superblocks with random decisions (Section 3.3.2.1). On the one hand, this approach is chosen to simulate many different traces because it is infeasible to explore all possibilities with increasing number of decision options. On the other hand, this leads to the known problem that the maximal observed completion time often underestimates the WCCT [16]. Analogous to the last paragraph, this only affects the general and the hybrid model.

(4) Consider Fig. 3.14 again: It is evident that slightly varying the slot size can result in a large change of the worst-case completion time. If the slot length becomes larger, it is possible that unserved memory requests can already be treated in the currently active slot instead of waiting for the next active slot. Hence, the WCCT is decreased by up to the length of the TDMA cycle. Since the time consumption time for sending mailbox messages and for accessing the main memory varies each time, the effective slot length on an SPE is not constant (Section 3.2.2) and consequently, the maximal number of possible requests. As Fig. 3.9 implies, the probability that sending a message takes the full $\frac{S}{2}$ is approximately 10^{-5}. Assuming statistical independence, this leads to the probability of 10^{-10} for sending two messages, which is necessary for just one slot. For more slots, the probability of full time consumption is further reduced.



(a) $6{\cdot}10^4$ processing cycles were executed for the dedicated task and $1.8{\cdot}10^5$ for the others.



(b) $6{\cdot}10^4$ processing cycles were executed for the dedicated task and $1.8{\cdot}10^5$ for the others.

Figure 3.15: Comparing the predicted worst-case with the maximal observed completion (CT) and execution times (ET).

3.6 Conclusions

Processing elements in a multiprocessor system that share a common resource are greatly influenced by the interference of other elements. The worst-case completion time of tasks executed in such architectures can be significantly increased compared to single processor systems. In order to gain experimental results for a MPSoC, namely the Cell Broadband Engine, this chapter demonstrated a possibility how to implement the TDMA scheduler for this specific hardware. But this implementation also showed the limitation of this hardware: Due to the observed variability of the communication time and the synchronization overhead, the Cell Broadband Engine is not suitable for hard real-time systems. Several work-around were necessary to obtain meaningful experimental results. Especially, finding an upper bound for the synchronization overhead and dealing with its large distribution was challenging and reduces the overall system predictability.

An experimental setup has been created to compare the analytical WCCT with the maximal observed completion time of two example tasks. The discrepancy between these two values are up to 37 % depending on the resource access model. The main reason for this gap is that the analytical model assumes a constant communication time for accessing the shared resource, which is generally not the case for real hardware systems. However, this difference is reduced the more structure is applied to the task: For the dedicated model for example, its value is as low as 10 %.

Nevertheless, the conclusions of [11] and [10] can be approved. The separation of communication and execution by introducing dedicated phases increases the timing predictability: The dedicated access model leads to the most accurate estimation of the WCCT, mainly because of the predetermined order of resource accesses and computations. One restriction for results of the general and the hybrid tasks remains: Because of the randomly chosen order of operations during the experiments, it is not guaranteed that the worst case occurs during a finite execution time.

4 Memory Interference Delay Analysis for Multiple Shared Resources

4.1 Introduction

In this chapter, the performance of a multiprocessor system containing two shared resources is analyzed (see Fig. 4.1). More precisely, we are interested in finding a preferably tight bound of the worst-case completion time of a task executed on a specific processing element. Several papers have been published that analyze the influence of memory contention in multiprocessor platforms with a single shared resource.

As discussed in the previous chapters, [10] analyzes the WCCT of tasks sharing one resource with TDMA policy. Tasks are modeled as sequence of superblocks. Schranzhofer et al. [17] extended their model with adaptive resource arbiters, which combines static arbitration slots with dynamic arbitration segments. During the dynamic segments, the First-Come, First-Served (FCFS) strategy is applied in order to improve the response time of tasks. Their work shows a way how to solve the problem of determining the WCCT of a task with a dynamic programming approach.

The derivation of an upper bound of the delay that a task is suffering due to memory contention is covered in [12] for one shared resource with Round-Robin, Fixed Priority or FCFS arbitration. Their presented approach allows to take the influence of an arbitrary number of interfering tasks into account

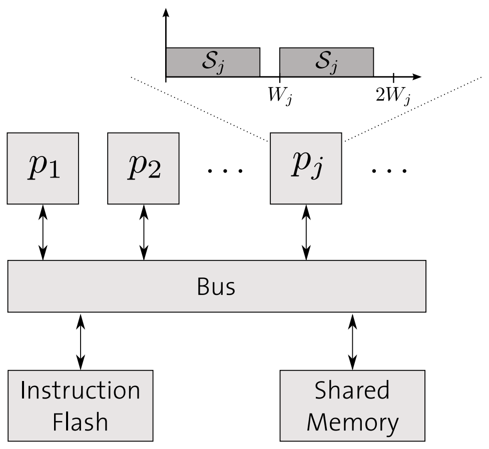


Figure 4.1: Architecture overview

by abstracting them as arrival curves [18]. One advantage of arrival curves is the abstraction disregards timing correlations between processing elements. However, this advantage results in an overestimation of the upper bound because an arrival curve $\alpha(\Delta)$ represents the maximum amount of time required to perform memory requests in any time interval Δ.

Recent work of Lv et al. [19] presents a way how to use Abstract Interpretation to analyze the cache behavior of a task. They apply Timed Automata (TA) to model the precise timing information of a task. With an additional TA, it is possible to analyze the timing behavior of a shared resource with any access policy. Their work includes the FCFS as well as the TDMA arbitration. The drawback of this approach is that it is not scalable, i.e., the runtime of the analysis becomes infeasible for an increasing number of states due to the combinatorial explosion.

Above all, these papers only cover the performance analysis for one shared resource. Introducing a second shared resource complicates the analysis and the existing methods cannot be easily applied. To the best of our knowledge, deriving a tight WCCT of a task accessing multiple shared resources with FCFS arbitration is still an open problem.

For the moment, we are considering two processing elements in order to describe the fundamental problems of deriving a tight upper bound of the completion time. These processing cores are connected to an instruction flash and a data memory over a crossbar bus as sketched in Fig. 4.4a. This crossbar bus connects the processing elements to the FIFO queues of the shared resources. Thus, the resource arbitration is realized with First-Come, First-Served. In addition, resource accesses to any of these shared resources are assumed to be non-buffered, i.e., a task is stalled until the access has been performed. Since at most one request can be served at any time, a

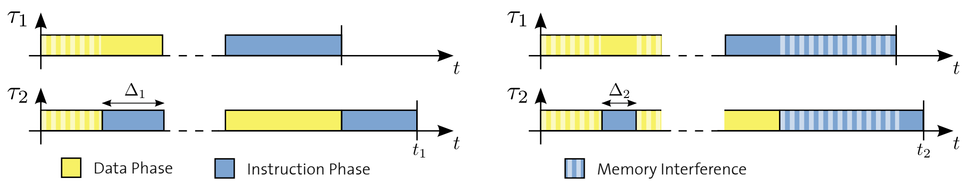


Figure 4.2: Two tasks divided into different phases.

request from any other processing element is delayed. This memory delay is not known in advance and consequently, an upper bound has to be safely estimated.

Figure 4.2 shows an example of two tasks τ_1 and τ_2 and their execution is divided into phases of data accesses and phases of instruction fetches and execution. Whenever both tasks are simultaneously accessing the same resource, the tasks are delayed as a result of memory interference. The first essential problem for the analysis of the FCFS arbitration is that the memory interference delays of one task depend on the exact behavior of the other task, which itself is influenced by the first one. As a matter of fact, the complexity of this problem becomes larger for an increasing number of tasks.

As in Section 2.2.3, the communication time is the time an access to the shared resource requires to be served when not interfered. The instruction time is the time that a processing element needs to perform an instruction. In general, these values are not constant and it is only possible to specify a lower and an upper bound. This has been shown in the measurements of Section 3.4.3.3 for the communication time. Consequently, the worst-case of a phase does not necessarily lead to the worst-case of the complete task. Consider the first instruction phase of task τ_2. In the first case (left), the instruction phase takes Δ_1 time units, which results in the task's completion time t_1. In the second case (right), the same phase finishes earlier after $\Delta_2 < \Delta_1$ due to faster instruction fetch or faster execution. Despite the reduced completion time of this phase, the task's completion time t_2 for this case might become larger than t_1. The concrete realization of this phase influences the subsequent behavior of both tasks: It is possible that phases are shifted, such that two phases of τ_1 and τ_2 overlap which previously had no interference, and consequently increases the total completion time. It is not trivial to take this influence into account, since it impacts future time instances. Because of this domino effect, it is not sufficient to iteratively estimate the worst-case delay for each phase. The analysis has to include the whole task under analysis and the interfering task as well, which blows up the possible state space. Dealing with these issues, the contributions of this chapter can be summarized in the following way:

- We identify the difficulties of finding a valid and tight WCCT in a multiprocessor system containing two shared resources with FCFS access policy.
- A first approach that calculates an untight bound is shown, which is then further improved by simplifying the problem.
- We show the limitations of using either Timed Automata or Dynamic Programming applied to determine the WCCT.
- The usage of TDMA arbitration is motivated in order to increase the timing predictability of the system and to directly analyze the worst-case completion time of an arbitrary number of processing elements.

The remaining part of this chapter is outlined as follows: Section 4.2 complements the task model and the model of shared resources for the proposed architecture and introduces additional notation.Section 4.3 addresses the problem of estimating an upper bound of the worst-case completion time for a given task and FCFS arbitration. We present the performance analysis for an equivalent architecture with TDMA buses in Section 4.4. Experimental results show the differences between FCFS and TDMA for the considered architecture in Section 4.5. Finally, Section 4.6 concludes this chapter.

4.2 System Model

This section extends the system model summarized in Section 2.2. Following the assumptions of [10], we assume that the hardware platform has no timing anomalies [3] and that the execution of tasks and the resource arbiter are initialized synchronously. Otherwise, it would be necessary to take an infinite number of offsets into account.

4.2.1 Architecture

The considered hardware architecture in Fig. 4.1 is a multiprocessor system consisting of processing elements $p_j \in \mathcal{P}$. These processing cores can access two shared resources. Each processing element has a local memory for its exclusive data (which is not explicitly shown), whereas the remaining data resides in the shared memory. Additionally, instructions are shared among all processing elements and are located in the instruction flash. As a result, the execution of a task not only requires fetching data from the main memory, but also fetching instructions from the flash resulting in communication for both operations.

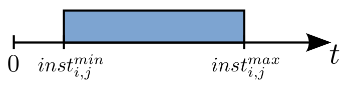


Figure 4.3: Minimal and maximal instruction times of superblock $s_{i,j}$.

4.2.2 Task Model

In addition to Section 2.2.2, the task model is extended in order to describe the behavior of introducing an instruction flash. We are assuming one task τ_j per processing element p_j that consists of a sequence of superblocks $\mathcal{S}_j$ and that is repeated periodically with processing cycle W_j. A general solution for more than one task per processing core is outlined in [12]. Thus, for the remaining part of this chapter, τ_j and $\mathcal{S}_j$ is used interchangeably. Furthermore, the superblocks are executed sequentially, i.e., a superblock is activated as soon as the previous one has finished.

Instead of the maximal execution time $exec_{i,j}^{max}$ for a superblock $s_{i,j}$, the new parameter $inst_{i,j}^{max}$ is used to express the maximal time for executing one single instruction of superblock $s_{i,j}$ on processing element p_j. This upper bound has to be thought as pure execution time without communication time needed to fetch the instruction from the shared flash. The introduction of this new parameter is necessary because contrary to [12] and [10], the actual execution time cannot be realized arbitrarily between $exec_{i,j}^{min}$ and $exec_{i,j}^{max}$. Refer to Fig. 4.3 for an example concerning the instruction time of a superblock.

Similar to the maximal number of data accesses $\mu_{i,j}^{max}$, the maximal number of instruction fetches is denoted as $\nu_{i,j}^{max}$. Accordingly, the maximal execution time of a superblock $s_{i,j}$ without communication can be expressed as $\nu_{i,j}^{max} \cdot inst_{i,j}^{max}$. The *cache profile* of a superblock, which also includes the lower bounds, is therefore defined as:

$$c_{i,j}^{prof} = \{\mu_{i,j}^{min}, \mu_{i,j}^{max}, \nu_{i,j}^{min}, \nu_{i,j}^{max}, inst_{i,j}^{min}, inst_{i,j}^{max}\} \tag{4.1}$$

In this chapter, deriving an exact cache profile is not outlined but supposed to be given in advance.

4.2.2.1 Access Model

Each superblock is structured in phases. During the dedicated phases *acquisition* and *replication*, only memory requests are allowed. During the *execution* phase, instruction fetches and – depending on the access model –

memory requests are performed. The three different access models – *dedicated*, *general* and *hybrid* – describe the access behavior of a task to the shared resources, i.e., which kind of phases the task contains. Refer to Section 2.2.2.1 for detailed explanations concerning the different access models.

4.2.3 Model of Shared Resources

A resource arbiter is responsible to grant access to its attached shared resource. This resource is able to serve at most one request at any time and the communication time for transferring data to/from the processing element can be bounded by a constant C. In general, its actual value is not equal for different types of memories. Therefore, the upper bound for the communication time is denoted as C_μ for the data memory and C_ν for the instruction flash[1]. It would be safe to set C to the maximum of both values, but this results in overestimation. If several processing cores access a shared resource, the communication time might be increased due to contention. The maximum delay depends on the applied arbitration scheme and is described later.

Resource accesses are assumed to be non-buffered: A task whose request is not in the cache has to wait until the request has been served. The shared resources are supposed to be non-preemptive, thus a currently processed request cannot be interrupted by any other request. Moreover, the shared memories are implemented with single-port access. For the case of dual-port access to the instruction flash, there is no contention on this shared resource. For this case, the problem is reduced to the problem of finding the worst-case delay for a single shared resource and the time for fetching an instruction is included in the execution time. A potentially untight worst-case delay can be estimated according to [12] if the main memory applies FCFS arbitration.

4.2.3.1 First-Come, First-Served

The proposed architecture in Fig. 4.4a connects the processing elements and the shared resources over a crossbar bus, i.e., there exists a separate channel to the FIFO queue of each resource. Since it is assumed that one task can only request one resource access at any time, the maximal buffer size of the FIFO is equal to the total number $|\mathcal{P}|$ of processing cores. Consequently, a request arriving at this queue is delayed for at most $(|\mathcal{P}| - 1) \cdot C$ time

[1]We are dropping the index and writing C whenever the actual resource does not matter for the discussion.

units until the shared resource can perform the request resulting in a total worst-case delay:

$$|\mathcal{P}| \cdot C \tag{4.2}$$

As seen in Section 4.1, taking the upper bound C for each resource request as communication time might not result in the worst case. This property of FCFS is contrary to TDMA, where the upper bound C is considered during the worst-case analysis. Hence, the interval between the lower and the upper bound $[C^{min}, C^{max}]$ needs to be taken into account in order to obtain a safe WCCT.

4.2.3.2 Time Division Multiple Access

Instead of FCFS, resource arbitration for the same hardware system can be implemented with TDMA. Generally, there are two meaningful ways to introduce a TDMA bus. Figure 4.4b illustrates the first possibility where the processing cores are connected over one TDMA bus with one schedule Θ. The second option is to introduce a TDMA scheduler for each shared resource, as outlined in Fig. 4.4c. The schedule for the instruction flash is denoted as Θ_ν, while Θ_μ is the schedule for the data memory. The latter proposal is corresponding to the original crossbar topology, where the FIFO queue is replaced by a TDMA bus.

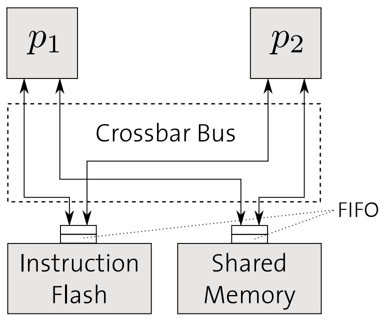
In this chapter, we are assuming a *regular* scheduler: Each processing element p_j has exactly one assigned slot with length δ_{p_j} and starting time σ_{p_j}. This schedule is repeated after its length $L(\Theta)$, which is the sum of all slot lengths:

$$L(\Theta) = \sum_{j=1}^{|\mathcal{P}|} \delta_{p_j}, \quad \delta_{p_j} \geq C \; \forall j \tag{4.3}$$

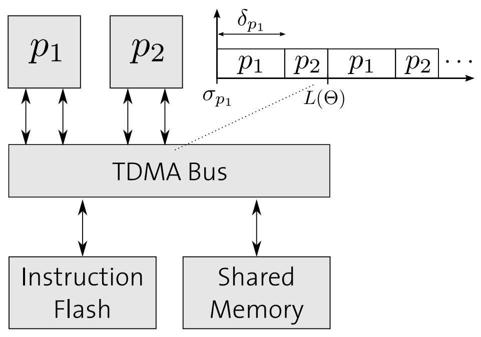
Moreover, the slots do not have to be of the same length for the case of two different schedulers. Nevertheless, their size has to be at least the communication time C since it is required that at least one request can be served during an active slot. Otherwise, no resource request would be served by the TDMA arbiter. See Tab. 4.1 for more details concerning the minimal slot lengths for the different schedulers.

A resource request issued $C - \epsilon > 0$ time units before the slot end cannot be served within the active slot and is therefore stalled until the activation of the next available slot. This results in the worst-case delay for one request under TDMA arbitration:

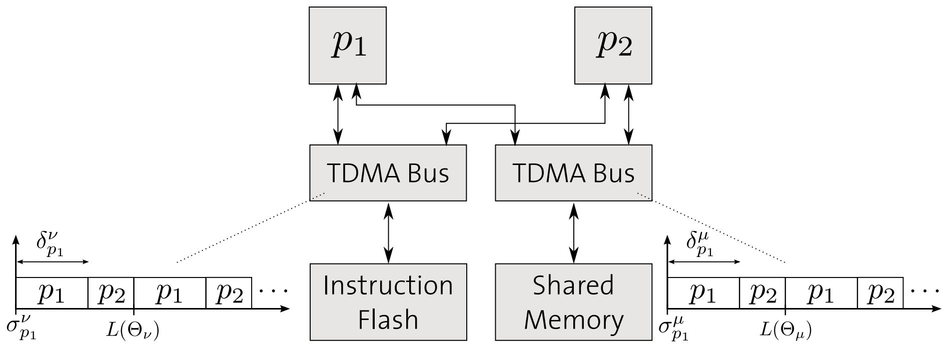
$$(C - \epsilon) + (L(\Theta) - \delta_{p_j}) + C, \quad \epsilon < C \tag{4.4}$$



(a) Two processing elements connected over a *crossbar* topology.



(b) Two processing elements connected over *one TDMA bus* with schedule Θ.



(c) Two processing elements connected over *two TDMA buses* with schedules Θ_μ and Θ_ν.

Figure 4.4: Hardware architecture containing different interconnect buses.

Schedule	Minimum Slot Length
Θ	$\delta_{p_j} > \max(C_\mu, C_\nu)$
Θ_μ	$\delta^{\mu}_{p_j} > C_\mu$
Θ_ν	$\delta^{\nu}_{p_j} > C_\nu$

Table 4.1: Minimum slot lengths for TDMA.

For the minimal slot length $\delta_{p_j} = C$ for example, the worst-case delay of one resource access is bounded by $L(\Theta) + C$.

4.2.4 Determination of the WCCT

The objective of the remaining chapter is to find methods to determine the worst-case completion time of a task. These methods depend on the tasks and on the applied arbitration policy. The influence of the arbitration is taken into account by Eq. (4.2) for FCFS and by Eq. (4.4) for TDMA.

To start with a lower bound for the WCCT, consider a task τ_j executed on a single processor system. Then, the resource requests cannot be interfered and the determination of the worst-case execution time of τ_j is trivial once its cache profile is available:

$$wcet_j = \sum_{i=1}^{|\mathcal{S}_j|} \Big(\nu_{i,j}^{max} \cdot (C_\nu + inst_{i,j}^{max}) + (\mu_{i,j}^{max,a} + \mu_{i,j}^{max,e} + \mu_{i,j}^{max,r}) \cdot C_\mu \Big) \quad (4.5)$$

4.3 Worst-Case Delay Estimation for FCFS

The FCFS arbitration introduces a new degree of freedom compared to a static access policy: The performance analysis has to deal with the situation where the worst-case memory delay of a task depends on the interfering tasks which by themselves depend on all others. In the related work of [12], a comparable problem occurred and was solved by considering one task under analysis and by introducing a traffic delay curve for the interfering tasks. However, this approach cannot be directly applied for more than one shared resources. Hence, it is necessary to find a new method in order to estimate an upper bound of the worst-case completion time of a task.

For several tasks simultaneously accessing the shared resources, the maximal value for the WCCT can be determined by taking the worst-case delay for each access to the shared resources. This worst case occurs whenever the performed resource request arrives at the last position of the FIFO queue. Following this idea, the maximal WCCT of a task τ_j can be expressed as:

$$\begin{aligned} wcct_j^{up} = \sum_{i=1}^{|\mathcal{S}_j|} \Big(& \nu_{i,j}^{max} \cdot (|\mathcal{P}| C_\nu + inst_{i,j}^{max}) \\ & + (\mu_{i,j}^{max,a} + \mu_{i,j}^{max,e} + \mu_{i,j}^{max,r}) \cdot |\mathcal{P}| C_\mu \Big) \end{aligned} \quad (4.6)$$

While this maximal WCCT is trivial to calculate, it is often too pessimistic because it does not include any information of the interfering tasks. Actually, this bound would only result if the same task was executed on every processing element. Hence, the objective of this section is to find a tighter WCCT than this maximal WCCT. First, the dynamic programming approach and its limitations are discussed. Second, an algorithm is presented that derives an untight upper bound and third, this bound is further improved by reducing the uncertainty of the task model. Finally, an unscalable solution using timed automata is presented, which leads to the motivation of introducing TDMA as arbitration policy.

4.3.1 Dynamic Programming

Dynamic programming approaches have been widely used to find optimal solutions for many different optimization problems. In this context, finding a valid upper bound can be regarded as optimization: The objective is to maximize the WCCT of a task under analysis with respect to the interfering tasks.

In the related work of [12], the proposed algorithm uses this kind of approach to deal with an aspect of the problem, namely to handle the introduced complexity of multiple flows. In contrast, the algorithm of [17] uses dynamic programing to efficiently obtain the WCCT and thus, solves the complete problem. The usage of dynamic programming applied to the proposed problem might be promising and is therefore discussed in this section.

Often, optimizations cannot be solved by considering all possibilities due to the large state space. For instance, the runtime of a recursive algorithm grows exponentially with increasing size of the problem and might not be computable due to time and memory requirements. The strength of dynamic programming is to divide the problem in smaller subproblems, which can be optimized, and to use these solutions to optimize the entire problem [20]. It still works recursively, but the runtime can be reduced since the subproblems are of smaller size.

In order to apply dynamic programming to any problem, it has to satisfy the *principle of optimality*. This principle states that optimizing a subproblem leads to the overall optimum [20]. This requirement is hard to prove for our system containing a crossbar topology and the example in the introduction (Fig. 4.2) already indicates that the local optimality does not necessarily lead to the overall optimality. Second, it is a challenge to find the subproblems at all and to define them in a way that they are composable. To the best of our knowledge, finding the WCCT of a task under these circumstances cannot be properly solved with dynamic programming.

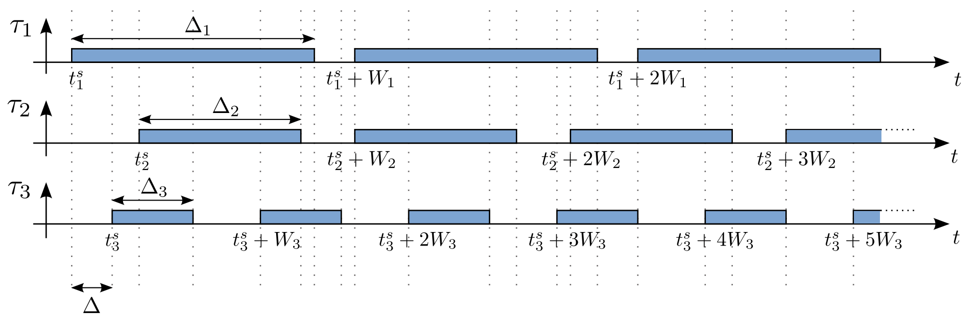


Figure 4.5: Example of three tasks and the resulting time slices Δ.

4.3.2 Untight Bound

The worst-case completion time of Eq. (4.6) is too pessimistic because it does not take any information of the interfering tasks into account. Hence, the maximum interference delay can be reduced with Algorithm 1 as a first improvement. Essentially, the idea is to start with the upper bound of Eq. (4.6) for each task and then to exclude data and instruction fetches that cannot be interfered by all other tasks. Consider Fig. 4.5 for an illustrative example. The algorithm divides the execution of tasks into time slices Δ and if not all tasks are active during a specific time slice, the worst-case completion times for the active tasks can be reduced. For instance, task τ_1 is not interfered during the first time slice.

The algorithm takes the set of tasks and the communication times as input parameter. A task τ_j is defined by its cache profile $c_{i,j}^{prof}$ for each superblock, its processing cycle W_j and its first activation time t_j^s. Additionally, the new parameter $\Lambda_{i,j}$ for each superblock $s_{i,j}$ is introduced, which is the maximal possible completion time assuming $|\mathcal{P}| - 1$ interfering processing elements:

$$\Lambda_{i,j} = \nu_{i,j}^{max} \cdot (|\mathcal{P}|C_\nu + inst_{i,j}^{max}) + \mu_{i,j}^{max} \cdot |\mathcal{P}|C_\mu \tag{4.7}$$

Actually, this is the maximal WCCT for a single superblock of Eq. (4.6). Equation (4.7) already implies that the sequence of superblocks contains general phases with at most μ^{max} data accesses and not more than ν^{max} instruction fetches. This simplifies the handling of the superblocks because it does not matter which phase of which access model is currently analyzed. It is easily possible to transform a sequence of superblocks of any access model to this sequence of general phases.

Variable	Description
d_j	memory interference delay that can be excluded during one processing cycle
Δ_j	remaining time of task assuming maximal WCCT
$\Lambda_{i,j}$	maximal WCCT of superblock $s_{i,j}$
t_j^c	starting time of current ($t_j^c \leq t$) or next ($t_j^c > t$) processing cycle
$wcct_j$	task's resulting WCCT

Table 4.2: State variables of task τ_j used in Algorithm 1.

4.3.2.1 Overview

To start with, an overview of the algorithm is provided. During the main loop described in Section 4.3.2.2, the set $\mathcal{A}$ of currently active tasks is considered. For this, a time variable t determines which task has not finished yet. A task τ_j is supposed to be running if its activation time t_j^c has already past, i.e., $t_j^c \leq t$. Moreover, the time slice Δ is used to express the next time instance $t + \Delta$ where a task either starts or terminates. See Tab. 4.2 for a full description of the used the state variables of task τ_j.

The state variable d_j is introduced to express the overall memory interference delay of task τ_j that can be excluded during one processing cycle. The objective of the algorithm is to reduce the maximal WCCT $wcct_j^{up}$ of Eq. (4.6) by d_j. In general, different processing cycles result in different values for d_j. Hence, only the minimal value of d_j for all cycles improves the WCCT. After each cycle, the value of d_j is reset and recalculated for the next cycle.

For each task in $\mathcal{A}$, the *minimum access delay* that can be interfered during the time slice Δ is computed, see Section 4.3.2.4. This delay is then multiplied with the number of inactive tasks $|\mathcal{P}| - |\mathcal{A}|$ and added to d_j. After the termination of a task τ_j, $wcct_j^{up}$ is reduced by d_j resulting in the improved $wcct_j$.

In order to calculate the minimum access delay, the *time window* T is used to iterate over the sequence of superblocks beginning from the relative time $t - t_j^c$ until $t - t_j^c + \Delta$. If a superblocks completely lies within the time slice Δ, the memory interference delay of all accesses can be reduced. Otherwise, the boundaries have to be differently treated. This procedure is covered in Section 4.3.2.3.

Algorithm 1 Calculate $wcct_j$ of task τ_j $\forall j \in \{1, \ldots, |\mathcal{P}|\}$

```
procedure WCCT-UNTIGHT($\{\tau_j : \forall j\}, C_\mu, C_\nu$)
 1: $wcct_j = 0,\ \Delta_j = \sum_{i=1}^{|\mathcal{S}_j|} \Lambda_{i,j},\ t^c_j = t^s_j,\ d_j = 0,\ \forall j \in \{1, \ldots, |\mathcal{P}|\}$
 2: $t = \min_{\forall j}\{t^s_j\},\ \Delta = 0$
 3: while $t < \max_{\forall j}\{t^s_j\} + \operatorname{lcm}_{\forall j}\{W_j\}$ do
 4:     $\mathcal{A} = \{\tau_j : t^c_j \le t, \forall j \in \{1, \ldots, |\mathcal{P}|\}\},\ \Delta = \min\left(\min_{\forall \tau_j \in \mathcal{A}}\{\Delta_j\},\ \min_{\forall \tau_j \notin \mathcal{A}}\{t^c_j - t\}\right)$
 5:     for all $\tau_j \in \mathcal{A}$ do
 6:         if $|\mathcal{A}| < |\mathcal{P}|$ then
 7:             let $i$ be the index of the currently active superblock $s_{i,j}$
 8:             $t' = \sum_{k=1}^{i-1} \Lambda_{k,j} - d_j$
 9:             while $t' < t - t^c_j + \Delta$ do
10:                 calculate $T$ according to Eq. (4.8)
11:                 $d_j = d_j + (|\mathcal{P}| - |\mathcal{A}|) \cdot$ MIN-DELAY$(s_{i,j}, T, C_\mu, C_\nu)$
12:                 $i = i + 1$, calculate $t'$ according to Eq. (4.9)
13:             end while
14:         end if
15:         $\Delta_j = \Delta_j - \Delta$
16:         if $\Delta_j = 0$ then
17:             $\Delta_j = \sum_{i=1}^{|\mathcal{S}_j|} \Lambda_{i,j},\ wcct_j = \max(wcct_j, \Delta_j - d_j),\ t^c_j = t^c_j + W_j,\ d_j = 0$
18:         end if
19:     end for
20:     $t = t + \Delta$
21: end while
```

4.3.2.2 Main Loop

As a first step, the state variables are initialized (Step 1). The value of the maximal WCCT of task τ_j during each loop iteration is stored in $wcct_j$. Δ_j denotes the remaining completion time of task τ_j. Its value is set to the sum of the maximal completion times of all superblocks (which is equivalent to Eq. (4.6)). The sum of all delays that can be excluded is characterized by d_j and the rest of the algorithm calculates this delay for each cycle of each task.

Beginning from the minimum starting time, the main loop is processed until the maximum starting time plus the hyperperiod is reached (Steps 2 and 3). The hyperperiod is calculated as lowest common multiple of all processing cycles in order to cover all possible offsets between the tasks. Based on the set $\mathcal{A}$ of all active tasks, the new time slice Δ is determined (Step 4). As an example, the dotted lines in Fig. 4.5 indicate all time slices.

For every task in $\mathcal{A}$, the excluded delay d_j is iteratively increased during Steps 6 to 14. The exact procedure is outlined in the next two sections. Afterwards, the remaining execution time Δ_j of task τ_j is reduced by the time slice Δ (Step 15). If this value becomes zero, it is reset to the upper bound of Eq. (4.6) and the worst-case completion time of task τ_j is obtained by subtracting the excluded delay d_j from this upper bound (Step 17). Moreover, the activation time is set to the beginning of the new cycle and the sum of excluded delays d_j is reinitialized.

4.3.2.3 Deriving the Time Windows

This section describes how to calculate the minimum time of task $\tau_j \in \mathcal{A}$ during Δ that cannot be interfered by all tasks. Therefore, this part is only executed if not all tasks are active (Step 6). The temporary variable t' is initialized with the relative starting time of the currently active superblock $s_{i,j}$. This starting time is derived from summing up the upper completion time $\Lambda_{k,j}$ of all finished superblocks minus the excluded delay (Step 8). A time window T for each active superblock is determined beginning with the relative time $t - t_j^c$ until the relative end of the time slice $t - t_j^c + \Delta$ (Steps 9 to 13):

$$T = \begin{cases} t' + \Lambda_{i,j} - (t - t_j^c) & t' < t - t_j^c \;\wedge\; t - t_j^c \leq t' + \Lambda_{i,j} - \Delta \quad \textbf{(1)a} \\ \Delta & t' < t - t_j^c \;\wedge\; t - t_j^c > t' + \Lambda_{i,j} - \Delta \quad \textbf{(1)b} \\ \Lambda_{i,j} & t' \geq t - t_j^c \;\wedge\; t' + \Lambda_{i,j} \leq t - t_j^c + \Delta \quad \textbf{(2)} \\ t - t_j^c + \Delta - t' & \text{otherwise} \quad \textbf{(3)} \end{cases} \tag{4.8}$$

For the following computations, consider Fig. 4.6 as illustrative example. Basically, there are three different types of time windows, namely for the first, each succeeding and the last superblock within the time slice Δ. Equation (4.8) is used to calculate the time window by distinguishing these cases:

(1) For the first superblock, the time window is the remaining time of the superblock and calculated as difference of the relative ending time of the superblock $t' + \Lambda_{i,j}$ and the relative time $t - t_j^c$ **(1)a**. However, if the time slice Δ is smaller than this remaining time, the time window is equal to Δ **(1)b**.

(2) For each superblock that is completely within Δ, the time window is set to the upper completion time $\Lambda_{i,j}$.

(3) For the last superblock, the time window is derived by subtracting the relative activation time t' of the superblock from the relative ending time $t - t_j^c + \Delta$ of the time slice.

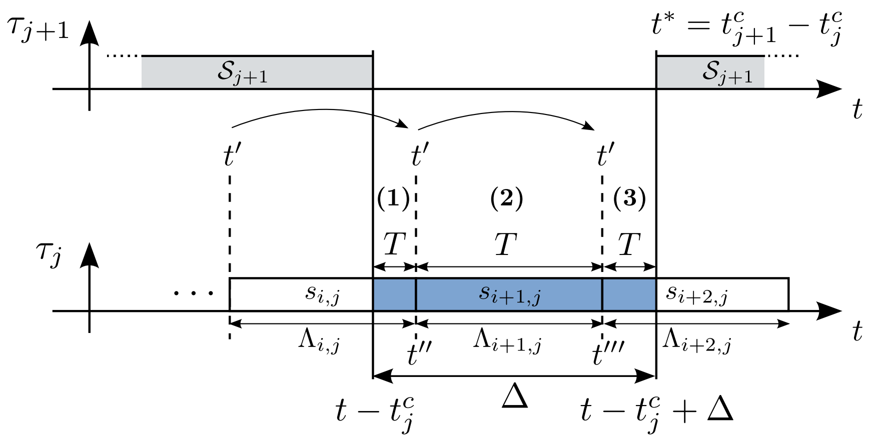


Figure 4.6: Deriving the time windows T for three superblocks that are active during the time slice Δ.

An example of two tasks is shown in Fig. 4.6. The time windows T are calculated for task τ_j, while τ_{j+1} is inactive until its activation time t^c_{j+1}. In the figure, the time instance t^* is relative to the activation time t^c_j of task τ_j. If $t^* > t'''$, then all three cases **(1)a**, **(2)** and **(3)** are considered. However, if $t''' > t^* > t''$, then only the time windows T for cases **(1)a** and **(3)** are determined because there is no superblock that lies completely within Δ. And if $t^* < t''$, only **(1)b** is regarded.

The minimum access delay of superblock $s_{i,j}$ during T is the number of performed requests during this time period. Its value is determined by algorithm MIN-DELAY and multiplied by the number of inactive tasks, i.e., $|\mathcal{P}| - |\mathcal{A}|$ (Step 11). This is actually the delay that can be excluded. Afterwards, the next superblock is processed by incrementing the index variable i and setting the temporary variable t' to the relative activation time of the next superblock (Step 12):

$$t' = \begin{cases} t - t^c_j + T & t' < t - t^c_j \quad \textbf{(1)} \\ t' + T & \text{otherwise} \quad \textbf{(2)}\ \&\ \textbf{(3)} \end{cases} \tag{4.9}$$

4.3.2.4 Calculating the Minimum Access Delay

Algorithm 2 calculates the minimum access delay of superblock $s_{i,j}$ during the time window T under the assumption that each request is delayed for $|\mathcal{P}|C$ time units. The resulting memory access delay is defined as number of performed requests multiplied with the appropriate communication time.

For the dedicated cases, it is already clear which operation is performed during T. Therefore, it is sufficient to multiply the number of performed

Algorithm 2 Calculate minimum delay of superblock $s_{i,j}$ that can be interfered during T

```
procedure MIN-DELAY($s_{i,j}, T, C_\mu, C_\nu$)
1: if $\mu^{max} = 0$ then
2:     return $\left\lfloor \frac{T}{|\mathcal{P}|C_\nu + inst^{max}} \right\rfloor \cdot C_\nu$
3: else if $\nu^{max} = 0$ then
4:     return $\left\lfloor \frac{T}{|\mathcal{P}|C_\mu} \right\rfloor \cdot C_\mu$
5: else if $T < \Lambda$ then
6:     return $\min_{\forall m \in \mathcal{M}} \{mC_\mu + f(m)C_\nu\}$
7: else
8:     return $\mu^{max} \cdot C_\mu + \nu^{max} \cdot C_\nu$
9: end if
```

requests with the corresponding communication time (Steps 1 to 4). And for the trivial case $T = \Lambda_{i,j}$, each request μ^{max} and ν^{max} can be excluded (Step 8).

For every other time window $T < \Lambda_{i,j}$, the algorithm determines those $m \in \{0, \ldots, \mu_{i,j}^{max}\}$ and $n \in \{0, \ldots, \nu_{i,j}^{max}\}$ that can be performed during this time period. For this, the minimal number of instruction fetches during T if m data access happen is calculated as[2]:

$$n = f(m) = \left\lfloor \frac{\{T - m \cdot |\mathcal{P}|C_\mu\}^+}{|\mathcal{P}|C_\nu + inst_{i,j}^{max}} \right\rfloor \tag{4.10}$$

Because of the floor operation in Eq. (4.10), this function is not injective, i.e., it is possible that several m are mapped to the same n (see Fig. 4.7 for an example). Consequently, only the largest m should be chosen for equal function values. Hence, the following set contains all those data requests m:

$$\mathcal{M} = \{m : 0 \leq m < \mu^{max}, f(m) > f(m+1)\} \cup \{\mu^{max}\} \tag{4.11}$$

Finally, the tuple $(m', f(m'))$ is chosen that minimizes the resulting resource access delay $m' \cdot C_\mu + f(m') \cdot C_\nu$ (Step 6).

4.3.2.5 Time Complexity

The time complexity of the algorithm is discussed in this section. The loop of Steps 9 to 13 analyzes the superblocks of each task τ_j. This can be

[2] $\{z\}^+ = \max(0, z)$

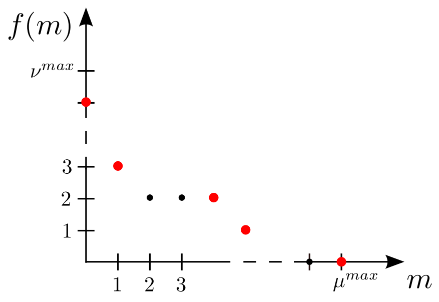


Figure 4.7: Different possibilities (m, n) of performed requests during the time window T.

bounded by the total number of superblocks $|\mathcal{S}_j|$. The algorithm iterates the time over the interval between the minimal starting time and the maximal starting time plus the hyperperiod (Steps 2 and 3). During this interval, the maximum number of considered cycles of task τ_j is calculated as:

$$R_j^{max} = \left\lceil \frac{\underset{\forall j}{\operatorname{lcm}}\{W_j\} + \underset{\forall j}{\max}\{t_j^s\} - \underset{\forall j}{\min}\{t_j^s\}}{W_j} \right\rceil \tag{4.12}$$

Consequently, the calculation of the minimum access delay for τ_j is executed at most $|\mathcal{S}_j| \cdot R_j^{max}$ times. The overall time complexity for the algorithm can be bounded by Eq. (4.13), which is scalable for an increasing number of superblocks and tasks:

$$\mathcal{O}\Big(\sum_{j=1}^{|\mathcal{P}|}(|\mathcal{S}_j| \cdot R_j^{max})\Big) \tag{4.13}$$

4.3.3 Recursive Approximation

This section presents a way to approximate the worst-case completion time for each task. In order to simplify the problem, we are reducing the uncertainty by assuming:

- Constant communication times C_μ and C_ν.
- Constant instruction times ($inst^{min} = inst^{max}$).
- Fixed number of resource accesses ($\mu^{min} = \mu^{max}$, $\nu^{min} = \nu^{max}$).

Although these assumptions may not reflect the timing behavior of multiprocessor systems, they allow an approximation of the WCCT. We are only

Variable	Description
d_j	memory interference delay due to contention during one phase
Δ_j	remaining time of the current superblock assuming WCET
$\Gamma_{i,j}$	WCET of superblock $s_{i,j}$
k_j	index of current superblock
t_j^c	starting time of current ($t_j^c \leq t_j$) or next ($t_j^c > t_j$) processing cycle
t_j	task's current time
$wcct_j$	task's resulting WCCT

Table 4.3: State variables of task τ_j used in Algorithm 3.

considering tasks containing dedicated superblocks, i.e., the instruction and the data phases are strictly separated, therefore eliminating the need to optimally choose the right operation at any time. Otherwise, the runtime of the presented Algorithm 3 becomes prohibitive even for a few superblocks per task.

4.3.3.1 Overview

The presented Algorithm 3 recursively estimates the worst-case completion time of two tasks τ_1 and τ_2. Basically, if they access the same shared resource during a phase, two cases are distinguished:

(1) τ_1 wins the first contention.

(2) τ_2 wins the first contention.

Each other request during the same phase of the superblocks is maximally delayed for C resulting in the access latency of $2C$. Comparing the final worst-case completion times of both possibilities, the case that leads to the larger WCCT is taken for each task during each recursive instance. The recursion works over the whole sequence of superblocks because the optimality of a subproblem does not have to result in global optimality as discussed in Section 4.1.

The algorithm can be divided into two functional parts. The first one takes care of the state variables for each task. They are initialized the first time and updated during each recursion. The former mechanism is outlined in Section 4.3.3.2 and the latter in Section 4.3.3.4. The second part is the actual calculation of the worst-case delay, see Section 4.3.3.3.

An overview of the used state variables is provided in Tab. 4.3. The notation is similar to Algorithm 1, but differing in the following points: First, each task τ_j has its own time variable t_j. Second, the variable k_j is used to index the currently active superblock. Third, the worst-case delay during the time slice Δ is expressed by d_j. Fourth, Δ_j denotes the remaining worst-case execution time of a superblock instead of the maximal worst-case completion time. At the activation of a superblock $s_{i,j}$, this value is initialized with its WCET:

$$\Gamma_{i,j} = \mu_{i,j}^{max} \cdot C_\mu + \nu_{i,j}^{max} \cdot (C_\nu + inst_{i,j}^{max}) \tag{4.14}$$

4.3.3.2 Initialization

The ending time is set to the latest starting time plus the hyperperiod (Step 1). The hyperperiod is defined in the same way as in Algorithm 1 as the lowest common multiple of all processing cycles. As a result, the time variable t_j for each task τ_j is processed from the task's starting time t_j^s until $t_j^s + \underset{\forall j}{\operatorname{lcm}}\{W_j\}$. Then, the state variables are initialized in Step 2 for each task. The remaining calculation is implemented by Algorithm 4 and described in Section 4.3.3.3.

Algorithm 3 Recursively calculate $wcct_j$ of task τ_j $\forall j \in \{1, 2\}$

```
procedure WCCT-RECURSIVE($\{\tau_j : \forall j\}, C_\mu, C_\nu$)
1: $t_{end} = \max_{\forall j}\{t_j^s\} + \operatorname{lcm}_{\forall j}\{W_j\}$
2: $wcct_j = 0$, $t_j = t_j^s$, $t_j^c = t_j^s$, $k_j = 1$, $\Delta_j = \Gamma_{k_j,j}$, $d_j = 0$, $\forall j \in \{1, 2\}$
3: return WORST-CASE-DELAY($\{\tau_j : \forall j\}, C_\mu, C_\nu$)
```

4.3.3.3 Worst-Case Delay

This section describes how to calculate the worst-case delay d_j for each task τ_j, which is implemented by Algorithm 4. The set $\mathcal{A}$ contains the active tasks with minimal t_j. The time slice Δ is set to either the earliest ending phase (Δ_j), the nearest new cycle time ($t_j^c - t_j$) or to the next time variable ($t_{j''} - t_{j'}$), whichever results in the minimal value for Δ (Steps 4 to 8). For the latter, j' is the index of the task with smaller current time and j'' is the index of the other task (Step 7). Then, the number of interfering requests μ and ν during Δ are computed (Step 10). There two reasons, for which both values become zero:

- Only one task is currently analyzed and consequently, the task is not interfered.

- If two tasks are analyzed, it is possible that there is no interference: Since there are only dedicated phases, it is possible that the tasks do not access the same resource during Δ.

In both cases, the worst-case delays d_j are all zero and the tasks are processed for Δ time units without interference (Step 12).

Algorithm 4 Recursively calculate $wcct_j$ of task τ_j $\forall j \in \{1,2\}$

```
procedure Worst-Case-Delay($\{\tau_j : \forall j\}, C_\mu, C_\nu, [\mathcal{A}, \Delta, t_{end}]$)
 1: State-Variables($\{\tau_j : \forall j\}, \mathcal{A}, \Delta$)
 2: if $\min_{\forall j}\{t_j\} \geq t_{end}$ then
 3:     return $\{wcct_j : \forall j\}$
 4: else if $t_1 = t_2$ then
 5:     $\mathcal{A} = \{\tau_j : t_j^c \leq t, \forall j \in \{1,2\}\}$, $\Delta = \min\left(\min_{\forall \tau_j \in \mathcal{A}}\{\Delta_j\}, \min_{\forall \tau_j \notin \mathcal{A}}\{t_j^c - t_j\}\right)$
 6: else
 7:     let $\tau_{j'}$ be the task with smaller current time and $\tau_{j''}$ the other task
 8:     $\mathcal{A} = \{\tau_{j'}\}$, $\Delta = \min(\Delta_{j'}, t_{j''} - t_{j'})$
 9: end if
10: let $\mu$ and $\nu$ be the number of interfering requests during $\Delta$
11: if $\mu = 0$ and $\nu = 0$ then
12:     return Worst-Case-Delay($\{\tau_j : \forall j\}, C_\mu, C_\nu, \mathcal{A}, \Delta, t_{end}$)
13: else
14:     let $j_1$ be one task of $\mathcal{A}$ and $j_2$ the other task
15:     if $\mu > 0$ then
16:         $d_{j_1}^1 = (\mu - 1)C_\mu$, $d_{j_2}^1 = \mu C_\mu$,    $d_{j_1}^2 = \mu C_\mu$, $d_{j_2}^2 = (\mu - 1)C_\mu$
17:     else
18:         $d_{j_1}^1 = (\nu - 1)C_\nu$, $d_{j_2}^1 = \nu C_\nu$,    $d_{j_1}^2 = \nu C_\nu$, $d_{j_2}^2 = (\nu - 1)C_\nu$
19:     end if
20:     for all $k \in \{1,2\}$ do
21:         $d_j = d_j^k, \forall j$
22:         $wcct^k$ =Worst-Case-Delay($\{\tau_j^k : \forall j\}, C_\mu, C_\nu, \mathcal{A}, \Delta, \max_{\forall j}\{t_j^c + W_j\}$)
23:     end for
24:     return $\{\max_{\forall k}\{wcct_j^k\} : \forall j\}$
25: end if
```

Otherwise, new state variables are introduced (Steps 14 to 18): The first ones with superscript 1 are used to analyze the case where the first resource request of task τ_{j_1} is issued immediately before the request of τ_{j_2}. For task τ_{j_1}, this results in the worst-case delay of $(\mu - 1)C_\mu$ and $(\mu - 1)C_\nu$, respectively, while for the other task τ_{j_2}, the maximal worst-case delay is assumed. On the other hand, the second state variables with superscript 2 are computed for the contrary case.

Afterwards, the worst-case completion times for both cases are recursively determined taking the maximum of the next cycle time as ending time for the recursion (Step 22). The larger WCCT for both cases is then taken as the WCCT of the task (Step 24).

4.3.3.4 State Variables

For each recursion of Algorithm 4, the state variables are updated for all active tasks $\tau_j \in \mathcal{A}$ by Algorithm 5. The current time t_j is increased by the processed time slice Δ and the worst-case delay d_j. This worst-case delay is reset and the remaining worst-case execution time Δ_j of the current superblock is adjusted (Step 2). If this time becomes zero, the next superblock is activated if the task has not finished (Step 7). Otherwise, the worst-case completion time is updated and the state variables are reinitialized (Step 5).

Algorithm 5 Handling of the state variables of task τ_j $\forall j \in \{1, 2\}$

```
procedure STATE-VARIABLES($\{\tau_j : \forall j\}, \mathcal{A}, \Delta$)
 1: for all $j \in \mathcal{A}$ do
 2:     $t_j = t_j + \Delta + d_j$, $\Delta_j = \Delta_j - \Delta$, $d_j = 0$
 3:     if $\Delta_j = 0$ then
 4:         if $k_j = |\mathcal{S}_j|$ then
 5:             $wcct_j = \max(wcct_j, t - t_j^c)$, $t_j^c = t_j^c + W_j$, $t_j = t_j^c$, $k_j = 1$, $\Delta_j = \Gamma_{k_j,j}$
 6:         else
 7:             $k_j = k_j + 1$, $\Delta_j = \Gamma_{k_j,j}$
 8:         end if
 9:     end if
10: end for
```

4.3.3.5 Time Complexity

Due to the recursion, Algorithm 3 needs exponential time for calculating the WCCT of two tasks. In the worst case, each request of both tasks results in a recursive decision. Therefore, the time complexity can be expressed as:

$$\mathcal{O}(2^n), \quad n = \max_{\forall j} \left\{ \sum_{i=1}^{|\mathcal{S}_j|} \mu_{i,j}^{max} \right\} + \max_{\forall j} \left\{ \sum_{i=1}^{|\mathcal{S}_j|} \nu_{i,j}^{max} \right\} \tag{4.15}$$

Since the whole sequence of superblocks has to be taken into consideration, this approach is not scalable for increasing number of superblocks.

4.3.4 Timed Automata

A Timed Automaton is defined as a finite automaton extended with real-valued clock variables [21]. These clocks progress synchronously and clock constrains describe the timing behavior of the automaton. It is possible to model the proposed architecture with $|\mathcal{P}|$ processing elements and two shared resources as a network of parallel Timed Automata.

One possibility of using TAs is presented in the following. One TA for each task has been designed to model its timing behavior. An example is shown in Fig. 4.8a for task τ_{j+1}[3]. The automaton starts from the state *inactive* and processes all states until one of the ending states *violation* or *finished* is reached. The former models a deadline violation, i.e., if the completion time of a task is larger than its processing cycle. The latter is activated when the task has terminated and the global clock `t` is at least `EndTime`, which is equal to $\max\limits_{\forall j}\{t_j^s\} + \operatorname{lcm}\limits_{\forall j}\{W_j\}$.

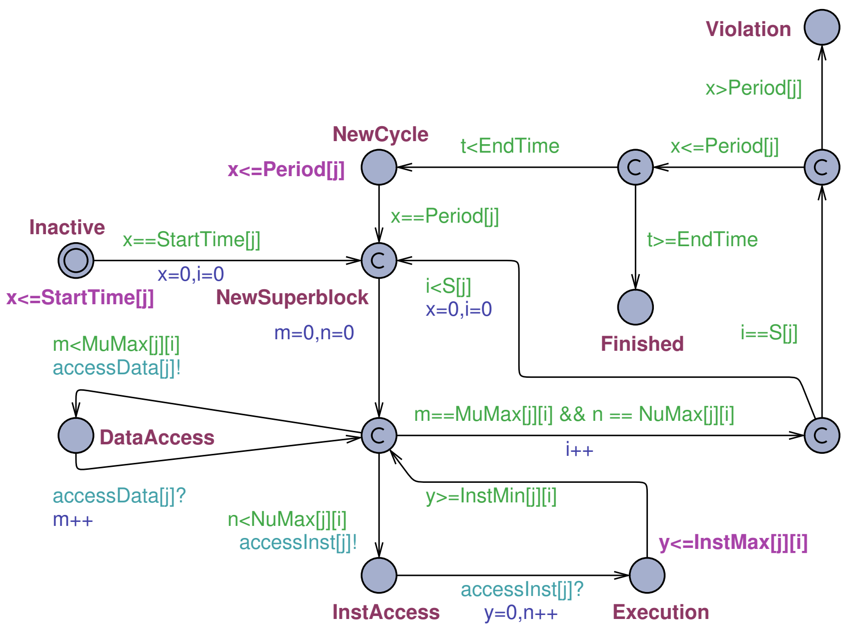
During the active time of the task, its superblocks are sequentially processed starting with the index $i = 0$ until $i \geq |\mathcal{S}_j|$. The counters `m` and `n` are used for the performed data accesses and instructions of the current superblock, respectively. The accesses of task τ_{j+1} to the shared resources are implemented by using the channels `accessInst[j]` and `accessData[j]`.

An additional TA is used for every shared resource. This is outlined in Fig. 4.8b for a FCFS bus taken from [19]. The channel `accessBus` has to be set to `accessData` for the shared memory and to `accessInst` for the instruction flash. There is another adaption necessary because the communication times are not constant: The access time `MemTime` from the original TA is replaced by the parameters (`BusTimeMin`, `BusTimeMax`). They are set to either (C_μ^{min}, C_μ^{max}) or (C_ν^{min}, C_ν^{max}) depending on the applied resource.

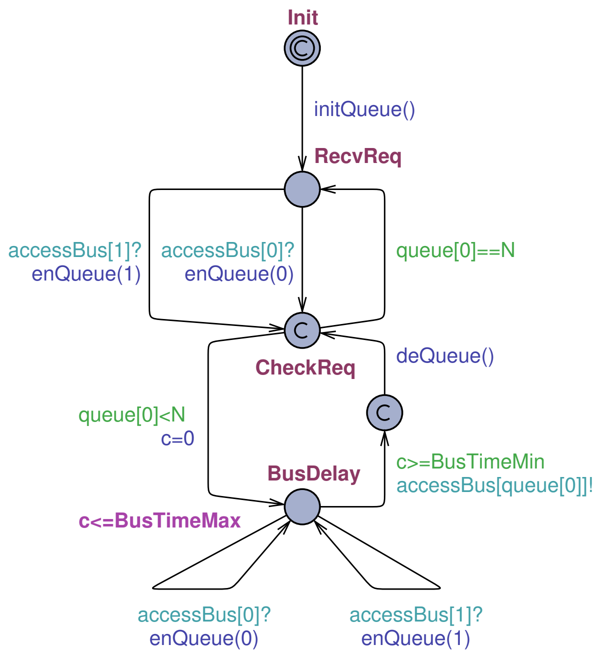
The parallel network of TAs is then analyzed with a model-checker such as Uppaal. It verifies if their is no possibility to reach the *violation* states, i.e., if all possible completion times are smaller than the corresponding processing cycles. The advantage of this approach is that it allows to take all the variabilities of communication and instruction time into account. However, it is not directly possible to obtain each WCCT. Even worse, it does not solve the problem in general because the runtime needed for the model-checking is not feasible for an increasing number of superblocks due to the state space explosion. The Timed Automata are non-deterministic automata and during the model-checking, this non-determinism is eliminated by considering all possibilities, which results in exponential time complexity.

In order to overcome this problem, the next section replaces the original crossbar bus with one and two TDMA buses, respectively. The TDMA explicitly removes the memory interference since a task is only allowed to

[3] The used notation for the TAs is taken from Uppaal [22], where all indexes start from zero.



(a) Timed Automaton for one task.



(b) Timed Automaton for a FCFS bus [19] extended with variable bus delay between `BusMinTime` and `BusMaxTime` instead of constant bus delay `MemTime`.

Figure 4.8: Timed Automata modeling arbitrary number of tasks accessing two FCFS buses. For the data bus, the channel `accessBus` is defined by `accessData`. For the instruction bus, `accessBus` is defined by `accessInst`.

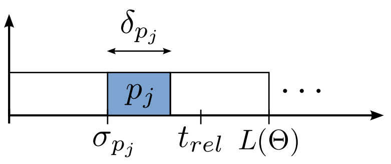


Figure 4.9: Regular TDMA schedule Θ.

access the shared resource during predefined slots. This allows to analyze the WCCT of the same tasks with reduced time complexity.

4.4 Worst-Case Delay Estimation for TDMA

As it is possible to design a system equivalent to the original one containing the crossbar bus, this section shows the performance analysis for the TDMA arbitration. TDMA is a standard access policy in industrial hardware systems often used to increases timing predictability. Generally, the big advantage compared to FCFS is that the performance analysis is simplified by removing the influence of the interfering tasks. The worst-case delay analysis only depends on a given task under analysis and the TDMA scheduler(s). Hence, a further favorable property of TDMA is that the presented performance analysis can be directly applied to any number of tasks.

In this section, we adapt the analysis methodologies of [10] for the modified architectures containing either one TDMA bus with schedule Θ (Fig. 4.4b) or two TDMA buses with schedules Θ_μ and Θ_ν (Fig. 4.4c). In the following, the case for one TDMA scheduler is covered by the more general solution of two TDMA schedulers. For this purpose, the schedules Θ_μ and Θ_ν are equal to Θ and additionally, the right communication times for the particular resource accesses have to be taken.

The original algorithm for an acquisition / replication phase Wct-AR of [10] is used to receive the WCCT for performing a specified number of requests. Starting from time instance t, Algorithm 6 calculates this WCCT for a regular schedule Θ and a particular communication time C.

The time $t_{rel} \in [0, L(\Theta))$ (Step 4) determines the relative position within the schedule Θ as illustrated in Fig. 4.9. If the slot assigned to p_j is currently available, i.e., if $t_{rel} \in \left[\sigma_{p_j}, \sigma_{p_j} + \delta_{p_j}\right)$, the number of performed requests λ^c during the rest of the slot is computed (Steps 5 to 12). If all pending requests can be served, the algorithm returns.

Otherwise, the time t' is set to the starting time of the next slot (Step 13). The number of performed requests n during the assigned slot determines the number of cycles that are necessary to serve all remaining requests (Step 14). Steps 15 to 18 calculate the resulting completion time.

Algorithm 6 Calculate WCCT for performing λ resource requests

```
procedure WCCT-AR($\Theta, p_j, \lambda, t, C$)
 1: if $\lambda = 0$ then
 2:     return $t$
 3: end if
 4: $t_{rel} = t - \left\lfloor \frac{t}{L(\Theta)} \right\rfloor \cdot L(\Theta)$
 5: if $\sigma_{p_j} \leq t_{rel} < \sigma_{p_j} + \delta_{p_j}$ then
 6:     $\lambda^c = \left\lfloor \frac{\sigma_{p_j} + \delta_{p_j} - t_{rel}}{C} \right\rfloor$
 7:     if $\lambda^c \geq \lambda$ then
 8:         return $t + \lambda C$
 9:     else
10:         $\lambda = \lambda - \lambda^c$
11:     end if
12: end if
13: let $t'$ be the starting time of the next time slot after $t$
14: $n = \left\lfloor \frac{\delta_{p_j}}{C} \right\rfloor$, $c = \{\lceil \frac{\lambda}{n} \rceil - 1\}^+$, $\lambda = \lambda - c \cdot n$
15: if $\lambda > n$ then
16:     return $t + t' + L(\Theta) \cdot (c + 1) + (\lambda - n) \cdot C$
17: else
18:     return $t + t' + L(\Theta) \cdot c + \lambda C$
19: end if
```

Based on this algorithm, the next two sections describe the determination of the WCCT of a phase. Since the execution of an instruction requires an instruction fetch first, the calculation of the worst-case completion time of this resource access is calculated identically.

4.4.1 WCCT for a Dedicated Phase

For a dedicated phase, either the maximal number of instruction fetches ν^{max} or the maximal number of data accesses μ^{max} is equal to zero. Thus, the WCCT as calculated in Algorithm 7 (Steps 1 to 7) is the time needed to perform all of these requests plus the starting time t.

For a data phase (Step 7), it is possible to directly apply the algorithm WCT-AR. For an instruction phase (Steps 2 to 5) however, the WCCT is obtained in a different way: First, the completion time of an instruction fetch (WCT-AR) is computed followed by the execution of the instruction ($inst^{max}$). This is iteratively repeated until all ν^{max} requests are served. This proceeding is necessary because the active slot of schedule Θ_ν depends on the current time t, which progresses during the execution of the instruction.

Algorithm 7 Calculate WCCT for a general phase of task τ_j

```
procedure WCCT-PHASE($\Theta_\mu, \Theta_\nu, p_j, \mu^{max}, \nu^{max}, inst^{max}, t, C_\mu, C_\nu$)
 1: if $\mu^{max} = 0$ then
 2:     while $\nu^{max} > 0$ do
 3:         $t =$WCT-AR$(\Theta_\nu, p_j, 1, t, C_\nu) + inst^{max}$, $\nu^{max} = \nu^{max} - 1$
 4:     end while
 5:     return $t$
 6: else if $\nu^{max} = 0$ then
 7:     return WCT-AR$(\Theta_\mu, p_j, \mu^{max}, t, C_\mu)$
 8: else
 9:     $t_\nu =$WCT-AR$(\Theta_\nu, p_j, 1, t, C_\nu) + inst^{max}$
10:     $t_\nu =$WCCT-PHASE$(\Theta_\mu, \Theta_\nu, p_j, \mu^{max}, \nu^{max} - 1, inst^{max}, t_\nu, C_\mu, C_\nu)$
11:     $t_\mu =$WCT-AR$(\Theta_\mu, p_j, 1, t, C_\mu)$
12:     $t_\mu =$WCCT-PHASE$(\Theta_\mu, \Theta_\nu, p_j, \mu^{max} - 1, \nu^{max}, inst^{max}, t_\nu, C_\mu, C_\nu)$
13:     return $\max(t_\nu, t_\mu)$
14: end if
```

4.4.2 WCCT for a General Phase

For a general phase, the question is whether to perform a data access or an instruction. This decision has to be made in the way that it maximizes the WCCT of the whole phase. Steps 9 to 13 of Algorithm 7 implement a recursive approach by examining all possibilities.

On one hand, the case is considered where the first performed operation is an instruction. Similar to the dedicated proceeding, the worst-case completion time to perform one instruction is calculated (Step 9) and the WCCT of the remaining operations is recursively determined (Step 10). On the other hand, the same procedure is implemented for performing a data request as first operation (Steps 11 and 12). This recursion is repeated until the phase becomes dedicated. After these steps, the maximum of both completion times is the final WCCT of the phase (Step 13).

4.4.3 Time Complexity

An equivalent representation of this approach is a binary tree with height $n = \mu^{max} + \nu^{max}$ and consequently, the time complexity of Algorithm 7 is:

$$\mathcal{O}(2^n), \quad n = \mu^{max} + \nu^{max} \tag{4.16}$$

In contrast to the FCFS arbitration, the optimality of subproblems – assuming the worst-case for each phase – results in the overall optimality during the analysis of TDMA. Hence, the recursion works over individual superblocks instead of considering the whole task. This results in a computationally efficient solution compared the recursive approach of Section 4.3.3.

4.5 Experiments

For the experiments, the cache profiles c^{prof} for the example tasks are randomly generated. Starting with the hybrid access model, the data accesses during the dedicated phases of any superblock are shifted to the execution phase in order to create the tasks for the general model. For the dedicated access model, the data accesses during the execution phase are shifted to the dedicated phases. This procedure ensures that the worst-case execution times are equal for the different access models. The parameters for the uniform random distributions are summarized in Eq. (4.17). In addition, the communication times are set to $C_\mu = 30\,\mu s$ and $C_\nu = 20\,\mu s$.

$$\begin{aligned} \mu^{max,a} &\in \{1, \dots, 2\} \\ \mu^{max,e} &\in \{1, \dots, 4\} \\ \mu^{max,r} &\in \{1, \dots, 2\} \\ \nu^{max} &\in \{1, \dots, 6\} \\ inst^{max} &\in [50, 200]\,\mu s \end{aligned} \tag{4.17}$$

4.5.1 FCFS Arbitration

For the FCFS arbitration, Figure 4.10 shows the results for two dedicated tasks for an increasing number of superblocks. The upper WCCT is the maximal possible WCCT if each resource request is delayed for $|\mathcal{P}|C$ (Eq. (4.6)). The untight WCCT is calculated with Algorithm 1 and the recursively estimated WCCT with Algorithm 3. For the sake of comparison, the worst-case execution time of the tasks is also plotted, assuming there is no interference at all (Eq. (4.5)). The traced WCCT is calculated by taking the maximal completion time of all possible traces considering constant communication and instruction times.

The upper and the untight bounds are equal for the first task with shorter completion time, since no memory delay can be excluded by Algorithm 1. For the second task however, the untight bound improves with increasing number of superblocks compared to the upper bound. For example, this improvement is $1 - \frac{8.8124}{9.5124} \approx 7.4\,\%$ for ten superblocks. The average relative

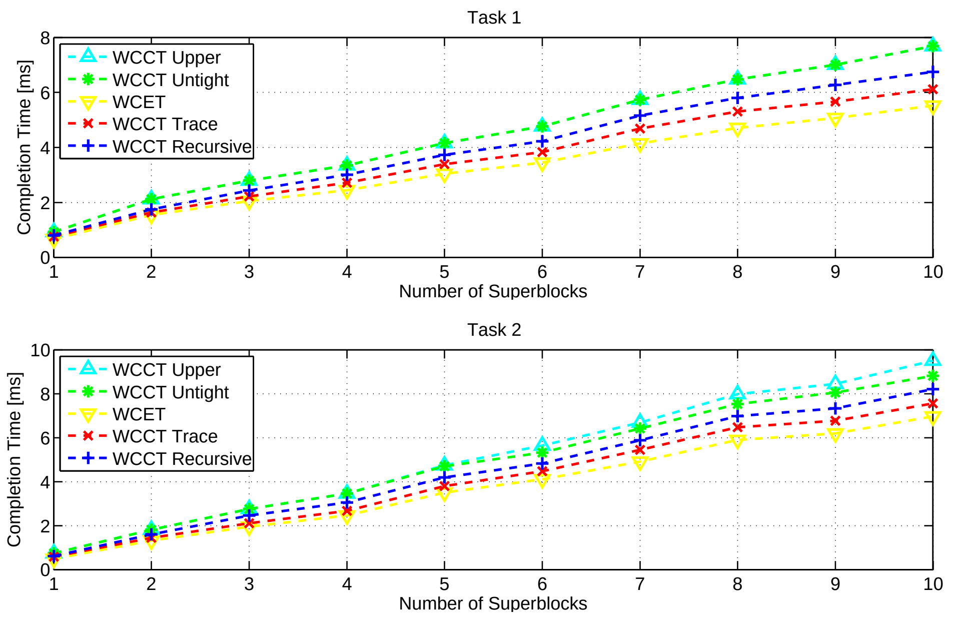


Figure 4.10: Several completion times for two example tasks with FCFS arbitration.

difference between the untight WCCT and the traced WCCT is about 24 % for the first and 22.5 % for the second task. Although it is not safe to take the traced completion time as WCCT, this relative difference indicates that the WCCT computed by Algorithm 1 is not tight for this specific setup.

Nevertheless, Fig. 4.10 also indicates that the untight WCCT for the corresponding general and hybrid tasks must lie between the upper and the untight WCCT. The former is trivial because the upper WCCT cannot be exceeded by any task model. The latter is a conclusion of the schedulability relationship between the different access models [11]. This relationship states that the WCCT of a general task is at least the WCCT of the corresponding hybrid task, which is for its part at least the WCCT of the dedicated task. Although mentioned in the context of TDMA arbitration, the justification for FCFS policy is analogical.

The time complexity of the untight and the recursive algorithm is compared in Fig. 4.11. The time measurement was executed on a mobile Intel Pentium Dual-Core 64-bit processor with 2 GHz frequency. Only the dedicated access model is analyzed because even for three superblocks, the analysis for the general tasks becomes infeasible due to the combinatorial explosion of considering all possibilities. The figure shows that the time consumption of the recursive approach grows exponentially because the whole sequence of superblocks is taken into account. Hence, the algorithm is only computable

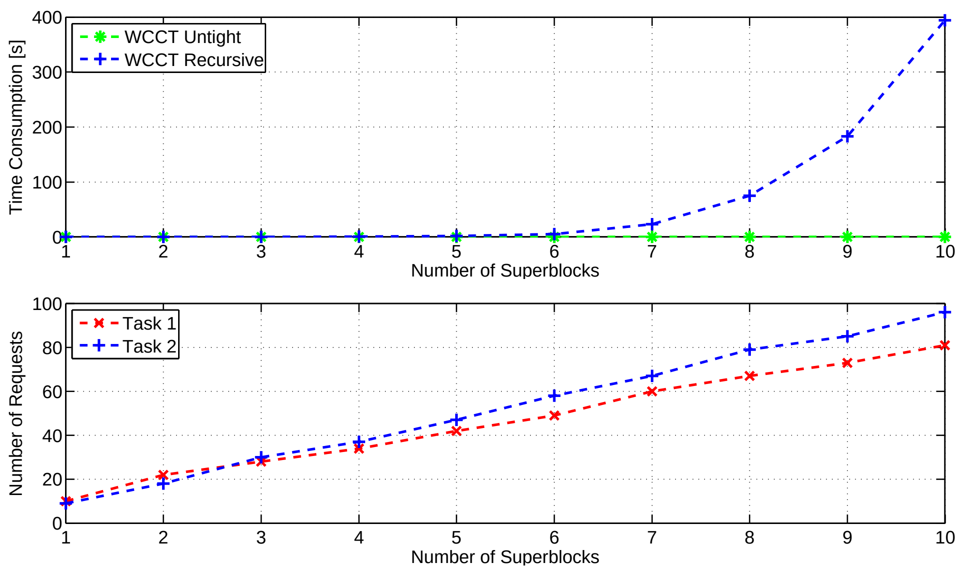


Figure 4.11: Comparison of the time consumption of the algorithms for an increasing number of resource requests (data and instruction).

for a small number of superblocks.

4.5.2 TDMA Arbitration

In this section, the timing behavior for TDMA arbitration is analyzed. The system containing two TDMA buses is compared to the system with one TDMA bus as well as to the original crossbar topology. For the FCFS arbitration, the untight bound of Algorithm 1 is taken because it is the only safe bound for all access models and for an arbitrary number of processing elements.

4.5.2.1 Experimental Setup

In the following, the configuration of the schedulers is described. This configuration is only chosen for this specific setup and might be far from optimal. In general, it is the task of the system designer to choose these parameters optimally. This leads to a multi-objective optimization for a given set of tasks, which can be solved with evolutionary algorithms [23]. For this experiment, each processing element has the same TDMA bandwidth by choosing all the slot lengths equally:

$$\delta_j^\mu = \delta^\mu,\ \delta_j^\nu = \delta^\nu, \quad \forall j \in \{1, \dots, |\mathcal{P}|\} \tag{4.18}$$

The cycle lengths are configured taking different slot lengths:

$$\begin{aligned} L(\Theta_\mu) &= |\mathcal{P}| \cdot \delta^\mu, \quad \delta^\mu \in \{30, 40, \ldots, 100\}\,\mu s \\ L(\Theta_\nu) &= |\mathcal{P}| \cdot \delta^\nu, \quad \delta^\nu \in \{20, 30, \ldots, 50\}\,\mu s \end{aligned} \tag{4.19}$$

The relative starting times of the TDMA slots are defined by Eq. (4.20). The TDMA schedule Θ_μ dedicated for accessing the shared memory begins with a slot for processing element p_1 and each succeeding core p_j has the next active slot. The TDMA schedule Θ_ν granting access to the instruction flash is configured with the first slot assigned to $p_{j'}$ with $j' = \lfloor |\mathcal{P}| \div 2 \rfloor$, which is the "middle" processing element. Equal to Θ_μ, the slot for each subsequent processing element is activated afterwards.

$$\sigma_j^\mu = (j-1) \cdot \delta^\mu \bmod L(\Theta_\mu), \quad \sigma_j^\nu = \left(j - \left\lfloor \frac{|\mathcal{P}|}{2} \right\rfloor \right) \cdot \delta^\nu \bmod L(\Theta_\nu) \tag{4.20}$$

For the following experiment, the system contains ten processing elements ($|\mathcal{P}| = 10$) and task τ_{10} is analyzed. Every task consists of 20 superblocks and is repeated after the period $W_j = 200\,ms$. In fact, the interfering tasks are only relevant for calculating the WCCT for FCFS. Moreover, the schedule Θ_μ is taken for the single TDMA bus topology because the slot length has to be at least $\max(C_\mu, C_\nu) = 30\,\mu s$.

4.5.2.2 Results

For the single TDMA bus, the difference between the computed untight WCCT of FCFS and the tight WCCT of TDMA is shown in Fig. 4.12 for different slot lengths δ plotted on the x-axis. Positive bars indicate those slot lengths, where the worst-case completion time of the system with TDMA arbitration is smaller than the untight WCCT with FCFS arbitration (additionally marked with an arrow). Although the results are highly dependable of the chosen parameters, there exists a slot length for this concrete case, for which the TDMA system performs better than the original crossbar system. This slot length is actually the minimum slot length of $30\,\mu s$ but the improvement is rather small.

Figure 4.13 shows the same comparison for the dual TDMA bus. Different slot lengths δ^μ and δ^ν are plotted on the x- and y-axis. Introducing a second TDMA bus increases the design space and consequently the possible slot lengths, for which the TDMA outperforms the WCCT of the FCFS system. In some cases, only the task with dedicated access has a positive difference.

Taking the communication times as slot lengths minimizes the overall WCCT for TDMA and is moreover smaller than the best WCCT of the single TDMA

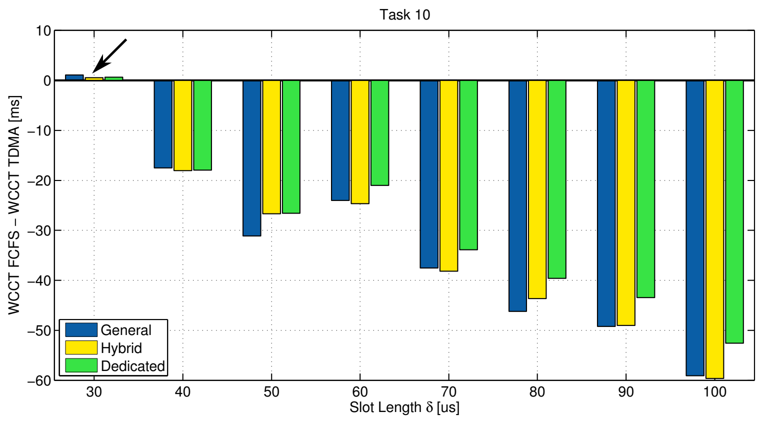


Figure 4.12: Difference of the WCCT of FCFS and single bus TDMA.

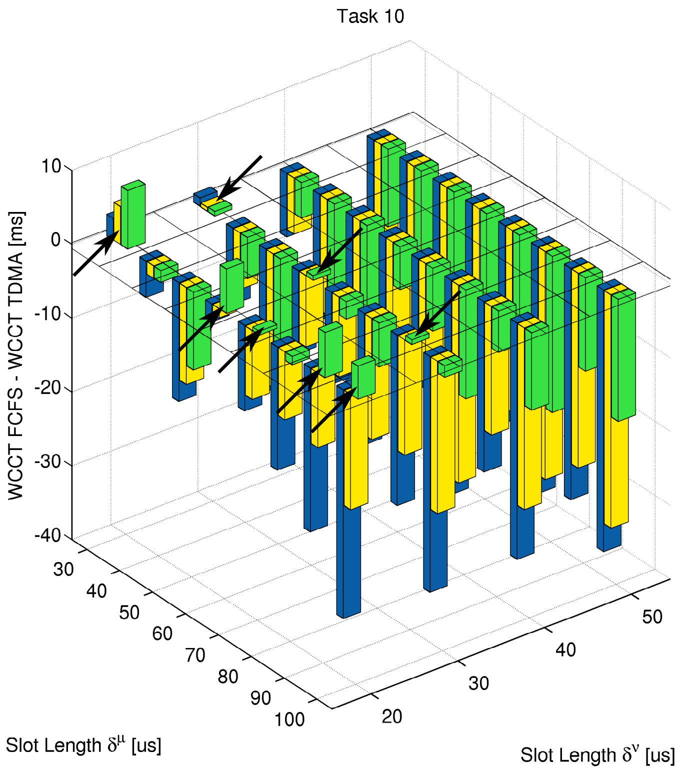


Figure 4.13: Difference of the WCCT of FCFS and dual bus TDMA.

system. Instead of giving each task the same TDMA bandwidth, it would make sense to prefer those tasks with many resource accesses. For instance, one possibility would be to give tasks with many instruction fetches higher bandwidth for the instruction flash and tasks with many data requests larger slot lengths for the shared memory. However, the optimal design of the schedulers is out of scope.

4.6 Conclusions

This chapter showed the difficulties that a second shared resource in a multiprocessor system introduces under FCFS arbitration. Taking all variabilities like communication and instruction time into account is still an open problem. The main challenge is to find a computationally efficient method to determine the worst-case completion times of tasks. Several approaches failed because it was not possible to reduce the large state space by solving smaller subproblems. In the end, it was only feasible to analyze the system by inserting structure in terms of the dedicated task model and reducing the uncertainty of the cache profile and the memory latencies. However, this leads to restrictive assumptions which do not really reproduce the behavior of modern hardware systems. All in all, we are dealing with the trade-off between scalable performance analysis and degree of freedom concerning less restrictive assumptions.

One possible solution is to design the tasks in a time-triggered way, i.e., the superblocks are activated at predefined time instances. Then, the problem state space is reduced enabling the recursive approaches to terminate after each superblock. On the other hand, the experiments showed that it is possible to design an equivalent TDMA architecture which increases timing predictability. The advantage of analyzing a system with TDMA compared to FCFS is that the presented methodology is scalable with respect to the number of tasks and the number of superblocks. This is why the TDMA arbitration is our preferred solution.

For TDMA, the problem of finding good slot lengths for each task and schedule is a multi-objective optimization problem. The objective is to minimize the WCCT of all tasks choosing from different possible slot lengths. One approach to solve such problems is covered by the theory of evolutionary algorithms. Finally, a way to estimate the WCCT of a task executed in the original crossbar topology considering variable communication times and cache profiles is proposed as future work.

5 Conclusions and Outlook

5.1 Conclusions

In the first part of this thesis, we showed the efforts needed to implement a predictable TDMA scheduler on an unpredictable MPSoC, namely the Cell Broadband Engine. The Element Interconnect Bus (EIB) of this hardware architecture is designed for high average throughput and high average latency. However, the measurements on this hardware system yield that this design strategy has the drawback of reduced timing predictability. There are two reasons for analyzing the timing behavior of the EIB. First, only average values are published by the developers of the Cell Broadband Engine. For the worst-case analysis however, only upper bounds are safe. And since the exact structure of the EIB is unknown, it had to be assumed as black box. Second, the measurements were necessary in order to estimate the parameters needed for the performance analysis. In the theoretical model, these parameters are assumed to be given, but only elaborate determination of them enable the derivation of methodologies to calculate the WCCT of a task.

In particular, the communication time is the time to serve one request to the shared resource assuming no interference. This parameter is critical for the performance analysis: On the one hand, its exact value has a large impact on the result. On the other hand, the fact that it is often only possible to determine an upper bound for a hardware system, results in overestimation of the WCCT. Although the bound is safe, it reduces the timing predictability in the sense that the difference between the actual worst case and the resulting bound increases. For instance, cache mechanisms like a translation look-aside buffer (TLB) are applied for optimizing the average latency, but the average and the worst case drift apart.

The second critical parameter is the synchronization overhead. This is the upper bound of the message sending delays between the scheduler (PPE) and the processing element (SPE) for one TDMA slot. The actual sending latency influences each effective slot length that is available to the SPE. Similar to the communication time, the actual values are smaller than the bound for most of our experiments, which also results in overestimation of the WCCT. Furthermore, this parameter determines the minimal length of each slot, and hence, the granularity of the scheduler. Due to the large distribution of the measured values, we decided to sacrifice on predictability and to opt for a highly reliable system, i.e., 99.999 % reliability. This is not a solution for a hard real-time system, but provides reasonable guarantees for many industrial embedded systems. A reduced minimal slot length is essential because the WCCT of a task is increased the larger the TDMA slot length becomes.

In the second part of this thesis, the theoretical model has been extended with a second globally shared resource – an instruction flash. The execution of an instruction on a processing element requires an instruction fetch first. Consequently, a task executed on a processing element can be delayed due to contention on both shared resources. The worst-case behavior has been analyzed for FCFS as well as for TDMA arbitration in order to find bounds that guarantee real-time constraints. For FCFS, finding a tight WCCT is complicated as this dynamic policy reduces the analyzability of the system. Several effects have been characterized that make the performance analysis difficult. For instance, the memory interference delay depend on all tasks and due to the domino effect, obtaining a WCCT for one task under analysis is only possible by considering the whole sequence of superblocks on all processing elements. Thus, the challenge is to find a computationally efficient method, which is still an open problem.

One solution is to restrict the assumptions by assuming constant communication times and instruction times. But as a matter of fact, the assumptions of the theoretical model are a trade-off between simplifying the performance analysis and accuracy of the result. In the case of TDMA, this normally results in overestimation. For FCFS however, too restrictive assumptions can lead to unsafe bounds. Therefore, our recommendation is to avoid FCFS for multiple shared resources if timing predictability is required and to use the static TDMA arbitration instead. The introduced structure increases the analyzability of the system by explicitly removing memory interference. A further advantage of TDMA is that the time complexity is proportional to the number of tasks under analysis. And the experiments showed that there exist schedulers, for which the tight WCCT of TDMA is smaller than the untight WCCT of FCFS.

The practical experiments on the Cell Broadband Engine approved the schedulability relationship between the access models. To be more specific, the dedicated access model, where communication is separated from execution, has the smallest WCCT compared to the other models. And the difference to the maximally measured completion time is the smallest for this access model. The schedulability relationship also holds for two shared resources. In fact, when applying FCFS, only the dedicated model could be analyzed while the analysis of the other models turned out to be too complex. To recapitulate, it is a necessity to introduce structure in terms of static arbitration such as TDMA and dedicated access model to achieve analyzable systems.

5.2 Outlook

Due to limited time, several aspects were not analyzed in detail. Concerning the synchronization of the processing elements, there are some promising possibilities to reduce the large distribution of the synchronization overhead. For example, one solution is to use memory spinlock techniques: The synchronization is realized with DMA transfers and synchronization variables. This approach has been briefly taken into consideration, but abandoned because the implemented solution only marginally reduces the maximum measured delays. However, a more elaborate method would possibly improve the results. Another solution is to define one SPE as master that synchronizes the other SPEs. At least, the available average latencies between two SPEs indicate a possible improvement.

Only little time has been spent on the design of the TDMA schedulers for the two shared resources. Depending on the tasks, the slot lengths can be optimized. One idea is to take the structures of the tasks into account. For example, the bandwidth of a task to the resources can be related to the number of accesses of this task in relation to the other tasks accesses. A more general way is to use evolutionary multiobjective optimization algorithms to find optimal slot lengths.

A
Technical Issues Concerning the Cell Broadband Engine

A.1 Mailbox Functionality

A.1.1 SPU Channels

Each SPE has an incoming (*SPU Read Inbound Mailbox*) and an outgoing (*SPU Write Outbound Mailbox*) mailbox managed by the MFC as it has been summarized in Section 2.1. In this thesis, only blocking functions are considered. For the SPE, the functions for accessing the mailboxes are provided by the library `spu_mfcio.h` (see Chapter 4 of [24], *Programming Support for MFC Input and Output*):

```
#include <stdint.h>
#include <spu_mfcio.h>

int main()
{
    uint32_t msg = 0;

    // reading from the inbound mailbox
    msg = spu_read_in_mbox();

    // writing to the outbound mailbox
    spu_write_out_mbox(msg);

    return 0;
}
```

For the PPE, the runtime management library `libspe2.h` (see Chapter 7 of [25], *SPE MFC Problem State Facilities*) is responsible for the mailbox handling. The following listing shows a basic example of how to use the provided mailbox functions as well as a simple error treatment:

```c
#include <stdint.h>
#include <libspe2.h>
#include <assert.h>

int main()
{
    uint32_t msg = 0;
    int stat = 0;

    // create context
    spe_context_ptr_t ctxs = spe_context_create(0, NULL);

    // writing to the inbound mailbox
    stat = spe_in_mbox_write(ctxs, &msg, 1, SPE_MBOX_ALL_BLOCKING);
    assert(stat == 1);

    // reading from the outbound mailbox
    while (stat == 0) {
        stat = spe_out_mbox_read(_ctxs, msg, 1);
    }
    assert(stat == 1);

    return 0;
}
```

This is the standard way to access mailboxes and is accomplished by the SPU channels. The runtime management library functions for the PPE that provide this possibility invoke a system call [8]. The performance of mailboxes can be improved by directly accessing the corresponding MMIO registers of an SPE. Note that this improvement is only possible for communication between PPE and SPE and not for two SPEs. Chapter *Direct SPE access for applications* in [25] gives a general overview of this possibility. The final solution as it is used for this thesis is presented in the following section.

A.1.2 Memory-Mapped I/O

First of all, the context of an SPE needs to be created on the PPE with the special flag `SPE_MAP_PS` in order to access the SPE's memory-mapped registers as described in [25]. These registers reside in the so called *Problem-State Area* of an SPE (see chapter *Problem-State Memory-Mapped Registers* of [6]). The address of this area is obtained by calling the function `spe_ps_area_get`.

Basically, the functions `_spe_in_mbox_write` and `_spe_out_mbox_read` included by the library `cbe_mfc.h` access the addresses `SPU_Out_Mbox` and

SPU_In_Mbox relatively to the beginning of the Problem-State Area. The following listing for the PPE only shows the essential parts:

```c
#include <stdint.h>
#include <libspe2.h>
#include <cbe_mfc.h>

int main()
{
    uint32_t msg = 0;

    // create context
    spe_context_ptr_t ctxs = spe_context_create(SPE_MAP_PS, NULL);

    // access problem state area
    spe_spu_control_area_t ps = (spe_spu_control_area_t*)
        spe_ps_area_get(ctxs, SPE_CONTROL_AREA);

    // writing to the inbound mailbox
    _spe_in_mbox_write(ps, &msg);

    // reading from the outbound mailbox
    msg = _spe_out_mbox_read(ps);

    return 0;
}
```

A.2 Time Measurement

A.2.1 Power Processor Element

On the PPE, the *GNU C Library* [26] and its functionality are available and therefore, several methods for time measurement are supported. The function `clock_gettime` is used to retrieve following high-resolution clock values:

- **Real Time:** System-wide real-time clock.
- **Process Time:** High-resolution per-process timer from the CPU.

The function returns the measured time in the structure `timespec` which contains two fields – one for seconds and one for nanoseconds. For our measurement, this structure is converted to microseconds, which is sufficiently accurate. Since there is a need of a system-wide time measurement, the real-time clock is used. An SPE cannot read this value, which would be inaccurate anyway due to the delay between the SPE and the PPE. This is a general problem of distributed systems.

A faster way for measuring the elapsed time is to work with the PPE's 64-bit *time-base* register. Actually, every time function of the GNU C Library somehow abstracts this clock, but might invoke several system calls. By directly accessing this register, much overhead can be saved resulting in a more accurate time measurement. The easiest but not most intuitive way for reading the time-base register is provided by the assembler intrinsics library `ppu_intrinsics.h` (see chapter *PPU-Specific Intrinsics* of [25]):

```c
#include <ppu_intrinsics.h>
#include <stdint.h>

int main()
{
    uint64_t clk = __mftb();
    return 0;
}
```

This function returns the PPE's clock value, which is updated at a ticking rate called *time-base frequency* [15]. Its value is $f_{TB} = 79.8$ MHz for the Sony PlayStation 3 and $f_{TB} = 25$ MHz for the IBM Full-System Simulator. To obtain the time value, the clock value is converted to time units:

$$t = \frac{clk}{f_{TB}} \; [s] \qquad \text{(A.1)}$$

A.2.2 Synergistic Processor Element

Each SPE has its own 32-bit decrementer that can be used for time measurements. It is actually a 32-bit register comparable to the PPE's time-base register, but in contrast, the clock value is decremented each tick. Instead of directly calling the SPE decrementer, the SPU Timer Library [27] offers a more convenient way for reading the clock value. It provides a virtual, monotonically increasing 64-bit clock with the same time-base frequency as for the PPE. Thus, the conversion to time units happens equally.

A further advantage of the SPU Timer Library is that the interrupt handler `spu_clock_slih` for the *SPU Decrementer Event* is provided (see section *SPU Event Definitions* of [6]). This event happens when the decrementer has reached zero and the value has to be reset, which is the case every $\frac{2^{32}}{79.8 \cdot 10^6} = 53.821\ldots$ seconds for the PlayStation 3. An example and more details concerning the interrupt handling are shown in the next section.

A.3 Interrupt Handling on the SPE

In order to separate the actual execution from the synchronization part, an interrupt handler is used on the SPE. To avoid the periodic polling of the inbound mailbox for new mails, the *SPU Inbound Mailbox available event* as described in section *SPU Event Definitions* of [6] can be used. This event generates an interrupt which is handled by the implemented interrupt-handling function.

Unfortunately, interrupt handling is not well documented for the Cell Broadband Engine. Principally, there exist two different levels for interrupt handling. The *First-Level Interrupt Handler* (FLIH) has to be implemented with assembler instructions and is therefore complicated and unsafe to use. The *Second-Level Interrupt Handler* (SLIH) is a more user-friendly solution and its usage is actually very easy.

First, the interrupt-handling function is registered with `spu_slih_register` and the event handling is activated by writing the event mask with the function `spu_write_event_mask`. There is just one thing that has to be considered for more than one events: To avoid overwriting the currently active handled events, the new one has to be OR'ed with the old ones reading them with `spu_read_event_mask`. Second, the interrupt-handling function unmasks the handled event with bit manipulation.

According to chapter *Synergistic Processor Unit Channels*, section *SPU Event Definitions* of [6], the *SPU Inbound Mailbox Available Event* is raised if the channel count of the inbound mailbox changes from zero (empty) to non-zero (not-empty). Hence, it is important to read all available messages until the mailbox becomes empty again. The complete procedure can be found in the following listing. The first function `mfc_in_mbox_event` shows the usage of the interrupt handler, while the main function presents the registration of events. Note that the additional *SPU Decrementer Event* is not relevant for the *SPU Inbound Mailbox Available Event*, but the example shows the usage of two different events.

```c
#include <stdint.h>
#include <spu_mfcio.h>

mfc_in_mbox_event(uint32_t status)
{
    // remove handled event from event mask
    status &= ~MFC_IN_MBOX_AVAILABLE_EVENT;

    uint32_t ppe_msg;
    while (spu_stat_in_mbox() > 0)
    {
        // reading inbound mailbox
        ppe_msg = spu_read_in_mbox();

        // processing of message
        ...
    }

    return status;
}

int main()
{
    uint32_t mask;

    // register inbound mail available event
    spu_slih_register(MFC_IN_MBOX_AVAILABLE_EVENT, mfc_in_mbox_event);
    mask = spu_read_event_mask();
    spu_write_event_mask(mask | MFC_IN_MBOX_AVAILABLE_EVENT);

    // register decrement interrupt handler
    spu_slih_register(MFC_DECREMENTER_EVENT, spu_clock_slih);
    mask = spu_read_event_mask();
    spu_write_event_mask(mask | MFC_DECREMENTER_EVENT);

    ...

    return 0;
}
```

A.3.1 Interrupt-Safe Critical Sections

During a running DMA transfer, the program should not be interrupted (see section *Interrupt-Safe Critical Sections* in [24]). The MFC mnemonic `mfc_begin_critical_section` marks the beginning of the critical section and returns the current interrupt state, which is restored after the critical section with `mfc_end_critical_section`:

```
    // begin of critical section
    uint32_t mach_stat = mfc_begin_critical_section();

    ...

    // end of critical section
    mfc_end_critical_section(mach_stat);
```

B

Implemented Software

B.1 Scheduling Framework

This application implements a TDMA scheduler on top of the Cell Broadband Engine. The PPE is used as scheduler and the SPEs execute an assigned task.

B.1.1 Requirements

Following external libraries have been used:

- RapdiXML as XML Parser:
  http://rapidxml.sourceforge.net
- Google CTemplate as Template Engine:
  http://ctemplate.sourceforge.net

B.1.2 Source

- This folder contains the source code:

  `source/sched`

- This folder contains the MATLAB files, configuration and results:

  `matlab/cell_tdma`

B.1.3 Usage

The program is called by specifying an XML file as parameter:

```
./sched xml-file
```

Thus, the TDMA scheduler is completely configurable by the specified XML file. Its format is covered in the subsequent sections and summarized in the following. Placeholders are represented by "...":

```xml
<?xml version="1.0" encoding="utf-8"?>
<ppe>
    <configuration>
        ...
    </configuration>
    <scheduler type="..." mode="...">
        ...
    </scheduler>
    <processing>
        ...
    </processing>
</ppe>
```

B.1.3.1 Configuration

This part is used for the general settings:

```xml
<configuration>
    <delay name="mailbox">
        <length unit="(us|ms|s)">...</length>
    </delay>
    <delay name="memory">
        <length unit="(us|ms|s)">...</length>
    </delay>
    <delay name="start">
        <length unit="(us|ms|s)">...</length>
    </delay>
    <!-- Template File for MATLAB -->
    <file name="template" path="..." />
    <file name="matlab" path="..." />
</configuration>
```

Parameter	Description
`delay name="mailbox"`	upper bound of mailbox sending latency
`delay name="memory"`	upper bound of DMA transfer latency
`delay name="start"`	starting delay between first synchronization and TDMA schedule
`file name="template"`	template file for the MATLAB output file
`file name="matlab"`	based on the template file, this MATLAB file is generated containing the results of the measurement

B.1.3.2 Scheduler

This part is used for the scheduler specific configuration:

```
<scheduler type="(tdma|none)" mode="(terminate|skip_window)">
    <duration>
        <length unit="(us|ms|s)">...</length>
    </duration>
    <window>
        <length unit="(us|ms|s)">...</length>
    </window>
     <slot spe="(0|...|7)">
        <length unit="(us|ms|s)">...</length>
    </slot>
    <slot...
    </slot>
    ...
</scheduler>
```

Parameter	Description
`scheduler type`	type of scheduler: `tdma`: use TDMA scheduling `none`: use no scheduling
`scheduler mode`	reaction of scheduler if violation of upper bound of mailbox sending latency: `terminate`: terminate the measurement `skip_window`: skip to the next time window
`window`	length of time window
`slot spe`	slot length for specified SPE

B.1.3.3 Processing

This part is used for the task specific configuration:

```
<processing>
    <task spe="(1|...|7)">
        <period>
            <length unit="(us|ms|s)"></length>
        </period>
        <superblock iterations="...">
            <acq mu="..." />
            <exe mu="..." lambda="..." />
            <rep mu="..." />
        </superblock>
        <superblock...
        </superblock>
        ...
    </task>
    <task...
    </task>
    ...
</processing>
```

Parameter	Description
`task spe`	task configuration for specified SPE
`period`	processing cycle
`superblock iterations`	superblock repeated specified times
`acq mu`	number of memory accesses during acquisition phase
`exe mu`	number of memory accesses and …
`exe lambda`	… computations during execution phase
`rep mu`	number of memory accesses during replication phase

B.2 Measurement Tools

B.2.1 Mailbox Functions

This application measures the time behavior of mailbox functions and prints the resulting values. The PPE sends a dummy value to the SPE, which immediately acknowledges the reception. The measured values are:

- Reading the mailbox: Time to read from the mailbox.
- Writing the mailbox: Time to write to the mailbox.
- The sum: Called round-trip time.

B.2.1.1 Source

- This folder contains the source code:

  ```
  source/mbox_rtt
  ```

- This folder contains the MATLAB files and results:

  ```
  matlab/cell_rtt
  ```

B.2.1.2 Usage

```
./mbox_rtt #spu_threads #iterations outputmode
          mailboxmode limit [filename]
```

Parameter	Description
`#spu_threads`	number of SPU threads: For each thread, a whole measurement round with `#iterations` cycles will be executed.
`#iterations`	number of iterations to execute per thread
`outputmode`	`output`: the result after each cycle is printed `silent`: only the maximum and the average values are displayed
`mailboxmode`	`libspe`: standard method using SPU channels `cbemfc`: direct method using MMIO registers
`limit`	number in the range $[0, 1]$: Corresponds to reliability
`filename`	if specified, the result is appended to the file instead of using the standard output. Useful for large number of iterations

B.2.2 DMA Functions

This application measures the time behavior of the DMA functions of the SPE and prints the resulting values. The following values are measured:

- Reading from the memory: Time of DMA get operation.
- Writing to the memory: Time of DMA put operation.

B.2.2.1 Source

- This folder contains the source code:

  `source/dma_transfer`

- This folder contains the MATLAB files and results:

  `matlab/cell_dma`

B.2.3 Usage

```
./dma_transfer  #iterations outputmode
```

Parameter	Description
`#iterations`	number of iterations to execute per thread
`outputmode`	`output`: the result after each cycle is printed `silent`: only the maximum and the average values are displayed

B.3 Analysis Tools

B.3.1 First-Come, First-Served

B.3.1.1 Source

This folder contains the MATLAB files and results:

`matlab/wcct_fcfs`

B.3.1.2 Functions

- `calc_wcct(task, C, start_time)`
  Calculates the worst-case completion time (WCCT) of the specified set of two tasks accessing two shared resources with FCFS arbitration policy.
- `calc_wcct_trace(task, C, start_time)`
  Calculates the worst-case completion time (WCCT) of the specified set of tasks considering all possible traces.
- `calc_wcct_untight(task, C, start_time)`
  Calculates an untight worst-case completion time (WCCT) of the specified set of tasks accessing two shared resources with FCFS arbitration policy.
- `calc_wcct_upper(task, C)`
  Calculates the maximal possible worst-case completion time (WCCT) of the specified set of tasks assuming worst-case delay for each resource request.
- `calc_wcet(task, C)`
  Calculates the worst-case execution time (WCET) of the specified set of tasks.
- `get_max_requests(task, state, delta, C)`
  Returns the maximum number of data and instruction accesses for a given `task` and time window `delta`. Only for dedicated tasks!
- `get_min_delay(phase, delta, p, C)`
  Returns the minimum delay a task can suffer during time window `delta` assuming `p` interfering tasks.
- `get_phases(blocks)`
  Transforms a sequence of superblocks to a sequence of phases, which are basically general superblocks.

- `get_wcet_of_phase(phase, C)`
  Returns the worst-case execution time (WCET) of the specified phase.

- `get_worst_case_delay(task, state, delta, C)`
  Returns the worst-case delay of two tasks. For a specific time window `delta`, the worst-case delay is obtained by assuming the worst-case number of interfering data accesses and instruction fetches.

- `perform_request(task, C)`
  Performs a request simulating the behavior of a (perfect) shared resource, i.e., the communication time is assumed to be constant.

- `update_state(task, state, delta, delay)`
  Updates the remaining time of a phase for each task. If delta becomes zero, the next phase is activated. If the last phase has finished, the WCCT set accordingly.

- `update_task(task)`
  Activates the next superblock of task.

- `wcct_recursive(task, state, C, max_time)`
  Recursively determines the WCCT of the tasks of current state until `max_time`.

- `wcct_trace(task, C)`
  Returns the WCCT of the tasks by recursively determine each possible trace.

B.3.2 Time Division Multiple Access

B.3.2.1 Source

This folder contains the MATLAB files and results:

`matlab/wcct_tdma`

B.3.2.2 Functions

- `calc_wcct(task, sched_data, sched_inst, C_data, C_inst)`
  Calculates the worst-case completion time (WCCT) of the specified set of tasks with two TDMA schedules for data and instruction cache.

- `get_phases(blocks)`
  Transforms a sequence of superblocks to a sequence of phases, which are basically general superblocks.

- `wct_ar(sched, accesses, current_time, C)`
  Calculates the worst-case completion time of a dedicated phase with specified number of accesses.

-
  ```
  wct_e(sched_data, sched_inst, accesses, instructions,
        current_time, C_data, C_inst, inst_time)
  ```
  Calculates the worst-case completion time of a general phase with specified number of data accesses and instructions with maximal instruction time.

Bibliography

[1] L. Thiele and R. Wilhelm, "Design for timing predictability," *Real-Time Systems*, vol. 28, no. 2, pp. 157–177, 2004.

[2] H. Kopetz and G. Grunsteidl, "TTP-A time-triggered protocol for fault-tolerant real-time systems," in *Fault-Tolerant Computing, 1993. FTCS-23. Digest of Papers., The Twenty-Third International Symposium on*, pp. 524–533, IEEE, 2002.

[3] R. Wilhelm, D. Grund, J. Reineke, M. Schlickling, M. Pister, and C. Ferdinand, "Memory Hierarchies, Pipelines, and Buses for Future Architectures in Time-Critical Embedded Systems," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 28, no. 7, pp. 966–978, 2009.

[4] S. Schliecker, M. Negrean, and R. Ernst, "Bounding the shared resource load for the performance analysis of multiprocessor systems," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2010*, pp. 759–764, IEEE, 2010.

[5] 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*, pp. 245–254, ACM, 2009.

[6] International Business Machines Corporation, Sony Computer Entertainment Incorporated, Toshiba Corporation, *Cell Broadband Engine Architecture, Version 1.02*, October 2007.

[7] M. Kistler, M. Perrone, and F. Petrini, "Cell Multiprocessor Communication Network: Built for Speed," *IEEE micro*, vol. 26, no. 3, pp. 10–23, 2006.

[8] J. Abellan, J. Fernandez, and M. Acacio, "CellStats: A Tool to Evaluate the Basic Synchronization and Communication Operations of the Cell BE," in *Proceedings of the 16th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP 2008)*, pp. 261–268, IEEE Computer Society, 2008.

[9] International Business Machines Corporation, Sony Computer Entertainment Incorporated, Toshiba Corporation, *Software Development Kit for Multicore Acceleration, Programming Tutorial, Version 3.1, Chapter Programming the SPEs*, 2008.

[10] A. Schranzhofer, J. Chen, and L. Thiele, “Timing Analysis for TDMA Arbitration in Resource Sharing Systems,” in *2010 16th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 215–224, IEEE, 2010.

[11] A. Schranzhofer, R. Pellizzoni, J. Chen, L. Thiele, and M. Caccamo, “Worst-Case Response Time Analysis of Resource Access Models in Multi-Core Systems,” in *Design Automation Conference (DAC), 2010 47th ACM/IEEE*, pp. 332–337, IEEE, 2010.

[12] R. Pellizzoni, A. Schranzhofer, J. Chen, M. Caccamo, and L. Thiele, “Worst Case Delay Analysis for Memory Interference in Multicore Systems,” in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2010*, pp. 741–746, IEEE, 2010.

[13] D. Bovet, M. Cesati, and A. Oram, *Understanding the Linux Kernel.* O’Reilly & Associates, Inc. Sebastopol, CA, USA, 2002.

[14] J. Gray and D. Siewiorek, “High-Availability Computer Systems,” *Computer*, vol. 24, no. 9, pp. 39–48, 2002.

[15] International Business Machines Corporation, Sony Computer Entertainment Incorporated, Toshiba Corporation, *Cell Broadband Engine Programming Handbook, Version 1.12, Chapter Time Base and Decrementers*, April 2009.

[16] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström, “The worst-case execution-time problem — overview of methods and survey of tools,” *ACM Trans. Embed. Comput. Syst.*, vol. 7, no. 3, pp. 1–53, 2008.

[17] A. Schranzhofer, R. Pellizzoni, J. Chen, L. Thiele, and M. Caccamo, “Timing Analysis for Resource Access Interference on Adaptive Resource Arbiters.” Submitted to the *17th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), 2011.*

[18] L. Thiele, S. Chakraborty, and M. Naedele, “Real-time calculus for scheduling hard real-time systems,” in *International Symposium on Circuits and Systems ISCAS 2000*, vol. 4, (Geneva, Switzerland), pp. 101–104, 2000.

[19] M. Lv, N. Guan, W. Yi, and G. Yu, "Combining Abstract Interpretation with Model Checking for Timing Analysis of Multicore Software,"

[20] Bertsekas, Dimitri P., *Dynamic Programming and Optimal Control.* Athena Scientific, 1995.

[21] J. Bengtsson and W. Yi, "Timed Automata: Semantics, Algorithms and Tools," *Lectures on Concurrency and Petri Nets*, pp. 87–124, 2004.

[22] G. Behrmann, A. David, and K. Larsen, "A Tutorial on Uppaal," *Formal methods for the design of real-time systems*, pp. 33–35, 2004.

[23] A. Eiben and J. Smith, *Introduction to Evolutionary Computation.* Natural Computing Series, Springer, 2003.

[24] International Business Machines Corporation, Sony Computer Entertainment Incorporated, Toshiba Corporation, *C/C++ Language Extensions for Cell Broadband Engine Architecture, Version 2.6*, August 2008.

[25] International Business Machines Corporation, Sony Computer Entertainment Incorporated, Toshiba Corporation, *SPE Runtime Management Library, Version 2.3*, October 2008.

[26] Free Software Foundation Incorporated, *The GNU C Library.* Online Reference: http://www.gnu.org/software/libc/.

[27] International Business Machines Corporation, Sony Computer Entertainment Incorporated, Toshiba Corporation, *SPU Timer Library Programmer's Guide and API Reference, Version 3.0*, October 2007.