Linköping Studies in Science and Technology
Thesis No. 1503

Predictable Real-Time Applications on Multiprocessor Systems-on-Chip

by

Jakob Rosén

Submitted to Linköping Institute of Technology at Linköping University in partial fulfilment of the requirements for degree of Licentiate of Engineering

Department of Computer and Information Science
Linköping University
SE–581 83 Linköping, Sweden

Linköping 2011

This is a Swedish Licentiate's Thesis.
The Licentiate's degree comprises 120 ECTS credits of postgraduate studies.

ISBN 978-91-7393-090-1, ISSN 0280-7971
Printed by LiU-Tryck, Linköping, Sweden 2011

Electronic version:
http://urn.kb.se/resolve?urn=urn:nbn:se:liu:diva-70138

PRESS PLAY ON TAPE

Predictable Real-Time Applications on Multiprocessor Systems-on-Chip

by

Jakob Rosén

September 2011
ISBN 978-91-7393-090-1
Linköping Studies in Science and Technology
Thesis No. 1503
ISSN 0280-7971
LIU-TEK-LIC-2011:42

ABSTRACT

Being predictable with respect to time is, by definition, a fundamental requirement for any real-time system. Modern multiprocessor systems impose a challenge in this context, due to resource sharing conflicts causing memory transfers to become unpredictable. In this thesis, we present a framework for achieving predictability for real-time applications running on multiprocessor system-on-chip platforms. Using a TDMA bus, worst-case execution time analysis and scheduling are done simultaneously. Since the worst-case execution times are directly dependent on the bus schedule, bus access design is of special importance. Therefore, we provide an efficient algorithm for generating bus schedules, resulting in a minimized worst-case global delay.

We also present a new approach considering the average-case execution time in a predictable context. Optimization techniques for improving the average-case execution time of tasks, for which predictability with respect to time is not required, have been investigated for a long time in many different contexts. However, this has traditionally been done without paying attention to the worst-case execution time. For predictable real-time applications, on the other hand, the focus has been solely on worst-case execution time optimization, ignoring how this affects the execution time in the average case. In this thesis, we show that having a good average-case global delay can be important also for real-time applications, for which predictability is required. Furthermore, for real-time applications running on multiprocessor systems-on-chip, we present a technique for optimizing for the average case and the worst case simultaneously, allowing for a good average case execution time while still keeping the worst case as small as possible. The proposed solutions in this thesis have been validated by extensive experiments. The results demonstrate the efficiency and importance of the presented techniques.

This research work was funded in part by CUGS (the National Graduate School of Computer Science, Sweden).

Department of Computer and Information Science
Linköpings universitet
SE–581 83 Linköping, Sweden

Acknowledgments

So, here I am. The end of this winding road has been reached, and the thesis is finally ready. However, I would never have come to this point without the many people who gave me support, inspiration and courage during the past years. Let me first start by thanking my supervisors Zebo Peng and Petru Eles, for giving me the opportunity to become a graduate student and for always believing in me. I am very grateful for that. The working environment at IDA has been excellent, and I wish I could thank every one of my colleagues individually. If you work here and read this, consider yourself thanked!

Carl-Fredrik Neikter did a fantastic job contributing to the framework which this thesis is built upon, and I truly enjoyed our collaboration. Alexandru Andrei spent many late nights helping me with technical issues during the writing of our first paper, and for that I am thankful. I also want to thank Soheil Samii and Sergiu Rafiliu for our frequent and (usually) very enjoyable "board meetings", where we discussed all kinds of subjects related to our work.

Anton Blad and Fredrik Kuivinen have been my friends for a long time, and I really enjoyed the luxury of also having them as colleagues for a while. In particular, I want to thank them for the fun moments we spent developing the TRUT64 device (and testing it!), and of course also for just being such great guys. Furthermore, I want to express gratitude to Traian Pop for introducing me to the world of research in a friendly manner, and for throwing the best birthday parties. A big thank you must also go to everyone who participated in the official duck feeding sessions at the university pond, and to the geese and the ducks for kindly accepting the bread that I brought.

Finally, I thank my family for giving me love and support. Always.

Jakob Rosén

Linköping, August 2011

Contents

List of Figures

Abbreviations

ACET	Average-Case Execution Time
ACGD	Average-Case Global Delay
BSA	Bus Scheduling Approach
CFG	Control Flow Graph
DS	Density Region Based Sizes
ISS	Initial Slot Sizes
NWCET	Naive Worst-Case Execution Time
QoS	Quality of Service
SSA	Slot Size Adjustments
TDMA	Time Division Multiple Access
WCET	Worst-Case Execution Time
WCGD	Worst-Case Global Delay

1

Introduction

Embedded real-time systems have become a key part of our society, helping us in almost every aspect of our daily living. Of significant importance is the class of safety-critical embedded real-time systems, to which we entrust our lives, for instance at hospitals, in aeroplanes and in cars. These systems must be reliable and, consequently, it is of crucial importance that they are *predictable* with respect to time. However, predictability is desirable not only for this kind of traditional hard real-time systems, but for any system exhibiting real-time properties.

This thesis describes techniques for designing predictable embedded real-time systems in a multiprocessor environment. Besides serving as an introduction to the topic, this chapter presents a summary of the state-of-the-art research within the field. After briefly describing the contributions of this thesis, the chapter ends with an outline of what follows next.

1.1 Multiprocessor Real-Time Systems

For real-time systems, correctness of a program not only depends on the produced computational results, but also on its ability to deliver these on time, according to specified time constraints. Therefore, for a real-time application, predictability with respect to time is of uttermost importance. The obvious example is safety-critical hard real-time systems, such as medical and avionic applications, for which failure to meet a specified deadline not only renders the computations useless, but also can have catastrophic consequences. However, predictability is getting more and more desirable also for other classes of embedded applications, for instance within the domains of multimedia and telecommunication, for which QoS guarantees are desired [8].

As these kinds of applications become increasingly complex, they also require more computational power in terms of hardware resources. Generally, the trend in processor design is to increase the number of cores as means to improve the performance and power efficiency, and microprocessors with hundreds of cores are expected to arrive on the market in the not-so-distant future [4]. In order to satisfy the demands of complex and resource-demanding embedded systems, for which predictability is required, multicore systems implemented on a single chip are used to an increasing extent [16, 36].

To achieve predictability with respect to time, schedulability analysis techniques are applied, assuming that the worst-case execution time (WCET) of every task is known. A lot of research has been carried out within the area of worst-case execution time analysis [21]. However, according to the proposed techniques, each task is analyzed in isolation, as if it was running on a monoprocessor system. Consequently, it is assumed that memory accesses over the bus take a constant amount of time to process, since no bus conflicts can occur. For multiprocessor systems with a shared communication infrastructure, however, transfer times depend on the bus load and are therefore no longer constant, causing the traditional methods to produce incorrect results [32]. The main obstacle when performing timing analysis on multiprocessor systems is that the scheduling of tasks assumes that their worst-case execution times are known, but to calculate these worst-case execution times, knowledge about the task schedule is required. Clearly, the traditional method of separating WCET analysis and task scheduling no longer works, and new approaches are required.

Of great interest when measuring the execution time of a real-time

application is the global delay, defined as the time it takes to execute the application from its beginning to the very end. In this thesis, we propose an approach for designing predictable real-time embedded systems on multiprocessor system-on-chip architectures. We show how it is possible to design predictable systems using a TDMA bus architecture, and we also propose algorithms for generating intelligent bus schedules minimizing the worst-case global delay (WCGD) of the application.

Furthermore, we take a new approach to hard real-time system design by, in addition, also considering the effects on the average-case global delay (ACGD) while making sure that the WCGD is kept to a near-minimum. Optimization techniques for improving the average case execution time of an application, for which predictability with respect to time is not required, have been investigated in nearly every scientific discipline involving a computer. However, this has traditionally been done without paying attention to the worst-case execution time. For predictable applications, on the other hand, the focus has been solely on worst-case execution time optimization, which still is a hot research topic [6, 7]. To the best of our knowledge, this is the first time the combination of these two concepts has been investigated within the context of achieving predictability.

1.2 Related Work

The fundament for achieving predictability is worst-case execution time (WCET) analysis, and a lot of research has been carried out within this area. Wilhelm et al. present an overview of the existing methods and tools [33]. None of them can, however, be applied directly to multiprocessor systems with a shared communication infrastructure, since these techniques assume a monoprocessor environment. Yan and Zhang present a new approach for worst-case execution time analysis on multicore processors with shared L2 caches [37]. They describe their work as a first, important step towards a complete framework rather than a full solution to the problem.

One approach is to use an additive bus model, assuming that conflicts on the bus do not affect the calculated worst-case execution times significantly compared to when running the same program on a monoprocessor platform. It has been shown that this is a good assumption if the bus load is kept below 60% [2], but even for such low bus loads, no worst-case execution time guarantees can be made. Furthermore,

increasing the number of processor cores will also increase the bus congestion [35] and, thus, the additive bus model is likely to not perform well for future architectures, even when strict time-predictability is not required.

Schoeber and Puschner present a technique for achieving predictability on TDMA bus-based chip multiprocessors [30]. Similar to the approach in this thesis, the output from the worst-case execution time analysis is used to improve the bus schedule. However, in order to avoid the problem of scheduling tasks, they assume that the number of cores are greater than the number of tasks.

In a recently published paper, Lv et al. use abstract modeling in Uppaal to calculate the worst-case execution time of tasks running on multiprocessor systems with a shared communication infrastructure [14]. The contribution of their approach is a very general framework with support for many kinds of buses. However, their solution does not handle computer architectures exhibiting timing anomalies.

Within the context of response time analysis [9], Schliecker et al. propose a technique using accumulated busy times instead of considering each memory access individually [29]. The result is a framework for multiprocessor analysis that computes worst-case response times with good precision, but without providing any hard timing guarantees. A more recent approach by the same authors provides conservative bounds, but without support for noncompositional architectures [28]. Schranzhofer et al. present a technique for response time analysis for TDMA bus-based multiprocessor systems with shared resources [31]. The analysis is safe, but does not support the presence of timing anomalies. Pellizzoni and Caccamo propose an approach for calculating the delay caused by bus interference for tasks running on systems with several connected peripherals [19]. They also provide a corresponding schedulability analysis framework.

Edwards and Lee argue in favor of hardware customized for achieving timing predictability, in contrast to today's platforms optimized solely for good average case performance [5]. Lickly et al. present an example of one such processor [12]. However, no such hardware exists on the market today. Paolieri et al. propose a predictable multiprocessor hardware architecture, using custom bus arbiters, designed for running hard and soft real-time tasks concurrently [17]. For hard real-time tasks, it provides a maximum bound on the memory access transfer time. The big advantage of this approach is that traditional worst-case execution

time analysis techniques can be used without modifications. However, applications with many hard real-time tasks will make this upper bound become large, potentially increasing the pessimism.

1.3 Contribution

The main contributions of this thesis are:

1. We propose a novel technique to achieve predictability on multiprocessor systems by doing worst-case execution time analysis and scheduling simultaneously [1, 23, 24]. With respect to a given TDMA bus schedule, tasks are scheduled at the same time as their worst-case execution times are calculated, and the resulting worst-case global delay of the application is obtained.

2. To generate good bus schedules, we have constructed efficient optimization algorithms that minimize the worst-case global delay of the given application [23].

3. Combining optimization for the worst case and the average case, we have developed an approach to achieve a good average-case global delay while still keeping the worst-case delay as small as possible [25].

1.4 Thesis Organization

The remaining part of the thesis is outlined as follows. In the next chapter, the system model is described in detail. The overall approach for achieving predictability is then presented in Chapter 3. In Chapter 4, we discuss the underlying worst-case analysis framework necessary for implementing our approach. Chapter 5 presents algorithms for optimization of the worst-case global delay using several bus scheduling approaches. Experimental results are presented at the end of the chapter. The algorithms for combining WCGD and ACGD optimization are presented in Chapter 6, together with a motivation for why the average case is important also for predictable real-time systems. The chapter ends with experimental results validating our approach. Finally, Chapter 7 presents our conclusions.

2

System Model

This chapter starts by describing the hardware platform that is assumed throughout the rest of the thesis. Next, the software application model is explained. Finally, we describe the model of the TDMA-based communication infrastructure.

2.1 Hardware Architecture

As hardware platform, we have considered a multiprocessor system-on-chip architecture with a shared communication infrastructure, as shown in Figure 2.1, typical for the new generation of multiprocessor system-on-chip designs [11]. Each processor has its own cache for storing data and instructions, and is connected to a private memory via the bus. For interprocessor communication, a shared memory is used. All memory accesses to the private memories are cached, as opposed to accesses to the shared memory which, in order to avoid cache coherence problems, are not cached. All memory devices are accessed using the same, shared bus. However, in the case of private memory accesses, the bus is used only when an access results in a cache miss.

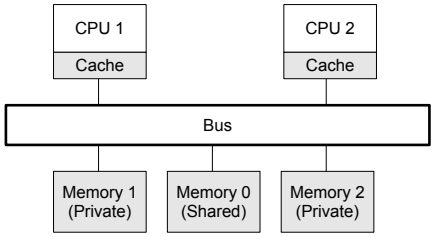


Figure 2.1 *Hardware Model*

Within the context of worst-case execution time analysis, hardware platforms can be divided into compositional architectures and noncompositional architectures [34], depending on whether or not the platform exhibits timing anomalies [13, 22]. Timing anomalies occur when a local worst-case scenario, such as a cache miss instead of a hit, does not result in the worst case globally. This complicates the worst-case execution time analysis significantly, since no local assumptions can be made. Compositional architectures, such as the ARM7, do not exhibit timing anomalies, and the analysis can therefore be divided into disjunct subproblems, simplifying the analysis procedure. Noncompositional architectures, on the other hand, require a far more complicated and time-consuming analysis. The PowerPC 775 is an example of a noncompositional architecture [34]. As will be described further on, our approach works for both compositional architectures and noncompositional architectures.

2.2 Application Model

The functionality of a software application is captured by a directed acyclic task graph, $G(\Pi, \Gamma)$. Its nodes Π represent computational tasks, and the edges Γ represent data dependencies between them. A task cannot start executing before all of its input data is available. Communication between tasks mapped on the same processor is performed by using the corresponding private memory, and is handled in the same way as other memory requests during the execution of a task. Interprocessor communication, or so called *explicit communication*, is done via the shared memory and is modeled as two communication tasks – one for transmitting and one for receiving – in the task graph. The transmitting communication task is assigned to the same processor as

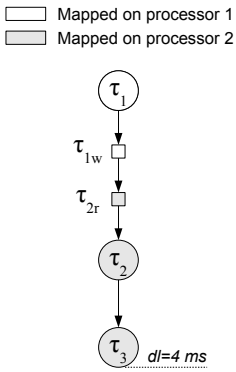


Figure 2.2 *Task graph*

the task that is sending data to the shared memory, and similarly the receiving communication task is assigned to the processor fetching the same data. An example is shown in Figure 2.2 where τ_{1w} and τ_{2r} represent the transmitting and receiving task, respectively.

A computational task cannot communicate with other tasks during its execution, which means that it will not access the shared memory. However, the task is accessing data, used in the computations, from its private memory and program instructions are continuously fetched. Consequently, the bus is accessed every time a cache miss occurs, resulting in what we define as *implicit communication*. As opposed to explicit communication, implicit communication has not been taken into account in previous approaches for real-time application system-level scheduling and optimization [20, 27].

The task graph has a deadline which represents the maximum allowed execution time of the entire application, known as the maximum global delay. Individual tasks can have deadlines as well. The example task graph in Figure 2.2 has a global delay of 4 milliseconds. The application is assumed to be running periodically, with a period greater than or equal to the application deadline.

2.3 Bus Model

A precondition for achieving predictability is to use a predictable bus architecture. Therefore, we are using a TDMA-based bus arbitration policy, which is suitable for modern system-on-chip designs with QoS

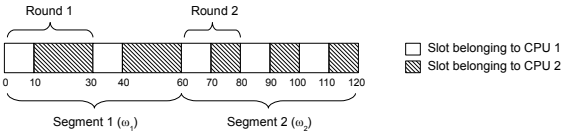


Figure 2.3 *Example of a bus schedule*

constraints [8, 18, 26].

The behavior of the bus arbiter is defined by the *bus schedule*, consisting of sequences of slots representing intervals of time. Each slot is owned by exactly one processor, and has an associated start time and an end time. Between these two time instants, only the processor owning the slot is allowed to use the bus. A bus schedule is divided into *segments*, and each segment consists of a *round*, that is, a sequence of slots that is repeated periodically. See Figure 2.3 for an example.

The bus arbiter stores the bus schedule in a dedicated external memory, and grants access to the processors accordingly. If processor CPU_i requests access to the bus in a time interval belonging to a slot owned by a different processor, the transfer will be delayed until the start of the next slot owned by CPU_i. A bus schedule is defined for one period of the application, and is then repeated periodically. A table representation of the bus schedule in Figure 2.3 can be found in Figure 2.4.

To limit the required amount of memory on the bus controller needed to store the bus schedule, a TDMA round can be subject to various complexity constraints. A common restriction is to let every processor own, at most, a specified number of slots per round. Also, one can let the sizes be the same for all slots of a certain round, or let the slot order be fixed.

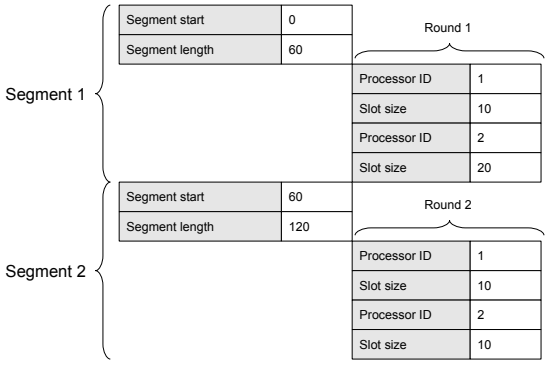


Figure 2.4 *Bus schedule table representation*

3

Predictability Approach

This chapter begins with a motivational example illustrating the problems encountered when designing predictable multiprocessor-based real-time systems. It then continues by describing our overall approach to achieve predictability for such systems.

3.1 Motivational Example

Consider two tasks running on a multiprocessor system with two processors and a shared communication infrastructure according to Chapter 2. Each task has been analyzed with a traditional WCET tool, assuming a monoprocessor system, and the resulting Gantt chart of the worst-case scenario is illustrated in Figure 3.1a. The dashed intervals represent cache misses, each of them taking six time units to serve, and the white solid areas represent segments of code not using the bus. The task running on processor 2 is also, at the end of its execution, transferring data to the shared memory, and this is represented by the black solid area.

However, since the tasks are actually running on a multiprocessor system with a shared communication infrastructure, they do not have

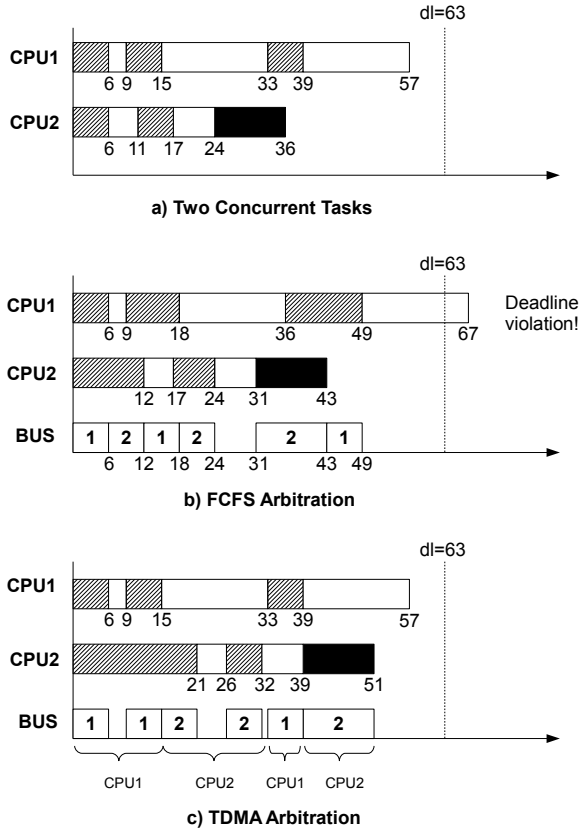


Figure 3.1 *Motivational example*

exclusive access to the bus handling the communication with the memories. Hence, some kind of arbitration policy must be applied to distribute the bus bandwidth among the tasks. The result is that when two tasks request the bus simultaneously, one of them has to wait until the other has finished transferring. This means that transfer times are no longer constant. Instead, they now depend on the bus conflicts resulting from the execution load on the different processors. Figure 3.1b shows the corresponding Gantt chart when the commonly used FCFS arbitration policy is applied.

The fundamental problem when performing worst-case execution time analysis on multiprocessor systems is that the load on the other processors is in general not known. For a task, the number of cache

misses and their location in time depend on the program control flow path. This means that it is very hard to foresee where there will be bus access collisions, since this will differ from execution to execution. To complicate things further, the worst-case control flow path of the task will change depending on the bus load originating from the other concurrent tasks. In order to solve this and introduce predictability, we use a TDMA bus schedule which, a priori, determines exactly when a processor is granted the bus, regardless of what is executed on the other processors. Given a TDMA bus schedule, the WCET analysis tool calculates a corresponding worst-case execution time. Some bus schedules will result in relatively short worst-case execution times, whereas others will be very bad for the worst case. Therefore, it is important that a clever bus schedule, optimized to reduce the worst case, is used. Algorithms for this will be presented in Chapter 5. Note that regardless of what bus schedule is given as input to the WCET analysis algorithm, the corresponding worst-case execution time will always be safe. Figure 3.1c shows the same task configuration as previously, but now the memory accesses are arbitrated according to a TDMA bus schedule.

3.2 Overall Approach

For a task running on a multiprocessor system, as described in Chapter 2, the problem for achieving predictability is that the duration of a bus transfer depends on the bus congestion. Since bus conflicts depend on the task schedule, WCET analysis cannot be performed before that is known. However, task scheduling traditionally assumes that the worst-case execution times of the tasks to be scheduled are already calculated.

To solve this circular dependency, we have developed an approach based on the following principles:

1. A TDMA-based bus access policy, according to Section 2.3, is used for arbitration. The bus schedule, created at design time, is enforced during the execution of the application.

2. The worst-case execution time analysis is performed with respect to the bus schedule, and is integrated with the task scheduling process, as described in Figure 3.3.

We illustrate our overall approach with a simple example. Consider the application in Figure 3.2a. It consists of three tasks – τ_1, τ_2 and τ_3 – mapped on two processors. The static cyclic scheduling process is

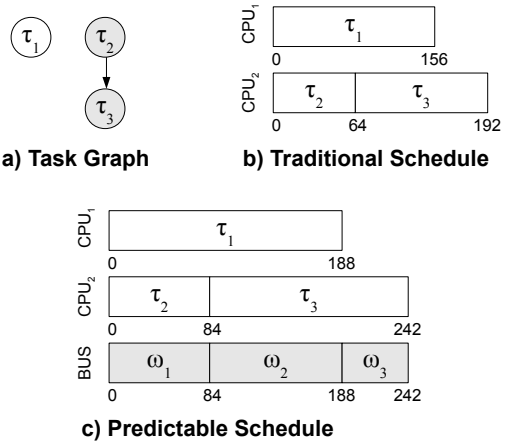


Figure 3.2 *Overall approach example*

based on a list scheduling technique, and is performed in the outer loop described in Figure 3.3 [10]. Let us, as is done traditionally, assume that worst-case execution times have been obtained using techniques where each task is considered in isolation, ignoring conflicts on the bus. These calculated worst-case execution times are 156, 64, and 128 time units for τ_1, τ_2, and τ_3, respectively. The deadline is set to 192 time units, and would be considered as satisfied according to traditional list scheduling, using the already calculated worst-case execution times, as shown in Figure 3.2b. However, this assumes that no conflicts, extending the bus transfer durations (and implicitly the memory access times), will ever occur on the bus. This is, obviously, not the case in reality and thus results obtained with the previous assumption are wrong.

In our predictable approach, the list scheduler will start by scheduling the two tasks τ_1 and τ_2 in parallel, with start time 0, on their respective processor (line 2 in Figure 3.3). However, we do not yet know the end times of the tasks, and to gain this knowledge, worst-case execution time analysis has to be performed. In order to do this, a bus schedule which the worst-case execution times will be calculated with respect to (line 6 in Figure 3.3) must be selected. This bus schedule is, at the moment, constituted by one bus segment ω, as described in Section 2.3. Given this bus schedule, worst-case execution times of tasks τ_1 and τ_2 will be computed (line 7 in Figure 3.3). Based on this output, new bus schedule candidates are generated and evaluated (lines 5-8 in Figure 3.3), with the goal of obtaining those worst-case execution times that

lead to the shortest possible worst-case global delay of the application.

Assume that, after selecting the best bus schedule, the corresponding worst-case execution times of tasks τ_1 and τ_2 are 167 and 84 respectively. We can now say the following:

- Bus segment ω_1 is the first segment of the application bus schedule, and will be used for the time interval 0 to 84.
- Both tasks τ_1 and τ_2 start at time 0.
- In the worst case, τ_2 ends at time 84 (the end time of τ_1 is still unknown, but it will end later than 84).

Now, we go back to step 3 in Figure 3.3 and schedule a new task, τ_3, on processor CPU_2. According to the previous worst-case execution time analysis, task τ_3 will, in the worst case, be released at time 84, scheduled in parallel with the remaining part of task τ_1. A new bus segment ω, starting at time 84, will be selected and used for analyzing task τ_3. For task τ_1, the already fixed bus segment ω_1 is used for the time interval between 0 and 84, after which the new segment ω is used. Once again, several bus schedule candidates are evaluated, and finally the best one, with respect to the worst-case global delay, is selected. Assume that the segment ω_2 is finally selected, and that the worst-case execution times for tasks τ_1 and τ_3 are 188 and 192 respectively, making task τ_3 end at 276. Now, ω_2 will become the second bus segment of the application bus schedule, ranging from time 84 to 188, and this part of the bus schedule will be fixed. Now, we repeat the same procedure with the remaining part of τ_3 (which now ends at time 242 instead of 276, since ω_3 assigns all bus bandwidth to CPU_2). The final, predictable schedule is shown in Figure 3.2c, and leads to a WCGD of 242.

An outline of the algorithm can be found in Figure 3.3. We define Ψ as the set of tasks active at the current time t, and this is updated in the outer loop. In the beginning of the loop, a new bus segment ω, starting at t, is generated and the resulting bus schedule candidate is evaluated with respect to each task in Ψ. Based on the outcome of the WCET analysis, the bus segment ω is improved for each iteration. The bus segments previously generated before time t remain unaffected. After selecting the best segment ω, θ is set to the end time of the task in Ψ that finished first. The time t is updated to θ and we continue with the next iteration of the outer loop.

Communication tasks are treated as a special class of computational tasks, which are generating a continuous flow of cache misses with no

```
01:  θ=0
02:  while not all tasks scheduled
03:    schedule new task at t ≥ θ
04:    Ψ=set of all tasks that are active at time t
05:    repeat
06:      select bus segment ω for the
           time interval starting at t
07:      determine the WCET of all tasks in Ψ
08:    until termination condition
09:    θ=earliest time a task in Ψ finishes
10:  end while
```

Figure 3.3 *Overall approach*

computational cycles in between. The number of cache misses is specified such that the total amount of data transferred on the bus, due to these misses, equals the maximum length of the explicit message. Therefore, from an analysis point of view, no special treatment is needed for explicit communication. In the following, when we talk about cache misses, it applies to both explicit and implicit communication.

4

Worst-Case Execution Time Analysis

In order to calculate the WCET of a task, the analysis needs to be aware of the TDMA bus, taking into account that processors must be granted the bus only during their assigned time slots. This chapter describes the modifications required in order to adopt a traditional WCET algorithm to our predictable approach, without increasing the overall time-complexity.

4.1 TDMA-Based WCET Analysis

Performing worst-case execution time analysis with respect to a TDMA bus schedule requires not only knowledge about the number of cache misses for a certain program path, but also their location with respect to time. Hence, traditional ILP-based methods for worst-case execution time analysis cannot be applied. Instead, each memory access needs to be considered with respect to the bus schedule, granting access to the bus only during the slots belonging to the requesting processor. However, to collect the necessary information used by our worst-case execution time analysis framework, the same techniques used in traditional

methods can be utilized.

Calculating the worst-case execution time has to be done with respect to the particular hardware architecture on which the task being analyzed is going to be executed. Factors such as the instruction set, pipelining complexity, caches and so on must be taken into account by the analysis. For an application running on a compositional architecture, the analysis can be divided into subproblems processed in a local fashion, for instance on basic block level. We can be sure that the local worst-case always contributes to the worst-case globally, allowing for fast analysis techniques without the need to analyze every single program path individually. This is, unfortunately, not the case when using noncompositional architectures. The presence of timing anomalies will force the analysis to consider all possible program paths explicitly, naturally causing the analysis time to explode as the size of the tasks increase.

For a predictable multiprocessor system with a shared communication structure, as described in Chapter 2, it is necessary to search through all feasible program paths and match each possible bus transfer to slots in the actual bus schedule, keeping track of exactly when a bus transfer is granted the bus in the worst case. This means that the execution time of a basic block will vary depending on when it is executed. Fortunately, for an application running on a compositional architecture, efficient search-tree pruning techniques dramatically reduce the search space, allowing for local analysis, just as for traditional WCET techniques.

4.2 Compositional WCET Analysis Flow

A typical program flow for a WCET tool operating on compositional architectures is shown in the left path of Figure 4.1 [33]. First, a control flow graph (CFG) is generated. A value analysis is then performed to find program characteristics such as data address ranges and loop bounds. To take into account performance-enhancing features of modern hardware, cache and pipeline analyses are carried out next. A path analysis identifies the feasible paths and an ILP formulation for calculating the worst-case program path is then produced. The information traditionally provided in this ILP formulation is, however, not sufficient for calculating the WCET on a multiprocessor system since not only the number of cache misses are needed for each basic block, but

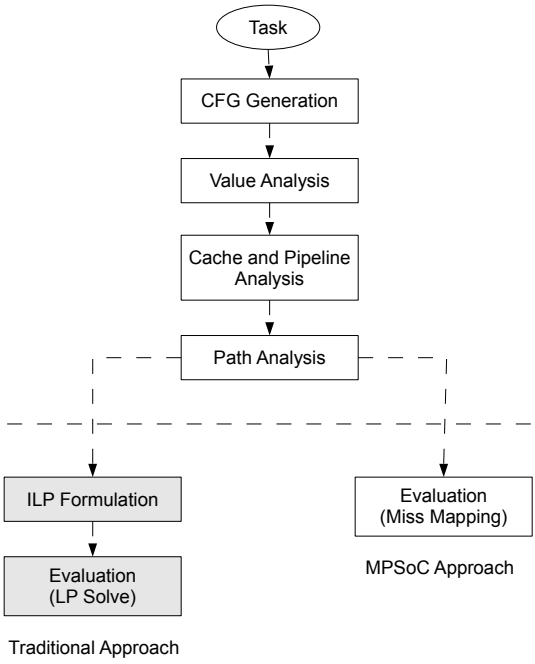


Figure 4.1 *WCET tool program flow*

also their positions with respect to time. If necessary, an underlying WCET tool has to be modified to provide this information. A more in-depth description can be found in the work published by Neikter [3].

Our TDMA-based approach for compositional WCET analysis is illustrated in the right path of Figure 4.1. After the path analysis, the information from the previous steps is used to calculate the worst-case program path by mapping the cache misses to the corresponding bus slots in the TDMA schedule. We will now show the idea behind this with a simple example.

4.2.1 Monoprocessor WCET Example

Consider a task τ executing on a system with two processors (processor 1 and processor 2). The task is being mapped on processor 1, and has start time 0. First, an annotated control flow graph, as illustrated in Figure 4.2, is constructed. The rectangular elements **B**, **C**, **H**, **E**, **F** in

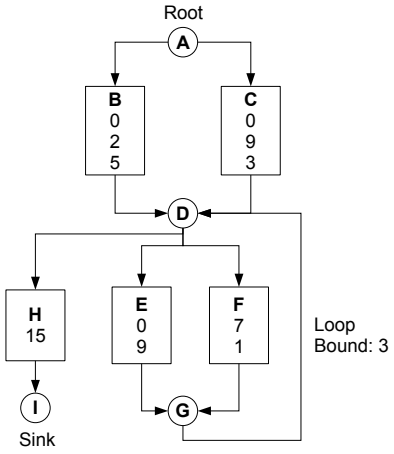


Figure 4.2 *Example CFG*

the graph represent basic blocks, and the circles **A**, **D**, **G**, **I** represent control nodes gluing them together. The loop starting at control node **G** will run at most three times, so the loop bound is consequently set to 3. The annotated numbers in the basic blocks represent consecutive cycles of execution, in the worst case, not accessing the bus. For instance, basic block **B** will, when executed, immediately – after 0 clock cycles – issue a cache miss. After this, 2 cycles will be spent without bus accesses before the next (and last) cache miss occurs. Finally, 5 bus access-free cycles will be executed before the basic block ends. Hence, the execution time of basic block **B** will be $(0 + k_1 + 2 + k_2 + 5)$ where k_1 and k_2 represent the transfer times of the first and second cache miss respectively. Note that usually, loop unrolling is performed in order to decrease the pessimism of the analysis. This example is, however, purposely kept as simple as possible, and therefore the loop has not been unrolled.

For a typical monoprocessor system, all cache misses take the same constant amount of time to process, and the execution time of basic block **B** would be known immediately. However, for multiprocessor architectures such as the one described in Chapter 2, we must calculate the individual transfer times with respect to a given TDMA schedule.

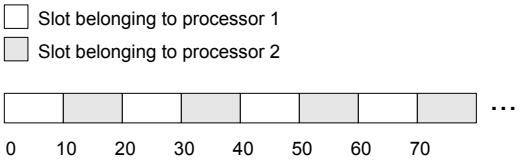


Figure 4.3 *Example TDMA bus schedule*

4.2.2 Multiprocessor WCET Example

Instead of a monoprocessor system, assume a multiprocessor system, as described in Chapter 2, using the bus schedule in Figure 4.3. Processor 1, on which the task is running, gets a bus slot of size 10 processor cycles periodically assigned to it every 20th cycle. In this particular example, a cache miss takes 10 cycles for the bus to transfer, resulting in the bus being granted to processor 1 *only* at times t satisfying $t \equiv 0 \pmod{20}$, where $\equiv$ is the congruence operator.

To calculate the worst-case program path, we must evaluate all feasible program paths in the control flow graph. In the very simple example in Figure 4.2, there are 30 program paths[1] to explore, growing exponentially with the number of branches and loop bounds. Fortunately, due to the nature of the compositional architecture and the TDMA bus, not all of them have to be investigated explicitly. In fact, in a task graph with all loops unrolled, each basic block would need to be investigated exactly once, as will be explained in the following.

Let us denote the worst-case start time of a basic block $\mathbf{Z}$ by $s(\mathbf{Z})$, and the end time in the worst case by $e(\mathbf{Z})$. The execution time of a basic block $\mathbf{Z}$, in the worst case, is then defined as $w(\mathbf{Z}) = e(\mathbf{Z}) - s(\mathbf{Z})$. Without considering bus conflicts, as in traditional methods, the worst-case execution time of the basic blocks would be $w_{\text{trad}}(\mathbf{B}) = 27, w_{\text{trad}}(\mathbf{C}) = 32, w_{\text{trad}}(\mathbf{E}) = 19, w_{\text{trad}}(\mathbf{F}) = 18$ and $w_{\text{trad}}(\mathbf{H}) = 15$. The corresponding worst-case program path becomes $\mathbf{C}, \mathbf{E}, \mathbf{E}, \mathbf{E}, \mathbf{H}$ resulting in a worst-case execution time of $27{+}19{\cdot}3{+}15 = 104$ clock cycles. However, this assumes that all cache misses take the same amount of time to transfer, and this is false in a multiprocessor system with a shared communication structure. In our TDMA-based approach, the execution time of a basic block depends on its start time in relation to the bus schedule. We start from the root node and successively calculate the execution time of each basic block with respect to the worst-case

[1] $2 + 2^2 + 2^3 + 2^4 = 30$

start time. At the same time, the worst-case path is calculated.

With respect to the TDMA schedule in figure 4.3, the worst-case start times of the basic blocks connected directly to the root node is 0, since they will never execute at any other time instant. The execution time of block **B**, in the worst case, is $w(\mathbf{B}) = 0 + 10 + 2 + 18 + 5 = 35$ whereas the corresponding execution time of block **C** is $w(\mathbf{C}) = 0+10+9+11+3 = 33$. Note that $w(\mathbf{B}) > w(\mathbf{C})$, even though the relation is the opposite in the traditional case above where $w_{\text{trad}}(\mathbf{B}) < w_{\text{trad}}(\mathbf{C})$. In order to decide which one of these two basic blocks is on the critical path, two very important observations must be made based on the predictable nature of the TDMA bus (and the compositionality considered in this section).

1. *The absolute end time of a basic block can never increase by letting it start earlier.* That is, a basic block **Z** with $s(\mathbf{Z}) = x$ and $e(\mathbf{Z}) = y$, any start time $x' < x$ will result in an end time $y' \leq y$. The execution time of the particular basic block can increase, but the increment can never exceed the difference $x - x'$ in start time. This means that for a basic block **Z**, the basic block will never end later than $e(\mathbf{Z})$ as long as it start before (or at) $s(\mathbf{Z})$. This guarantees that the worst-case calculations will never be violated, no matter what program path is taken. Note that $w(\mathbf{Z})$ is the execution time in the worst case, with respect to $e(\mathbf{Z})$, and that the time spent by executing **Z** can be greater than $w(\mathbf{Z})$ for an earlier start time than $s(\mathbf{Z})$.

2. Consider a basic block **Z** with worst-case start time $s(\mathbf{Z}) = x$ and worst-case end time $e(\mathbf{Z}) = y$. If we, instead, assume a worst-case start time of $s(\mathbf{Z}) = x''$ where $x'' > x$, the corresponding resulting absolute end time $e(\mathbf{Z}) = y''$ will always satisfy the relation $y'' \geq y$. This means that the greatest assumed worst-case start time $s(\mathbf{Z})$ will also result in the greatest absolute end time $e(\mathbf{Z})$.

Based on the second observation, we can be sure that the maximum absolute end time for the basic block (**E**, **F** or **H**) succeeding **B** and **C** will be found when the worst-case start time is set to 35 rather than 33. Therefore, we conclude that **B** is on the worst-case program path and, since they are not part of a loop, **B** and **C** do not have to be investigated again.

Next follow three choices. We can enter the loop by executing either **E** or **F**, or we can go directly to **H** and end the task immediately. Due

to observation 2 above, we can conclude that the worst-case absolute end time of **H**, and thus the entire task, will be achieved when the loop iterates the maximum possible number of times, which is 3 iterations, since that will maximize $s(\mathbf{H})$. Therefore, the next step is to calculate the worst-case execution time for basic blocks **E** and **F** respectively for each of the three iterations, before finally calculating the worst-case execution time of **H**. In the first iteration, the worst-case start time is $s(\mathbf{E_1}) = s(\mathbf{F_1}) = 35$ and the execution times become $w(\mathbf{E_1}) = 0+15+9 = 24$ and $w(\mathbf{F_1}) = 7+28+1 = 36$ for **E** and **F** respectively. We conclude that the worst-case program path so far is $\mathbf{B}, \mathbf{F}$ and the new start time is set to $s(\mathbf{E_2}) = s(\mathbf{F_2}) = 35 + 36 = 71$. In the second loop iteration, we get $w(\mathbf{E_2}) = 0+19+9 = 28$ and $w(\mathbf{F_2}) = 7+12+1 = 20$. Hence, in this iteration, **E** contributes to the worst-case program path and the new worst-case start time becomes $s(\mathbf{E_3}) = s(\mathbf{F_3}) = 99$. In the final iteration, the execution times are $w(\mathbf{E_3}) = 0 + 11 + 9 = 20$ and $w(\mathbf{F_3}) = 7 + 24 + 1 = 32$ respectively, resulting in the new worst-case start time $s(\mathbf{H}) = 131$. We now know that the worst-case program path is $\mathbf{B}, \mathbf{F}, \mathbf{E}, \mathbf{F}, \mathbf{H}$, and since **H** contains no cache misses, and therefore always takes 15 cycles to execute, the WCET of the entire task is $e(\mathbf{H}) = 146$.

As shown in this example, in a loop-free control flow graph, each basic block has to be visited once. For control flow graphs containing loops, the number of investigations will be the same as for the case where all loops are unrolled according to their respective loop bounds. The result, when the graph is traversed, is a time-complexity not higher than for traditional monoprocessor worst-case execution time analysis techniques.

4.3 Noncompositional Analysis

In the presence of timing anomalies, it is no longer possible to do local assumptions about the global worst case execution time. Therefore, for such architectures, every program path has to be analyzed explicitly. This is the case, not only for multiprocessor systems, but for any worst-case execution time framework operating on a noncompositional platform. Also, all steps in Figure 4.1, from the cache and pipeline analyses and forward, must be integrated since it, for noncompositional architectures, is impossible to assume safe initial cache and pipeline states for a basic block, regardless of the allowed pessimism. Since also tradi-

tional WCET analysis operating on noncompositional hardware has to perform a global search through all program paths, the modifications in order to make it aware of the TDMA bus is, in theory, straight-forward. To adapt a traditional noncompositional WCET analysis technique to the class of multiprocessor systems described in Chapter 2, for each considered cache miss, the bus schedule has to be searched in order to find the start and end times of the corresponding bus transfer. This operation is of linear complexity and will therefore not increase the total, already exponential, complexity of the traditional worst-case execution time analysis.

5

Bus Schedule Optimization

Given any TDMA bus schedule, the WCET analysis framework described in Chapter 4 calculates a safe worst-case execution time. This means that the WCET of a task is directly dependent on the bus schedule. This chapter describes how to generate a bus schedule, while satisfying various efficiency requirements. At the end of the chapter, we present experimental results showing the efficiency of our approach.

5.1 WCGD Optimization

Since the bus schedule is directly affecting the worst-case execution time of the tasks, and consequently also the worst-case global delay of the application, it is important that it is chosen carefully. Ideally, when constructing the bus schedule, we would like to allocate a time slot for each individual cache miss on the worst-case control flow path, granting access to the bus immediately when it is requested. There are, however, two significant problems preventing us from doing this. The first one is that several processors can issue a cache miss at the same time instant, creating conflicts on the bus. The second problem is that allocating bus

slots for each individual memory transfer would create a very irregular bus schedule, requiring an unfeasible amount of memory space on the bus controller.

In order to solve the problem of irregular, memory consuming bus schedules, some restrictions on the TDMA round complexity need to be imposed. For instance, an efficient strategy is to allow each processor to own a maximum number of slots per round. Other limitations can be to let each round have the same slot order, or to force the slots in a specific round to have the same size. In this chapter, we assume that every processor can own at most one bus slot per round. The slots in a round can have different sizes, and the order can be set without restrictions. However, it is straight-forward to adapt this algorithm to more (or less) flexible bus schedule design rules. In addition to the main algorithm, we present a simplified algorithm for the special case where all slots in a round must be of the same size.

The problem of handling cache miss conflicts is solved by distributing the bus bandwidth such that the transfer times of cache misses, contributing directly to the worst-case global delay, are minimized. This is done in the inner loop of the overall approach outlined in Figure 3.3. For the optimization process, we start by defining a cost function that estimates the worst-case global delay as a function of the bandwidth distribution. A detailed description will follow in the next section.

5.2 Cost Function

Given a set of active tasks $\tau_i \in \Psi$ (see Figure 3.3), the goal is now to generate a close to optimal bus segment schedule with respect to Ψ. An optimal bus schedule, however, is a bus schedule taking into account the global context, minimizing the global delay of the application. This global delay includes tasks not yet considered and for which no bus schedule has been defined. This requires knowledge about future tasks, not yet analyzed, and, therefore, we must find ways to approximate their influence on the global delay.

In order to estimate the global delay, we first build a schedule S^λ of the tasks not yet analyzed, using a list scheduling technique. When building S^λ we approximate the WCET of each task by its respective worst-case execution time in the naive case, where no conflicts occur on the bus and any task can access the bus at any time. From now on we refer to this conflict-free WCET as NWCET (Naive Worst-Case

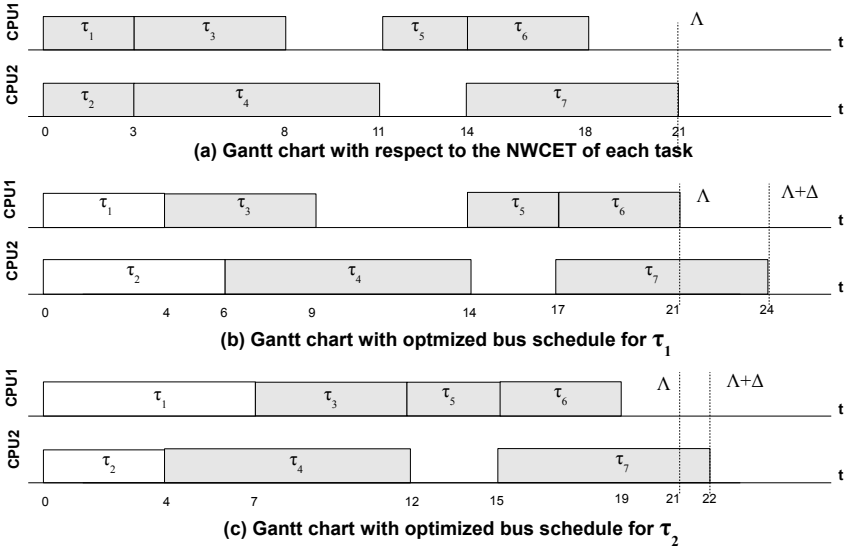


Figure 5.1 *Estimating the global delay*

Execution Time).

When optimizing the bus schedule for the tasks $\tau \in \Psi$, we need an approximation of how the WCET of one task $\tau_i \in \Psi$ affects the global delay. Let D_i be the union of the set of all tasks depending directly on τ_i in the process graph, and the singleton set containing the first task in S^λ that is scheduled on the same processor as τ_i. We now define the tail λ_i of a task τ_i recursively as:

- $\lambda_i = 0$, if $D_i = \emptyset$
- $\lambda_i = \max_{\tau_j \in D_i}(x_j + \lambda_j)$, otherwise.

where $x_j = \text{NWCET}_j$ if τ_j is a computation task. For communication tasks, x_j is an estimation of the communication time, depending on the length of the message. Intuitively, λ_i can be seen as the length of the longest (with respect to the NWCET) chain of tasks that are affected by the execution time of τ_i. Without any loss of generality, in order to simplify the presentation, only computation tasks are considered in the examples of this section. Consider Figure 5.1a, illustrating a Gantt chart of tasks scheduled according to their NWCETs. Direct data dependencies exist between tasks τ_4 & τ_5, τ_5 & τ_6, and τ_5 & τ_7; hence,

for instance, $D_3 = \{\tau_5\}$ and $D_4 = \{\tau_5, \tau_7\}$. The tails of the tasks are: $\lambda_7 = \lambda_6 = 0$ (since $D_7 = D_6 = \emptyset$), $\lambda_5 = 7$, $\lambda_4 = \lambda_3 = 10$, $\lambda_2 = 18$ and $\lambda_1 = 15$.

Since our concern when optimizing the bus schedule for the tasks in Ψ is to minimize the global delay, a cost function taking λ_i into account can be formulated as follows:

$$C_{\Psi,\theta} = \max_{\tau_i \in \Psi}(\theta + \mathrm{WCET}_i^{\theta} + \lambda_i) \tag{5.1}$$

where WCET_i^{θ} is defined as the length of that portion of the worst case execution path of task τ_i which is executed after time θ.

5.3 Optimization Approach

The optimization algorithm is outlined in Figure 5.2.

1. Calculate initial slot sizes.
2. Calculate an initial slot order.
3. Analyze the WCET of each task $\tau \in \Psi$ and evaluate the result according to the cost function.
4. Generate a new slot order candidate and repeat from 3 until all candidates are evaluated.
5. Generate a new slot size candidate and repeat from 2 until the exit condition is met.
6. The best configuration according to the cost function is then used.

Figure 5.2 *The optimization approach*

These steps will now be explained in detail, starting with the inner loop that decides the order of the slots. Given a specific slot size, we search the order of slots that yields the best cost.

5.3.1 Slot Order Selection

At step 2 of the algorithm in Figure 5.2, a default initial order is set. When step 4 is reached for the first time, after calculating a cost for the current slot configuration, the task $\tau_i \in \Psi$ that is maximizing the cost function in Equation 5.1 is identified. We then construct $n - 1$

new bus schedule candidates, n being the number of tasks in the set Ψ, by moving the slot corresponding to this task τ_i, one position at a time, within the TDMA round. The best configuration with respect to the cost function is then selected. Next, we check if any new task τ_j, different from τ_i, now has taken over the role of maximizing the cost function. If so, the procedure is repeated, otherwise it is terminated.

5.3.2 Determination of Initial Slot Sizes

At step 1 of the algorithm in Figure 5.2, the initial slot sizes are dimensioned based on an estimation of how the slot size of an individual task $\tau_i \in \Psi$ affects the global delay.

Consider λ_i, as defined in Section 5.2. Since it is a sum of the NWCETs of the tasks forming the tail of τ_i, it will never exceed the accumulative WCET of the same sequence of tasks. Consequently, if we for all $\tau_i \in \Psi$ define

$$\Lambda = \max_{\tau_i \in \Psi}(\text{NWCET}_i^{\theta} + \lambda_i) \tag{5.2}$$

where NWCET_i^{θ} is the NWCET of task $\tau_i \in \Psi$ counting from time θ, a lower limit of the global delay can be calculated by $\theta + \Lambda$. This is illustrated in Figure 5.1a, for $\theta = 0$. Furthermore, let us define Δ as the amount by which the estimated global delay increases due to the time each task $\tau_i \in \Psi$ has to wait for the bus.

See Figure 5.1b for an example. Contrary to Figure 5.1a, τ_1 and τ_2 are now considered using their real WCETs, calculated according to a particular bus schedule ($\Psi = \{\tau_1, \tau_2\}$). The corresponding expansion Δ is 3 time units. Now, in order to minimize Δ, we want to express a relation between the global delay and the actual bus schedule. For task $\tau_i \in \Psi$, we define m_i as the number of remaining cache misses on the worst case path, counting from time θ. Similarly, also counting from θ, l_i is defined as the sum of each code segment and can thus be seen as the length of the task minus the time it spends using the bus or waiting for it (both m_i and l_i are determined by the WCET analysis). Hence, if we define the constant k as the time it takes to process a cache miss when ignoring bus conflicts, we get

$$\text{NWCET}_i^{\theta} = l_i + m_i k \tag{5.3}$$

As an example, consider Figure 5.3a showing a task execution trace, in the case where no other tasks are competing for the bus. A black box

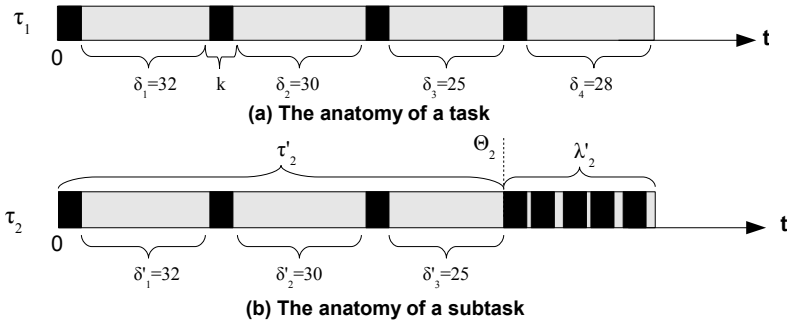


Figure 5.3 *Close-up of two tasks*

represents the idle time, waiting for the transfer, due to a cache miss, to complete. In this example $m_1 = 4$ and $l_1 = \delta_1 + \delta_2 + \delta_3 + \delta_4 = 115$.

Let us now, with respect to the particular bus schedule, denote the average waiting time of task τ_i by d_i. That is, d_i is the average time task τ_i spends waiting, due to other processors owning the bus and the actual time of the transfer itself, every time a cache miss has to be transferred on the bus. Then, analogous to Equation 5.3, the WCET of task τ_i, counting from time θ, can be calculated as

$$\mathrm{WCET}_i^{\theta} = l_i + m_i d_i \tag{5.4}$$

The dependency between a set of average waiting times d_i and a bus schedule can be modeled as follows. Consider the distribution P, defined as the set $p_1, \ldots, p_n$, where $\sum p_i = 1$. The value of p_i represents the fraction of bus bandwidth that, according to a particular bus schedule, belongs to the processor running task $\tau_i \in \Psi$. Given this model, the average waiting times can be rewritten as

$$d_i = \frac{1}{p_i} k \tag{5.5}$$

Putting Equations 5.2, 5.4, and 5.5 together and noting that Λ has been calculated as a maximum over all $\tau_i \in \Psi$, we can formulate the following

system of inequalities:

$$\theta + l_1 + m_1 \frac{1}{p_1} k + \lambda_1 \leq \theta + \Lambda + \Delta$$

$$\vdots$$

$$\theta + l_n + m_n \frac{1}{p_n} k + \lambda_n \leq \theta + \Lambda + \Delta$$

$$p_1 + \cdots + p_n = 1$$

What we want is to find the bus bandwidth distribution P that results in the minimum Δ satisfying the above system. Unfortunately, solving this system is difficult due to its enormous solution space. However, an important observation that simplifies the process can be made, based on the fact that the slot distribution is represented by continuous variables p. Consider a configuration of $p_1, \ldots, p_n, \Delta$ satisfying the above system, and where at least one of the inequalities are not satisfied by equality. We say that the corresponding task τ_i is not on the critical path with respect to the schedule, meaning that its corresponding p_i can be decreased, causing τ_i to expand over time without affecting the global delay. Since the values of p must sum to 1, decreasing p_i, allows for increasing the percentage of the bus given to the tasks τ that are on the critical path. Even though the decrease might be infinitesimal, this makes the critical path shorter, and thus Δ is reduced. Consequently the smallest Δ that satisfies the system of inequalities is achieved when every inequality is satisfied by equality. As an example, consider Figure 5.1b and note that τ_5 is an element in both sets D_3 and D_4 according to the definition in Section 5.2. This means that τ_5 is allowed to start first when both τ_3 and τ_4 have finished executing. Secondly, observe that τ_5 is on the critical path, thus being a direct contributor to the global delay. Therefore, to minimize the global delay, we must make τ_5 start as early as possible. In Figure 5.1b, the start time of τ_5 is defined by the finishing time of τ_4, which also is on the critical path. However, since there is a block of slack space between τ_3 and τ_5, we can reduce the execution time of τ_2 and thus make τ_4 finish earlier, by distributing more bus bandwidth to the corresponding processor. This will make the execution time of τ_1 longer (since it receives less bus bandwidth), but as long as τ_3 ends before τ_4, the global delay will decrease. However, if τ_3 expands beyond the finishing point of τ_4, the former will now be on the critical path instead. Consequently, making task τ_3 and τ_4 end at the same time, by distributing the bus bandwidth such that the sizes of

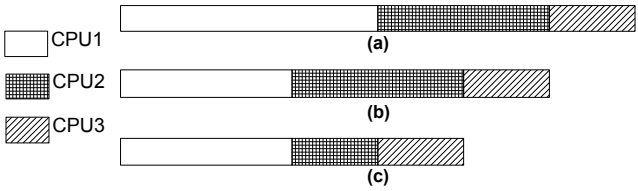


Figure 5.4 *Calculation of new slot sizes*

τ_1 and τ_2 are adjusted properly, will result in the earliest possible start time of τ_5, minimizing Δ. In this case the inequalities corresponding to both τ_1 and τ_2 are satisfied by equality. Such a distribution is illustrated in Figure 5.1c.

The resulting system consists of $n+1$ equations and $n+1$ variables (p_1, ..., p_n and Δ), meaning that it has exactly one solution, and even though it is nonlinear, it is simple to solve. Using the resulting distribution, a corresponding initial TDMA bus schedule is calculated by setting the slot sizes to values proportional to P.

5.3.3 Generation of New Slot Size Candidates

One of the possible problems with the slot sizes defined as in Section 5.3.2 is the following: if one processor gets a very small share of the bus bandwidth, the slot sizes assigned to the other processors can become very large, possibly resulting in long wait times. By reducing the sizes of the larger slots while trying to keep their mutual proportions, this problem can be avoided.

We illustrate the idea with an example. Consider a round consisting of three slots ordered as in Figure 5.4a. The slot sizes have been dimensioned according to a bus distribution $P = \{0.49, 0.33, 0.18\}$, calculated using the method in Section 5.3.2. The smallest slot, belonging to CPU 3, has been set to the minimum slot size k, and the remaining slot sizes are dimensioned proportionally [1] as multiples of k. Consequently, the initial slot sizes become $3k$, $2k$ and k. In order to generate the next set of candidate slot sizes, we define P' as the actual bus distribution of the generated round. Considering the actual slot sizes, the bus distribution becomes $P' = \{0.50, 0.33, 0.17\}$. Since very large slots assigned to a certain processor can introduce long wait times for tasks running

[1] While slot sizes, in theory, do not have to be multiples of the minimum slot size k, in practice this is preferred as it avoids introducing unnecessary slack on the bus.

on other processors, we want to decrease the size of slots, but still keep close to the proportions defined by the bus distribution P. Consider once again Figure 5.4a. Since, $p'_1 - p_1 > p'_2 - p_2 > p'_3 - p_3$, we conclude that slot 1 has the maximum deviation from its supposed value. Hence, as illustrated in Figure 5.4b, the size of slot 1 is decreased one unit. This slot size configuration corresponds to a new actual distribution $P' = \{0.40, 0.40, 0.20\}$. Now $p'_2 - p_2 > p'_3 - p_3 > p'_1 - p_1$, hence the size of slot 2 is decreased one unit and the result is shown in Figure 5.4c. Note that in the next iteration, $p'_3 - p_3 > p'_1 - p_1 > p'_2 - p_2$, but since slot 3 cannot be further decreased, we recalculate both P and P', now excluding this slot. The resulting sets are $P = \{0.60, 0.40\}$ and $P' = \{0.67, 0.33\}$, and hence slot 1 is decreased one unit. From now on, only slot 1 and 2 are considered, and the remaining procedure is carried out in exactly the same way as before. When this procedure is continued as above, all slot sizes will converge towards k which, of course, is not the desired result. Hence, after each iteration, the cost function (Equation 5.1) is evaluated and the process is continued only until no improvement is registered for a specified number π of iterations. The best ever slot sizes (with respect to the cost function) are, finally, selected. Accepting a number of steps without improvement makes it possible to escape certain local minima (in our experiments we use $8 < \pi < 40$, depending on the number of processors).

5.3.4 Density Regions

A problem with the technique presented above is that it assumes that the cache misses are evenly distributed throughout the task. For most tasks, this is not the case in reality. A solution to this problem is to analyze the internal cache miss structure of the actual task and, accordingly, divide the worst case path into disjunct intervals, so called *density regions*. A *density region* is defined as an interval of the path where the distance between consecutive cache misses (δ in Figure 5.3) does not differ more than a specified number. In this context, if we denote by α the average time between two consecutive cache misses (inside a region), the density of a region is defined as $\frac{1}{\alpha+1}$. A region with high density, close to 1, has very frequent cache misses, while the opposite holds for a low-density region.

Consequently, in the beginning of the optimization loop, we identify the next density region for each task $\tau_i \in \Psi$. Now, instead of constructing a bus schedule with respect to each entire task $\tau_i \in \Psi$, only the

interval $[\theta..\Theta_i)$ is considered, with Θ_i representing the end of the density region. We call this interval of the task a subtask since it will be treated as a task of its own. Figure 5.3b shows a task τ_2 with two density regions, the first one corresponding to the subtask τ_2'. The tail of τ_2' is calculated as $\lambda_2' = \lambda_2'' + \lambda_2$, with λ_2'' being defined as the NWCET of τ_2 counting from Θ_2. Furthermore, in this particular example $m_2' = 3$ and $l_2' = \delta_1' + \delta_2' + \delta_3' = 87$.

Consider Figure 3.3 illustrating the overall approach. Analogous to the case where entire tasks are analyzed, when a bus schedule for the current bus segment has been decided, θ' will be set to the finish time of the first subtask. Just as before, the entire procedure is then repeated for $\theta = \theta'$.

However, modifying the bus schedule can cause the worst-case control flow path to change. Therefore, the entire cache miss structure can be transformed during the optimization procedure (lines 4 and 5 in Figure 5.2), resulting in possible changes with respect to both subtask density and size. We solve this problem by using an iterative approach, adapting the bus schedule to possible changes of the subtask structure while making sure that the total cost is decreasing. This procedure will be described in the following paragraphs.

Subtask Evaluation

First, let us in this context define two different cost functions, both based on Equation 5.1. Let $\tau_i^{'\text{end}}$ be the end time of subtask τ_i', and define $\tau^{'\text{end}}$ as:

$$\tau^{'\text{end}} = \min_{\tau_i \in \Psi}(\tau_i^{'\text{end}}) \tag{5.6}$$

Furthermore, let $\text{NWCET}_i^{\tau^{'\text{end}}}$ be the NWCET of the task τ_i, counting from $\tau^{'\text{end}}$ to the end of the task. The *subtask cost* $C'_{\Psi,\theta}$ can now be defined as:

$$C'_{\Psi,\theta} = \max_{\tau_i \in \Psi}(\tau^{'\text{end}} + \text{NWCET}_i^{\tau^{'\text{end}}} + \lambda_i) \tag{5.7}$$

Hence, the subtask cost is a straight-forward adaption of the cost function in Equation 5.1 to the concept of subtasks. Instead of using the worst-case execution time of the entire task, only the part corresponding to the first density region after time θ is considered. The rest of

the task, from the end of the first density region to the end of the entire task, is accounted for in the tail, with respect to its corresponding NWCET.

In order to more accurately approximate how the subtask affects the worst-case global delay, we also introduce its *complementary task cost* $C''_{\Psi,\theta}$ in addition to the subtask cost. Let $\text{WCET}_i^{\tau'^{\text{end}}}$ be the worst-case execution time of task τ_i starting from time τ'^{end}. We here assume that $\text{WCET}_i^{\tau'^{\text{end}}}$ has been calculated with respect to a tailored bus segment, starting after τ'^{end}. The bus schedule representing this bus segment is calculated considering the cache miss structure of the corresponding part of the task, for instance by using the algorithm described in Section 5.3.2 for calculating initial slot sizes. This way we can approximate the transfer delays of the cache misses between τ'^{end} and the end of the task, instead of using the corresponding NWCET (as is done when calculating the subtask cost). The complementary task cost can be defined as:

$$C''_{\Psi,\theta} = \max_{\tau_i \in \Psi}(\tau'^{\text{end}} + \text{WCET}_i^{\tau'^{\text{end}}} + \lambda_i) \tag{5.8}$$

Note that the only difference between this cost function and the previous one in Equation 5.7 is that we now use a calculated WCET for the remaining part of the task, instead of the NWCET. Consequently, the complementary task cost is always greater than or equal to the subtask cost. The problem with using the NWCET, as done when calculating the subtask cost, is that small subtasks tend to be favored. The complimentary cost is more precise, but also more time-consuming to calculate. Therefore the idea is to use it only when necessary.

With the two cost functions defined, we can formulate an algorithm for subtask evaluation, presented in Figure 5.5. In step 2, the tasks $\tau_i \in \Psi$ are analyzed, in their entirety, in order to achieve an initial cache miss structure. This structure is then used to identify the first subtask τ'_i of each task (step 3), and to calculate an initial bus schedule (step 4). In order to evaluate the bus schedule, the complementary cost is evaluated in step 5. In step 8, the bus schedule is modified with respect to the subtasks τ_i. The algorithms, changing the slot sizes and order of the current TDMA round, used for these modifications can be found in Section 5.3.3 and Section 5.3.1. In step 9, the first corresponding subtask τ'_i of each task $\tau_i \in \Psi$ is reidentified with respect to the new cache miss structure, and an updated cost is calculated (by using the

less expensive subtask cost function). If this cost is an improvement of the previous cost[2], we also evaluate the complementary cost $C''_{\Psi,\theta}$. If the new complementary cost is lower than the best cost $C^{\text{best}}_{\text{inner}}$ found so far in the inner loop, we update $C^{\text{best}}_{\text{inner}}$ to this new lowest cost.

We then try to modify the bus schedule further until no more improvements are found (steps 8-12). Consequently, reaching step 13 means two things. Either we have found the best bus schedule, or the worst-case control flow path has changed during the iterations, resulting in a different cache miss structure, not suitable for the generated bus schedule (again, note that the steps 8-12 try to improve the initial sizes calculated, with respect to a specific density, in step 4). If $C^{\text{best}}_{\text{inner}} = C^{\text{best}}_{\text{outer}}$, we did not manage to improve the existing best cost from the last time the inner loop was visited, and the algorithm is halted. If $C^{\text{best}}_{\text{inner}} < C^{\text{best}}_{\text{outer}}$, on the other hand, we identify new subtasks with respect to the improved bus schedule (step 3), and repeat the procedure. Note that this algorithm will always converge since it never accepts solutions that lead to higher costs.

5.4 Simplified Algorithm

For the case where all slots of a round have to be of the same, round-specific size, calculating the distribution P makes little sense. Therefore, we also propose a simpler, but quality-wise equally efficient algorithm, tailor-made for this class of more limited bus schedules. The slot ordering mechanisms are still the same as for the main algorithm, but the procedures for calculating the slot sizes are now vastly simplified. The algorithm is summarized in Figure 5.6.

In step 1, we start by using the smallest possible slot size, since this will minimize the maximum transfer delay. Next, an initial slot order, chosen arbitrarily, is specified in step 2. The slot order candidates are then generated just as in the general algorithm, by changing the position of the slot belonging to the processor on the critical path. After finding the best order for a particular slot size, the latter is modified by, for instance, increasing it k steps. After an appropriate slot size is found, it can also be "'fine tuned"' by increasing or decreasing the size by a very small amount, less than k. Since all processors get the same amount

[2]In the opposite case, for which no improvement of the cost was made, there is no need to calculate $C''_{\Psi,\theta}$ since $C'_{\Psi,\theta} < C''_{\Psi,\theta}$.

1. Set $C_{\text{outer}}^{\text{best}} = \infty$.
2. Calculate initial slot sizes with respect to all tasks $\tau_i \in \Psi$.
3. For each task $\tau_i \in \Psi$, calculate the WCET and identify the corresponding first subtask τ_i'.
4. Calculate the initial slot sizes with respect to the subtasks τ_i'.
5. Calculate the complementary task cost $C_{\Psi,\theta}^{''}$.
6. If $C_{\Psi,\theta}^{''} < C_{\text{outer}}^{\text{best}}$, set $C_{\text{outer}}^{\text{best}} = C_{\Psi,\theta}^{''}$.
7. Set $C_{\text{inner}}^{\text{best}} = C_{\text{outer}}^{\text{best}}$.
8. Modify the bus schedule with respect to the cache miss structure of τ_i'.
9. Once again, for each task $\tau_i \in \Psi$, calculate the WCET and identify the first corresponding subtask τ_i'.
10. Calculate the subtask cost $C_{\Psi,\theta}^{'}$.
11. If $C_{\Psi,\theta}^{'} < C_{\text{inner}}^{\text{best}}$, calculate the complementary task cost $C_{\Psi,\theta}^{''}$ and, if $C_{\Psi,\theta}^{''} < C_{\text{inner}}^{\text{best}}$, set $C_{\text{inner}}^{\text{best}} = C_{\Psi,\theta}^{''}$.
12. Repeat from 8 until no improvements have been made for N iterations.
13. If $C_{\text{inner}}^{\text{best}} < C_{\text{outer}}^{\text{best}}$ then set $C_{\text{outer}}^{\text{best}} = C_{\text{inner}}^{\text{best}}$ and goto 3
14. Use the bus schedule corresponding to $C_{\text{outer}}^{\text{best}}$ for the interval between θ and the end time of the subtask that finished first, and update θ to this end time.

Figure 5.5 *Subtask evaluation algorithm*

of bus bandwidth, the concept of density regions is not useful in this simplified approach.

5.5 Memory Consumption

As stated in Section 2.3, a TDMA bus schedule is composed of segments. Therefore, the amount of memory space needed to store the bus schedule is defined by the number of segments and the complexity restrictions imposed, by the system designer, on the underlying TDMA rounds. In order to calculate an upper bound on the number of segments needed, we make the observation that a new segment is created at every time

1. Initialize the slot sizes to the minimum size k.
2. Calculate an initial slot order.
3. Analyze the WCET of each task $\tau \in \Psi$ and evaluate the result according to the cost function.
4. Generate a new slot order candidate and repeat from 3 until all candidates are evaluated.
5. Increase the slot sizes one step.
6. If no improvements were achieved during a specified number of iterations then exit. Otherwise repeat from 2.
7. The best configuration according to the cost function is then used.

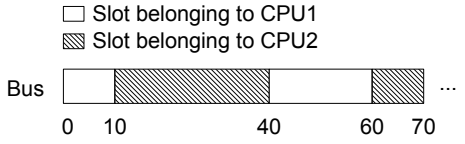
Figure 5.6 *The simplified optimization approach*

t when at least one task starts or finishes. For the case when density regions are not used, these are also the only times when a new segment will be created. Hence, an upper bound on the number of segments is $2|\Pi|$, where Π is the set of all tasks as defined in Chapter 2.

When using density regions, the start and finish of every region can result in a new segment each. Therefore, tasks divided into very many small density regions will result in bus schedules consuming more memory. A straight-forward solution is to limit, according to the available controller memory, the minimum size of a density region. For instance, if the minimum density region size for a task τ_i is $x\%$ of the task length l_i as defined above, the number of generated segments becomes at most $2|\Pi|\frac{100}{x}$.

5.6 Experimental Results

The complete flow illustrated in Figure 3.3 has been implemented and used as a platform for the experiments presented in this section. The bus schedule synthesis was carried out on a general purpose PC with a dual core Pentium 4 processor, running at 2.8 GHz. We have developed a WCET analysis tool, using SymTA/P from Braunschweig University as starting point, based on the considerations in Chapter 4. A system-on-chip design according to Chapter 2, consisting of several ARM7 cores, is assumed for the worst-case execution time analysis. For these examples,



Slot start	Owner
0	CPU1
10	CPU2
40	CPU1
60	CPU2
70	CPU1
...	...

Figure 5.7 *BSA1 bus schedule*

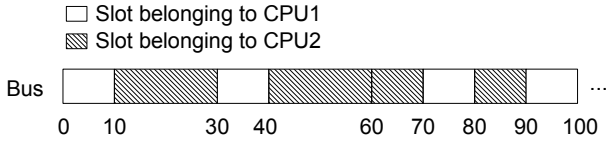
we have assumed that 12 clock cycles are required to handle a memory access due to a cache miss.

5.6.1 Bus Schedule Approaches

To evaluate our algorithms, we defined four bus schedule approaches of varying complexity. The least restrictive approach, BSA1, imposes no restrictions at all and is therefore mostly of interest for comparisons with the other approaches. Since there is no requirement for regularity, a BSA1 schedule is composed by only one segment, consisting of a (very complex) round having the same size as the segment itself. Each processor can own any number of slots of different sizes, and the order of the slots is arbitrary. An example of a BSA1 bus schedule and its table representation can be found in Figure 5.7.

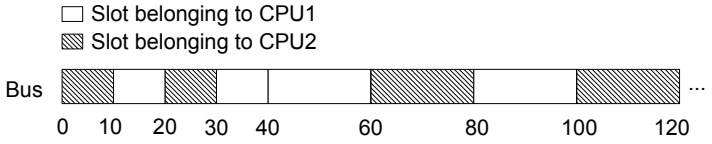
With the more restrictive BSA2, each processor can own at most one bus slot per round. However, the slots in a round can still have different sizes, and the order can be set arbitrarily. Imposing this restriction on the round dramatically decreases the memory needed to store the bus schedule, since the regularity can be used to store it in an efficient fashion. An example of a BSA2 bus schedule is depicted in Figure 5.8. The first segment starts at time unit 0 and ends at time unit 60, immediately followed by the second segment. The main algorithm in this chapter assumes that a BSA2 bus schedule is used.

BSA3 is as BSA2 but with the additional restriction that all slots



Segment start	Owner	Slot size
0	CPU1	10
	CPU2	20
60	CPU2	10
	CPU1	10
...	...	...

Figure 5.8 *BSA2 bus schedule*

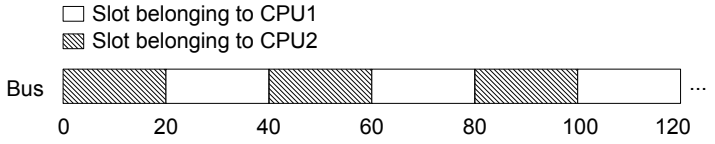


Segment start	Owner	Slot size
0	CPU2	10
	CPU1	
40	CPU1	20
	CPU2	
...	...	...

Figure 5.9 *BSA3 bus schedule*

in a round must be of the same size, regardless of owner. This further decreases the amount of memory required on the bus arbiter, since only one size has to be stored for each round, regardless of the number of slots. The order is, however, still arbitrary, just as for BSA2. An example is illustrated in Figure 5.9. The simplified algorithm explained in this chapter operates on BSA3 bus schedules.

We have also defined a fourth approach, BSA4, which is as BSA3 but with the very strong restriction of allowing only bus schedules constituted by one segment (and thus one round). This requires almost no



Segment start	Owner	Slot size
0	CPU2	20
	CPU1	

Figure 5.10 *BSA4 bus schedule*

memory at all on the bus arbiter. Since this approach is extremely limited, it is interesting mostly for comparisons with the other approaches, just as for BSA1. An example is shown in Figure 5.10

5.6.2 Synthetic Benchmarks

Our first set of experiments was done using benchmarks consisting of randomly generated task graphs with 50 to 200 tasks. The individual tasks were generated according to control flow graphs extracted from various C programs, such as algorithms for sorting, searching, matrix multiplications and DSP processing. We have run experiments for configurations consisting of 2 to 10 processors, and for each configuration, 50 randomly generated task graphs were used and an average WCGD was calculated.

For comparison, this set of experiments was carried out using each of the four bus scheduling approaches defined in Section 5.6.1. In addition, to use as a baseline for evaluating our algorithms, the WCGD was also calculated assuming immediate access to the bus for all processors, resulting in no memory access being delayed. Note that this is an unrealistic assumption, even for a hypothetical optimal bus schedule, resulting in optimistic and unsafe results. Nevertheless, this would be the (wrong!) result if traditional worst-case execution time analysis methods were used.

The result of our experiments is depicted in Figure 5.11. The diagram corresponding to each bus scheduling approach represents how many times larger the respective average WCGD is, in relation to the baseline. As can be seen, not surprisingly, BSA1 produces the shortest

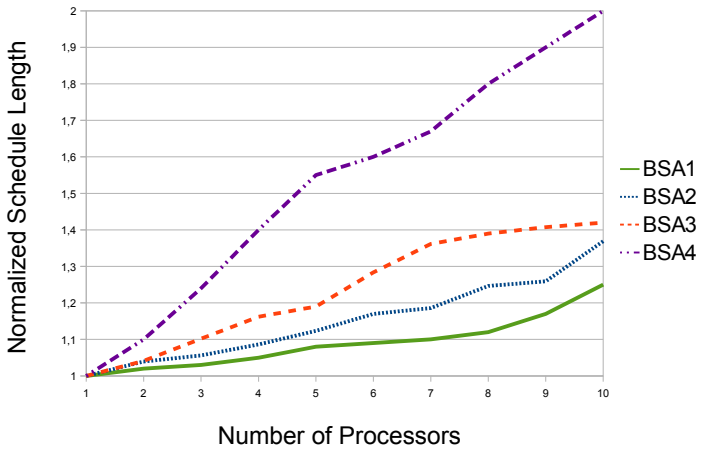


Figure 5.11 *Four bus access policies*

worst-case global delays. This is expected, since the corresponding bus schedules have no restrictions with respect to flexibility. The results produced by BSA2 and BSA3 are, however, not at all far behind. This shows that the price for obtaining regular bus schedules, which can be fitted into memories with a relatively small capacity, is very low. The poor flexibility provided by BSA4, on the other hand, is not enough, and for large bus schedules, the results become inferior.

In a second set of experiments, we have compared the BSA2 and BSA3 bus scheduling approaches, since they are the important alternatives from a practical viewpoint. In particular, we were interested in the efficiency of these policies for applications with different cache miss patterns. A cache miss pattern of a particular task is, in this context, characterized by the standard deviation of the set of time-intervals between all consecutive cache misses. Three classes of applications, each one representing a different level of cache miss irregularity, were created. Every application was composed, according to a randomized task graph, by 20 randomly generated tasks, and each class contained 30 applications. For all tasks, the average distance between consecutive caches misses was 73 clock cycles.

The first class of applications was generated with a uniformly distributed cache miss pattern, corresponding to a standard deviation of

0 clock cycles. The other two classes had a more irregular cache miss structure, corresponding to standard deviations of 50 and 150 clock cycles, respectively. Just as for the previous set of examples, the unsafe traditional case, where no processor ever has to wait for the bus, is used as a baseline. A comparison of the resulting average worst-case global delays is shown in Figure 5.12. It is expected that the two approaches produce the same worst-case global delays for very regular cache miss structures since, most of the time, all processors will demand an equal amount of bus bandwidth. However, as the irregularity of the cache miss structure increases, the ability of BSA2 to distribute the bandwidth more freely becomes more and more of an advantage.

We also carried out a third set of experiments, demonstrating the efficiency of the successive steps of our main bus access optimization algorithm. The same three classes of applications were used as for the previous set, as well as the same baseline. The results are presented in Figure 5.13. The ISS bar represents the average worst-case global delay obtained using the initial slot sizes, calculated as described in Section 5.3.2. The SSA bar corresponds to the average WCGD after slot size adjustments, as described in Section 5.3.3, have been performed as well. Finally, the DS bar shows the result of also applying the concept of density regions, according to Section 5.3.4, in addition to the previous two steps. As expected, density regions are efficient for irregular cache miss patterns, but do not help if the structure is uniformly distributed.

The execution time, for the whole flow, of an example consisting of 100 tasks on 10 processors is 120 minutes for the BSA2 algorithm and 5 minutes for the simplified BSA3 version.

5.6.3 Real-Life Example

In order to validate the real-world applicability of this approach, we have also analyzed a smart phone. It consists of a GSM encoder, GSM decoder and an MP3 decoder, mapped on four ARM7 processors. The GSM encoder and decoder are mapped on one processor each, whereas the MP3 decoder is mapped on two processors. The software applications have been partitioned into 64 tasks, and the size of one such task is between 70 and 1304 lines of C code for the GSM codec, and 200 and 2035 lines for the MP3 decoder. We have assumed a 4-way set associative instruction cache with a size of 4 kilobytes and a direct mapped data cache of the same size. The worst-case global delay was calculated using the four bus scheduling approaches defined in Section

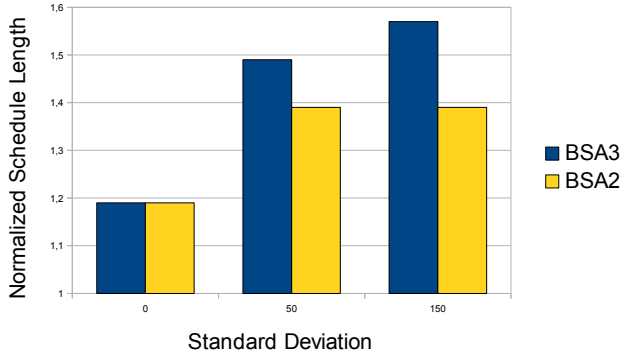


Figure 5.12 *Comparison between BSA2 and BSA3*

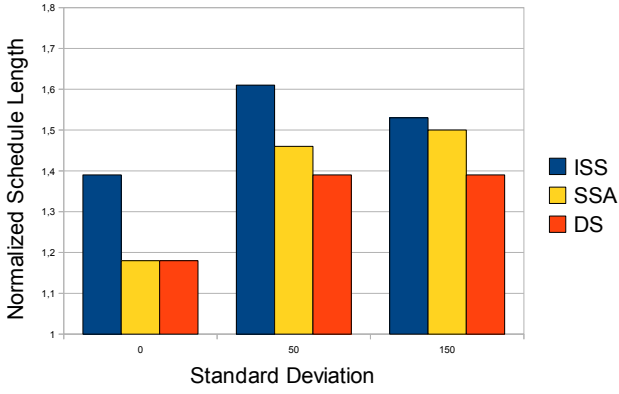


Figure 5.13 *BSA2 optimization steps*

5.6.1. For comparison, we also calculated the WCGD assuming, unrealistically, immediate access to the bus for each processor, as done by traditional worst-case execution time analysis techniques. Table 5.1 shows, for each of the four bus scheduling approaches, how many times larger the obtained safe worst-case global delay is compared to the unrealistic counterpart. As can be seen, the results are coherent with the experiments in Section 5.6.2.

BSA1	BSA2	BSA3	BSA4
1.17	1.31	1.33	1.62

Table 5.1 *Results for the smart phone*

6

Worst/Average-Case Optimization

When optimizing a system platform for a hard real-time application, the goal is, traditionally, to minimize the worst-case global delay. However, the worst-case program path of an application is in most cases taken very seldom, and doing so generally leads to a much longer execution time than what can be expected on average, resulting in a gap between the WCGD and the average-case global delay (ACGD). Therefore, when designing periodic systems, there will be a significant interval of time after the program has finished until the next period starts. During this time interval, the processors are free to be used for anything, as long as they are ready and available at the start of the new period. Thus, instead of just letting them be idle, doing nothing, we can utilize this slack for performing computations not requiring strict predictability, or we can just shut off the processors to save energy. Consequently, it can be of great interest that the average-case global delay is as short as possible, even for hard real-time systems. In this chapter, we present an efficient algorithm for obtaining predictable real-time applications with a low average-case global delay, while still keeping the WCGD close to a minimum.

6.1 Motivation

Consider the hard real-time application in Figure 6.1a, composed of the three tasks τ_1, τ_2 and τ_3 running on two processors. The application is periodically executed with a period equal to the worst-case global delay. After finishing the last computation, the processors are powered off, until the start of the next period, to save energy. Consequently, in the average case, the processors are shut off between the time instants ACGD and WCGD, and since the goal is to reduce the power consumption as much as possible, we would like to maximize this time interval. However, we also want to have a short period and, therefore, the worst-case global delay must remain small. Hence, optimizing for the average case without caring for the worst case is not suitable for this kind of systems. On the other hand, if we optimize for the ACGD while also making sure that the WCGD is kept at a near-minimum, it is possible to benefit from a substantial reduction of power consumption while extending the application period only marginally. This is exactly the case in Figure 6.1b, where it can be seen that the energy consumed for running the application is dramatically reduced, but since the WCGD is increased only by a small amount, the application period can still be kept low.

The interval between the end time of the application and the WCGD can, obviously, also be used for other purposes than switching off the processors. In Figure 6.1c, we have used the remaining time, after the end of the application, for running best effort calculations, represented by the tasks τ_4, τ_5 and τ_6. These tasks can, if needed, be preempted at the end of the application period.

Another example, outside the traditional hard real-time domain, can be found in Figure 6.2. The application, consisting of the three tasks τ_1, τ_2 and τ_3 running on two processors is, at the end of its execution, writing a produced result to a FIFO buffer and is then immediately restarted. An external consumer is periodically reading data from the other end of the buffer. For such systems, a small ACGD allows for high data rates, but in order to guarantee a minimum rate and to help the system designer dimension the buffer, the WCGD must be known and, preferably, be small as well.

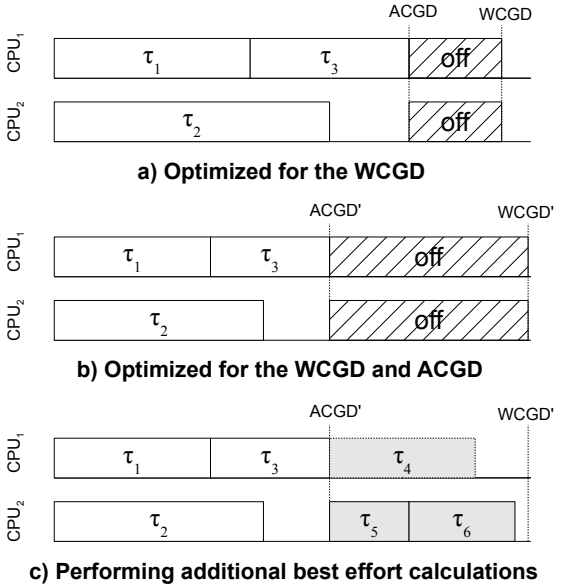


Figure 6.1 *Motivational example for a hard real-time system*

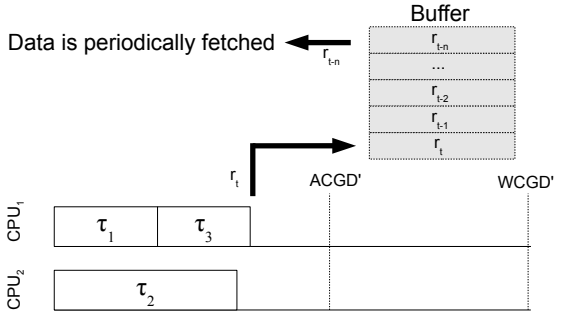


Figure 6.2 *Motivational example for a buffer-based system*

6.2 Average-Case Execution Time Estimation

When calculating the worst-case execution time of a task, one tries to find the worst-case program path with respect to the specified bus schedule. The optimization algorithm then locates where, with respect to the worst-case program path, to allocate bandwidth. This technique is not directly applicable to average-case execution time analysis, since there is generally no particular program path corresponding to the average-case execution time of a task.

To evaluate how good a bus schedule is from the point of view of the average-case execution time, the application has to be executed a large number of times so that the end time of each run can be recorded and used to calculate a mean. This is, naturally, a rather time-consuming process, and therefore, using this method repetitively inside an optimization loop leads to unmanageably long analysis times. Also, in order for the optimization algorithm to know where to allocate bus bandwidth for a certain task, the locations, with respect to time, of the cache misses for an average execution of the task have to be approximated. We solve these two problems by using a histogram-based technique, where task measurement data is used to create task profiles then given as input to the optimization algorithm.

In order to build the memory access histogram, N sets of input data are generated for each task. This data is randomized with respect to a distribution representing typical input patterns for the particular task in question. Every task is then executed, in isolation, N times and, for each execution, a trace file containing the locations of the cache misses is generated. Using this information, we want to find out where, in time, cache misses are most likely to occur so that bus bandwidth can be assigned accordingly. This is done by building, for each task, a histogram over bus accesses in time (not considering the actual time spent waiting for and using the bus), with respect to all N measured executions.

Figure 6.3 shows an example of a histogram based on 1000 executions. The y-value of the histogram denotes how many of the N measured executions of the task accessed the bus during the time interval represented by the corresponding x-value. For instance, in this example, it can be seen that all measured executions access the bus at the very start of the task due to instruction cache misses. During the time regions denoted by t_1, t_2, t_3 and t_4, most measured executions access the bus, and therefore chances are high that so will be the case during

an average execution of the task. Consequently, making sure that the task gets a lot of bandwidth during these time periods is most likely a good idea.

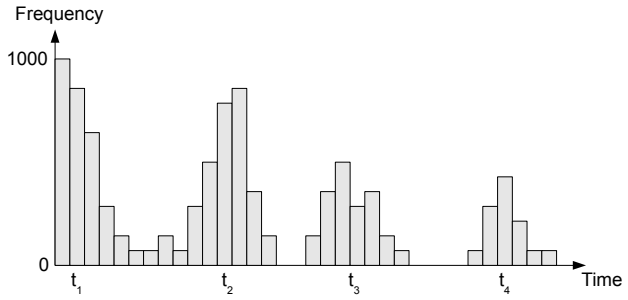


Figure 6.3 *Example histogram for* 1000 *executions*

Given the histogram and a specified bus schedule, we can obtain an estimation of the average-case task execution time with respect to the bus schedule. By using the frequency data on the y-axis, a set of N^H representative hypothetical program paths can be constructed. These hypothetical paths can then be used to approximate the average-case execution time by analyzing their respective global delay and calculating the mean. We will illustrate how this is done by a simple example.

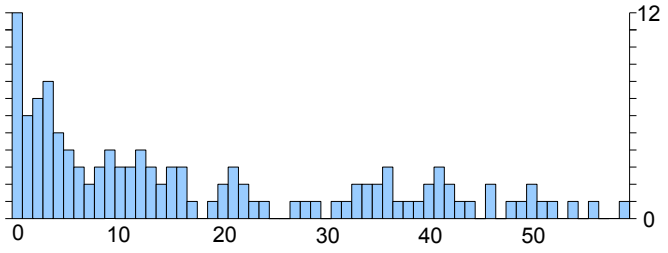


Figure 6.4 *Example histogram for 12 executions*

Consider the histogram in Figure 6.4. In this example, 12 executions have been carried out ($N = 12$), and we want to use this information to estimate the average-case execution time of the measured task with respect to a certain bus schedule. For each execution of the task during the construction of the histogram, the corresponding execution path length, defined as the end-to-end delay excluding bus transfer times, and the number of cache misses are recorded. This information is sorted,

with respect to the execution path length, and presented in Figure 6.5. We can see that the shortest execution path was 8 clock cycles long (excluding transfer times) and had 5 cache misses. The longest path, on the other hand, measured 60 clock cycles and generated 15 cache misses.

Path Length	Number of Misses
8	5
9	6
11	5
12	8
18	9
28	12
32	11
34	12
39	11
48	14
57	16
60	15

Figure 6.5 *Example table for hypothetical path classification*

For this example, we want to create three hypothetical control flow paths ($N^H = 3$). Consequently, we start by dividing the measured execution paths in Figure 6.5 into three equally sized groups, with respect to their execution path lengths. The first group (Group 1) contains the four shortest executions (lengths 8, 9, 11 and 12), whereas the two other groups (Group 2 and Group 3) contain the medium-length executions (lengths 18, 28, 32, 34) and the longest executions (lengths 39, 48, 57, 60), respectively. The idea is to now construct a hypothetical execution path for each of the three groups, capturing their characteristics.

First, we want to compute the representative execution path length and corresponding number of cache misses for each of the three hypothetical paths. This is done by calculating the averages for each group. The hypothetical path corresponding to Group 1 will thus have an execution path length of $(8 + 9 + 11 + 12)/4 = 10$ and be assigned $(5 + 6 + 5 + 8)/4 = 6$ cache misses. The path corresponding to Group 2 will have a length of 28 with 11 cache misses, and for Group 3 the corresponding values will be 51 and 14, respectively. We can now, with respect to the histogram, assign cache misses to the hypothetical paths.

We will show how to do this for Group 1, with the shortest execution path lengths.

Position	Misses	P	r_1	r_2	r_3	Included
0	12	1.00	0.13	-	-	Yes
3	8	0.67	0.71	0.75	0.55	Yes
2	7	0.58	0.53	-	-	Yes
1	6	0.50	0.22	-	-	Yes
4	5	0.42	0.96	0.46	-	No
5	4	0.33	0.31	-	-	Yes
9	4	0.33	0.76	0.34	-	No
6	3	0.25	0.43	0.29	-	No
8	3	0.25	0.60	0.24	-	Yes
7	2	0.17	0.43	0.41	-	No

Figure 6.6 *Example table for cache miss selection*

The idea is to now position the 6 cache misses inside the hypothetical path corresponding to Group 1. To help us do this, we have the histogram in Figure 6.4, telling where it is appropriate to assign cache misses. Since the hypothetical path shall be of length 10, only the data from time 0 to 9 is of interest. This data is sorted and transformed into a table, as shown in the two leftmost columns of Figure 6.6. The third column represents the probability of a cache miss occurring at the actual position. For example, this probability is 1 for position 0, since a cache miss was issued there in all 12 measured executions.

In order to choose if a specific cache miss is to be included, we draw a random number $r \in [0..1] \subseteq \mathbb{R}$ and compare that number with the cache miss probability. Since we only have 6 cache misses to assign, we start with the most probable cache misses, at the top of the table, to make sure that they get the chance to be included. We draw one random number for each cache miss, going from top to bottom. If the random number is lower than or equal to the corresponding probability, the cache miss is included. If the random number instead is greater, the cache miss is excluded – for now. After repeating this procedure for each cache miss in the table, we can start a new iteration and begin from the top again, or we can choose to include the most probable cache misses that were excluded earlier and consider the hypothetical path as

generated.

Since the table is sorted with respect to the cache miss probability, we start at the first row, corresponding to location 0, and move downwards step by step. The column labeled r_1 contains the random numbers drawn during the first iteration of the drawing process. The first generated random number is 0.13, which of course is less than 1 so the hypothetical path will have a cache miss at location 0. The underlining of the random value in Figure 6.6 means that it was lower than the corresponding probability and the cache miss was included. We move to the next row, corresponding to location 3. The new generated random number is 0.71, which is greater than the corresponding probability of 0.67. Hence, we do not put a cache miss at position 3 of the hypothetical path right now.

This procedure goes on until we reach the last row of the table. Then, we can see that four cache misses were included in total, and they will be found on location 0, 1, 2 and 5 in the hypothetical path. However, we need two more cache misses. We can now choose to include the two previously excluded cache misses with the highest probability, which in this case corresponds to locations 3 and 4. Another option is to perform another iteration of drawing to give the less probable cache misses a new chance as well. In Figure 6.6, the random numbers of a second iteration is shown in the column labeled r_2. If the corresponding cache miss was already included, no random number is drawn, leaving the cell empty. When the final cache miss is selected, the algorithm is halted. In this particular example, three iterations were needed in order to find all of the 6 cache misses, resulting in a hypothetical path with cache misses on locations 0, 1, 2, 3, 5 and 8. The maximum number of iterations, denoted as N^R, is specified by the designer, and makes sure that the algorithm will eventually halt. When the number of iterations has reached N^R, the remaining - not yet selected - cache misses with the highest probability are chosen, just as described above.

We can now obtain the end-to-end delay, with respect to a specified bus schedule, corresponding to each execution path by mapping the cache misses to the bus slots. It is here assumed that each execution has the same probability, and therefore all hypothetical paths will have probability $1/N^H$ (since they are constructed by an equal number of executions[1]). The ACGD of the task is then calculated by computing

[1]If not, it is straight-forward to use individual probabilities for the hypothetical paths instead.

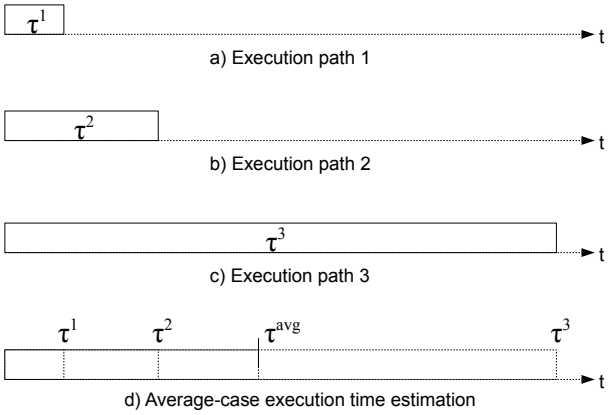


Figure 6.7 *Three hypothetical execution paths and the corresponding average-case execution time estimation*

the mean of all N^H execution path end-to-end delays. An example can be found in Figure 6.7. Figures 6.7a-c depict three hypothetical execution paths – τ^1, τ^2 and τ^3 – of a task τ. Their execution times are 200, 520 and 1870 time units, respectively. The average-case execution time of task τ can be estimated to $(200 + 520 + 1870)/3 = 863$ time units, as illustrated in Figure 6.7d.

Using this estimation approach, we want to design a bus schedule that produces a good ACGD. However, since we also want to keep the WCGD as small as possible, the worst-case program path must be considered during the optimization process. This requires a new technique for optimization, which will be described in the next section.

6.3 Combined Optimization Approach

We assume that the steps in Figure 3.3 have been carried out, and that we have the result in the form of a task schedule, s_0^{worst}, and a bus schedule, B_0, corresponding to the smallest possible WCGD. These are, together with a designer-specified limit on the maximum allowed worst-case global delay and the memory access histogram data for the tasks $\tau_i \in G(\Pi, \Gamma)$, taken as input parameters to our combined optimization approach, as illustrated in Figure 6.8. As output from the algorithm, a bus schedule B_{final}, optimized for both ACGD and WCGD, is returned

```
Algorithm:  Combined ACGD and WCGD optimization
Input:  - Initial bus schedule B_0
        - Initial worst-case schedule s_0^worst
        - Maximum worst-case global delay WCGD_max
        - Memory access histograms of the tasks τ_i ∈ G(Π, Γ)
Output: - The resulting bus schedule B_final
        - The resulting worst-case task schedule s_final^worst
        - The ACGD estimation ACGD_est
01:  Calculate initial average-case schedule s_0^avg
02:  k = 0
03:  repeat
04:    [B_{k+1}, s_{k+1}^worst, s_{k+1}^avg] = improve(B_k, s_k^worst, s_k^avg)
05:    k=k+1
06:  until termination condition
07:  B_final = B_k
08:  s_final^worst = s_k^worst
09:  ACGD_est = length(s_k^avg)
```

Figure 6.8 *Combined optimization approach*

together with the final worst-case task schedule $s_{\text{final}}^{\text{worst}}$ and average-case task schedule length ACGD_{est}.

In the first step of our algorithm in Figure 6.8, the average-case schedule s_0^{avg} is calculated with respect to B_0. Then an iterative function, denoted as `improve` on line 4 in Figure 6.8, tries to improve the bus schedule with respect to both the average and the worst case. The termination condition is reached when no more improvements can be found, and the algorithm then exits and returns the best bus schedule B_{final} and corresponding worst-case task schedule $s_{\text{final}}^{\text{worst}}$. The application is finally simulated a large number of times in order to measure the real ACGD.

6.4 Bus Access Optimization for ACGD and WCGD

The `improve` function in Figure 6.8 takes as input parameters a bus schedule B_k, the current worst-case task schedule s_k^{worst}, the current average-case task schedule s_k^{avg} and WCGD_{max}. As output, we get the improved bus schedule B_{k+1} together with the corresponding worst-case task schedule s_{k+1}^{worst} and average-case task schedule s_{k+1}^{avg}. A description of the computations performed in the `improve` function will now follow.

The goal is to modify the bus schedule so that the average-case global delay of the application is reduced, while the WCGD is increased as little as possible. To do so, the effects on both the ACGD and WCGD have to be considered for each possible modification. However, performing average-case and worst-case execution time analysis with respect to several bus schedule candidates is a time-consuming process. Therefore, it is desirable to identify the most interesting parts of the bus schedule, where a modification is likely to result in positive effects for the global delay, and then perform execution time analysis with respect to modifications of these parts only. Consequently, we start the `improve` function by investigating which parts of the bus schedule to modify for a decreased ACGD, without initially considering the effects on the WCGD. Only the most interesting parts are then investigated with respect to both the ACGD and WCGD.

6.4.1 Task and Bus Segments

The first step of the `improve` function is to generate the average-case task schedule s_k^{avg} by performing ACET analysis (Section 6.2), for each task, with respect to the bus schedule B_k. From the execution time analysis, we can extract interesting properties, such as bus transfer times and the number of memory accesses, of certain time intervals, and these properties are then used to determine how much the corresponding parts of the bus schedule can be improved with respect to the ACGD (and later also how to modify the bus schedule).

In order to find suitable time intervals, we first divide both the bus schedule B_k and the average-case task schedule s_k^{avg} into segments. For this, we distinguish between two different kind of segments: *task segments* and *bus segments*. Bus segments are defined just as in Section 2.3, i.e. as intervals of the bus schedule where the same TDMA round is repeated. Consequently, the bus schedule can be regarded as a disjunctive set B of bus segments.

A task segment can be of arbitrary size and represents an interval of time, with respect to the task schedule. Every task schedule can be seen as a disjunctive set of task segments, Ξ. The division of the task schedule into task segments is done with respect to the cache miss structure of the application, with the goal of obtaining a uniform distribution of cache misses in each segment. We will now show how this is done with a small example.

Consider the task graph in Figure 6.9. We have five tasks – τ_1, τ_2, τ_3,

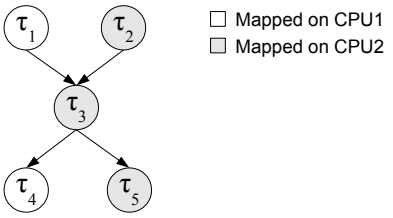


Figure 6.9 *Example task graph*

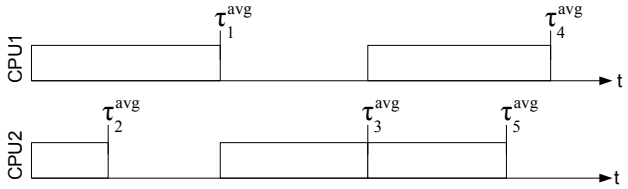


Figure 6.10 *Average-case chart*

τ_4 and τ_5 – mapped on two processors. A Gantt chart of the application, in the case where the estimated average-case execution time is used for scheduling the tasks, is illustrated in Figure 6.10.

In this example, N^H is set to $N^H = 2$, and hence each task has two hypothetical execution paths. These are depicted in Figure 6.11. In order to handle the effect of interprocessor data dependencies, we divide the task schedule into a disjunctive set of *dependency intervals*. A new dependency interval begins whenever a task with an interprocessor dependency is released, or when the longest hypothetical execution path of a task also executing in a previous dependency interval ends. In this example, we get six dependency intervals, denoted by Di_1, Di_2, Di_3, Di_4, Di_5 and Di_6. A task segment spanning over multiple dependency intervals can lead to bandwidth assignment conflicts within the segment, and therefore we do not allow such a division. Consequently, a task segment must belong to exactly one dependency interval.

The task segments are now created by calculating the average cache miss density for all overlapping execution paths, and then dividing the dependency intervals with respect to these average densities so that every subinterval gets a close to uniform distribution on each active processor. These subintervals are the actual task segments. The result, in our particular example, can be seen in Figure 6.12. In this simple case, we distinguish between two different levels of average cache miss

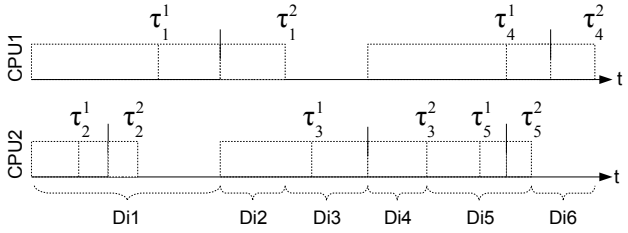


Figure 6.11 *Average-case chart with corresponding execution paths*

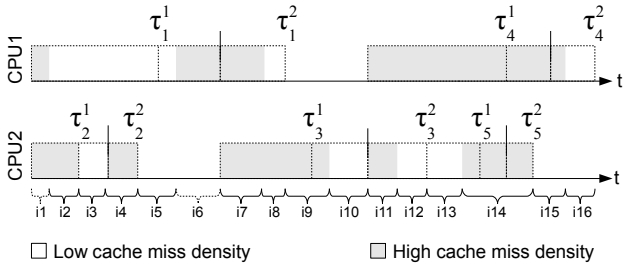


Figure 6.12 *Average-case chart with density regions*

density. Areas with white color represent regions with low cache miss density, whereas the gray intervals represent high density regions. In this case we get 16 task segments, i_1-i_{16}. Dependency interval Di_1, for instance, is divided into task segments i_1-i_6.

What we, initially, would like to do is to identify the areas of the bus schedule, represented by task segments and bus segments, for which modifications can result in the most beneficial change of average-case global delay. In order to perform this identification, we start with an investigation of how the bus bandwidth is distributed to the task segments and bus segments, in the average case.

6.4.2 Bus Bandwidth Distribution Analysis

Based on the information from the average-case execution time analysis, for each task segment $\Omega_i \in \Xi$ and bus segment $\omega_j \in B$, we want to determine the following:

1. The desired bus bandwidth, that, when given to this particular interval of the bus schedule, minimizes the global delay in the aver-

age case. For a specific task segment $\Omega_i \in \Xi$, this bandwidth is denoted $\overline{P}_{\Omega_i}$ and is a vector of n elements, $\overline{P}_{\Omega_i} = p_{\Omega_i}(1), \ldots, p_{\Omega_i}(n)$, where n is the number of processors in the system and each element represents the desired bandwidth for the corresponding processor. Similarly, $\overline{P}_{\omega_j} = p_{\omega_j}(1), \ldots, p_{\omega_j}(n)$ is the corresponding vector for a bus segment $\omega_j \in B$. The bandwidth is represented as the fraction of the total bus bandwidth, thus satisfying:

$$\sum_k p_{\Omega_i}(k) = 1, \qquad \sum_k p_{\omega_j}(k) = 1$$

Detailed descriptions for how to perform these calculations can be found in Appendix A.1.1.

2. The bandwidth currently given to each processor during the specific interval $\Omega_i \in \Xi$ and $\omega_j \in B$, represented by the vectors $\overline{P}^{bus}_{\Omega_i} = p^{bus}_{\Omega_i}(1), \ldots, p^{bus}_{\Omega_i}(n)$ and $\overline{P}^{bus}_{\omega_j} = p^{bus}_{\omega_j}(1), \ldots, p^{bus}_{\omega_j}(n)$ respectively. Just as for the desired bandwidth, the elements are representing fractions of the total bandwidth, therefore summing up to one. Appendix A.1.2 describes how these two vectors are calculated.

For a specific task segment $\Omega_k \in \Xi$, let $\alpha_{\Omega_k} \in [1..n] \subseteq \mathbb{Z}$, where n is the number of processors, denote the processor on which the task on the critical path, with respect to the average-case task schedule, is executed during that particular time interval. Let us define the scalar $p^{\Delta}_{\Omega_k}$, for a task segment $\Omega_k \in \Xi$, as the difference between the desired task bandwidth and the provided bus bandwidth for processor α_{Ω_k}. That is, $p^{\Delta}_{\Omega_k} = p_{\Omega_k}(\alpha_{\Omega_k}) - p^{bus}_{\Omega_k}(\alpha_{\Omega_k})$.

Since many task segments can overlap a bus segment, several different processors can execute tasks that are on the (same) critical path during the particular time interval represented by the bus segment. Hence, let A_{ω_k} be the set of processors α_{ω_k} that are running tasks on the critical path during the interval represented by bus segment $\omega_k \in B$. We then define $p^{\Delta}_{\omega_k}$ as:

$$p^{\Delta}_{\omega_k} = max\Big(\bigcup_{\alpha_i \in A_{\omega_k}} \big(p_{\omega_k}(\alpha_i) - p^{bus}_{\omega_k}(\alpha_i)\big)\Big) \tag{6.1}$$

A high $p^{\Delta}_{\Omega_i}$ for a task segment $\Omega_i \in \Xi$ or $p^{\Delta}_{\omega_k}$ for a bus segment $\omega_j \in B$ means that the corresponding interval of the bus schedule has

room for improvement with respect to the average-case global delay. Therefore, time intervals with a high corresponding $p_x^{\Delta}, x \in (\Xi \cup B)$ are interesting from an optimization point of view, whereas intervals with a low $p_x^{\Delta}, x \in (\Xi \cup B)$ do not need further investigation. We can now limit the search space by just looking at parts of the bus schedule with a corresponding $p_x^{\Delta}, x \in (\Xi \cup B)$ exceeding a specified threshold.

If we wanted to just optimize for the average case, we would start by modifying the region represented by the segment (task or bus) with the highest $p_x^{\Delta}, x \in (\Xi \cup B)$. However, a large decrease in average-case global delay is not necessarily good, if that makes the WCGD increase too big. When deciding which region of the bus schedule to improve, one must also take into account the effect on the worst-case global delay. In fact, what we want to improve is the ratio between average-case improvement and worst-case extension, with respect to the global delay. In order for our optimization algorithm to decide how good a bus schedule is for both the ACGD and WCGD, a cost function is specified in the next section.

6.4.3 Cost Function

We denote with length(s) the length of schedule s. Let $s_{\text{old}}^{\text{worst}}$ and $s_{\text{old}}^{\text{avg}}$ be worst-case and average-case schedules, for the same application, generated with respect to a bus schedule B_{old}. After creating a new bus schedule B_{new}, by modifying a suitable interval of B_{old}, we obtain updated task schedules $s_{\text{new}}^{\text{worst}}$ and $s_{\text{new}}^{\text{avg}}$. If an improvement was made, the task schedules will satisfy $\text{length}(s_{\text{new}}^{\text{avg}}) < \text{length}(s_{\text{old}}^{\text{avg}})$ and $\text{length}(s_{\text{new}}^{\text{worst}}) > \text{length}(s_{\text{old}}^{\text{worst}})$[2] (since the WCGD is expected to grow). If the bus schedule B_{new} does not lead to an improvement with respect to the ACGD, it is discarded and not considered further by the optimization algorithm. Provided that the new bus schedule B_{new} actually results in an improvement, a good measure of how good the bus modification is can be given by the ratio:

$$q_{s_{\text{new}}^{\text{worst}}, s_{\text{new}}^{\text{avg}}, s_{\text{old}}^{\text{worst}}, s_{\text{old}}^{\text{avg}}} = \frac{\text{length}(s_{\text{old}}^{\text{avg}}) - \text{length}(s_{\text{new}}^{\text{avg}})}{\text{length}(s_{\text{new}}^{\text{worst}}) - \text{length}(s_{\text{old}}^{\text{worst}})} \tag{6.2}$$

[2]For the special case when $\text{length}(s_{\text{new}}^{\text{worst}}) <= \text{length}(s_{\text{old}}^{\text{worst}})$, the ratio in Equation 6.2 is set to ∞

Consequently, a suitable cost function for our optimization algorithm can be expressed as:

$$C(s_{\text{new}}^{\text{worst}}, s_{\text{new}}^{\text{avg}}, s_{\text{old}}^{\text{worst}}, s_{\text{old}}^{\text{avg}}) = -q_{s_{\text{new}}^{\text{worst}}, s_{\text{new}}^{\text{avg}}, s_{\text{old}}^{\text{worst}}, s_{\text{old}}^{\text{avg}}} \tag{6.3}$$

With respect to this cost function, we can now evaluate a set of bus schedule candidates and choose the best one.

6.4.4 Bus Schedule Optimization

We create a new bus schedule candidate $B_{k'}$ from bus schedule B_k (taken as input parameter on line 4 in Figure 6.8) by modifying the part of B_k corresponding to the time interval represented by a task segment $\Omega_i \in \Xi$ or bus segment $\omega_j \in B$. To calculate the cost of $B_{k'}$, we must compute the schedules $s_{k'}^{worst}$ and $s_{k'}^{avg}$. This is done by invoking the execution time analysis framework twice, for the entire application, and that is relatively costly from a computation-time perspective. Therefore, as stated previously, the solution is to limit the search space and thus only generate candidates that are likely to perform good. It can be assumed that improving areas corresponding to segments with a low $p_{\Omega_i}^{\Delta}$ or $p_{\omega_j}^{\Delta}$, for $\Omega_i \in \Xi$ and $\omega_j \in B$ respectively, will not lead to the best results, since the room for average-case global delay improvements is small. Therefore, we define Ξ' as the set of the t task segments $\Omega_i \in \Xi$ which have the greatest corresponding $p_{\Omega_i}^{\Delta}$ values. Similarly, B' is defined as the set of b bus segments $\omega_j \in B$ with the greatest corresponding $p_{\omega_i}^{\Delta}$. The $t+b$ segments in $\Xi' \cup B'$ are selected for further investigation. High t and b values, set by the designer, allow the algorithm to evaluate more bus schedule candidates, but at the expense of computation time.

For each segment in $\Xi' \cup B'$, we want to generate several bus schedule candidates and evaluate them with respect to the cost function defined in Equation 6.3. When no more bus schedule candidates are left to evaluate for any segment in $\Xi' \cup B'$, the candidate associated with the lowest cost is kept and returned as bus schedule B_{k+1} (line 4 in Figure 6.8).

The first bus schedule candidate $B_{k'_0}$ for a specific segment in $\Xi' \cup B'$ is generated by inserting a new bus segment into the previously generated bus schedule B_k. This new bus segment is constituted by a TDMA round r, generated so that the bus bandwidth during the corresponding interval is assigned according to the desired bus bandwidth $\overline{P}_{\Omega_i}$ or $\overline{P}_{\omega_j}$, depending on if the segment in $\Xi' \cup B'$ being investigated is a task

segment $\Omega_i \in \Xi'$ or a bus segment $\omega_j \in B'$. With respect to the bus schedule candidate $B_{k'_0}$, the schedules $s^{\text{worst}}_{k'_0}$ and $s^{\text{avg}}_{k'_0}$ are generated and $B_{k'_0}$ is then evaluated according to the cost function in Equation 6.3. To create the next bus schedule candidate $B_{k'_i}$ (where now, in this case, $i = 1$) for the same segment, round r is modified according to the outcome of the execution time analysis, using the hill climbing optimization technique to adjust the bandwidth distribution. Other modifications, such as slot order permutations, can also be carried out depending on the restrictions imposed on TDMA complexity. The procedure of improving round r – each improvement resulting in a new bus schedule candidate – is repeated a specified number of times or until no further improvements are found, and then the next segment in $\Xi' \cup B'$ is investigated. The best bus schedule candidate $B_{k'}$ is then chosen as the new bus schedule B_{k+1} for the application, and the function returns. The `improve` function is summarized in Figure 6.13.

Note that adding new bus segments will increase the complexity of the bus schedule. Since the memory on the bus arbiter is limited, there might be a limit for how many bus segments we can allow. Once this maximum number of bus segments is reached, we cannot increase the number of segments of the bus schedule without first deleting at least one, already existing, bus segment. Therefore, immediately after inserting the new bus segment, resulting in the bus schedule candidate $B_{k'_0}$, and before making any improvements to the corresponding round r constituting it, we evaluate the effect of merging every pair of consecutive bus segments in the bus schedule using the ACGD and WCGD analyses and computing the resulting cost. The best merge is then kept, and we continue by generating more bus schedule candidates $B_{k'_i}$ (by trying to improve r, as usual). Note that this is only a problem when improving the bus schedule with respect to task segments $\Omega_i \in \Xi'$, since improving with respect to bus segments $\omega_j \in B'$ does not increase the bus schedule complexity.

6.5 Experimental Results

We have evaluated our framework using an extensive set of generated C programs. The programs were constructed with respect to randomized task graphs consisting of between 20 and 200 tasks, mapped on 2 to 8 processors. The individual tasks were generated according to control flow graphs corresponding to programs for commonly used computa-

```
01: Perform an average-case execution time analysis.
02: Divide the resulting task schedule into a set of
    task segments Ξ.
03: Calculate current and desired bus bandwidth,
    P̄_{Ω_i} and P̄_{ω_j}, with respect to the ACGD only, for
    each task segment and bus segment.
04: Calculate Ξ′ and B′.
05: For each element in Ξ′ ∪ B′, generate a set of
    bus schedule candidates and evaluate them
    according to the cost function in Equation 6.3.
06: Return the candidate that generates the lowest cost,
    while keeping the WCGD below WCGD_max.
```

Figure 6.13 *The* `improve` *function*

tions such as sorting, searching, matrix multiplications and DSP processing. In total, 8000 applications were generated and evaluated. To calculate the memory access histograms, as described in Section 6.2, 1000 executions were carried out for each task.

As hardware platform, we have used the MPARM multiprocessor cycle-accurate simulator from Bologna University [15], configured according to the model in Section 2.1, using 8 ARM7 cores running at 200 MHz. An AMBA AHB-compliant bus arbiter, enforcing the bus model in Section 2.3, was implemented and incorporated into the simulation framework. The bus speed was set to 100 MHz, resulting in a memory access taking 13 CPU clock cycles to serve. In order to restrict the amount of memory on the controller, we imposed the following restrictions on TDMA round complexity:

1. A processor can own at most one slot in a TDMA round.

2. The slot order is fixed, and cannot be changed during the optimization procedure.

The values of t and b, described in Section 6.4.4, were set to 100 and 50, respectively. We also limited the total number of bus segments allowed in the bus schedule to 1000.

Using the approach described in Chapters 3 and 5, for each of the applications, we started by generating a bus schedule minimizing the worst-case global delay, completely ignoring the average case. Let us denote this initial, minimized, worst-case global delay by WCGD_0, and let ACGD_0 be the ACGD calculated with respect to the same bus schedule.

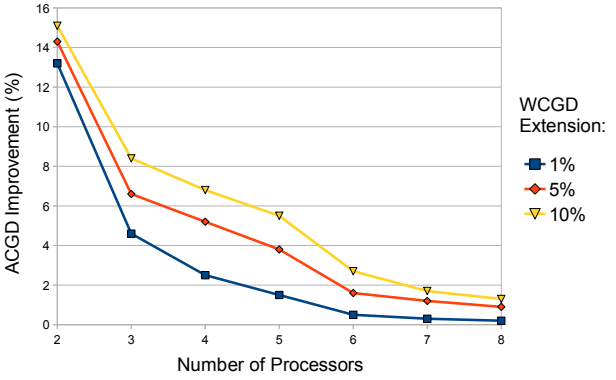


Figure 6.14 *Relative ACGD improvement*

The bus schedule, optimized for the worst case, and the corresponding worst-case task schedule were then sent as input parameters to the algorithm outlined in Figure 6.8, together with the generated memory access histograms for each task in the application task graph. A maximum allowed WCGD was also supplied.

We now investigated how much the ACGD can be decreased, given a maximum allowed increment (with respect to $WCGD_0$) of the resulting WCGD. For all applications, we performed the optimization procedure three times, allowing WCGD increments of 1%, 5% and 10% respectively. For each of these allowed increments, a corresponding average ACGD improvement was calculated. The result is presented in Figure 6.14. For instance, for two processors, accepting an 1% extension with respect to $WCGD_0$ leads to an average ACGD improvement of 13.2%. Accepting a greater WCGD increment naturally results in a more substantial ACGD reduction. It can be observed that using a lower number of processors allows for a higher ACGD decrement, with respect to $ACGD_0$. This is explained by the fact that fewer competing processors leave more room for tailoring the bus schedule for a specific processor, allowing for a more flexible design.

In a second experiment, we investigated how optimizing for the ACGD, without considering the WCGD at all, affects the latter. The idea is to show that optimizing only for the ACGD leads to unreasonably high worst-case global delays, compared to when optimizing for both. For this second experiment, we used the very same generated test examples as in the first experiment, allowing for direct comparisons with

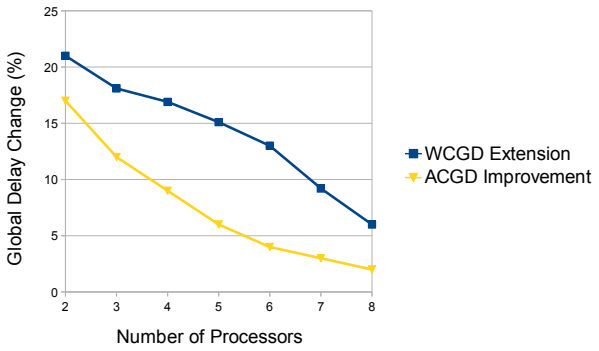


Figure 6.15 *Relative ACGD improvement and WCGD extension*

the already calculated WCET_0 and ACET_0. Initially, an algorithm for optimizing the bus schedule, taking into account only the ACGD, was applied to the test applications, and then the WCGD was calculated with respect to that bus schedule. Let us denote the resulting ACGD and WCGD by ACGD'_0 and WCGD'_0 respectively. In Figure 6.15, we have plotted the relative average extension of WCGD'_0 compared to WCGD_0, and the average reduction of ACGD'_0 compared to ACGD_0. As can be seen, not taking the WCGD into consideration when optimizing the bus schedule leads to higher worst-case global delays, whereas the corresponding ACGD improvement is only slightly better than when also optimizing for the WCGD and accepting a WCGD increase of 10%. For instance, for a 5 processor application, the WCGD extension compared to the optimal case (WCGD_0) is 15% whereas the improvement of the ACGD relative ACGD_0 is 6.0%. By looking in Figure 6.14, we can see that when optimizing for both ACGD and WCGD simultaneously, for 5 processors we can obtain a 5.5% (instead of 6%) improvement with only a 10% (compared to 15%) degradation of the WCGD.

All experiments were executed on a dual core Pentium 4 processor running at 2.8 GHz. The time to process one application ranged from 10 minutes to 4 hours, depending on the application complexity.

7

Conclusions

In this thesis, we have presented an approach to achieve time-predictability for real-time applications running on modern multiprocessor systems-on-chip, taking into consideration potential memory access conflicts between concurrent tasks. In particular, we have focused on the issue of bus access optimization. Efficient algorithms for minimizing the worst-case global delay have been proposed, making sure that the price to pay for predictability, in terms of time, becomes as small as possible.

We have also presented an entirely new, within the domain of real-time systems, concept of optimizing for the average-case global delay while keeping the WCGD as small as possible. The proposed algorithms show that this dual approach is more efficient than optimizing only for either the worst-case or the average-case.

All algorithms were validated by an extensive sets of experiments.

Bibliography

[1] A. Andrei, P. Eles, Z. Peng, and J. Rosén, "Predictable Implementation of Real-Time Applications on Multiprocessor Systems-on-Chip," in *VLSI Conference*, 2008.

[2] D. Bertozzi, A. Guerri, M. Milano, F. Poletti, and M. Ruggiero, "Communication-Aware Allocation and Scheduling Framework for Stream-Oriented Multi-Processor Systems-on-Chip," in *DATE*, 2006, pp. 3–8.

[3] C.-F. Neikter, "Cache Prediction and Execution Time Analysis on Real-Time MPSoC," 2008, Master Thesis, LIU-IDA/LITH-EX-A–08/046–SE.

[4] J. M. Calandrino, J. H. Anderson, and D. P. Baumberger, "A Hybrid Real-Time Scheduling Approach for Large-Scale Multicore Platforms," in *ECRTS*, 2007.

[5] S. A. Edwards and E. A. Lee, "The Case for the Precision Timed (PRET) Machine," *EECS Department Technical Report No. UCB/EECS-2006-149, University of California, Berkeley*, 2006.

[6] H. Falk, "WCET-aware Register Allocation based on Graph Coloring," in *DAC*, 2009.

[7] H. Falk and J. C. Kleinsorge, "Optimal Static WCET-aware Scratchpad Allocation of Program Code," in *DAC*, 2009.

[8] K. Goossens, J. Dielissen, and A. Radulescu, "AEthereal Network on Chip: Concepts, Architectures, and Implementations," *IEEE Design & Test of Computers*, vol. 2/3, pp. 115–127, 2005.

[9] M. Joseph and P. Pandya, "Finding Response Times in a Real-Time System," *The Computer Journal*, vol. 29, 1986.

[10] E. C. Jr and R. Graham, "Optimal Scheduling for Two-Processor Systems," *Acta Inform.*, vol. 1, pp. 200–213, 1972.

[11] I. A. Khatib, D. Bertozzi, F. Poletti, L. Benini, and et al., "A Multiprocessor Systems-on-Chip for Real-Time Biomedical Monitoring and Analysis: Architectural Design Space Exploration," in *DAC*, 2006, pp. 125–131.

[12] B. Lickly, I. Liu, S. Kim, H. D. Patel, S. A. Edwards, and E. A. Lee, "Predictable Programming on a Precision Timed Architecture," *EECS Department Technical Report No. UCB/EECS-2008-40, University of California, Berkeley*, 2008.

[13] T. Lundqvist and P. Stenström, "Timing Anomalies in Dynamically Scheduled Microprocessors," in *The 20th IEEE Real-Time Systems Symposium (RTSS)*, 1999, pp. 12–21.

[14] M. Lv, N. Guan, W. Yi, and G. Yu, "Combining Abstract Interpretation with Model Checking for Timing Analysis of Multicore Software," in *The 31st IEEE Real-Time Systems Symposium (RTSS)*, 2010.

[15] MPARM homepage,
"http://www-micrel.deis.unibo.it/sitonew/research/mparm.html".

[16] F. Nemati, M. Behnam, and T. Nolte, "Independently-Developed Real-Time Systems on Multi-Cores with Shared Resources," in *23rd EUROMICRO Conference on Real-Time Systems (ECRTS'11)*, July 2011.

[17] M. Paolieri, E. Quiñones, F. J. Cazorla, G. Bernat, and M. Valero, "Hardware Support for WCET Analysis of Hard Real-Time Multicore Systems," in *ISCA*, 2009, pp. 57–68.

[18] S. Pasricha, N. Dutt, and M. Ben-Romdhane, "Fast Exploration of Bus-based On-chip Communication Architectures," in *CODES+ISSS*, 2004, pp. 242–247.

[19] R. Pellizzoni and M. Caccamo, "Impact of Peripheral-Processor Interference on WCET Analysis of Real-Time Embedded Systems," *IEEE Transactions on Computers*, vol. 59, no. 3, pp. 400–415, 2010.

[20] P. Pop, P. Eles, Z. Peng, and T. Pop, "Analysis and Optimization of Distributed Real-Time Embedded Systems," *ACM Transactions on Design Automation of Electronic Systems*, vol. Vol. 11, pp. 593–625, 2006.

[21] P. Puschner and A. Burns, "A Review of Worst-Case Execution-Time Analysis," *Real-Time Systems*, vol. 2/3, pp. 115–127, 2000.

[22] J. Reineke, B.Wachter, S. Thesing, R.Wilhelm, I. Polian, J. Eisinger, and B. Becker, "A Definition and Classification of Timing Anomalies," in *International Workshop on WCET Analysis*, 2006, pp. 23–28.

[23] J. Rosén, A. Andrei, P. Eles, and Z. Peng, "Bus Access Optimization for Predictable Implementation of Real-Time Applications on Multiprocessor Systems-on-Chip," in *The 28th IEEE Real-Time Systems Symposium (RTSS)*, 2007, pp. 49–60.

[24] J. Rosén, P. Eles, Z. Peng, and A. Andrei, "Predictable Worst-Case Execution Time Analysis for Multiprocessor Systems-on-Chip," in *The 6th International Symposium on Electronic Design, Test and Applications (DELTA)*, 2011.

[25] J. Rosén, C.-F. Neikter, P. Eles, Z. Peng, P. Burgio, and L. Benini, "Bus Access Design for Combined Worst and Average Case Execution Time Optimization of Predictable Real-Time Applications on Multiprocessor Systems-on-Chip," in *The 17th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2011.

[26] E. Salminen, V. Lahtinen, K. Kuusilinna, and T. Hamalainen, "Overview of Bus-Based System-on-Chip Interconnections," in *ISCAS*, 2002, pp. 372–375.

[27] S. Schliecker, M. Ivers, and R. Ernst, "Integrated Analysis of Communicating Tasks in MPSoCs," in *The International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2006, pp. 288–293.

[28] S. Schliecker, M. Negrean, and R. Ernst, "Bounding the Shared Resource Load for the Performance Analysis of Multiprocessor Systems," in *Proc. of Design, Automation, and Test in Europe (DATE)*, Dresden, Germany, 2010.

[29] S. Schliecker, M. Negrean, G. Nicolescu, P. Paulin, and R. Ernst, "Reliable Performance Analysis of a Multicore Multithreaded System-On-Chip," in *The International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2008.

[30] M. Schoeberl and P. Puschner, "Is Chip-Multiprocessing the End of Real-Time Scheduling?" in *9th International Workshop on Worst-Case Execution Time (WCET) Analysis*, 2009.

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

[32] L. Thiele and R. Wilhelm, "Design for Timing Predictability," *Real-Time Systems*, vol. 28, no. 2/3, pp. 157–177, 2004.

[33] 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 Transactions on Embedded Computing Systems (TECS)*, vol. 7, no. 3, 2008.

[34] 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," *IEEE Transactions on CAD of Integrated Circuits and Systems*, 2009.

[35] S. Williams, A. Waterman, and D. Patterson, "Roofline: An Insightful Visual Performance Model for Multicore Architectures," *Communications of the ACM*, vol. 52, no. 4, 2009.

[36] W. Wolf, *Computers as Components: Principles of Embedded Computing System Design.* Morgan Kaufman Publishers, 2005.

[37] J. Yan and W. Zhang, "WCET Analysis for Multi-Core Processors with Shared L2 Instruction Caches," in *14th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2008, pp. 80–89.

A

Bus Bandwidth Calculations

A.1 Bus Bandwidth Calculations

This section presents the theoretical foundation for the bus bandwidth calculations needed by the optimization algorithm in Section 6.3. We demonstrate how to calculate the desired bus bandwidth that theoretically will result in the lowest possible average-case global delay. This is done for both task segments and bus segments, as defined in Section 6.3. Furthermore, we show how to calculate the current bus bandwidth that is actually being distributed to the active processors. While this procedure is trivial for bus segments, it requires more computations for task segments.

A.1.1 Calculation of the Desired Bus Bandwidth

We will now describe how to calculate the desired bandwidth $\overline{P}_{\Omega_i}$ for a task segment $\Omega_i \in \Xi$, or $\overline{P}_{\omega_i}$ for a bus segment $\omega_i \in B$, as required by the algorithm in Section 6.3. Since a bus segment can be seen as a particular variant of a task segment, we will only talk about the latter

in this section.

The overall idea is the same as for the approach used in Chapter 5, where the the initial slot sizes of a TDMA bus schedule are determined by formulating a set of inequalities. In that section, we calculate the bandwidth distribution that theoretically minimizes the worst-case global delay. Now, on the other hand, we want to do the same, but with respect to the ACGD instead. While these are similar procedures sharing many computation steps, there are several significant differences between them.

First, it is assumed that an average-case execution time analysis has been performed with respect to a bus schedule, partly or fully tailored for the worst case. The desired bus bandwidth for a specific part of this bus schedule is, in this context, the distribution of bandwidth that will reduce the average-case global delay as much as possible.

Let τ_i^s be the first task after $\tau_i \in T_j$ that is scheduled on the same processor. Furthermore, let D_i^1 be the set of all tasks $\tau_j \in G(\Pi, \Gamma)$ which task τ_i^s depends directly on in the task graph $G(\Pi, \Gamma)$. Now, D_i is defined as the set $\{\tau_i\} \cup D_i^1$. This means that task τ_i^s will be released only when all tasks in D_i have finished executing.

Now, for a task segment $\Omega_k \in \Xi$, let us define the following functions:

- $f_k^{\mathrm{o}}(\tau_i^j)$ is defined as the number of time units of the execution path τ_i^j (of task τ_i) that executes during the time interval represented by $\Omega_k \in \Xi$.

- $f_k^{\mathrm{m}}(\tau_i^j)$ is defined as the number of cache misses on the execution path τ_i^j (of task τ_i) that is issued during the time interval represented by $\Omega_k \in \Xi$.

- $f_k^{\mathrm{s}}(\tau_i^j)$ is defined as the number of time units execution path τ_i^j (of task τ_i) spends executing code during the time interval represented by $\Omega_k \in \Xi$, excluding transfer times.

We also define the following functions, not specific to a particular task segment:

- $f^{\mathrm{p}}(\tau_i^j)$ is defined as the length, in time units, of the execution path τ_i^j (of task τ_i).

- $u(\tau_i)$ is defined as the processor number that task τ_i is executing on.

Remember that the desired bandwidth, represented as the fraction of the total bandwidth, for a task $\tau_i \in T_j$ running on processor β during the time interval represented by Ω_j is defined as $p_{\Omega_j}(\beta)$. Let us, for convenience, denote this fraction as p_i. If a is defined as the time it takes for the bus to immediately transfer a cache miss, the average waiting time can then be modeled in terms of the desired bandwidth as follows:

$$d_i = \frac{1}{p_i} a \tag{A.1}$$

Let us define G_β as the set of tasks $\tau_i \in G(\Pi, \Gamma)$ executing on processor β, and let N be the number of processors. The number of hypothetical paths is denoted by N^H. We can now estimate the average-case global delay, as a function of the bus bandwidth given during the task segment $\Omega_k \in \Xi$, by the following formula:

$$\max_{\beta=0..N} \sum_{\tau_q \in \mathrm{G}_\beta} \max_{\tau_i \in \mathrm{D}_q} \frac{\sum_{j=1..N^H} f^{\mathrm{p}}(\tau_i^j) - f_{\mathrm{k}}^{\mathrm{o}}(\tau_i^j) + f_{\mathrm{k}}^{\mathrm{m}}(\tau_i^j) d_{u(\tau_i)} + f_{\mathrm{k}}^{\mathrm{s}}(\tau_i^j)}{N^H} \tag{A.2}$$

We want to calculate the bandwidth distribution resulting in the shortest global delay. This can be done by solving - with respect to the bandwidth distribution p_1, p_2, ..., p_N and the global delay t - the following system of inequalities:

$$\begin{aligned}
\sum_{\tau_q \in \mathrm{G}_1} \max_{\tau_i \in \mathrm{D}_q} \frac{\sum_{j=1..N^H} f^{\mathrm{p}}(\tau_i^j) - f_{\mathrm{k}}^{\mathrm{o}}(\tau_i^j) + f_{\mathrm{k}}^{\mathrm{m}}(\tau_i^j) \frac{a}{p_{u(\tau_i)}} + f_{\mathrm{k}}^{\mathrm{s}}(\tau_i^j)}{N^H} &\leq t \\
\sum_{\tau_q \in \mathrm{G}_2} \max_{\tau_i \in \mathrm{D}_q} \frac{\sum_{j=1..N^H} f^{\mathrm{p}}(\tau_i^j) - f_{\mathrm{k}}^{\mathrm{o}}(\tau_i^j) + f_{\mathrm{k}}^{\mathrm{m}}(\tau_i^j) \frac{a}{p_{u(\tau_i)}} + f_{\mathrm{k}}^{\mathrm{s}}(\tau_i^j)}{N^H} &\leq t \\
&\vdots \\
\sum_{\tau_q \in \mathrm{G}_\mathrm{N}} \max_{\tau_i \in \mathrm{D}_q} \frac{\sum_{j=1..N^H} f^{\mathrm{p}}(\tau_i^j) - f_{\mathrm{k}}^{\mathrm{o}}(\tau_i^j) + f_{\mathrm{k}}^{\mathrm{m}}(\tau_i^j) \frac{a}{p_{u(\tau_i)}} + f_{\mathrm{k}}^{\mathrm{s}}(\tau_i^j)}{N^H} &\leq t \\
p_1 + p_2 + \cdots + p_n &= 1
\end{aligned}$$

Consequently, we want to find the bandwidth distribution $p_1, \ldots, p_n$ that results in the smallest possible global delay t. This minimization problem can be solved using standard techniques.

A.1.2 Calculation of the Current Bus Bandwidth

Finally, we will now show how to calculate the current bus bandwidth used in the optimization algorithm in Section 6.3. Finding the current bandwidth $\overline{P}^{bus}_{\omega_j}$ for a bus segment $\omega_j \in B$ is trivial, since it is alone determined by the TDMA round constituting the bus segment.

For a task segment $\Omega_i \in \Xi$, $\overline{P}^{bus}_{\omega_i}$ can be calculated as follows. For a task segment $\Omega_k \in \Xi$, let O_{Ω_k} denote the set of overlapping bus segments in B. Now, for each $\Omega_k \in \Xi$, let us define the function $g_{\Omega_k} : O_{\Omega_k} \to [0..1] \subseteq \mathbb{R}$, mapping every bus segment $\omega_j \in O_{\Omega_k}$ to the fraction of its total coverage of Ω_k, with respect to time. Hence, for any $\Omega_k \in \Xi$, the following holds:

$$\sum_{\omega_j \in O_{\Omega_k}} g_{\Omega_k}(\omega_i) = 1 \tag{A.3}$$

The current bandwidth of a task segment $\Omega_k \in \Xi$ can now be calculated as:

$$\overline{P}^{bus}_{\Omega_k} = \sum_{\omega_j \in O_{\Omega_k}} g_{\Omega_k}(\omega_j) \cdot \overline{P}^{bus}_{\omega_j} \tag{A.4}$$

LINKÖPINGS UNIVERSITET

Avdelning, institution
Division, department

Institutionen för datavetenskap

Department of Computer and Information Science

Datum
Date

2011-09-30

Språk
Language

[ ] Svenska/Swedish
[X] Engelska/English
[ ] _____

Rapporttyp
Report category

[X] Licentiatavhandling
[ ] Examensarbete
[ ] C-uppsats
[ ] D-uppsats
[ ] Övrig rapport

_____

ISBN 978-91-7393-090-1

ISRN LIU-TEK-LIC-2011:42

Serietitel och serienummer
Title of series, numbering

ISSN 0280-7971

Linköping Studies in Science and Technology

Thesis No. 1503

URL för elektronisk version

http://urn.kb.se/resolve?urn=urn:nbn:se:liu:diva-70138

Titel
Title

Predictable Real-Time Applications on Multiprocessor Systems-on-Chip

Författare
Author

Jakob Rosén

Sammanfattning
Abstract

Being predictable with respect to time is, by definition, a fundamental requirement for any real-time system. Modern multiprocessor systems impose a challenge in this context, due to resource sharing conflicts causing memory transfers to become unpredictable. In this thesis, we present a framework for achieving predictability for real-time applications running on multiprocessor system-on-chip platforms. Using a TDMA bus, worst-case execution time analysis and scheduling are done simultaneously. Since the worst-case execution times are directly dependent on the bus schedule, bus access design is of special importance. Therefore, we provide an efficient algorithm for generating bus schedules, resulting in a minimized worst-case global delay.

We also present a new approach considering the average-case execution time in a predictable context. Optimization techniques for improving the average-case execution time of tasks, for which predictability with respect to time is not required, have been investigated for a long time in many different contexts. However, this has traditionally been done without paying attention to the worst-case execution time. For predictable real-time applications, on the other hand, the focus has been solely on worst-case execution time optimization, ignoring how this affects the execution time in the average case. In this thesis, we show that having a good average-case global delay can be important also for real-time applications, for which predictability is required. Furthermore, for real-time applications running on multiprocessor systems-on-chip, we present a technique for optimizing for the average case and the worst case simultaneously, allowing for a good average case execution time while still keeping the worst case as small as possible. The proposed solutions in this thesis have been validated by extensive experiments. The results demonstrate the efficiency and importance of the presented techniques.

Nyckelord
Keywords

Computer Systems, Embedded Systems, Real-Time Systems, Predictability, Multiprocessor Systems

Department of Computer and Information Science
Linköpings universitet

Licentiate Theses

Linköpings Studies in Science and Technology
Faculty of Arts and Sciences

No 17 **Vojin Plavsic:** Interleaved Processing of Non-Numerical Data Stored on a Cyclic Memory. (Available at: FOA, Box 1165, S-581 11 Linköping, Sweden. FOA Report B30062E)

No 28 **Arne Jönsson, Mikael Patel:** An Interactive Flowcharting Technique for Communicating and Realizing Algorithms, 1984.

No 29 **Johnny Eckerland:** Retargeting of an Incremental Code Generator, 1984.

No 48 **Henrik Nordin:** On the Use of Typical Cases for Knowledge-Based Consultation and Teaching, 1985.

No 52 **Zebo Peng:** Steps Towards the Formalization of Designing VLSI Systems, 1985.

No 60 **Johan Fagerström:** Simulation and Evaluation of Architecture based on Asynchronous Processes, 1985.

No 71 **Jalal Maleki:** ICONStraint, A Dependency Directed Constraint Maintenance System, 1987.

No 72 **Tony Larsson:** On the Specification and Verification of VLSI Systems, 1986.

No 73 **Ola Strömfors:** A Structure Editor for Documents and Programs, 1986.

No 74 **Christos Levcopoulos:** New Results about the Approximation Behavior of the Greedy Triangulation, 1986.

No 104 **Shamsul I. Chowdhury:** Statistical Expert Systems - a Special Application Area for Knowledge-Based Computer Methodology, 1987.

No 108 **Rober Bilos:** Incremental Scanning and Token-Based Editing, 1987.

No 111 **Hans Block:** SPORT-SORT Sorting Algorithms and Sport Tournaments, 1987.

No 113 **Ralph Rönnquist:** Network and Lattice Based Approaches to the Representation of Knowledge, 1987.

No 118 **Mariam Kamkar, Nahid Shahmehri:** Affect-Chaining in Program Flow Analysis Applied to Queries of Programs, 1987.

No 126 **Dan Strömberg:** Transfer and Distribution of Application Programs, 1987.

No 127 **Kristian Sandahl:** Case Studies in Knowledge Acquisition, Migration and User Acceptance of Expert Systems, 1987.

No 139 **Christer Bäckström:** Reasoning about Interdependent Actions, 1988.

No 140 **Mats Wirén:** On Control Strategies and Incrementality in Unification-Based Chart Parsing, 1988.

No 146 **Johan Hultman:** A Software System for Defining and Controlling Actions in a Mechanical System, 1988.

No 150 **Tim Hansen:** Diagnosing Faults using Knowledge about Malfunctioning Behavior, 1988.

No 165 **Jonas Löwgren:** Supporting Design and Management of Expert System User Interfaces, 1989.

No 166 **Ola Petersson:** On Adaptive Sorting in Sequential and Parallel Models, 1989.

No 174 **Yngve Larsson:** Dynamic Configuration in a Distributed Environment, 1989.

No 177 **Peter Åberg:** Design of a Multiple View Presentation and Interaction Manager, 1989.

No 181 **Henrik Eriksson:** A Study in Domain-Oriented Tool Support for Knowledge Acquisition, 1989.

No 184 **Ivan Rankin:** The Deep Generation of Text in Expert Critiquing Systems, 1989.

No 187 **Simin Nadjm-Tehrani:** Contributions to the Declarative Approach to Debugging Prolog Programs, 1989.

No 189 **Magnus Merkel:** Temporal Information in Natural Language, 1989.

No 196 **Ulf Nilsson:** A Systematic Approach to Abstract Interpretation of Logic Programs, 1989.

No 197 **Staffan Bonnier:** Horn Clause Logic with External Procedures: Towards a Theoretical Framework, 1989.

No 203 **Christer Hansson:** A Prototype System for Logical Reasoning about Time and Action, 1990.

No 212 **Björn Fjellborg:** An Approach to Extraction of Pipeline Structures for VLSI High-Level Synthesis, 1990.

No 230 **Patrick Doherty:** A Three-Valued Approach to Non-Monotonic Reasoning, 1990.

No 237 **Tomas Sokolnicki:** Coaching Partial Plans: An Approach to Knowledge-Based Tutoring, 1990.

No 250 **Lars Strömberg:** Postmortem Debugging of Distributed Systems, 1990.

No 253 **Torbjörn Näslund:** SLDFA-Resolution - Computing Answers for Negative Queries, 1990.

No 260 **Peter D. Holmes:** Using Connectivity Graphs to Support Map-Related Reasoning, 1991.

No 283 **Olof Johansson:** Improving Implementation of Graphical User Interfaces for Object-Oriented Knowledge- Bases, 1991.

No 298 **Rolf G Larsson:** Aktivitetsbaserad kalkylering i ett nytt ekonomisystem, 1991.

No 318 **Lena Strömbäck:** Studies in Extended Unification-Based Formalism for Linguistic Description: An Algorithm for Feature Structures with Disjunction and a Proposal for Flexible Systems, 1992.

No 319 **Mikael Pettersson:** DML-A Language and System for the Generation of Efficient Compilers from Denotational Specification, 1992.

No 326 **Andreas Kågedal:** Logic Programming with External Procedures: an Implementation, 1992.

No 328 **Patrick Lambrix:** Aspects of Version Management of Composite Objects, 1992.

No 333 **Xinli Gu:** Testability Analysis and Improvement in High-Level Synthesis Systems, 1992.

No 335 **Torbjörn Näslund:** On the Role of Evaluations in Iterative Development of Managerial Support Systems, 1992.

No 348 **Ulf Cederling:** Industrial Software Development - a Case Study, 1992.

No 352 **Magnus Morin:** Predictable Cyclic Computations in Autonomous Systems: A Computational Model and Implementation, 1992.

No 371 **Mehran Noghabai:** Evaluation of Strategic Investments in Information Technology, 1993.

No 378 **Mats Larsson:** A Transformational Approach to Formal Digital System Design, 1993.

No 380 **Johan Ringström:** Compiler Generation for Parallel Languages from Denotational Specifications, 1993.

No 381 **Michael Jansson:** Propagation of Change in an Intelligent Information System, 1993.

No 383 **Jonni Harrius:** An Architecture and a Knowledge Representation Model for Expert Critiquing Systems, 1993.

No 386 **Per Österling:** Symbolic Modelling of the Dynamic Environments of Autonomous Agents, 1993.

No 398 **Johan Boye:** Dependency-based Groudness Analysis of Functional Logic Programs, 1993.

No 402 **Lars Degerstedt:** Tabulated Resolution for Well Founded Semantics, 1993.

No 406 **Anna Moberg:** Satellitkontor - en studie av kommunikationsmönster vid arbete på distans, 1993.

No 414 **Peter Carlsson:** Separation av företagsledning och finansiering - fallstudier av företagsledarutköp ur ett agent-teoretiskt perspektiv, 1994.

No 417 **Camilla Sjöström:** Revision och lagreglering - ett historiskt perspektiv, 1994.

No 436 **Cecilia Sjöberg:** Voices in Design: Argumentation in Participatory Development, 1994.

No 437 **Lars Viklund:** Contributions to a High-level Programming Environment for a Scientific Computing, 1994.

No 440 **Peter Loborg:** Error Recovery Support in Manufacturing Control Systems, 1994.

FHS 3/94 **Owen Eriksson:** Informationssystem med verksamhetskvalitet - utvärdering baserat på ett verksamhetsinriktat och samskapande perspektiv, 1994.

FHS 4/94 **Karin Pettersson:** Informationssystemstrukturering, ansvarsfördelning och användarinflytande - En komparativ studie med utgångspunkt i två informationssystemstrategier, 1994.

No 441 **Lars Poignant:** Informationsteknologi och företagsetablering - Effekter på produktivitet och region, 1994.

No 446 **Gustav Fahl:** Object Views of Relational Data in Multidatabase Systems, 1994.

No 450 **Henrik Nilsson:** A Declarative Approach to Debugging for Lazy Functional Languages, 1994.

No 451 **Jonas Lind:** Creditor - Firm Relations: an Interdisciplinary Analysis, 1994.

No 452 **Martin Sköld:** Active Rules based on Object Relational Queries - Efficient Change Monitoring Techniques, 1994.

No 455 **Pär Carlshamre:** A Collaborative Approach to Usability Engineering: Technical Communicators and System Developers in Usability-Oriented Systems Development, 1994.

FHS 5/94 **Stefan Cronholm:** Varför CASE-verktyg i systemutveckling? - En motiv- och konsekvensstudie avseende arbetssätt och arbetsformer, 1994.

No 462 **Mikael Lindvall:** A Study of Traceability in Object-Oriented Systems Development, 1994.

No 463 **Fredrik Nilsson:** Strategi och ekonomisk styrning - En studie av Sandviks förvärv av Bahco Verktyg, 1994.

No 464 **Hans Olsén:** Collage Induction: Proving Properties of Logic Programs by Program Synthesis, 1994.

No 469 **Lars Karlsson:** Specification and Synthesis of Plans Using the Features and Fluents Framework, 1995.

No 473 **Ulf Söderman:** On Conceptual Modelling of Mode Switching Systems, 1995.

No 475 **Choong-ho Yi:** Reasoning about Concurrent Actions in the Trajectory Semantics, 1995.

No 476 **Bo Lagerström:** Successiv resultatavräkning av pågående arbeten. - Fallstudier i tre byggföretag, 1995.

No 478 **Peter Jonsson:** Complexity of State-Variable Planning under Structural Restrictions, 1995.

FHS 7/95 **Anders Avdic:** Arbetsintegrerad systemutveckling med kalkylprogram, 1995.

No 482 **Eva L Ragnemalm:** Towards Student Modelling through Collaborative Dialogue with a Learning Companion, 1995.

No 488 **Eva Toller:** Contributions to Parallel Multiparadigm Languages: Combining Object-Oriented and Rule-Based Programming, 1995.

No 489 **Erik Stoy:** A Petri Net Based Unified Representation for Hardware/Software Co-Design, 1995.

No 497 **Johan Herber:** Environment Support for Building Structured Mathematical Models, 1995.

No 498 **Stefan Svenberg:** Structure-Driven Derivation of Inter-Lingual Functor-Argument Trees for Multi-Lingual Generation, 1995.

No 503 **Hee-Cheol Kim:** Prediction and Postdiction under Uncertainty, 1995.

FHS 8/95 **Dan Fristedt:** Metoder i användning - mot förbättring av systemutveckling genom situationell metodkunskap och metodanalys, 1995.

FHS 9/95 **Malin Bergvall:** Systemförvaltning i praktiken - en kvalitativ studie avseende centrala begrepp, aktiviteter och ansvarsroller, 1995.

No 513 **Joachim Karlsson:** Towards a Strategy for Software Requirements Selection, 1995.

No 517 **Jakob Axelsson:** Schedulability-Driven Partitioning of Heterogeneous Real-Time Systems, 1995.

No 518 **Göran Forslund:** Toward Cooperative Advice-Giving Systems: The Expert Systems Experience, 1995.

No 522 **Jörgen Andersson:** Bilder av småföretagares ekonomistyrning, 1995.

No 538 **Staffan Flodin:** Efficient Management of Object-Oriented Queries with Late Binding, 1996.

No 545 **Vadim Engelson:** An Approach to Automatic Construction of Graphical User Interfaces for Applications in Scientific Computing, 1996.

No 546 **Magnus Werner :** Multidatabase Integration using Polymorphic Queries and Views, 1996.

FiF-a 1/96 **Mikael Lind:** Affärsprocessinriktad förändringsanalys - utveckling och tillämpning av synsätt och metod, 1996.

No 549 **Jonas Hallberg:** High-Level Synthesis under Local Timing Constraints, 1996.

No 550 **Kristina Larsen:** Förutsättningar och begränsningar för arbete på distans - erfarenheter från fyra svenska företag. 1996.

No 557 **Mikael Johansson:** Quality Functions for Requirements Engineering Methods, 1996.

No 558 **Patrik Nordling:** The Simulation of Rolling Bearing Dynamics on Parallel Computers, 1996.

No 561 **Anders Ekman:** Exploration of Polygonal Environments, 1996.

No 563 **Niclas Andersson:** Compilation of Mathematical Models to Parallel Code, 1996.

No 567 **Johan Jenvald:** Simulation and Data Collection in Battle Training, 1996.
No 575 **Niclas Ohlsson:** Software Quality Engineering by Early Identification of Fault-Prone Modules, 1996.
No 576 **Mikael Ericsson:** Commenting Systems as Design Support—A Wizard-of-Oz Study, 1996.
No 587 **Jörgen Lindström:** Chefers användning av kommunikationsteknik, 1996.
No 589 **Esa Falkenroth:** Data Management in Control Applications - A Proposal Based on Active Database Systems, 1996.
No 591 **Niclas Wahllöf:** A Default Extension to Description Logics and its Applications, 1996.
No 595 **Annika Larsson:** Ekonomisk Styrning och Organisatorisk Passion - ett interaktivt perspektiv, 1997.
No 597 **Ling Lin:** A Value-based Indexing Technique for Time Sequences, 1997.
No 598 **Rego Granlund:** C^3Fire - A Microworld Supporting Emergency Management Training, 1997.
No 599 **Peter Ingels:** A Robust Text Processing Technique Applied to Lexical Error Recovery, 1997.
No 607 **Per-Arne Persson:** Toward a Grounded Theory for Support of Command and Control in Military Coalitions, 1997.
No 609 **Jonas S Karlsson:** A Scalable Data Structure for a Parallel Data Server, 1997.
FiF-a 4 **Carita Åbom:** Videomötesteknik i olika affärssituationer - möjligheter och hinder, 1997.
FiF-a 6 **Tommy Wedlund**: Att skapa en företagsanpassad systemutvecklingsmodell - genom rekonstruktion, värdering och vidareutveckling i T50-bolag inom ABB, 1997.
No 615 **Silvia Coradeschi**: A Decision-Mechanism for Reactive and Coordinated Agents, 1997.
No 623 **Jan Ollinen:** Det flexibla kontorets utveckling på Digital - Ett stöd för multiflex? 1997.
No 626 **David Byers:** Towards Estimating Software Testability Using Static Analysis, 1997.
No 627 **Fredrik Eklund:** Declarative Error Diagnosis of GAPLog Programs, 1997.
No 629 **Gunilla Ivefors:** Krigsspel och Informationsteknik inför en oförutsägbar framtid, 1997**.**
No 631 **Jens-Olof Lindh:** Analysing Traffic Safety from a Case-Based Reasoning Perspective, 1997
No 639 **Jukka Mäki-Turja:**. Smalltalk - a suitable Real-Time Language, 1997.
No 640 **Juha Takkinen**: CAFE: Towards a Conceptual Model for Information Management in Electronic Mail, 1997.
No 643 **Man Lin**: Formal Analysis of Reactive Rule-based Programs, 1997.
No 653 **Mats Gustafsson**: Bringing Role-Based Access Control to Distributed Systems, 1997.
FiF-a 13 **Boris Karlsson:** Metodanalys för förståelse och utveckling av systemutvecklingsverksamhet. Analys och värdering av systemutvecklingsmodeller och dess användning, 1997.
No 674 **Marcus Bjäreland:** Two Aspects of Automating Logics of Action and Change - Regression and Tractability, 1998.
No 676 **Jan Håkegård**: Hierarchical Test Architecture and Board-Level Test Controller Synthesis, 1998.
No 668 **Per-Ove Zetterlund**: Normering av svensk redovisning - En studie av tillkomsten av Redovisningsrådets rekommendation om koncernredovisning (RR01:91), 1998.
No 675 **Jimmy Tjäder**: Projektledaren & planen - en studie av projektledning i tre installations- och systemutvecklingsprojekt, 1998.
FiF-a 14 **Ulf Melin**: Informationssystem vid ökad affärs- och processorientering - egenskaper, strategier och utveckling, 1998.
No 695 **Tim Heyer**: COMPASS: Introduction of Formal Methods in Code Development and Inspection, 1998.
No 700 **Patrik Hägglund:** Programming Languages for Computer Algebra, 1998.
FiF-a 16 **Marie-Therese Christiansson:** Inter-organisatorisk verksamhetsutveckling - metoder som stöd vid utveckling av partnerskap och informationssystem, 1998.
No 712 **Christina Wennestam:** Information om immateriella resurser. Investeringar i forskning och utveckling samt i personal inom skogsindustrin, 1998.
No 719 **Joakim Gustafsson:** Extending Temporal Action Logic for Ramification and Concurrency, 1998.
No 723 **Henrik André-Jönsson:** Indexing time-series data using text indexing methods, 1999.
No 725 **Erik Larsson:** High-Level Testability Analysis and Enhancement Techniques, 1998.
No 730 **Carl-Johan Westin:** Informationsförsörjning: en fråga om ansvar - aktiviteter och uppdrag i fem stora svenska organisationers operativa informationsförsörjning, 1998.
No 731 **Åse Jansson:** Miljöhänsyn - en del i företags styrning, 1998.
No 733 **Thomas Padron-McCarthy:** Performance-Polymorphic Declarative Queries, 1998.
No 734 **Anders Bäckström:** Värdeskapande kreditgivning - Kreditriskhantering ur ett agentteoretiskt perspektiv, 1998.
FiF-a 21 **Ulf Seigerroth:** Integration av förändringsmetoder - en modell för välgrundad metodintegration, 1999.
FiF-a 22 **Fredrik Öberg:** Object-Oriented Frameworks - A New Strategy for Case Tool Development, 1998.
No 737 **Jonas Mellin:** Predictable Event Monitoring, 1998.
No 738 **Joakim Eriksson:** Specifying and Managing Rules in an Active Real-Time Database System, 1998.
FiF-a 25 **Bengt E W Andersson:** Samverkande informationssystem mellan aktörer i offentliga åtaganden - En teori om aktörsarenor i samverkan om utbyte av information, 1998.
No 742 **Pawel Pietrzak:** Static Incorrectness Diagnosis of CLP (FD), 1999.
No 748 **Tobias Ritzau:** Real-Time Reference Counting in RT-Java, 1999.
No 751 **Anders Ferntoft:** Elektronisk affärskommunikation - kontaktkostnader och kontaktprocesser mellan kunder och leverantörer på producentmarknader, 1999.
No 752 **Jo Skåmedal:** Arbete på distans och arbetsformens påverkan på resor och resmönster, 1999.
No 753 **Johan Alvehus:** Mötets metaforer. En studie av berättelser om möten, 1999.

No 754 **Magnus Lindahl:** Bankens villkor i låneavtal vid kreditgivning till högt belånade företagsförvärv: En studie ur ett agentteoretiskt perspektiv, 2000.
No 766 **Martin V. Howard:** Designing dynamic visualizations of temporal data, 1999.
No 769 **Jesper Andersson:** Towards Reactive Software Architectures, 1999.
No 775 **Anders Henriksson:** Unique kernel diagnosis, 1999.
FiF-a 30 **Pär J. Ågerfalk:** Pragmatization of Information Systems - A Theoretical and Methodological Outline, 1999.
No 787 **Charlotte Björkegren:** Learning for the next project - Bearers and barriers in knowledge transfer within an organisation, 1999.
No 788 **Håkan Nilsson:** Informationsteknik som drivkraft i granskningsprocessen - En studie av fyra revisionsbyråer, 2000.
No 790 **Erik Berglund:** Use-Oriented Documentation in Software Development, 1999.
No 791 **Klas Gäre:** Verksamhetsförändringar i samband med IS-införande, 1999.
No 800 **Anders Subotic:** Software Quality Inspection, 1999.
No 807 **Svein Bergum**: Managerial communication in telework, 2000.
No 809 **Flavius Gruian:** Energy-Aware Design of Digital Systems, 2000.
FiF-a 32 **Karin Hedström:** Kunskapsanvändning och kunskapsutveckling hos verksamhetskonsulter - Erfarenheter från ett FOU-samarbete, 2000.
No 808 **Linda Askenäs:** Affärssystemet - En studie om teknikens aktiva och passiva roll i en organisation, 2000.
No 820 **Jean Paul Meynard:** Control of industrial robots through high-level task programming, 2000.
No 823 **Lars Hult:** Publika Gränsytor - ett designexempel, 2000.
No 832 **Paul Pop:** Scheduling and Communication Synthesis for Distributed Real-Time Systems, 2000.
FiF-a 34 **Göran Hultgren:** Nätverksinriktad Förändringsanalys - perspektiv och metoder som stöd för förståelse och utveckling av affärsrelationer och informationssystem, 2000.
No 842 **Magnus Kald:** The role of management control systems in strategic business units, 2000.
No 844 **Mikael Cäker:** Vad kostar kunden? Modeller för intern redovisning, 2000.
FiF-a 37 **Ewa Braf**: Organisationers kunskapsverksamheter - en kritisk studie av ”knowledge management”, 2000.
FiF-a 40 **Henrik Lindberg:** Webbaserade affärsprocesser - Möjligheter och begränsningar, 2000.
FiF-a 41 **Benneth Christiansson:** Att komponentbasera informationssystem - Vad säger teori och praktik?, 2000.
No. 854 **Ola Pettersson:** Deliberation in a Mobile Robot, 2000.
No 863 **Dan Lawesson**: Towards Behavioral Model Fault Isolation for Object Oriented Control Systems, 2000.
No 881 **Johan Moe:** Execution Tracing of Large Distributed Systems, 2001.
No 882 **Yuxiao Zhao:** XML-based Frameworks for Internet Commerce and an Implementation of B2B e-procurement, 2001.
No 890 **Annika Flycht-Eriksson:** Domain Knowledge Management in Information-providing Dialogue systems, 2001.
FiF-a 47 **Per-Arne Segerkvis**t: Webbaserade imaginära organisationers samverkansformer: Informationssystemarkitektur och aktörssamverkan som förutsättningar för affärsprocesser, 2001.
No 894 **Stefan Svarén:** Styrning av investeringar i divisionaliserade företag - Ett koncernperspektiv, 2001.
No 906 **Lin Han**: Secure and Scalable E-Service Software Delivery, 2001.
No 917 **Emma Hansson:** Optionsprogram för anställda - en studie av svenska börsföretag, 2001.
No 916 **Susanne Odar:** IT som stöd för strategiska beslut, en studie av datorimplementerade modeller av verksamhet som stöd för beslut om anskaffning av JAS 1982, 2002.
FiF-a-49 **Stefan Holgersson:** IT-system och filtrering av verksamhetskunskap - kvalitetsproblem vid analyser och beslutsfattande som bygger på uppgifter hämtade från polisens IT-system, 2001.
FiF-a-51 **Per Oscarsson:** Informationssäkerhet i verksamheter - begrepp och modeller som stöd för förståelse av informationssäkerhet och dess hantering, 2001.
No 919 **Luis Alejandro Cortes:** A Petri Net Based Modeling and Verification Technique for Real-Time Embedded Systems, 2001.
No 915 **Niklas Sandell:** Redovisning i skuggan av en bankkris - Värdering av fastigheter. 2001.
No 931 **Fredrik Elg:** Ett dynamiskt perspektiv på individuella skillnader av heuristisk kompetens, intelligens, mentala modeller, mål och konfidens i kontroll av mikrovärlden Moro, 2002.
No 933 **Peter Aronsson:** Automatic Parallelization of Simulation Code from Equation Based Simulation Languages, 2002.
No 938 **Bourhane Kadmiry**: Fuzzy Control of Unmanned Helicopter, 2002.
No 942 **Patrik Haslum**: Prediction as a Knowledge Representation Problem: A Case Study in Model Design, 2002.
No 956 **Robert Sevenius:** On the instruments of governance - A law & economics study of capital instruments in limited liability companies, 2002.
FiF-a 58 **Johan Petersson:** Lokala elektroniska marknadsplatser - informationssystem för platsbundna affärer, 2002.
No 964 **Peter Bunus:** Debugging and Structural Analysis of Declarative Equation-Based Languages, 2002.
No 973 **Gert Jervan:** High-Level Test Generation and Built-In Self-Test Techniques for Digital Systems, 2002.
No 958 **Fredrika Berglund:** Management Control and Strategy - a Case Study of Pharmaceutical Drug Development, 2002.
FiF-a 61 **Fredrik Karlsson:** Meta-Method for Method Configuration - A Rational Unified Process Case, 2002.
No 985 **Sorin Manolache:** Schedulability Analysis of Real-Time Systems with Stochastic Task Execution Times, 2002.
No 982 **Diana Szentiványi:** Performance and Availability Trade-offs in Fault-Tolerant Middleware, 2002.
No 989 **Iakov Nakhimovski:** Modeling and Simulation of Contacting Flexible Bodies in Multibody Systems, 2002.
No 990 **Levon Saldamli:** PDEModelica - Towards a High-Level Language for Modeling with Partial Differential Equations, 2002.
No 991 **Almut Herzog:** Secure Execution Environment for Java Electronic Services, 2002.

No 999 **Jon Edvardsson:** Contributions to Program- and Specification-based Test Data Generation, 2002.
No 1000 **Anders Arpteg:** Adaptive Semi-structured Information Extraction, 2002.
No 1001 **Andrzej Bednarski:** A Dynamic Programming Approach to Optimal Retargetable Code Generation for Irregular Architectures, 2002.
No 988 **Mattias Arvola:** Good to use! : Use quality of multi-user applications in the home, 2003.
FiF-a 62 **Lennart Ljung:** Utveckling av en projektivitetsmodell - om organisationers förmåga att tillämpa projektarbetsformen, 2003.
No 1003 **Pernilla Qvarfordt:** User experience of spoken feedback in multimodal interaction, 2003.
No 1005 **Alexander Siemers:** Visualization of Dynamic Multibody Simulation With Special Reference to Contacts, 2003.
No 1008 **Jens Gustavsson:** Towards Unanticipated Runtime Software Evolution, 2003.
No 1010 **Calin Curescu:** Adaptive QoS-aware Resource Allocation for Wireless Networks, 2003.
No 1015 **Anna Andersson:** Management Information Systems in Process-oriented Healthcare Organisations, 2003.
No 1018 **Björn Johansson:** Feedforward Control in Dynamic Situations, 2003.
No 1022 **Traian Pop:** Scheduling and Optimisation of Heterogeneous Time/Event-Triggered Distributed Embedded Systems, 2003.
FiF-a 65 **Britt-Marie Johansson:** Kundkommunikation på distans - en studie om kommunikationsmediets betydelse i affärstransaktioner, 2003.
No 1024 **Aleksandra Tešanovic:** Towards Aspectual Component-Based Real-Time System Development, 2003.
No 1034 **Arja Vainio-Larsson**: Designing for Use in a Future Context - Five Case Studies in Retrospect, 2003.
No 1033 **Peter Nilsson:** Svenska bankers redovisningsval vid reservering för befarade kreditförluster - En studie vid införandet av nya redovisningsregler, 2003.
FiF-a 69 **Fredrik Ericsson:** Information Technology for Learning and Acquiring of Work Knowledge, 2003.
No 1049 **Marcus Comstedt:** Towards Fine-Grained Binary Composition through Link Time Weaving, 2003.
No 1052 **Åsa Hedenskog:** Increasing the Automation of Radio Network Control, 2003.
No 1054 **Claudiu Duma:** Security and Efficiency Tradeoffs in Multicast Group Key Management, 2003.
FiF-a 71 **Emma Eliason:** Effektanalys av IT-systems handlingsutrymme, 2003.
No 1055 **Carl Cederberg:** Experiments in Indirect Fault Injection with Open Source and Industrial Software, 2003.
No 1058 **Daniel Karlsson:** Towards Formal Verification in a Component-based Reuse Methodology, 2003.
FiF-a 73 **Anders Hjalmarsson:** Att etablera och vidmakthålla förbättringsverksamhet - behovet av koordination och interaktion vid förändring av systemutvecklingsverksamheter, 2004.
No 1079 **Pontus Johansson:** Design and Development of Recommender Dialogue Systems, 2004.
No 1084 **Charlotte Stoltz:** Calling for Call Centres - A Study of Call Centre Locations in a Swedish Rural Region, 2004.
FiF-a 74 **Björn Johansson:** Deciding on Using Application Service Provision in SMEs, 2004.
No 1094 **Genevieve Gorrell:** Language Modelling and Error Handling in Spoken Dialogue Systems, 2004.
No 1095 **Ulf Johansson:** Rule Extraction - the Key to Accurate and Comprehensible Data Mining Models, 2004.
No 1099 **Sonia Sangari:** Computational Models of Some Communicative Head Movements, 2004.
No 1110 **Hans Nässla:** Intra-Family Information Flow and Prospects for Communication Systems, 2004.
No 1116 **Henrik Sällberg:** On the value of customer loyalty programs - A study of point programs and switching costs, 2004.
FiF-a 77 **Ulf Larsson:** Designarbete i dialog - karaktärisering av interaktionen mellan användare och utvecklare i en systemutvecklingsprocess, 2004.
No 1126 **Andreas Borg:** Contribution to Management and Validation of Non-Functional Requirements, 2004.
No 1127 **Per-Ola Kristensson:** Large Vocabulary Shorthand Writing on Stylus Keyboard, 2004.
No 1132 **Pär-Anders Albinsson:** Interacting with Command and Control Systems: Tools for Operators and Designers, 2004.
No 1130 **Ioan Chisalita:** Safety-Oriented Communication in Mobile Networks for Vehicles, 2004.
No 1138 **Thomas Gustafsson:** Maintaining Data Consistency in Embedded Databases for Vehicular Systems, 2004.
No 1149 **Vaida Jakoniené:** A Study in Integrating Multiple Biological Data Sources, 2005.
No 1156 **Abdil Rashid Mohamed**: High-Level Techniques for Built-In Self-Test Resources Optimization, 2005.
No 1162 **Adrian Pop:** Contributions to Meta-Modeling Tools and Methods, 2005.
No 1165 **Fidel Vascós Palacios:** On the information exchange between physicians and social insurance officers in the sick leave process: an Activity Theoretical perspective, 2005.
FiF-a 84 **Jenny Lagsten:** Verksamhetsutvecklande utvärdering i informationssystemprojekt, 2005.
No 1166 **Emma Larsdotter Nilsson:** Modeling, Simulation, and Visualization of Metabolic Pathways Using Modelica, 2005.
No 1167 **Christina Keller:** Virtual Learning Environments in higher education. A study of students' acceptance of educational technology, 2005.
No 1168 **Cécile Åberg:** Integration of organizational workflows and the Semantic Web, 2005.
FiF-a 85 **Anders Forsman:** Standardisering som grund för informationssamverkan och IT-tjänster - En fallstudie baserad på trafikinformationstjänsten RDS-TMC, 2005.
No 1171 **Yu-Hsing Huang:** A systemic traffic accident model, 2005.
FiF-a 86 **Jan Olausson:** Att modellera uppdrag - grunder för förståelse av processinriktade informationssystem i transaktionsintensiva verksamheter, 2005.
No 1172 **Petter Ahlström**: Affärsstrategier för seniorbostadsmarknaden, 2005.
No 1183 **Mathias Cöster**: Beyond IT and Productivity - How Digitization Transformed the Graphic Industry, 2005.
No 1184 **Åsa Horzella**: Beyond IT and Productivity - Effects of Digitized Information Flows in Grocery Distribution, 2005.
No 1185 **Maria Kollberg**: Beyond IT and Productivity - Effects of Digitized Information Flows in the Logging Industry, 2005.
No 1190 **David Dinka**: Role and Identity - Experience of technology in professional settings, 2005.

No 1191 **Andreas Hansson**: Increasing the Storage Capacity of Recursive Auto-associative Memory by Segmenting Data, 2005.
No 1192 **Nicklas Bergfeldt:** Towards Detached Communication for Robot Cooperation, 2005.
No 1194 **Dennis Maciuszek:** Towards Dependable Virtual Companions for Later Life, 2005.
No 1204 **Beatrice Alenljung:** Decision-making in the Requirements Engineering Process: A Human-centered Approach, 2005.
No 1206 **Anders Larsson:** System-on-Chip Test Scheduling and Test Infrastructure Design, 2005.
No 1207 **John Wilander:** Policy and Implementation Assurance for Software Security, 2005.
No 1209 **Andreas Käll:** Översättningar av en managementmodell - En studie av införandet av Balanced Scorecard i ett landsting, 2005.
No 1225 **He Tan:** Aligning and Merging Biomedical Ontologies, 2006.
No 1228 **Artur Wilk:** Descriptive Types for XML Query Language Xcerpt, 2006.
No 1229 **Per Olof Pettersson:** Sampling-based Path Planning for an Autonomous Helicopter, 2006.
No 1231 **Kalle Burbeck:** Adaptive Real-time Anomaly Detection for Safeguarding Critical Networks, 2006.
No 1233 **Daniela Mihailescu:** Implementation Methodology in Action: A Study of an Enterprise Systems Implementation Methodology, 2006.
No 1244 **Jörgen Skågeby:** Public and Non-public gifting on the Internet, 2006.
No 1248 **Karolina Eliasson:** The Use of Case-Based Reasoning in a Human-Robot Dialog System, 2006.
No 1263 **Misook Park-Westman:** Managing Competence Development Programs in a Cross-Cultural Organisation - What are the Barriers and Enablers, 2006.
FiF-a 90 **Amra Halilovic:** Ett praktikperspektiv på hantering av mjukvarukomponenter, 2006.
No 1272 **Raquel Flodström**: A Framework for the Strategic Management of Information Technology, 2006.
No 1277 **Viacheslav Izosimov:** Scheduling and Optimization of Fault-Tolerant Embedded Systems, 2006.
No 1283 **Håkan Hasewinkel:** A Blueprint for Using Commercial Games off the Shelf in Defence Training, Education and Research Simulations, 2006.
FiF-a 91 **Hanna Broberg:** Verksamhetsanpassade IT-stöd - Designteori och metod, 2006.
No 1286 **Robert Kaminski:** Towards an XML Document Restructuring Framework, 2006.
No 1293 **Jiri Trnka:** Prerequisites for data sharing in emergency management, 2007.
No 1302 **Björn Hägglund:** A Framework for Designing Constraint Stores, 2007.
No 1303 **Daniel Andreasson**: Slack-Time Aware Dynamic Routing Schemes for On-Chip Networks, 2007.
No 1305 **Magnus Ingmarsson:** Modelling User Tasks and Intentions for Service Discovery in Ubiquitous Computing, 2007.
No 1306 **Gustaf Svedjemo**: Ontology as Conceptual Schema when Modelling Historical Maps for Database Storage, 2007.
No 1307 **Gianpaolo Conte**: Navigation Functionalities for an Autonomous UAV Helicopter, 2007.
No 1309 **Ola Leifler**: User-Centric Critiquing in Command and Control: The DKExpert and ComPlan Approaches, 2007.
No 1312 **Henrik Svensson**: Embodied simulation as off-line representation, 2007.
No 1313 **Zhiyuan He:** System-on-Chip Test Scheduling with Defect-Probability and Temperature Considerations, 2007.
No 1317 **Jonas Elmqvist:** Components, Safety Interfaces and Compositional Analysis, 2007.
No 1320 **Håkan Sundblad:** Question Classification in Question Answering Systems, 2007.
No 1323 **Magnus Lundqvist**: Information Demand and Use: Improving Information Flow within Small-scale Business Contexts, 2007.
No 1329 **Martin Magnusson:** Deductive Planning and Composite Actions in Temporal Action Logic, 2007.
No 1331 **Mikael Asplund:** Restoring Consistency after Network Partitions, 2007.
No 1332 **Martin Fransson:** Towards Individualized Drug Dosage - General Methods and Case Studies, 2007.
No 1333 **Karin Camara:** A Visual Query Language Served by a Multi-sensor Environment, 2007.
No 1337 **David Broman:** Safety, Security, and Semantic Aspects of Equation-Based Object-Oriented Languages and Environments, 2007.
No 1339 **Mikhail Chalabine:** Invasive Interactive Parallelization, 2007.
No 1351 **Susanna Nilsson:** A Holistic Approach to Usability Evaluations of Mixed Reality Systems, 2008.
No 1353 **Shanai Ardi:** A Model and Implementation of a Security Plug-in for the Software Life Cycle, 2008.
No 1356 **Erik Kuiper:** Mobility and Routing in a Delay-tolerant Network of Unmanned Aerial Vehicles, 2008.
No 1359 **Jana Rambusch**: Situated Play, 2008.
No 1361 **Martin Karresand:** Completing the Picture - Fragments and Back Again, 2008.
No 1363 **Per Nyblom:** Dynamic Abstraction for Interleaved Task Planning and Execution, 2008.
No 1371 **Fredrik Lantz:**Terrain Object Recognition and Context Fusion for Decision Support, 2008.
No 1373 **Martin Östlund:** Assistance Plus: 3D-mediated Advice-giving on Pharmaceutical Products, 2008.
No 1381 **Håkan Lundvall:** Automatic Parallelization using Pipelining for Equation-Based Simulation Languages, 2008.
No 1386 **Mirko Thorstensson:** Using Observers for Model Based Data Collection in Distributed Tactical Operations, 2008.
No 1387 **Bahlol Rahimi:** Implementation of Health Information Systems, 2008.
No 1392 **Maria Holmqvist:** Word Alignment by Re-using Parallel Phrases, 2008.
No 1393 **Mattias Eriksson**: Integrated Software Pipelining, 2009.
No 1401 **Annika Öhgren:** Towards an Ontology Development Methodology for Small and Medium-sized Enterprises, 2009.
No 1410 **Rickard Holsmark:** Deadlock Free Routing in Mesh Networks on Chip with Regions, 2009.
No 1421 **Sara Stymne:** Compound Processing for Phrase-Based Statistical Machine Translation, 2009.
No 1427 **Tommy Ellqvist**: Supporting Scientific Collaboration through Workflows and Provenance, 2009.
No 1450 **Fabian Segelström:** Visualisations in Service Design, 2010.
No 1459 **Min Bao**: System Level Techniques for Temperature-Aware Energy Optimization, 2010.
No 1466 **Mohammad Saifullah**: Exploring Biologically Inspired Interactive Networks for Object Recognition, 2011

No 1468 **Qiang Liu**: Dealing with Missing Mappings and Structure in a Network of Ontologies, 2011.
No 1469 **Ruxandra Pop**: Mapping Concurrent Applications to Multiprocessor Systems with Multithreaded Processors and Network on Chip-Based Interconnections, 2011
No 1476 **Per-Magnus Olsson**: Positioning Algorithms for Surveillance Using Unmanned Aerial Vehicles, 2011
No 1481 **Anna Vapen**: Contributions to Web Authentication for Untrusted Computers, 2011
No 1485 **Loove Broms:** Sustainable Interactions: Studies in the Design of Energy Awareness Artefacts, 2011
FiF-a 101 **Johan Blomkvist:** Conceptualising Prototypes in Service Design, 2011
No 1490 **Håkan Warnquist:** Computer-Assisted Troubleshooting for Efficient Off-board Diagnosis, 2011
No 1503 **Jakob Rosén:** Predictable Real-Time Applications on Multiprocessor Systems-on-Chip, 2011