Analysis and Optimization Techniques for Real-Time Streaming Image Processing Software on General Purpose Systems

Mark Westmijze

Analysis and Optimization Techniques for Real-Time Streaming Image Processing Software on General Purpose Systems

Members of the dissertation committee:

Prof. dr. ir.	M.J.G. Bekooij	University of Twente (promotor)
Prof. dr. ir.	G.J.M. Smit	University of Twente
Prof. dr.	M. Huisman	University of Twente
Prof. dr.	H. Corporaal	Eindhoven University of Technology
Prof. dr.	A. Kumar	Technishe Universität Dresden
Dr.	T.P. Stefanov	Leiden University
Dr. ir.	M. Schrijver	ASML
Prof. dr.	J.N. Kok	University of Twente (chairman and secretary)

UNIVERSITY OF TWENTE. | DIGITAL SOCIETY INSTITUTE

Faculty of Electrical Engineering, Mathematics and Computer Science,
Computer Architecture for Embedded Systems (CAES) group.

DSI Ph.D. Thesis Series No. 18-002
Digital Society Institute
PO Box 217, 7500 AE Enschede, The Netherlands

This research has been conducted within the Netherlands Streaming (NEST) project (project number 10346). This research is supported by the Dutch Technology Foundation STW, which is part of the Netherlands Organisation for Scientific Research (NWO) and partly funded by the Ministry of Economic Affairs.

This thesis was printed by Ipskamp, The Netherlands.

ISBN 978-90-365-4569-3
ISSN 2589-7721 (DSI Ph.d-thesis serie No. 18-002)
DOI 10.3990/1.9789036545693

Analysis and Optimization Techniques for Real-Time Streaming Image Processing Software on General Purpose Systems

PROEFSCHRIFT

ter verkrijging van
de graad van doctor aan de Universiteit Twente,
op gezag van de rector magnificus,
prof. dr. T.T.M. Palstra,
volgens besluit van het College voor Promoties
in het openbaar te verdedigen
op vrijdag 29 juni 2018 om 12.45 uur.

door

Mark Westmijze

geboren op 29 mei 1984
te Deventer

Dit proefschrift is goedgekeurd door:

Prof. dr. ir. M.J.G. Bekooij (promotor)

Abstract

Commercial Off The Shelf (COTS) Chip Multi-Processor (CMP) systems are for cost reasons often used in industry for soft real-time stream processing. COTS CMP systems typically have a low timing predictability, which makes it difficult to develop software applications for these systems with tight temporal requirements. Restricting the way applications use the hardware and Operating System (OS) might alleviate this difficulty, so that certain types of applications could be run on COTS CMP systems with statistically verified temporal requirements. In this thesis we restrict the application domain to soft real-time medical image processing applications, which have a much more 'stable' usage of hardware resources than applications in general. Techniques at the application level are employed to improve the reproducibility (i.e. to reduce the variance) of the end-to-end latency of these imaging processing systems.

Firstly, we study the effectiveness of a number of scheduling heuristics that are intended to improve the reproducibility of a stream processing application that is executed on COTS multiprocessor systems. Experiments show that the proposed heuristics can reduce the end-to-end latency with almost 60%, and reduce the variation in the latency with more than 90%, when compared with a naive scheduling heuristic that does not consider execution times, dependencies and the memory hierarchy.

Secondly, we want to be able to integrate multiple real-time and best-effort applications on a single COTS CMP system without reducing the reproducibility of the real-time application too much. For this we examined the first component that is shared between different applications running on separate cores, the shared cache and in particular the bandwidth in the cache. We propose a technique that implements cache bandwidth reservation in software. This is achieved by dynamically duty-cycling best-effort applications based on their cache bandwidth usages measured with processor performance counters. With this technique we can control the latency increase of real-time applications that is caused by best-effort

applications.

Thirdly, we introduce the Probabilistic Time Triggered System (PTTS) model to analyze and optimize the end-to-end latency of a complete system that contains multiple time triggered interfaces. Our case study demonstrates the applicability of the PTTS model and the corresponding analysis techniques for an interventional X-ray system. We expect that the PTTS model is also applicable for other systems than medical image processing systems.

Contents

CHAPTER 1

Introduction

Commercial Off The Shelf (COTS) Chip Multi-Processor (CMP) systems are for cost reasons often used in industry for soft real-time stream processing. COTS CMP systems typically have a low timing predictability, which makes it difficult to develop software applications for these systems with tight temporal requirements. Since soft real-time requirements are probabilistic by definition and the performance of applications of these systems is usually measured and described statistically, it is almost impossible to formally verify that these requirements will be met. However, since the soft real-time requirements are probabilistic it is not necessary to formally guarantee an upper bound, only to statistically verify them. This also might prove difficult because the performance of these system is highly dependent on how the application uses the hardware, interacts with the OS and other applications directly or indirectly (due to shared resources). Restricting the way applications use the hardware and OS might alleviate this difficulty, so that certain types of applications could be run on a COTS CMP system with statistically verified temporal requirements. In this thesis we restrict the application domain to soft real-time medical image processing applications, which have a much more 'stable' usage of hardware resources than applications in general. We will introduce the concept of reproducibility in order to compare measured execution times with each other and the temporal requirements. The concept of reproducibility will allow us to think and reason about how well a given design adheres to the temporal requirements and in what direction the design can be improved, if the design does not satisfy the temporal requirements.

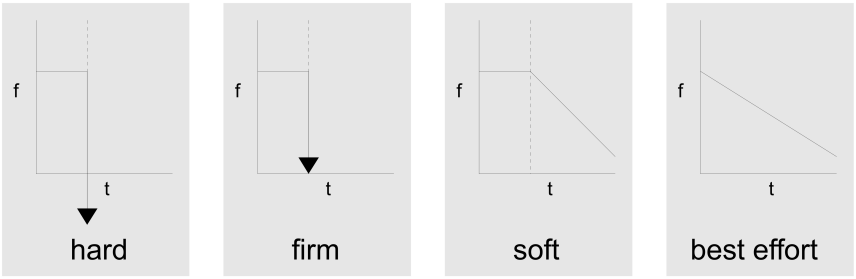


Figure 1.1: Real-time system types and their utility function

Firstly we will briefly introduce the different kinds of real-time systems and how they compare to best-effort systems. The difference between real-time systems in comparison to best-effort systems is that there are timing requirements. How the temporal requirements are formulated and what the consequences are in case the requirements are violated determines the kind of real-time system.

In the most stringent case, a hard real-time system, deadlines should never be violated because the consequence of a deadline violation can be something catastrophic. This will be the case for example, when the deadline violation might result in death or serious harm. The braking control system of a car is an example of a hard real-time system; the system should react within milliseconds to prevent collisions. The temporal requirements of an application can be described using a utility function [10]. A utility function gives for a certain execution time the value of the result. Such value is a measure of "quality". For hard real-time systems the value of the utility function decreases to negative infinity directly after the deadline, as shown in Figure 1.1, since a deadline miss is assumed to be harmful after the deadline. The vertical dashed line represents the deadline.

A slightly more lenient form of real-time systems is a firm real-time system where the result of a deadline violation is merely inconvenient. In this case the value of the utility function after the deadline is zero. An example is when an audio frame is not ready at the deadline. This is immediately noticeable as a click or noise in the produced sound, but this does not result in dangerous situations.

The type of real-time systems that we consider in this thesis are soft real-time systems. For soft real-time systems the value of the utility function after the deadline does not immediately drop to zero, but slowly

diminishes. An example of this is: generating the next frame in a video sequence can be slightly delayed without hardly any decrease of quality.

For best effort systems the value of the utility function immediately starts to decrease. I.e., faster is always better. See Figure 1.1 for a graphical depiction of the value of the utility function as a function of time for the different real-time systems.

Due to the importance of the timing constraints in many (hard) real-time systems the designers of such systems are allowed to justify elaborate hardware designs in order to satisfy those constraints. Furthermore, there are several models and techniques available to design and verify the temporal requirements of these systems. Many of these techniques are used to predict the execution time and especially the Worst Case Execution Time (WCET) of the system in order to guarantee that the temporal requirements are met. The hardware components that are used for hard real-time designs typically increase the cost of the designs, which make them more expensive than COTS architectures that are commonly used in servers, personal computers and increasingly also for soft real-time systems.

The differences between the underlying hardware architectures and their influence on the timing predictability of the software running on those architectures make the models that are developed for hard real-time systems not directly applicable for soft real-time applications, since the hard real-time models will result in over provisioning of the system, which often cannot be economically justified. In some cases it is even impossible to apply hard real-time analysis techniques to a soft real-time system due to the fact that an appropriate timing model of the used hardware is unavailable. Without these models it is impossible to derive WCETs, which implies that any formal guarantees related to the temporal requirements cannot be given.

Furthermore, it is important to note that the formal guarantees are only valid on the model, i.e., on an abstraction of reality. The correctness of the formal guarantees is only as good as the assumption that the model makes of the reality. For example, the guarantees of a model that does not take hardware or power failures into account will never correctly map to reality. In this case the guarantees will only hold as long as the underlying assumption of the model is valid, which in this case is that the hardware works perfectly and the power is always available. This implies that for any model, which is an abstraction of the reality,, by definition, the guarantees only hold for the model and not for the real-

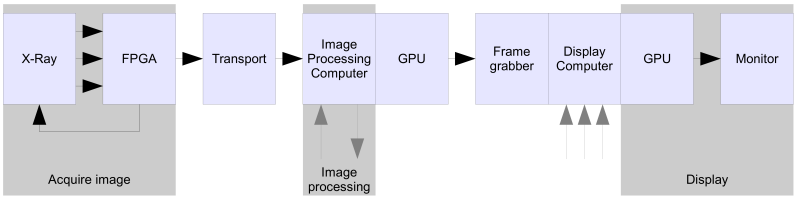


Figure 1.2: Overview of a complete interventional X-ray system

ity. How good formal guarantees of a certain model hold for reality is difficult to state, and an area where much remains unclear. Lee *et al* described this problem eloquently in [27] and [29]. Because all models lose, depending on the underlying assumptions, some accuracy, we can also conclude that for some systems we always need to empirically validate the final implemented system to the original model. If we already need to empirically validate the system anyway, it also becomes less stringent to require formal guarantees of the model on which the system is based.

1.1 Case study: interventional X-ray system

A practical example of a real-time system is an interventional X-ray system. With such a system a physician can make use of images captured with an X-ray imaging device to perform delicate medical procedures inside a patient, where the only visual feedback is provided by the images captured by the X-ray device. Therefore the latency constraint between the capturing of an image and displaying it should be short enough (< 200 ms) to provide sufficient eye-hand coordination. Furthermore, the variation of the latency, which is called jitter, must be sufficiently low such that the physician experiences a constant delay, which improves the eye-hand coordination and prevents fatigue. As mentioned above, in this system the constraints on the latencies are in the order of milliseconds. This allows us to abstract from some aspects of the components of a COTS CMP system, which we would not be able to do this, if our system specified latencies in the order of microseconds. For example, we do not take into account the variation of execution times that are introduced by arithmetic engines (i.e. the pipelines) of the processors because these variations are usually in the microsecond range.

Advanced image processing is necessary to obtain sufficient image quality from an X-ray sensor when used with a very low radiation dose. In an interventional X-ray application, only a fraction of the latency budget is available for image processing due to the latency that the detec-

tor and display introduce. Therefore, the image processing used to be performed on architectures like Application Specific Integrated Circuits (ASICs), Digital Signal Processors (DSPs) and Field-Programmable Gate Arrays (FPGAs). However, high performance CMP COTS hardware has become performance-wise so powerful and cost-effective, that the trend is to perform the image processing on this type of hardware despite the increased temporal uncertainty that this hardware may introduce. The use of COTS hardware seems to be acceptable as long as temporal constraints are only rarely violated. Furthermore, the usage of a COTS system would also allow for further integration of other parts of the system, which would also typically run on such a system. Among others, this reduces the overall complexity of the system, reduces costs with regard to logistics and increases reliability. However, in practice it is difficult to design such a system without systematic analysis and optimization methods.

Image processing systems such as interventional X-ray systems have soft real-time requirements. However, as discussed, there are currently no systematic real-time analysis and optimization techniques available for the programmer of soft real-time systems. This results in a trial and error type of programming style, which potentially results in systems with a suboptimal temporal behavior.

In this thesis we address the above mentioned issues. In particular this thesis is concerned with the analysis and optimization of soft real-time image processing systems. More specifically we focus on a specific real-time system, namely an interventional X-ray system that consists of a series of sub-systems, of which several are implemented on COTS CMP hardware. New models, analysis and optimization techniques will be presented that are used to reduce the temporal uncertainty introduced by these systems by means of adaptation of the operating system and application software running on these systems.

Figure 1.2 gives a high level overview of the components present in an interventional X-ray system. The system consists of three main parts. The first part is image acquisition, the second image processing and finally showing the processed image. The image acquisition is responsible for retrieving the X-ray image from a Charge-Coupled Device (CCD) and performs some rudimentary image processing algorithms. The image processing part performs computationally heavy image processing algorithms such as noise reduction. The display part shows the processed image and optionally integrates the output of other devices on a single display. The separate parts are either connected by Ethernet or by a DVI connection and are not synchronized to a single clock domain.

The remainder of this chapter is organized as follows. First we position our research more precisely in Section 1.2. Next we present an overview of the problems that we address in this thesis in Section 1.3. The problem statement is given in Section 1.4 and finally we present an outline of the remainder of this thesis in Section 1.5.

1.2 Positioning

At the time that the work for this thesis was performed it did fall between two popular research domains, namely the real-time domain and the high performance computing domain. The work was too empirical to nicely fit in the real-time domain. However, our need for reducing worst case behavior instead of optimizing average case behavior also did not put the work in the high performance computing domain. In this section we will first position the work against the real-time domain. Finally, we will discuss some work that has been performed in the last few years and which more closely resembles the work that is presented in this thesis.

There are several analysis methods available for the analysis of hard real-time systems. Each technique is applicable for certain combinations of applications and hardware platforms. Techniques for analyzing independent periodic and aperiodic tasks executing on a single core have been described by Butazzo *et el* [4].

Dataflow techniques described by Sriram and Bhattacharyya [41] can be used to analyze applications consisting of dependent tasks that execute concurrently on multiple cores.

Synchronous Data Flow (SDF) extends the model with timing and is first described by Lee *et al* [28]. Within the synchronous dataflow domain there are several analysis models used such as Homogeneous Synchronous Dataflow (HSDF) graphs [28] and Cyclo-Static Dataflow (CSDF) graphs [37]. Typically the hardware that is analyzed with dataflow analysis techniques is specifically designed for real-time systems. Techniques that increase the uncertainty of execution times are used more conservatively in such systems than in high performance systems (e.g. no shared caches, no speculative pipelining, etc). Therefore, dataflow techniques are typically used for software applications with a low degree of uncertainty/-dynamism that are running on hardware with a low degree of uncertainty. In fact the opposite is the case for *general purpose* applications, which typically show a high degree of temporal uncertainty and run on high-performance COTS hardware, which also introduces a high degree of uncertainty.

Many researchers have focused on deriving the WCET of applications on numerous architectures. For example, Thesing [48] introduced techniques to derive the WCET of tasks running on single core processors by modeling the pipeline of these processors. We cannot use such techniques since the processors that we consider are multi-core and use several ingenious and creative techniques to increase average case performance at the cost of an increase in WCET, for example by using shared data caches. An overview of techniques and methods to derive the WCET of applications can be found in [54].

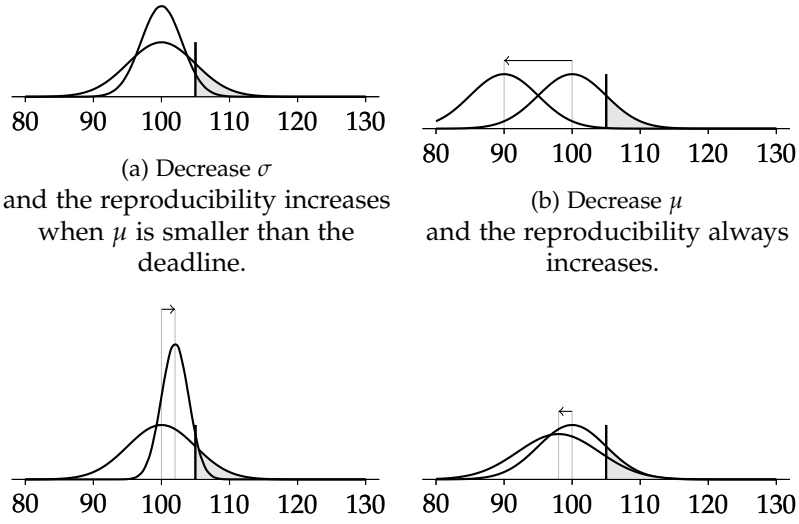
Most of the introduced techniques for hard real-time systems focus on predicting the execution times. Because these techniques are not applicable in our use-case we focus on *reproducibility*. We define reproducibility as follows: *an application on a system has a better reproducible behavior, when the probability of deadline violations is smaller. In a system that can run an application with a highly reproducible behavior, there is only a small influence of other applications that run on the system on the execution times of the tasks that belong to the considered reproducible application. We do not require all applications on a system to have a highly reproducible behavior, but only the subset of applications that have soft real-time requirements.*

It is interesting to specifically address the differences between this concept of reproducibility and composability [24]. Composability refers to whether applications influence each other, when executed concurrently on a platform. This allows applications to be designed, developed and verified in isolation. Reproducibility does not state that applications will not influence each other, only how well a given application adheres to its temporal requirements irrespective of other applications. This means that applications cannot be measured and verified in isolation. This makes composability a more desired feature than reproducibility. However, composability can usually not be achieved on COTS CMP systems, which makes this concept useless given our platform choice.

Figure 1.3 shows several graphs with a Gaussian Probability Density Function (PDF) with different parameters in order to graphically show the difference between the reproducibility represented by those PDFs. The PDF represents the latency distribution of an application running on COTS CMP hardware. The red vertical bar represents the deadline (105).

Figure 1.3a specifically shows two Gaussian functions where the variance (σ) is different.

In the other sub figures we show other circumstances under which the re-



(a) Decrease σ and the reproducibility increases when μ is smaller than the deadline.

(b) Decrease μ and the reproducibility always increases.

(c) Increase μ and decrease σ and the reproducibility might increase.

(d) Decrease μ and increase σ and the reproducibility might increase.

Figure 1.3: Example latency distributions and their reproducibility as function of μ and σ.

producibility between two PDFs differs. For example, in Figure 1.3b only the mean (μ) has decreased, which clearly reduces the probability of deadline violations. When the mean is increased, as shown in Figure 1.3c, the variance has to decrease enough in order to not increase the probability of deadline violations. Lastly, Figure 1.3d shows a situation where even though the variance has increased the reproducibility is still higher. This is because of the smaller probability of deadline violation since the mean has decreased enough so that it compensates for the larger variance.

In this thesis we focus on applications that behave similarly as applications typically analyzed using dataflow analysis techniques, but are run on high-performance COTS CMP hardware instead, which typically has a low degree of reproducibility. The techniques presented in this thesis can be used to increase the reproducibility of the execution times of soft real-time image processing applications on high-performance COTS CMP systems. Furthermore, the techniques that increase the timing reproducibility also have as side effect that the latency of the computation is decreased in most cases. In essence, we want to reduce the mean and the variance of the latency distribution as shown in Figure 1.3a and Figure 1.3b.

Another relevant observation is that software design methods typically focus on a systematic decomposition of a system into components that are developed in isolation in order to reduce the design complexity. However, most of the *typical* software design methods do not take temporal behavior into account during decomposition. The only way to derive the temporal behavior of such a system is to first build the complete application after which it is executed onto the system. This implies that the only way to check whether an application meets its real-time requirements is to program the system and to check the temporal behavior of the application against the real-time requirements.

On the other side of the spectrum we have design methods for (hard) real-time systems, for example [11], of which the correctness of the application depends on adhering to the (strict) real-time constraints and where the design process is built around a (formally defined) temporal model and (predictable) hardware that supports modeling, verifying and running hard real-time applications. In such a design process it is typical that the real-time constraints can be verified before programming the complete system since it can be proven that the application, as abstracted by a model, will meet its real-time constraints when executed on the system. This typically involves special hardware architectures where the temporal behavior of the application can be predicted. This is in contrast with the methods discussed previously, where this design problem is typically not possible due to the unpredictability of the commodity CMP hardware. However, due to the special hardware and conservative modeling, a hard real-time system is typically over-provisioned, which for hard real-time application can be justified because violating the real-time constraints is considered catastrophic.

For the real-time systems without hard constraints the cost of such special architectures cannot always be economically justified and may therefore be developed on commodity CMP. However, the combination of real-time applications and commodity CMP is not an area where sufficiently reliable models are available and this results in design methods where the temporal modeling is not properly taken into account, which results in iterative design methods where the real-time constraints are verified afterwards.

All approaches that do not make use of temporal analysis models will likely have many trial and error iterations. Without temporal modeling the designer of a system cannot properly reason about how changes to the design, e.g. decomposition, parallelization, pipelining, etc, change the temporal behavior of the system. The designer then either has to blindly traverse the design space, or rely on past experience to prune parts of the

design space to come to a system that satisfies its requirements.

Furthermore, for approaches that do not make use of temporal analysis models, it will also likely be the case that the specification of the real-time requirements is ambiguous because the basic definitions of the specification of the design, the temporal constraints of the complete design, components, modules, interfaces, etc, are not formally defined. This leads to informal temporal specifications, which by definition is troublesome for a real-time system where the correctness of the program depends on adhering to such temporal constraints.

Lastly, we will discuss some interesting work that has been published after the work for this thesis has been performed.

Lo *et al* introduce a framework, Heracles, that can colocate latency critical tasks with best effort tasks [31]. The temporal requirements of these latency critical tasks may be compared to our soft real-time requirements. The framework is able to determine what sort of best effort tasks may be combined on the same system as the latency critical tasks based on measured performance characteristics. These performance characteristics are measured similarly as we measure the memory bandwidth in the cache hierarchy in Chapter 4. However, instead of using these measurements to select tasks that do not interfere with each other we throttle the best effort application in order to reduce interference.

Novaković *et al* introduce similar techniques but for reducing the interference between virtualized environments [36].

Pellizzoni et al also focus on the interference that is caused by bottlenecks in the memory hierarchy but propose a more fine grained scheduling solution [38]. They propose to use memory servers that control how much bandwidth a group of tasks is allowed to use. Furthermore, they show how to use these memory servers to schedule tasks on the system.

1.3 Problem statement

The problem addressed in this thesis is the development of a systematic analysis and optimization approach for soft real-time medical image processing applications on commodity CMP systems.

This approach should help to pinpoint areas/components in the system that could cause a violation of throughput, latency and jitter requirements.

Furthermore, the approach should enable the optimization of the system such that the throughput, latency and jitter constraints are less often violated.

The first aspect that we determine is which hardware components have a significant effect on the reproducibility of the processing time of image processing applications on chip multi processor systems. More specifically, we determine how the use of these hardware components influences the reproducibility of the processing time. Besides hardware related reproducibility issues we also have to determine the software components that contribute significantly to a low reproducibility of image processing applications.

Given that we know the components that introduce uncertainty we have to define techniques that can increase the reproducibility. First we do this for a hardware platform on which only the real-time application is running and later we introduce techniques to allow additional non real-time applications to run simultaneously on the hardware platform.

Besides increasing the reproducibility and optimizing the processing times of image processing systems we would also like to analyze the performance characteristics of complete systems. This also includes the components that capture the X-ray image, transporting it to the image processing platform and finally is displaying the result to the physician. Given that we have a system that makes use of so called time triggered components we had to introduce a model that can capture the temporal characteristics of such a system.

These three steps combined will lead to image processing software on commodity CMP hardware with reproducible execution times and with the help of the model allows us to more efficiently traverse the design space.

1.4 Contribution

In this section we list the key results of the work described in this thesis.

Contribution 1 Identification of hardware and software components that contribute to the uncertainty of the performance of static image processing applications on commodity CMP systems that result in poor reproducibility.

Contribution 2 Techniques to improve the reproducibility of a static real-time streaming image processing application on commodity CMP systems by scheduling real-time tasks and allocating memory efficiently. The approach makes use of dataflow models and theory.

Contribution 3 An implementation of the approach mentioned in contribution 2 in a compiler that can transform a high-level description of a static streaming image processing application into a realization with highly reproducible timing.

Contribution 4 A technique to reduce the interference when running multiple applications on a single commodity CMP system by throttling tasks in order to reduce interference due to memory bandwidth contention. A reduction of the interference is achieved by extending the scheduler of the Linux kernel with a budget enforcement mechanism.

Contribution 5 A model that can be applied to reduce the latency of a stream processing system by adapting the phase difference between different clock signals of these time-triggered components. We propose an efficient probabilistic analysis approach for the derivation of the end-to-end latency distribution of real-time stream processing systems, that consist of subsystems with time triggered interfaces.

Contribution 6 Implementation of the proposed analysis algorithms from contribution 5 in tooling. The approaches mentioned in contributions 2 to 4 allow us to design and implement a system that has in many practical cases an acceptable temporal behavior so that it can be used as a soft real-time system.

1.5 Outline

The outline of this thesis is as follows. In Chapter 2 we will first describe the hardware and software components that are commonly used in a commodity CMP system. For each of those components we will discuss how they influence the uncertainty of the execution times of the tasks and thus the reproducibility of the systems. Based on the components that are introduced in Chapter 2 we present analysis techniques in Chapter 3 that can be used to reduce the introduced uncertainty of a COTS system on which we only run the real-time image processing application.

In Chapter 4 we relax the assumption that we run only one application on a CMP system and allow other applications to run besides the real-time

application. We assume that we can partition the CMP in such a way that the real-time application can run on a subset of the available cores. In this way the memory hierarchy becomes the most important component that is shared between applications. We investigate a technique that allows a system to determine how much bandwidth the best-effort application uses in the memory hierarchy and we adapt the speed of the best-effort application in order to reduce its bandwidth usage. This takes care that the real time application is never cache bandwidth limited. The presented technique requires that we know the bandwidth usage of the real-time application, the required CPU time while running 'in isolation', and the amount of slack (difference between actual execution time and deadline) of the real-time application.

In Chapter 5 the optimization of the latency and jitter of systems that consist of several time-triggered components by adapting their clock signals is presented. In order to derive which clock signals need modification of their phase differences we introduce a model that can be used to analyze the latency introduced by the complete system. In comparison with custom hardware we are constrained to components such as Graphical Processing Units (GPUs), which are time-triggered. These components with time-triggered interfaces make it impossible to use functionally deterministic modeling techniques from the real-time community such as SDF, etc, because time triggered systems do not adhere to the semantics of these functionally deterministic models.

The conclusions are presented in the last chapter. We state in this chapter under which conditions the concepts introduced in this thesis make commodity CMPs a viable architecture for soft real-time image processing applications.

CHAPTER 2

Sources of Jitter

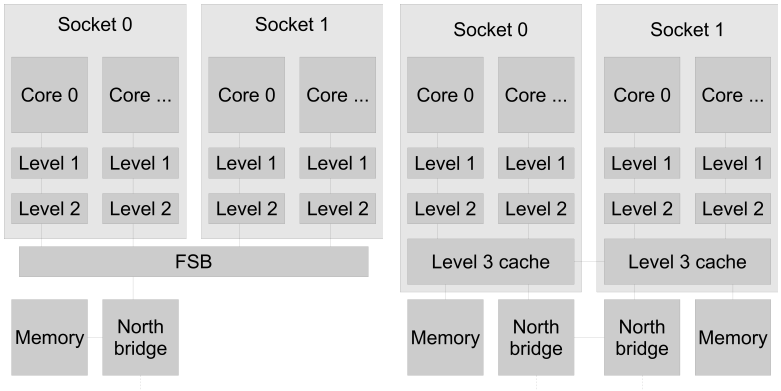
Abstract – *In this chapter we will examine the sources of jitter of real-time stream processing applications that are executed on a commodity CMP system. These sources of jitter decrease the reproducibility of the execution times. In particular we examine the potential influence of each jitter source for our medical X-ray image use case. Many of the results and insights will most likely also hold for many other stream processing applications.*

In this chapter we discuss the hardware and software components that introduce uncertainty in the latency and we estimate how much influence each source of jitter has on the end-to-end latency of a medical image processing application.

2.1 Hardware

The systems that we consider in this thesis are modern commodity CMP systems. Although all experiments in this thesis have been performed on processors from Intel, the result should be comparable on similar processors from AMD. The hardware architecture that is considered is shown in Figure 2.1b. Some, mainly older CMPs do not share the last level of the cache in each package, as shown in Figure 2.1a. This reduces bandwidth and increases unpredictability due to the cache hierarchy. Furthermore, the memory controller was not embedded in the processor, but in the

This chapter is based on [MW:1] , [MW:2] and [MW:3] .



(a) System architecture without shared caches and single memory controller.

(b) System architecture with multiple shared caches and multiple memory controllers.

Figure 2.1: Typical CMP architectures

north bridge. So even in the case that there are multiple memory controllers, the connection to the memory controller, the Front-Side Bus (FSB), is shared. This further increases congestion and decreases reproducibility. Since these architectures are now relatively uncommon, we do not consider these in this thesis.

In the following subsections we consider the following hardware features that lead to unreproducible behavior: functional units, caches, buses, Simultaneous Multi-Threading (SMT), Translation Lookaside Buffer (TLB) cache, dynamic frequency scaling and dynamic overclocking.

2.1.1 FUNCTIONAL UNITS

When an instruction is issued for execution it is placed in one of the execution engines, that can perform the specific instruction. These execution engines are deeply pipelined and because of the out-of-order execution of the instructions the latency between the start of an instruction and the end of it depends on several factors like the current status of the pipeline, data dependencies, etc. It is therefore not always feasible [55] to give accurate upper bounds on the execution times of each individual instruction. However, we are only interested in the execution of large numbers of instructions and therefore assume that the effects of the pipeline averages out.

2.1.2 CACHES

Due to the difference of the clock frequency between the processor and main memory a cache hierarchy is used. The cache hierarchy of typical CMPs can be split into a local cache hierarchy and a shared cache hierarchy.

For example, the cache hierarchy of the Nehalem microarchitecture consists of three levels. The last level of the cache – the level directly connected to the main memory – is shared between all the cores on the die [15]. When the accessed data by a core is only available locally (e.g. in a register, the first or second level of the cache) the latency of the access is not influenced by other cores and only depends on where it is available locally. The access to the data that is stored in the third level of the cache could be influenced by other cores if the total bandwidth to the third level of the cache is saturated [32], but due to the large size of the second level of the cache and the locality of reference of most streaming applications this is usually not the case. We therefore assume that accessing data that is available in the local cache hierarchy introduces neglectable jitter. When this is not the case some jitter will be introduced, because data has be loaded from main memory over a shared connection or has to be retrieved from another part of the cache hierarchy.

Reducing the communication between the cache and main memory will therefore mitigate some of the temporal effects of the cache.

Another technique for reducing the communication between the cache and main memory is preventing cache evictions of old data. Such cache evictions can be prevented by reusing the memory locations where old data (i.e. data that was used, but is never accessed again) resides for new data. The computation has to be scheduled in an order that would reuse the memory locations of old data before it is evicted from the cache.

Some jitter could also be introduced when data has to be retrieved from non local parts of the cache hierarchy (e.g. from the level 2 cache from another core). A reduction of non-local cache access would therefore reduce the amount of jitter that is introduced by this kind of access.

2.1.3 BUSES

The use of a single bus has been replaced by a interconnect called QuickPath [14] in the Nehalem architecture, and later in the Skylake architecture with the Ultra Path interconnect [34]. Where in a typical system that employs a single bus all cores can influence each other, the QuickPath in-

terconnect limits the influence to cores that share a QuickPath connection. In a multi-die system the communication between the level 3 caches of the cache hierarchy is routed through the QuickPath interconnect. However, in this thesis we focus on the sources of jitter that originate within a single die and assume that effects introduced by the QuickPath or Ultra Path interconnect can be neglected.

2.1.4 SIMULTANEOUS MULTI THREADING

With Simultaneous Multi-Threading (SMT) [26, 49], multiple threads use the same execution engines. This is done in order to increase the utilization of the execution engines of the processor. Multiple threads (two on the Nehalem micro architecture) issue instructions to the same execution engines. Those instructions essentially compete for the same instruction slots, but since a thread rarely uses all execution engines, issuing instructions from two threads will increase the total overall instructions issued per second. Even though the actual throughput of a thread will decrease dramatically, it will be active longer or more often since the OS does have more logical cores to schedule active threads to. Furthermore, SMT will also hide some of the latency due to cache misses since the non stalling thread can than issue more instructions into the engines. Hence the usage of SMT might decrease the jitter, which is beneficial for soft real-time applications. However, the streaming applications might have to instantiate more threads to take advantage of SMT, which could lead to additional synchronization jitter. The applicability of SMT therefore depends on the balance between the jitter reduction and the additional synchronization overhead.

2.1.5 TRANSLATION LOOKASIDE BUFFER

The Translation Lookaside Buffer (TLB) is used as a cache for the Memory Management Unit (MMU) that maps virtual addresses to physical addresses. On the Core 2 architecture from Intel the TLB could not hold enough entries to completely translate enough addresses to cover the last level of the cache. The newer Nehalem microarchitecture can still only cache 512 entries for 4 KB pages in its second level TLB cache per core. This implies that it can only store paging information for up to 2 MB. Since our streaming application alone will already access more than 2 MB data we could instantiate huge pages (of 2 or 4 MB each) so that our application does not thrash the TLB cache. However, due the high locality of reference of our application the introduced jitter by the TLB will be small.

2.1.6 DYNAMIC FREQUENCY SCALING AND DYNAMIC OVERCLOCKING

The clock frequency of a core in the Nehalem architecture can be scaled dynamically in order to reduce energy usage. Running a core on different clock frequencies introduces jitter. This technique is therefore disabled.

An additional technique (Turbo Boost [16, 32]) can dynamically overclock the base clock frequency of a chip, when certain conditions and thresholds (e.g. temperature) are not violated. Depending on how many cores are active the system may increase the base clock frequency of the chip further than specified as long as the temperature is low enough and the complete chip does not use too much energy. Since an increase in frequency will increase energy consumption it cannot apply Turbo Boost, when the chip is already consuming the maximum specified amount of energy. Furthermore, when the chip is already too warm, it will not activate Turbo Boost since that will also increase the temperature. In case the processor exceeds another critical thermal threshold it will even throttle the entire processor to ensure that the processor does not become unstable or is damaged by the high temperature.

It is therefore important to ensure that the chip (and the system) is adequately cooled and that the actual running state is monitored so that, if the system does throttle the base clock frequency due to a cooling failure, the application can respond accordingly. Due to the activation condition of Turbo Boost it will inherently introduce more jitter and reduce reproducibility. However, increasing the clock frequency will only decrease latency (assuming no performance anomalies). In [7] the Turbo Boost technique has been examined in detail. Charles *et al* concluded that the performance never decreased [7]. Since the performance would only increase with the Turbo Boost technique we could enable the technique and instantiate a buffer in order to make the iterations with a too low latency later available for consumption. This will reduce the number of iterations of which the results are produced too late. However, given our definition of reproducibility, the reproducibility would decrease since the execution times are not decreased uniformly since the Turbo Boost activation criteria might not always be met. So during measurements of the execution times it is necessary to disable Turbo Boost.

2.1.7 TIME TRIGGERED INTERFACES

Medical image processing systems such as interventional X-ray systems are nowadays often composed of a number of independent subsystems that have time triggered interfaces. Examples of these subsystems are the image sensor, the image enhancement general purpose computer, a GPU

for image composition and GUI, and the video wall for the combination of several displays.

In a system as shown in Figure 1.2 there are two GPUs in the complete processing chain. GPUs have time triggered interfaces. Furthermore, the CCD that captures the X-ray image also has a time triggered interface. In this system these interfaces are not synchronized with each other. This introduces another source of uncertainty in the end-to-end latency.

2.2 Software

2.2.1 operating system

An Operating System (OS) can have a significant influence on the jitter of an application because it is responsible for thread scheduling. Because the operating system decides where and when to execute threads, it is important to map the data in such a way that it is likely that a thread is executed on a core, where the data is already available. We want to achieve this by replacing the OS scheduler by statically mapping the computations to a limited number of threads. So that the operating system can schedule these threads efficiently and to reduce data movement between the local caches of the cores.

Furthermore, we have applied real-time scheduling patches from [33] to the Linux Kernel in order to achieve real-time scheduling possibilities. Also, the OS is responsible for swapping out memory pages when all physical pages are allocated and when a page fault occurs. Pages that are allocated to real-time applications should be locked in order to prevent that they are swapped out. All major OSs make it possible to lock pages. In our use case the data that was used by the application could always be completely loaded in main memory, so we did not experiment with locked and unlocked pages.

2.2.2 internal structure

The internal structure of the application has a large impact on the performance characteristics of the streaming application. A structure might hide or alleviate the impact of various latencies in a streaming application. For example: a simple method for introducing parallelism in an application is the fork join method where the computation that is performed inside a loop is parallelized.

OpenMP [9] is an Application Programming Interface (API) that is developed to easily implement this kind of parallelism. In Figure 2.2 a example

```
...
main(){
  init();
  #pragma omp parallel for
  for(int i=0; i<MAX; i++)
    doTask(i);
  deinit();
}
```

Figure 2.2: OpenMP Example where some data parallelism is introduced.

is shown that parallelizes a small loop. The loop is parallelized with the fork/join pattern (as shown in Figure 2.3), where a thread is created for each iteration in the loop and which are joined before the execution of the main thread can continue. The fork/join pattern waits for the slowest thread, and thus increases the probability that the end-to-end latency of the application is affected negatively (i.e. it decreases reproducibility). Therefore these patterns need to be used with care.

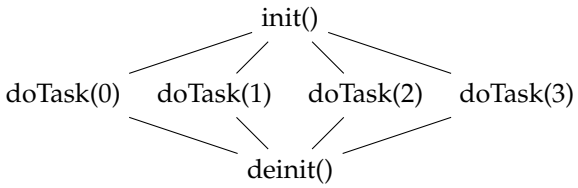


Figure 2.3: Task graph of Figure 2.2

2.3 Conclusion

This chapter presented some of the components in COTS CMP that introduce jitter sources. The following chapters examine how much jitter is introduced and how the hardware can be used in a way that mitigates those sources of jitter.

CHAPTER 3

Jitter reduction on CMP-systems

Abstract – *The real-time system research community has paid a lot of attention to the design of safety critical hard real-time systems for which the use of non-standard hardware and operating systems can be justified. However, stream processing applications like medical imaging systems are often not considered safety critical enough to justify the use of hard real-time techniques that would increase the cost of these systems significantly. Instead COTS hardware and OSes are used, and techniques at the application level are employed to improve the reproducibility (i.e. reduce the variance) of the end-to-end latency of these imaging processing systems.*

In this chapter, we study the effectiveness of a number of scheduling heuristics that are intended to reduce the latency and the jitter of stream processing applications that are executed on COTS multiprocessor systems. The proposed scheduling heuristics take the execution times of tasks into account as well as dependencies between the tasks, the data structures accessed by the tasks, and the memory hierarchy.

Experiments have been carried out on a quad core Symmetric Multiprocessing (SMP) Intel processor. These experiments show that the proposed heuristics can reduce the end-to-end latency with almost 60%, and reduce the variation in the latency with more than 90% when compared with a naive scheduling heuristic that does not consider execution times, dependencies and the memory hierarchy. The increased reproducibility makes it possible to efficiently design soft real-time image processing on COTS hardware.

Parts of this chapter are based on [MW:1] .

3.1 Introduction

This chapter presents techniques to improve the reproducibility when a single application is running on a COTS CMP system.

COTS CMP systems are nowadays often used for real-time (medical) image processing applications, of which an interventional X-ray application is an illustrative example. With an interventional X-ray system, a physician makes use of images captured with an X-ray imaging device to perform delicate medical procedures inside a patient, where the only visual feedback is provided by the images captured by the X-ray device. It is therefore desirable that the latency between the capturing of an image and displaying it is short enough (< 200 ms) to provide sufficient eye-hand coordination. Furthermore, the variation of the latency, which is called jitter, must be sufficiently low such that the physician experiences a constant delay, which improves the eye-hand coordination and prevents fatigue.

Due to low radiation limits advanced image processing is necessary to obtain sufficient image quality. In an interventional X-ray application, only a fraction of the latency budget is available for image processing due to the latency that the detector and display introduce. Therefore, the image processing used to be performed on architectures like ASICs, DSPs and FPGAs. However, high performance SMP COTS hardware has become performance-wise so powerful and cost-effective, that the trend is to perform the processing on this type of hardware despite the increased temporal uncertainty that this hardware may introduce. The use of COTS hardware seems to be acceptable as long as temporal constraints are rarely violated. Therefore, it is a valid approach to use heuristics for these systems during the design process, after which the systems performance is validated by means of extensive testing. The effort it takes to validate a system is greatly reduced when the execution times of the application is reproducible (i.e. when the jitter is low).

In this chapter we present a number of scheduling heuristics that are intended to reduce the latency as well as the jitter of streaming applications, such as the interventional X-ray application described above. We implemented these heuristics in a tool flow that can synthesize an application from a high level description of an image processing chain. The high level description of the image processing chain is static, i.e., there is no dynamic functional behavior. The tool flow was used to evaluate the scheduling heuristics on one image processing chain from the interventional X-ray application and on a set of synthetically generated image

processing chains. The synthesized applications were executed on COTS hardware with Intel Nehalem Central Processing Units (CPUs).

This chapter is structured as follows. First we discuss related work in Section 3.2, after which we recap in Section 3.3 which of the components that were presented in Chapter 2 are examined in this chapter. In Section 3.4 we present how our tool flow synthesizes a high level description of an image processing chain into an application. The experiments are described in Section 3.5. The results of the experimental evaluation can be found in Section 3.6. Finally we discuss the conclusions in Section 3.7.

3.2 Related Work

In [55], Wilhelm et al. discuss the components in an embedded system that affect the tightness of the computed worst-case execution times bounds by means of static timing analysis. The authors conclude that static timing analysis of systems with shared caches is very complex and that the computed bounds are often not tight. As our objective is to improve the typical behavior instead of the worst-case behavior of an application, we do not need to use formal timing analysis to derive the worst case behavior. Instead we measure the execution times and employ techniques to use the architecture in a way that reduces jitter.

Extensive measurements on a similar multiprocessor system as we consider in this chapter, are presented by Molka et al in [32]. However, only results are presented for a synthetic benchmark set, while we study the behavior of complete applications, which may provide other insights than a set of synthetic benchmarks.

An approach for improving the temporal behavior of a multiprocessor system with a shared cache by means of locking of cache lines, has been presented by Suhendra et al. in [43]. For the machine we consider in this chapter, this approach is not applicable because cache line locking is not supported.

In [2, 22], Anderson et al. and Kim et al. analyze the influence of thread scheduling on the behavior of the cache. However, the focus of the paper is mainly on the interaction of different applications, while we focus on the case that only one application is executed on the system.

In [6, 40, 56] Chakraborty et al, Schlieker et al. and Yan et al. introduce analysis methods to take the effect of caches and shared resources into account. However, these papers consider either the case of a single pro-

cessor system without a shared cache, or consider systems in which only the instruction cache is shared.

In [1], Albers et al. use another model for the mapping and partitioning of computation to threads, but the scheduling order of the application is not taken into account. Furthermore, the focus is on the reduction of latency and not primarily on the reduction of jitter.

3.3 Sources of jitter

In this chapter we examine the influence of hardware and software components that typically have a large influence on the jitter and latency. We focus on several components in the processor, such as the functional units, the caches, buses. Furthermore we examine the influence of techniques such as SMT and dynamic frequency scaling and dynamic overclocking. These components have been introduced and described in Section 2.1.

The software parts that typically influence the jitter and latency that we examine have been introduced in Section 2.2.

3.4 Tool flow

In this section we describe our tool flow that we use to synthesize an application from a high level description of the image processing chain. A detailed overview of all the options that have been implemented in the tool can be found in Appendix A. The tool flow and associated high level description language are designed in such a way that they can vary the usage of the two components that influence the jitter most, namely, the cache hierarchy and the OS. Our tool flow takes a high level description that only describes functional level parallelism and will then perform the following steps:

a. Introduction of data parallelism
b. Scheduling computational steps to threads
c. Allocate memory
d. Introduce synchronization
e. Generate code

In the following paragraphs we describe the steps in our tool flow in more detail. The tool flow takes as input a high level description in

which the functional behavior (e.g. code, functions, etc) is encapsulated in a *box* and boxes are connected together to describe the structure of the image processing chain. We will refer to this description as the structural description. Hence, the structural description exposes the functional level parallelism. Each box has a number of associated input and output ports that can be used to connect boxes together. A connection between an output and input port is associated with a memory buffer in order to store the data between the execution of the connected boxes. The applications that we describe with our high level description are *streaming* applications. In our description it means that the source boxes (i.e. boxes without inputs) are triggered periodically or are triggered at some external event (i.e. arrival of input data). Each execution of an (sub) box takes places in an *iteration*. An iteration would typically result in some output, for example: a video frame. Depending on the scheduling and mapping of the application it is possible that (sub) boxes from multiple iterations are executing at the same time. In this context we also define the *current* iteration as the *oldest* iterations that still has (sub) boxes to execute. See Figure 3.1 for an example where the image processing chain first applies a gain filter and secondly a convolution on the image that is produced by the source. Each edge that connects two boxes together represents a memory buffer.

3.4.1 INTRODUCTION OF DATA PARALLELISM

The structural description, that only contains functional level parallelism, is transformed into another description, which we call the instantiated description. This description also incorporates data level parallelisms, where the tool flow has instantiated data parallelism by splitting the boxes into *sub-boxes*. Under most circumstances, which we do not elaborate, it is not necessary to introduce more data parallelism than processors available in the hardware platform. Our tool therefore splits each box into as many sub-boxes as there are processors. The box is annotated with additional information that is used by the compiler to split the box into sub-boxes.

Each sub-box performs a part of the computation from the original box where it was instantiated from; the tool annotates each sub-box with the part of the computation that it has to perform. In our structural description we have also annotated each box with additional information that can be used to derive fine grained dependencies between sub-boxes. Without this information we would have to instantiate dependencies between all sub-boxes of subsequent boxes and this would limit the freedom during the scheduling step and thereby would introduce unnecessary

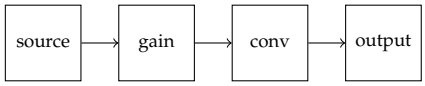


Figure 3.1: Structural description of a image processing application.

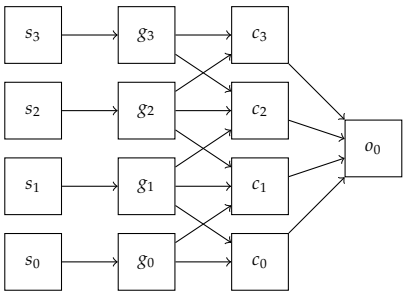


Figure 3.2: Instantiated description of Figure 3.1 which exposes data parallelism and fine-grained dependencies.

synchronization.

See Figure 3.2 for an example of how the tool flow transforms the structural description from Figure 3.1 and derives the fine grained dependencies. This implies that besides the high level description, also the structural description is statically defined at design time. In this example, there is a gain filter, where each pixel only depends on one pixel and therefore needs the minimal amount of dependencies. This is in contrast to the convolution filter, where each pixel depends on a region of pixels, and each sub-box of the convolution therefore depends on multiple sub-boxes of the gain filter. Lastly, we can see that our tool could not split the output box into multiple sub-boxes because the implementation of that box could not be parallelized.

3.4.2 SCHEDULING COMPUTATIONAL STEPS TO THREADS

At this step, we have a description of our image processing chain with data and functional level parallelism, and fine grained dependencies. We can now schedule and order the sub-boxes of this description to threads. In our tool we implemented several scheduling heuristics so that we can evaluate the influence of each scheduling heuristic on latency and jitter.

One-to-One

The One-to-One is the simplest scheduling heuristics where each sub-box is given its own thread. We will refer to this mapping method as

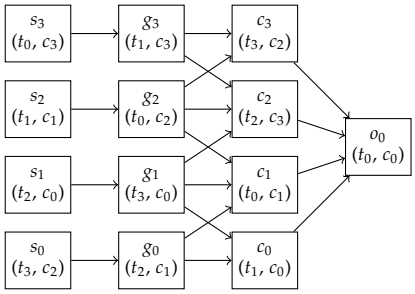


Figure 3.3: Instantiated description of Figure 3.2 with applied thread scheduling technique many-to-one. Subsequent tasks are not necessarily mapped onto the same thread. This leads to additional cache thrashing and unnecessary thread synchronization.

the *simple* method. In this method it is possible that some sub-boxes of the subsequent iteration execute before the end of a complete iteration, because there is no synchronization between iterations.

One-to-One without pipelining

Pipelining can significantly increase the amount of sub-boxes that can execute in parallel and, however, because the OS does not have a notion of which sub-boxes belong to the current iteration it may execute sub-boxes of subsequent iterations and thereby increase the latency. Pipelining over iterations can be prevented by adding a barrier between the last sub-box(es) of the current iteration and the first sub-box(es) of the next iteration. This method will be called the *barrier* method.

Many-to-One

The Many-to-One scheduling technique schedules and orders all sub-boxes to a configurable amount of threads, we call it the *fixed* method. For each processor that is available for the execution of the application one thread is instantiated. Each of these threads can be fixed to a specific core or to a subset of cores that share a cache level in order to prevent the OS from moving the thread and thereby trashing the cache. The scheduling and ordering is performed by first constructing a Homogeneous Synchronous Dataflow Graph (HSDFG) [30] that models the temporal behavior of the application. For each sub-box in the application an actor will be instantiated in the HSDFG. The dependencies are translated into the edges of the HSDFG. This HSDFG graph is used to construct a static schedule for each thread. See Figure 3.3 for an example mapping of Figure 3.2. The thread and core mapping are represented by the tuple, where t repre-

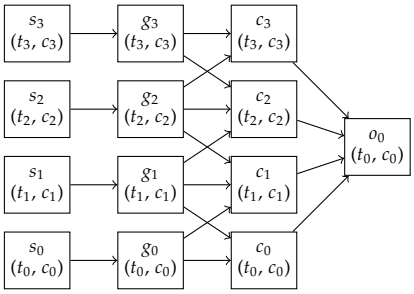


Figure 3.4: Instantiated description of Figure 3.2 with applied thread scheduling technique cache aware many-to-one. Subsequent tasks are mapped onto the same thread. This leads to better cache usage and reduced thread synchronization.

sents the thread and c the core. Note that subsequent sub-boxes (where the output of the first sub-box is used by the second) do not necessarily map onto the same thread and core.

The advantage of a static schedule is that the application controls the order in which the boxes are executed instead of the scheduler of the operating system. A disadvantage is that this technique does not take into account the state of the cache, which might result in a lot of cache trashing.

Many-to-One cache aware

This scheduling technique works almost in the same way as the fixed method, but it tries to schedule boxes to a thread taking the state of the cache into account in order to reduce cache misses. During the scheduling of sub-boxes this technique gives priority to sub-boxes of which the input data is most likely to be in the cache. We will refer to this mapping method as the *predictable* method. Figure 3.4 gives an example mapping of Figure 3.2. In this example subsequent sub-boxes are mapped on the same thread and core as much as possible.

Many-to-One cache aware reduced

Furthermore, a heuristic can be used to reduce the amount of active data (i.e. data that will be used in this or subsequent iterations) throughout an iteration (and thus the streaming application) that was generated using the predictable method. A Static Order Schedule (SOS) is constructed of the structural graph where actors of which execution results in the least amount of active data to be stored in memory are scheduled first. Then

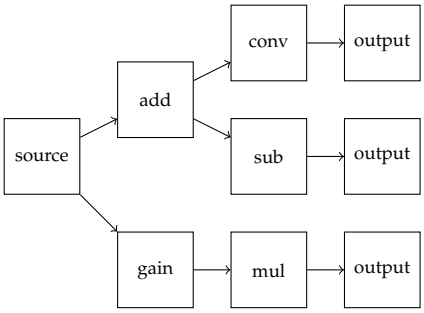


Figure 3.5: Structural description for larger streaming image processing application.

a back-tracking algorithm is used to check whether some choices of actor ordering would have resulted in a schedule with a smaller amount of active data during the execution of one iteration. Because the exponential complexity of this back-tracking algorithm, it is stopped, when a specific amount of tokens is reached or when it takes too long to explore the complete state space to find the optimal solution. This method will be referred to as the *reduced* method. The overhead of generating the SOS is only during design time and requires no additional computation during run-time.

The reduced method does not have any advantages on the example given in Figure 3.2, because there is basically only one linear mapping possible. Figure 3.5 shows a structural description where this method would give advantage. The reduce method would first complete process the lower boxes (*gain, mul, output*) before executing the add box in order to minimize the maximum amount of active data and increase locality of reference.

3.4.3 ALLOCATE MEMORY

When the thread mapping has completed, the memory allocation for the application can be computed. We have examined two memory allocation methods.

Simple

The *simple* memory allocation scheme allocates a separate memory range for each memory buffer.

Reuse

The *reuse* memory allocation schemes tries to reuse memory buffers from actors that have already finished their execution (within an iteration). First, an *interference graph* [5] is derived from the thread schedules. Secondly, a first fit heuristic is used to allocate memory for all memory buffers. The heuristic collects deallocated memory regions in a list and reallocates the first region with the correct size when a new allocation is performed.

3.4.4 GENERATE CODE

The final step of the application synthesis is the code generation. Four important pieces of code will be generated that are used to construct the complete application. Firstly, the *initialization code* will be generated. During the initialization memory will be allocated, synchronization primitives (e.g. barriers) will be created and configured and other data structures that configure the application will be configured. Secondly, the *thread code* will be generated. In each thread the call to the sub-boxes and synchronization statements are inserted. Thirdly, the *code* is generated that is responsible for starting the execution of all threads. Lastly, the *cleanup code* is generated that stops the threads, deallocates memory and cleans up all the used resources.

3.5 EXPERIMENTS

In this section, we present the experiments that we have run for the evaluation of the described techniques. Firstly, we describe the platform that we executed the experiments on in Section 3.5.1. Secondly, the applications that have been used as input for the experiments are elaborated in Section 3.5.2. Thirdly, we define the experiments that we have run in Section 3.5.3.

3.5.1 EXPERIMENTAL SETUP

The experiments were performed on a quad-core Core i7 860 from Intel, see Figure 3.6 for graphical overview. The four (hyper-threaded) cores share the third level of the cache that has a size of 8 MiB. A minimal Ubuntu installation [50] with the 2.6.32 Linux kernel was used as operating system. The system was running without a graphical user interface and unnecessary services were shutdown. Furthermore, the processors were run at 2.8 GHz with TurboBoost disabled.

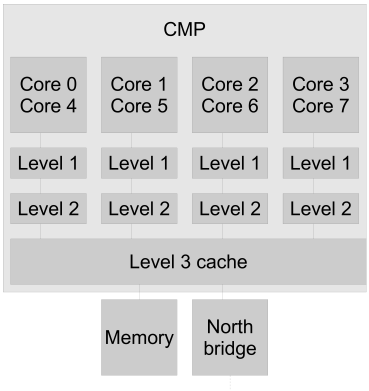


Figure 3.6: System architecture for experimental setup

In each experiment we measured the end-to-end latency of each iteration by collecting time stamps before and after its execution. The latency and jitter (variance) were calculated using these latencies. Since the instantiated and mapped descriptions of the applications, and the algorithms in the boxes are completely static (i.e. not data dependent) the variance in the measurements are completely introduced by the OS and the hardware.

3.5.2 EXPERIMENTAL INPUT

Figure 3.7 shows the topology of the structural description of an image processing chain from the interventional X-ray application. Due to the limited amount of image processing chains in the real-life interventional X-ray system we chose to generate additional artificial image processing chains. The additional image processing chains were generated in order to verify the proposed scheduling techniques work on different topologies. A tool was created that could generate random graphs that have similar characteristics as the actual image processing chains in the interventional X-ray system. Each graph was created with roughly 100 boxes with a topology that resembles the topology of the actual image processing chain. Furthermore, the number of input and output boxes was chosen to match with some of the interventional X-ray scenarios. After the graphs were generated each box was associated with a random image processing algorithm such as: averaging, addition and convolution. This results in image processing chains that do not produce any useful output, but still use the hardware in a realistic way. In total 250 image processing chains were generated.

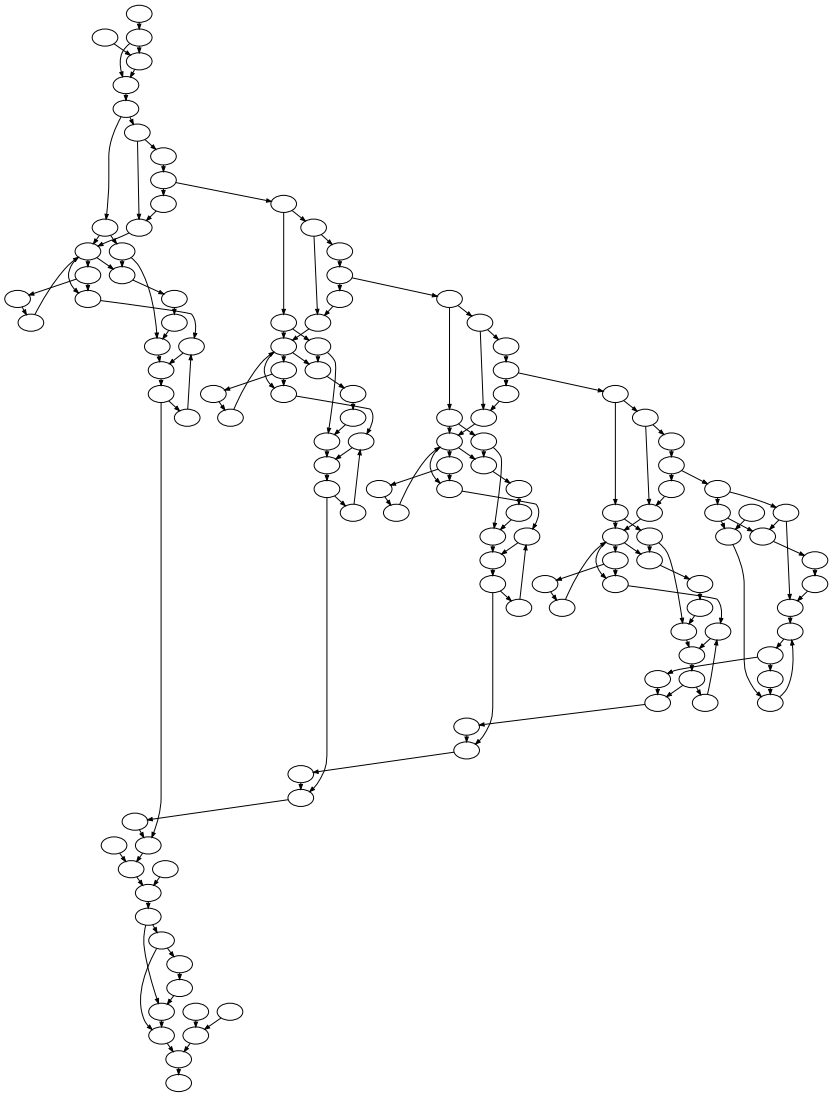


Figure 3.7: Topology of the structural description for the X-ray image processing chain

3.5.3 EXPERIMENTS

In each experiment the application was run for 100.000 iterations. As mentioned before, the boxes implement real algorithms and thus use the hardware in a realistic way. The boxes run on actual memory, as allocated by the memory allocation algorithms.

Execution on four physical cores

In this experiment we applied all the scheduling techniques and, where applicable, both memory allocation techniques to all the input applications. For the *fixed*, *predictable* and *reduced* techniques we instantiated four threads so that each of those threads could be mapped to one physical core (i.e. the affinity of the thread was reduced to one physical core). Hence, our application did not use SMT but we did not disable SMT altogether, so that the OS could still perform computation on the empty virtual core.

Execution on eight physical cores

For this experiment we only applied the reduced technique with both memory allocation techniques on one of the synthetic image processing chains. Furthermore we applied the techniques three times. We instantiated four, eight and sixteen threads respectively in each test. Each thread was mapped onto a single logical core.

3.6 RESULTS

3.6.1 EXECUTION ON FOUR PHYSICAL CORES

In Table 3.1 and Table 3.2 detailed results can be found from the real image processing chain and one of the synthetic chains, respectively.

First of all, the simple mapping technique does have a significant variation in latency. This is due to pipelining, as explained in Section 3.4.2. Pipelining is prevented in the barrier technique and we can see that this reduces the jitter significantly. The variance (*sigma*) is used as a measure for the jitter.

The mapping techniques fixed, predictable and reduce result in almost the same jitter regardless of the memory allocation method.

When these heuristics are used in conjunction with the memory reuse technique, a reduction in end-to-end latency is observed. Furthermore,

Thread scheduling	Memory allocation	Memory size (KiB)	$\bar{x}$ (μs)	σ (μs)	1855 μs 5516 μs
simple	naive	30635	49494	3400	
barrier	naive	30635	4940	37	
fixed	naive	30635	5439	29	
predictable	naive	30635	4561	24	
reduced	naive	30635	4117	30	
fixed	reuse	18250	3527	14	
predictable	reuse	14074	2823	18	
reduced	reuse	7486	1871	8	

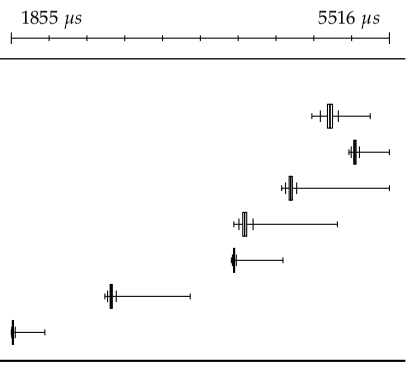


Table 3.1: results for X-ray image processing chain. Note that the boxplot for the first result has been omited in order to see more details in the other rows. In the results we clearly see that there are significant differences between the thread scheduling techniques. Furthermore, when those techniques are combined with a more effiecient memory allocation techniques those differences become even larger.

Thread scheduling	Memory allocation	Memory size (KiB)	$\bar{x}$ (μs)	σ (μs)	4288 μs — 12906 μs
simple	naive	75456	271618	23726	
barrier	naive	75456	12118	207	
fixed	naive	75456	11088	50	
predictable	naive	75456	10965	62	
reduced	naive	75456	10831	57	
fixed	reuse	11520	5271	11	
predictable	reuse	9792	4622	38	
reduced	reuse	5760	4309	25	

Table 3.2: results for synthetic image processing chain. We broadly see the same differences between the different techniques when compared with the results for the actual chain. However, the absolute differences are larger, mainly due to the larger difference is memory usage.

the jitter also reduced significantly. For example, the variance for the predictable thread scheduling technique with the memory reuse technique reduces from 30 μs to 8 μs, a reduction of 73%.

When we compare these techniques with the memory reuse heuristic is used we can see a significant difference between the fixed scheduling heuristic and the predictable and reduced scheduling heuristic. The memory reduction algorithm does not immediately seem to impact the average execution length or jitter, but this is due to the fact that in these cases all the accessed memory will fit in the cache. However, the reduction in memory usage might have additional benefits such as a high hit rate on the second level of the cache and it could be more robust against the cache thrashing that might be the result of other applications running on the same system. These possible benefits have not been explored in this thesis.

We also see a difference in the effectiveness of the techniques between the actual image processing chain and the synthetic one. For example the latency of actual image processing chain with the reduced technique reduces by 55%. The latency of the synthetic image processing chains reduces by 60%. On the actual image processing chain the predictable and reduced techniques do not decrease the latency as much as we observe in the synthetic image processing chains. Although we have not shown the box plots of all the synthetic chains, these observations hold for the complete set of synthetics chains.

In Figure 3.8 we have stacked a compact representation of the box plots of all the executions of the synthetic chains when they were scheduled with the reduced technique and the reuse memory allocation technique was applied. For clarity we have sorted the graphs on the median latency. In the figure we see that the typically observed latency is roughly the same for all the graphs and more importantly that the maximum observed latency is relative to the median. Hence, the techniques are generic and can be applied to different topologies which will result in reduced latency and jitter.

3.6.2 EXECUTION ON EIGHT PHYSICAL CORES

In this experiment SMT was evaluated on the synthetic image processing chains. In Table 3.3 we see that using more logical cores does increase the latency, most likely due to additional synchronization. However, there is no significant change in the jitter. We expected that there would be an increase in jitter because the execution of the threads on logical cores

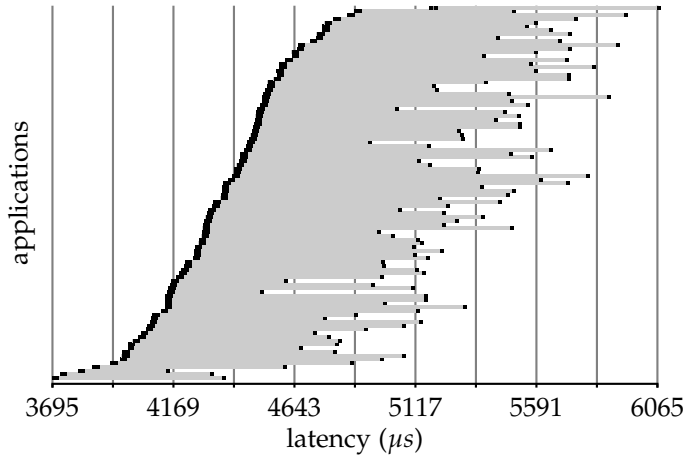


Figure 3.8: Detailed results for run with reduce reuse on 4 cores. Here several synthetically generated applications are mapped with the same thread scheduling technique and memory allocation techniques in order to determine how robust the techniques are.

Threads	$\bar{x}$ (μs)	σ (μs)	4825 μs — 8350 μs
4	4929	38	
8	5452	37	
16	7942	76	

Table 3.3: Hyper threading enabled on 4 cores. We see that in our application hyper threading leads to a latency increase due to a lack of execution engines and more cache trashing.

are highly dependent on each other, but we speculate that since both threads execute similar code they both receive a fair amount of time on the execution engines.

3.7 Conclusions

The scheduling heuristics that we have presented in this chapter are able to greatly improve the reproducibility of the execution times in our synthetic applications. We have shown that the heuristics both decrease jitter (i.e. reduce σ) and decrease the end-to-end latency (i.e. reduce μ) of the execution times. m

As stated in Section 1.2 we defined reproducibility as follows: *An ap-*

plication on a system has a better reproducible behavior when the probability of deadline violations is smaller. Since our thread scheduling and memory allocating techniques reduced both the latency and the jitter this will always reduce the reproducibility, regardless of the required deadline.

From the experiments we have observed that the reproducibility is only bad when our application and OS have maximum scheduling freedom (small grained tasks that can be scheduled on all cores). So by only restricting the scheduling freedom of the OS we can already improve the reproducibility considerably. We think this is a surprising result. We expected that its would be much harder to improve the reproducibility of a CMP COTS hardware. The reason of this result is most likely that the computational operations that we perform in a task, and the relations between the tasks themselves result in relatively stable behavior of the hardware. For example, the algorithms we examined are not data dependent, this results in stable usage of the pipelines, branch predictors, etc. If our experiments would be repeated with computational tasks that do not result in this stable usage of the hardware the result would likely be much different. For example, video compression algorithms are highly dependent on the data they compress. This will result in more random memory accesses, more mis-predicted branches, more pipeline flushes, etc.

We conclude that using CMP COTS hardware for soft real-time medical image processing application is possible. Note that we restricted the class of image processing applications where the the structural description is defined statically and the tasks are not data dependent. Especially when the scheduling freedom of the OS with regard to the computational tasks has been restricted sufficiently.

CHAPTER 4

Reduction of the jitter caused by shared bandwidth in the cache hierarchy

Abstract – *Systems with Chip Multi-Processors are currently used for several applications that have real-time requirements. In Chip Multi-Processor architectures, many hardware resources such as parts of the cache hierarchy are shared between cores. By using such resources, applications can significantly interfere with each other. In the previous chapter, we showed that a single X-ray imaging streaming application can be executed with high reproducibility on such systems. However, it was assumed that only one application would be running on the system. This prevents further system integration where multiple real-time and best-effort applications are executing on a single Chip Multi-Processor.*

In this chapter, we address the limited bandwidth in the cache hierarchy, which can cause threads to interfere with each other significantly. We propose a technique that implements cache bandwidth reservation in software. This is achieved by dynamically duty-cycling best-effort applications, based on their cache bandwidth usages using processor performance counters. Duty-cycling best-effort applications allow us to control the influence of those applications on real-time applications. With this technique we can control the latency increase of real-time applications that is caused by best-effort applications. When the temporal requirements allow for a (slight) latency increase this technique can be used to combine real-time and best-effort applications on the same system with a minimal reduction of best-effort performance. The results

This chapter is based on [MW:2] .

of the experiments with real-life applications indicate that we can control the increase of the latency to such an extent that we can almost completely eliminate the influence of bandwidth sharing in the cache. Hence, the technique increases the reproducibility of the real-time application.

4.1 Introduction

In this previous chapter we have shown that a soft real-time image processing application can be executed on a COTS CMP system. This chapter extends this work by allowing best-effort applications to run beside the soft real-time application. The interference introduced by the best-effort application is managed in order to not violate the temporal requirements of the soft real-time application.

COTS systems with General Purpose Processors (GPPs) have become so powerful that they can be used for applications that could previously only be performed on hardware such as digital signal processors, FPGAs or by ASICs. However, the techniques that are employed by modern GPPs to reach their performance tend to neglect the worst case performance in order to improve the best-case and average-case performance. This seemingly does not make them a favorable target for real-time applications. A lot of work in the real-time community has been focused on the definition of architectures on which real-time applications can run predictably (even when running concurrently with other applications) and modeling techniques that determine the worst case behavior of real-time applications.

Applying these modeling techniques on high performance GPPs often results in loose throughput and latency bounds as a result of the GPPs and CMP architecture. This leads to over-provisioned systems, which cannot be economically justified for most soft real-time applications.

We presented the streaming X-ray application on a CMP architecture in Chapter 1. In the previous chapter, we provided evidence that it is feasible to run streaming applications in such a way that drastically reduces the occurrences of worst case behavior. The jitter, variation of latency, of the executed application is sufficiently small to satisfy the soft real-time requirements of the X-ray application. However, one important assumption was that the system was only used for the X-ray application. A consequence is that a separate system is required for additional applications while the system that runs the X-ray application has spare processing

power most of the time. We therefore will study in this study whether we can relax that assumption, and run some best-effort application in parallel with our real-time streaming application. This in order to retain the highly reproducible behavior of the soft real-time application while allowing best effort application to run concurrently (although with possibly reduced execution speed).

The problem that must be addressed is that threads that execute concurrently on different cores on a CMP architecture can degrade each others performance significantly. There are several components in a CMP that cause cores to degrade each others performance such as the cache hierarchy, memory controller and CPU interconnect. They are discussed in more detail in Section 2.1.

In this chapter, we focus on space contention and bandwidth congestion within the cache hierarchy and study how this causes applications to degrade each others performance and propose and evaluate a technique to control the degradation.

The chapter is structured as follows. First we re-cap the case study, an interventional X-ray imaging system, in Section 4.2. Related work is described in Section 4.3. This allows us to give a detailed explanation of why we study the space contention and bandwidth congestion in CMP architectures in Section 4.4 and Section 4.5. Section 4.6 explains how we implemented a technique that reduces the bandwidth congestion and thereby controls the interference that is caused by the best-effort application. The experiments that we used for the evaluation are presented in Section 4.7 and the results of those experiments in Section 4.7.5. We summarize the conclusions in Section 4.8.

4.2 X-ray System

In an X-ray system there are real-time and best-effort applications that are nowadays run on multiple COTS systems. The real-time applications are streaming applications where images are typically processed at frame rates between 15 and 60 Frames per second. For precise hand-eye coordination and to prevent fatigue, the processed X-ray images should preferably have a low latency (e.g., 120 ms) and the jitter on the latency should be low (e.g., < 16 ms). These requirements enable the physician to smoothly control his equipment. Practical experience has shown that the occasional deviations from the requirements, in case they are small enough and infrequent do not affect the patient's safety. Due to the nature of these requirements we can use modeling techniques and architectures

that work most of time instead of always. Examples of best-effort applications are the Graphical User Interface (GUI), the storage of processed images and the retrieval and analysis of stored images.

Our aim is to run the soft real-time and best-effort applications on a single system while controlling the progress of the best-effort application in such a way that real-time requirements are satisfied and the performance of the best-effort application is maximized. We use the same hardware architecture as in Chapter 3 (see Figure 3.6).

In this chapter we only examined the case where threads of multiple applications are mapped onto a disjoint subsets of physical cores (e.g., the real-time threads on core 0 and 1 and a best-effort application on core 2 and 3, with SMT disabled). The first shared resource for which the threads have to contend is the bandwidth to and the space in the third level of the cache. Other components in the system may also be shared, but in this chapter we focus on how the cache influences the performance of multiple concurrently executing threads.

4.3 Related Work

In many papers cache partitioning techniques have been proposed to improve the performance and predictability of the cache. Cache partitioning splits the cache in disjoint sets that do not share cache lines in the cache hierarchy. This prevents cache thrashing between threads. Stone et al. presented and used this technique in order to minimize the miss rate of each application [42]. Chiou *et al.* presented a technique for cache partitioning based on columnization [8]. Iyer studied cache partitioning in order to improve the Quality of Service (QoS) [18]. Kannan *et al.* studied how dynamic cache partitioning can be added to an Operating System in order to improve the QoS for one or more applications and propose a methodology that can dynamically alter the cache partitioning at runtime [19]. Kim *et al.* also studied cache partitioning in a CMP and optimize the cache partitioning for fairness (i.e. an application with a small amount of cache accesses is penalized less compared to an application with a lot of cache accesses) in [21].

Cache partitioning is a form of performance isolation that at system level has been studied. Verghese *et al.* studied the performance isolation in CMP and focused on isolation in components such as the memory controller and disk access [52]. Nesbit *et al.* introduced Virtual Private Machines (VPMs) in [35], an abstraction supported by hardware modifications in the architecture of a CMP to enable applications to reserve a

certain amount of hardware resources. In the cache hierarchy space and bandwidth can be reserved in order to reduce the interference between applications. In [11], Hansson *et al.* introduced an embedded architecture that also implements performance isolation (on clock cycle level), hardware arbitration, and hard real-time scheduling techniques in order to present a virtual platform that provides complete isolation for different applications. In the newest generation of Intel processors a Cache Allocation Technology (CAT) technique has been implemented that allows the programmer to influence how the cache is used. But in this case only the space is allocated.

However, to the best of our knowledge there is no prior work that addresses the interference that is caused by bandwidth sharing in COTS CMP architectures.

	Bandwidth (GB/s)				
Cores	a	b	c	d	e
	Read				
1	11.5	11.5	19.9	19.9	19.8
2	6.9	7.9	19.8	19.8	16.0
3	7.0	5.6	19.7	19.8	10.4
4	4.8	4.4	17.2	19.7	8.0
	Write				
1	6.3	6.3	15.0	15.0	15.1
2	3.1	3.9	10.5	10.6	8.6
3	2.7	2.5	7.3	7.3	5.1
4	2.6	1.9	5.4	5.4	4.2
	Mixed read (50%) and write (50%)				
1	7.1	7.1	18.2	18.2	17.5
2	4.8	4.9	16.4	16.5	12.8
3	3.3	3.3	11.5	11.5	7.4
4	3.7	2.6	8.7	8.8	5.3

Table 4.1: Partitioning effects in GB/s per core: a) Does not fit in cache, not partitioned. b) Does not fit in cache, partitioned. c) Fits in cache, not partitioned. d) Fits in cache, partitioned. e) only measured thread fits in cache, partitioned

4.4 Motivation

In the previous work chapter, we have shown that under certain conditions COTS can be used for soft real-time image processing applications. The conditions were:

Soft real-time Because of the complexity of modern high-performance CMP architectures (e.g., Intel Nehalem) it can be hard or even impossible to derive tight WCETs. Therefore we do not formally derive the WCET, but we validate that the soft real-time constraints are met by measuring the actual jitter of the system.

Static streaming The application that we examine executes static algorithms. Additionally, the application is streaming, which means that the same algorithms are executed repeatedly on new data. This allows us to map and order the algorithms and allocate the used memory of the application in such a way that less cache thrashing occurs, which is one of the main causes of jitter in CMP architectures.

Single application On a system with only one application there is no contention on hardware resources with other applications.

In this chapter, we want to relax the third condition in order to facilitate the execution of additional best-effort applications during the execution of the real-time applications. When two applications share hardware resources, which can be the case in COTS, they can influence the execution time of each other. As mentioned before we focus on the allocation and bandwidth sharing in the cache and how that influences the interference between a real-time streaming application and other best-effort applications.

4.5 Cache Partitioning

In Section 4.3 we have mentioned that many authors have studied the partitioning of the cache as a means of performance isolation. With cache partitioning, threads do not longer evict cache lines from other threads (cache thrashing), ensuring that data remains in the cache and does not have to be retrieved from main memory. However, this technique only guarantees that other threads do not evict cache lines owned by a thread, not that the latency to retrieve that data is the same, because bandwidth to the third level of the cache is shared. Hence cache partitioning can be used to reduce some of the latency that is introduced by hardware sharing. However, it will not completely remove it. The contention can

only be removed when the bandwidth to the shared cache is controlled. This means there are two orthogonal problems: memory allocation in the cache (solved by partitioning and not studied in this chapter [1]) and bandwidth sharing to the shared level in the cache (studied in this chapter).

Molka *et al.* performed detailed benchmarking on the Nehalem architecture in order to determine the bandwidths between the core and different levels of the cache hierarchy [32]. We extended those benchmarks so that we can determine how much bandwidth a single application can consume on the connection between the second and the shared third level of the cache. The most important observation from those experiments was that a single core is able to consume a significant portion of the cache bandwidth when it is writing (e.g., 15 GB/s write bandwidth for a single core that only writes where the maximal aggregated write bandwidth is 21.9 GB/s, when three cores are writing at 7.3 GB/s). Application that only performs reads will only claim up to 26% of the available bandwidth on a Quad core (19.9 GB/s read bandwidth for a single core where the maximal aggregated read bandwidth is 78.8 GB/s).

With the following experiment we want to demonstrate that this is not the results of cache thrashing, but due to the bandwidth congestion. We recreated the experiments that were performed by Molka *et al* [32] with different memory configuration, namely:

A The accessed data does not fit in the cache and the cache is not partitioned. The main memory will therefore be accessed and cores will thrash the cache.

B The accessed data does not fit in the cache and the cache is partitioned. The main memory will be accessed, but cores will only evict their own cache lines.

C The accessed data of the combined cores does fit in the cache and the cache is not partitioned. The main memory should only be accessed when cache thrashing occurs.

D The accessed data does fit in the allocated cache partition. Main memory should not be accessed and cache thrashing does not occur between the cores.

E Only the accessed data of the measured cores does fit into its allocated cache partition and the data of the other cores does not fit into the cache and they access data in one combined cache partition.

[1] First available in the Broadwell architecture from Intel [13, 17]

Furthermore, we did run these five configurations in three different scenarios. In the three scenarios we either perform only data read, data writes or a mix of data reads and writes. The results of this experiment can be found in Table 4.1. From these experiments we make the following observations:

- When all the data fits in the cache and the cores are only reading (configuration C and D in the read scenario, i.e., columns 3 and 4) the read bandwidth per core is almost constant, regardless of the number of cores that are running. Only when all the cores are reading and the cache is unpartitioned we see a slightly lower read bandwidth (i.e., 17.2 GB/s for four partitioned cores versus 19.8 GB/s on average for the other cases).

- When the data does not fit in the cache (configuration A and B in all scenarios, i.e., columns 1 and 2) the read and write bandwidths are significantly degraded when more cores are running. This is caused by the limited bandwidth to main memory.

- When the data fits in the cache and the cores are writing (configuration C and D in the write scenario) the aggregated bandwidth of the system is maximal 21.9 GB/s (7.3 GB/s per core on 3 cores) while a single thread can consume 15 GB/s essentially consuming 68% of the available write bandwidth.

Therefore we conclude that most performance degradation is caused by the limited bandwidth, and in particular by the limited write bandwidth. Furthermore, cache partitioning does not necessarily reduce the interference. It may even degrade the performance in the situation where the performance of a memory intensive application is degraded more by the reduced cache space than by the cache thrashing of other applications.

4.6 Cache Bandwidth

In order to reduce the interference of the best-effort applications on the real-time applications to an acceptable level, we have to be able to determine when this occurs. We first determine the total current bandwidth usage, and thereafter use this information to reduce the bandwidth of the best-effort application.

4.6.1 BANDWIDTH USAGE ESTIMATION

The system has to detect when a core uses more bandwidth than allocated. To measure this we selected two performance counters in the Nehalem architecture:

- LEVEL_2_TRANSACTIONS.LOAD: Each load transaction is counted. However, a read transaction takes less bandwidth than a write transaction.
- LEVEL_2_LINES_OUT.ANY: Counts the number of lines that are written back to the third level of the cache.

The LEVEL_2_TRANSACTIONS.LOAD performance events could accurately estimate the read bandwidths as found in Table 4.1. The LEVEL_2_LINES_OUT.ANY performance can be used to determine the total (read + write) bandwidth. We could not precisely estimate the write bandwidth in all scenarios with only one performance counter, but we were able to roughly derive this value based on the total bandwidth and the read bandwidth. These counters can be used during the execution of the best-effort application to determine the currently used bandwidth and during design time to determine the worst case bandwidth usage of real-time and best-effort applications.

4.6.2 BANDWIDTH ARBITRATION

There is no hardware mechanism that can allocate bandwidth within the cache hierarchy of the Nehalem architecture to specific cores. We therefore implemented a mechanism in software that can be used to achieve the same effect, but on a coarser time scale. Our mechanism to reduce the bandwidth on a coarser time scale is to repeatedly suspend the best-effort applications for a certain time on a core that uses more bandwidth than allocated, which we will now refer to as duty-cycling. This mechanism results in a much lower (average) bandwidth and reduced performance for the best-effort applications on the core that is temporarily suspended. With this technique it is not possible to completely remove the interference because the duty-cycling of cores manages the bandwidth on a coarse level and only reacts after a best-effort application has consumed too much bandwidth.

4.6.3 DUTY-CYCLING

When the estimated read or write bandwidth of a core that is running best-effort applications reaches a certain threshold we assume that this

core is degrading the performance of the real-time application and that the core that runs that offending best-effort application must be duty-cycled. We implemented this by running a thread, with real-time scheduling priorities, at a higher priority than the best-effort applications in order to temporarily suspend any best-effort application that might consume too much bandwidth. A Linux kernel with real-time patches applied allowed us to precisely sample the performance counters, as mentioned in Section 4.6.1, at regular intervals (e.g., 100 μs) and duty-cycle the best-effort applications accordingly.

4.6.4 threshold and suspension time

The threshold for the read and write bandwidths depends on three items: available bandwidth of the specific processor, consumed bandwidth of the real-time applications and timing constraints of the real-time application. The available bandwidth of the processor in combination with the consumed bandwidths of the real-time applications determines the available bandwidth for the best-effort applications. The timing constraints (e.g., latency) of the real-time application in combination with the unobstructed performance of the real-time application determine how much performance degradation (e.g., latency increase) the real-time application can handle before a timing constraint is violated. The timing constraints and the performance of the real-time application therefore determine how long a best-effort application should be suspended in relation to the sample interval between the measurements of the performance counters.

4.7 Experiments

In this section we describe the experiments that we performed to evaluate the proposed technique.

4.7.1 setup

We used a Linux kernel with real-time patches applied [33]. For each core that would be monitored and duty-cycled, a thread with real-time priorities was instantiated and mapped to that core in order to periodically sample the performance counters. When a derived bandwidth crossed a certain threshold the thread would suspend the core for a specified amount of time. Furthermore, the threads of the real-time and best-effort applications were allocated to disjoint sets of cores.

In the experiments we could influence the following parameters: the type of real-time application, type and number of applications of which the

best-effort performance should be optimized, sampling interval, bandwidth threshold, suspension time. The experiments were categorized in three classes.

4.7.2 SYNTHETIC BANDWIDTH EXPERIMENTS

In our first set of experiments we ran the same program that we used to determine the bandwidth between the level 2 and level 3 of the cache. However, now we applied our duty-cycling technique on cores 2 through 4 and measured the bandwidth on core 1. The sampling interval was 100 μs and the suspension time 900 μs. The thresholds for duty-cycling was set at 25% of the maximal bandwidth. The results of this experiments can be found in Table 4.2 and relative to Table 4.1 in Table 4.3.

4.7.3 OPTIMIZED REAL-TIME STREAMING EXPERIMENTS

In the second set of experiments we ran a static streaming application on the first and second core of the processor and ran a set of best-effort applications on the third and fourth core. The X-ray application was generated and compiled with the tooling as presented in Chapter 3 with the techniques that reduces jitter on CMP, when executed as a single application. In this case the interference memory allocator and the ordering thread scheduling heuristic were used. With the performance counters we estimated the read and write bandwidths of the real-time application to be between 4 and 9 GB/s, which implies that there is enough read bandwidth left, but that the write bandwidth is almost saturated. The threshold for duty-cycling was set at 10%. For the sampling interval and suspension time two sets of parameters were used: 500 μs and 500 μs; and 100 μs and 900 μs for the sampling interval and suspension time respectively. During the experiment the latency of real-time stream processing were measured in order to determine the interference between the applications.

The applications that were used as best-effort applications:

- X-ray^i: The same application that is running as real-time application, but now as background task. Generated using the interference (optimized) memory allocator. Without cache thrashing most memory accesses would hit the cache and memory bandwidth between the cache and memory is low in comparison to the bandwidth between level 2 and level 3 of the cache.
- X-ray^s: The same application that is running as real-time application, but now as background task. Generated using the simple

memory allocator, which does not optimize memory accesses. The cache hit ratio is much lower in comparison with the X-ray^i application. The bandwidth between the cache and main memory is therefore also much higher.

- Syn 'C': An application that reads and writes to the third level of the cache as fast as possible.
- gcc: The gcc compiler while compiling an application.
- ffmpeg: A video transcoder while transcoding a video.

Table 4.4 summarizes the results from this experiment. In the table a row is shown for each set of best-effort applications that were used in the experiment. The first two columns show which best-effort application are run on cores 3 and 4. Each set of best-effort applications is run in four different configurations, namely: *baseline*, where only the real-time application is run on cores 1 and 2; *congested*, where the best-effort applications are executed without our proposed duty-cycle technique; and the two *duty-cycled* configuration, where the duty-cycle technique is applied with two sets of parameters as explained before.

4.7.4 UNOPTIMIZED REAL-TIME STREAMING EXPERIMENTS

In this third set of experiments we re-ran the same set of applications as in the second set of experiments but we now ran our streaming application with another memory allocator that does not optimize level 3 cache accesses (i.e. the simple memory heuristic from Chapter 3) and therefore has more accesses to the third level of the cache and to main memory. The results for this set of experiments can be found in Table 4.5.

4.7.5 RESULTS

In Table 4.2 the results of the first set of experiments are shown. The results in this table clearly show that the maximal reachable bandwidths for the real-time thread is much higher with the duty-cycling technique enabled then without the technique (see Table 4.1. For example, the bandwidth of configuration E under the write scenario without duty-cycling (i.e., column 10 in Table 4.1) drops from 15.1 GB/s, when only one core is writing to 4.2 GB/s, when all the cores are writing. When the cores 2 through 4 are duty-cycled and all the cores are writing (i.e., the last cell in column 10 in Table 4.2) the bandwidth only drops to 14.0 GB/s. In this case the degradation reduces 89.9% from 72.1% to 7.2%. Table 4.3 summarizes the performance degradation reduction between Table 4.1 and Table 4.2.

	Bandwidth (GB/s)				
Cores	a	b	c	d	e
	Read				
1	11.4	11.5	19.9	19.9	19.8
2	11.1	11.2	19.9	19.9	19.5
3	11.0	10.8	19.9	19.9	19.1
4	10.8	10.9	19.9	19.7	18.7
	Write				
1	6.3	6.4	15.0	15.0	15.1
2	6.1	6.1	14.6	15.0	14.5
3	6.0	6.0	14.2	14.2	14.1
4	5.7	5.8	14.1	14.0	14.0
	Read (50%) / Write (50%)				
1	7.3	7.3	18.2	18.2	17.6
2	7.0	7.1	18.0	18.0	17.0
3	7.0	6.8	17.7	17.6	16.6
4	6.9	6.7	17.6	17.2	16.1

Table 4.2: Partitioning effects in GB/s while duty-cycled. a) Does not fit in cache, not partitioned. b) Does not fit in cache, partitioned. c) Fits in cache, not partitioned. d) Fits in cache, partitioned. e) only measured thread fits in cache, partitioned

The second set of experiments has been used to estimate the benefits of duty-cycling in a realistic setup. The results for this set of experiments can be found in Table 4.4.

Although the X-ray^s application does not consume the most bandwidth in the cache hierarchy it degrades (increases) the latency of the real-time the most (i.e., the latency increases from 5867 μs to 13873 μs). The main reason that this application degrades the latency the most is because it is the only tested application in our benchmark set that also uses a lot of cache space and therefore introduces more cache thrashing than the other application. Nevertheless, the duty-cycling technique also decreases the latency degradation for this application.

The X-ray^i and Syn 'C' application use a significantly smaller amount of the cache space and therefore inflict less cache line evictions to the real-time application, but use much more bandwidth between the level 2 and level 3 of the cache. Also for these cases (i.e., row 1 and 3) we see a

	Bandwidth (GB/s)				
	Read				
Cores	A	B	C	D	E
2	91.3	91.7	100.0	100.0	92.1
3	88.9	88.1	100.0	100.0	92.6
4	89.6	91.5	100.0	0.0	90.7
	Write				
2	93.8	91.7	91.1	100.0	90.8
3	91.7	92.1	89.6	89.6	90.0
4	83.8	88.6	90.6	89.6	89.9
	Read (50%) / Write (50%)				
2	95.7	100.0	88.9	88.2	89.4
3	97.4	92.1	92.5	91.0	91.1
4	94.1	91.1	93.7	89.4	88.5

Table 4.3: Performance degradation reduction (%). a) Does not fit in cache, not partitioned. b) Does not fit in cache, partitioned. c) Fits in cache, not partitioned. d) Fits in cache, partitioned. e) only measured thread fits in cache, partitioned

significant degradation of the latency (i.e., to 9283 μs and 9219 μs for the two applications respectively). In both cases the latency degradation is reduced by the duty-cycling technique.

Gcc and ffmpeg (i.e. row 4 and 5) both exhibit much lower bandwidth usage to the third level of the cache and therefore do not degrade the latency to the same extent as the other cases.

Both sets of parameters (500/500 and 100/900 for the sampling interval and suspension time) reduced the degradation of the latency of the real-time application. The 100/900 parameters reduced the latency degradation more than the 500/500 parameters, but also results in less best-effort performance.

In the last set of experiments we see that the latency of the interfered real-time application is higher than in the second set. This is due to the fact that the unoptimized X-ray application consumes more bandwidth to the third level of the cache and to main memory and is therefore more susceptible to interference. However, relative to the baseline latency of the real-time application the increase is slightly smaller. Furthermore, we

Core 3	Core 4	Avg (μs)	Stdev	Max (μs)
		Baseline		
X-ray^i	X-ray^i	5867	15.22	6069
X-ray^s	X-ray^s	5867	15.22	6069
Syn 'C'	Syn 'C'	5867	15.22	6069
Syn 'C'	gcc	5867	15.22	6069
ffmpeg	ffmpeg	5667	15.22	6069
		Congested		
X-ray^i	X-ray^i	9283	19.68	9470
X-ray^s	X-ray^s	13873	87.85	14185
Syn 'C'	Syn 'C'	9219	48.43	10644
Syn 'C'	gcc	7691	161.95	9327
ffmpeg	ffmpeg	6131	149.37	7855
		Duty-cycled 500/500		
X-ray^i	X-ray^i	7339	156.93	8849
X-ray^s	X-ray^s	8305	277.14	9474
Syn 'C'	Syn 'C'	7370	90.93	7912
Syn 'C'	gcc	6755	95.48	7256
ffmpeg	ffmpeg	5977	94.55	7211
		Duty-cycled 100/900		
X-ray^i	X-ray^i	6136	88.55	7037
X-ray^s	X-ray^s	6304	66.3	6512
Syn 'C'	Syn 'C'	6037	46.87	7225
Syn 'C'	gcc	6050	41.68	6978
ffmpeg	ffmpeg	5881	22.62	6010

Table 4.4: Duty-cycling results. Latency of the real-time application

observe that also in this case the latency increase could be controlled by the duty-cycling technique.

4.8 Conclusion

Due to the shared level in the cache and the possible congestion to that level, applications can decrease the reproducibility of each other. This can make it difficult to combine soft real-time and best effort applications on the same system. We therefore proposed a technique that can limit the interference of best effort applications on soft real-time applications. For an industrial X-ray imaging application we show that the latency

Core 3	Core 4	Avg (μs)	Stdev	Max (μs)
		Baseline		
X-ray^i	X-ray^i	10705	28.56	10918
X-ray^s	X-ray^s	10705	28.56	10918
Syn 'C'	Syn 'C'	10705	28.56	10918
Syn 'C'	gcc	10705	28.56	10918
ffmpeg	ffmpeg	10705	28.56	10918
		Congested		
X-ray^i	X-ray^i	14458	211.81	16146
X-ray^s	X-ray^s	21236	62.56	21513
Syn 'C'	Syn 'C'	12742	18.85	12805
Syn 'C'	gcc	11990	187.30	12773
ffmpeg	ffmpeg	11554	259.21	12651
		Duty-cycled 500/500		
X-ray^i	X-ray^i 13092	248.73	14704	
X-ray^s	X-ray^s	14237	69.32	14506
Syn 'C'	Syn 'C'	12193	39.06	12336
Syn 'C'	gcc	11800	98.88	12197
ffmpeg	ffmpeg	11300	159.26	12260
		Duty-cycled 100/900		
X-ray^i	X-ray^i 11295	70.89	12551	
X-ray^s	X-ray^s	11315	69.58	11670
Syn 'C'	Syn 'C'	11540	46.48	11754
Syn 'C'	gcc	11767	144.83	12368
ffmpeg	ffmpeg	11059	50.94	11467

Table 4.5: Duty-cycling results. Not optimized for level 3 access. Latency of the real-time application

increase that is caused by the interference can be controlled while maintaining some best-effort performance. For synthetic benchmarks we show that there are scenarios where the duty-cycling technique can reduce the latency degradation by 89%. We therefore conclude that best-effort core duty-cycling based on bandwidth consumption is an essential method to control the interference in CMP architectures used for real-time streaming applications in combination with best-effort applications, such as the interventional X-ray application.

CHAPTER 5

End-to-End Latency Distribution Analysis for Probabilistic Time-Triggered Systems

Abstract – *Medical image processing systems are typically implemented with a number of independent subsystems that have time triggered interfaces. A critical design parameter for these systems is the latency between the instant that an image is captured and the instant that the enhanced image is displayed to the physician. Computation of an exact end-to-end latency distribution with existing techniques is often impractical as a result of the extremely large number of states that needs to be considered.*

In this chapter we introduce the PTTS model. We show that with this model an exact distribution of the end-to-end latency can efficiently be computed. The reason for this is that a cyclic triggering pattern describes the complete state space and that it is often feasible and practical to traverse this state space to derive an exact latency distribution. Furthermore, we present techniques to reduce the analysis time at the cost of accuracy for the cases that the search space is impractically large.

We demonstrate the applicability of the presented analysis technique by showing that several system configurations of an X-ray application can be quickly explored. This analysis makes the parameters of a system configuration explicit that significantly affects the end-to-end latency distribution.

This chapter is based on [MW:3] .

This chapter introduces a model to analyze the end-to-end latency of a system that is composed of COTS components with time triggered interfaces. Techniques to improve the reproducibility of execution times of the individual components, which allow us to do the analysis, have been presented in the previous chapters.

5.1 Introduction

With an interventional X-ray system, a physician makes use of images captured with an X-ray imaging device to perform delicate medical procedures with a catheter inside a patient, where the only visual feedback is provided by the images captured by the X-ray device. It is therefore desirable that the latency between the capturing of an image and displaying it is short enough (e.g. $<$200 ms) to provide sufficient eye-hand coordination. Furthermore, the jitter must be sufficiently small such that the physician experiences a constant delay, which improves the eye-hand coordination and prevents fatigue. Medical image processing systems such as interventional X-ray systems are nowadays often composed of a number of independent subsystems that have time triggered interfaces. Examples of these subsystems are the image sensor, the image enhancement general purpose computer, a GPU for image composition, a GUI and the video wall for the combination of several displays. The latency introduced by the individual subsystems as well as the end-to-end delay can vary significantly and is typically measured. These measurements usually give little insight in which parameters of the system configuration have a large effect on the distribution of the end-to-end latency.

The end-to-end latency distribution can be computed given the well known Probabilistic Timed Labeled Transition System (PTLTS) model [53]. However, the run-time of the exact analysis of the end-to-end latency distribution given an PTLTS model can become prohibitively long as a result of the large number of states that needs to be considered. A complete and deterministic end-to-end latency analysis is necessary, in order to detect rare events where data is overwritten before it is read.

In this chapter we introduce the PTTS model together with analysis techniques for the efficient derivation of an exact end-to-end latency distribution of time triggered systems that are composed by time triggered subsystems. The PTTS model is only suitable for systems that are composed of a chain of time-triggered subsystems that are connected by buffers with one location with destructive write and non-destructive read seman-

tics instead of the in the PTLTS considered queues with non-destructive write and destructive read semantics. We will show that systems with these buffers can be analyzed more efficiently than systems with queues. Furthermore, we will derive an expression for the time-complexity of our analysis algorithm. We also present a technique that reduces the state-space at the cost of accuracy by adapting parameters in the model.

Our case study demonstrates the applicability of the PTTS model and the corresponding analysis techniques for an interventional X-ray system. We expect that the PTTS model is also applicable for other systems than medical image processing systems.

The outline of this chapter is as follows. In Section 5.2 we discuss alternative analysis approaches for time-triggered systems. In Section 5.3 we define the PTTS model. In the subsequent Section 5.4 we present a didactic example that provides the intuition behind our end-to-end analysis techniques. A detailed description of our analysis algorithm is given in Section 5.6. A technique to reduce the running time of our analysis algorithm is discussed in Section 5.7. We demonstrate the applicability of the PTTS model for an interventional X-ray system in Section 5.8. Finally we state the conclusion in Section 5.9.

5.2 Related Work

The start of the execution of tasks in time triggered systems is determined by timers instead of events that denote the arrival of containers with input data of the tasks. Time triggered systems, in which the tasks are scheduled strictly periodically, have been studied extensively by Kopetz [23, 25]. In his approach the schedule of the tasks is computed at design time using the worst-case execution times of the tasks. The computed schedule should respect the precedence constraints between the tasks, as well as the throughput and end-to-end latency constraints of the task graph. Satisfaction of the precedence constraints guarantees functional deterministic behavior of the tasks graph, i.e. the results computed by the tasks are independent of the execution time of the tasks as long as the execution times are not larger than the at design time assumed worst-case execution times. The most important difference with the work presented in this chapter is that we do not make use of worst-case execution times but make use of Execution-Time Profiles (ETPs) to characterize the execution time of the tasks. The ETPs are probabilistic characterizations of the execution times by means of probability mass functions [3]. As a result we do not want to compute a periodic schedule at design-time, because it will be overly pessimistic. Another consequence is that the

execution of the systems that we consider in this chapter is not functionally deterministic, because we allow that data is overwritten before it is read depending on the execution times. Another time triggered model is proposed by Henzinger [12], which relies on a global clock and WCETs in order to compute permissible schedules. Our model does not use WCETs and can derive the distribution of the end-to-end latency instead of giving a guarantee that a system adheres to hard real-time constraints.

The PTLTS [47] model is a model that is suitable for the analysis of probabilistic time triggered systems. This model can be automatically derived from a description of a system in the POOSL language [46]. A PTLTS model can be converted into a Markov chain for which exact analysis techniques exist [39]. Furthermore, the model supports stochastic execution times [44]. The model can also be simulated, but this reduces the accuracy of the latency distribution. It should be noted that the obtained results are in general only valid for infinitely large intervals of time. A well known problem is that the number of states in these Markov chains is often so large that exact analysis is impractical. Therefore, approximation techniques based on simulation have been proposed that do not consider all states [39, 51]. In these proposals the PTLTS model is simulated till estimators indicate that it is likely that a sufficiently large part of the state space has been considered and sufficiently accurate results are obtained. However, usually no indication is provided whether the obtained results are an over approximation or an under approximation of the throughput and the latency nor is there a bound provided on the accuracy of the obtained analysis results. An important difference is that our PTTS model is only suitable for the analysis of a subset of the systems that can be analyzed with the PTLTS model, i.e. systems that can be modeled with time-triggered subsystems. However, we will show that because our PTTS model is less expressive there is additional structure in the state-space, which can be exploited to define significantly more efficient analysis techniques. Due to the less expressive model we can derive events, i.e. triggerings of the input subsystem, in the state space separately, which decrease the analysis complexity. Furthermore we show that our analysis results are valid for finite intervals of time.

Another model is Scenario Aware Data Flow (SADF) that extends SDF with stochastic scenarios in order to model the dynamic behavior and variable execution times of applications [45]. However, SADF lacks the expressiveness to model the time-triggered interfaces. The production and consumption rates of actors in the SADF model should be consistent. In our model the production rates of different subsystems may differ and our analysis algorithm is able to quantify the amount of data duplication

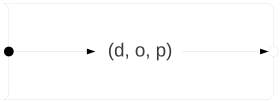


Figure 5.1: A PTTS component

or destruction. Stochastic Petri nets [20] can also be used to model the stochastic behavior of streaming application but suffer the same problem as the SADF model, namely that the modeling of time-triggered interfaces is not possible.

5.3 The PTTS Model Definition

In this section we define the PTTS model. The PTTS model consists of components with one input port and one output port. Such a component is depicted in Figure 5.1.

The input port of a component is a buffer with one location, which is read out time-triggered. More precisely the instant at which the component reads a sample is determined by a timer and not by the arrival time of data containers at the input port. The buffer at the input port has non-destructive read and destructive write semantics. Therefore, if no new container has arrived on the input port at the sampling moment, the previously arrived container is read again from the input port. If multiple containers arrive at the input port between two sampling instants, then the container is processed that arrived last.

Each component is characterized by three parameters d, o, and f. Here d defines the period at which the time-triggered input port triggers, o is the initial offset when the input port is triggered for the first time, and f is the Execution-Time Profile (ETP). The ETP is a Probability Mass Function (PMF) that characterizes the latency of a component, i.e. the interval between the triggering of the input port of the component and when data is produced at its output port. The PMF is defined at discrete points. The use of a PMF instead of a PDF simplifies the explanation of our end-to-end latency distribution calculation algorithm, while the algorithm would remain conceptually similar if PDFs would be used instead. Multiple containers can be under processing at the same point in time, in other words, the model allows auto-concurrency. That a later triggering of the component can result in an earlier production than an earlier triggering, i.e. out of order production, is not problematic because we only derive temporal behavior and not functional behavior with the PTTS model. The ETP can

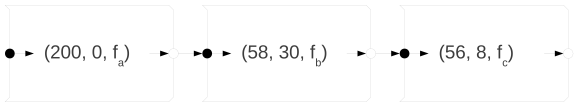


Figure 5.2: A chain of PTTS components

be obtained by measurement of the latency of a component in isolation.

Components in the PTTS model can correspond, for example, with a CCD in a video system that is triggered at 24 Frames per second (FPS), or a GPU that transfers a new image from video memory to a monitor at 60 FPS. It is also possible that a component corresponds to a task graph executing on multiple processors, but the PTTS model abstracts from this.

PTTS components are connected in a chain. The output of all but the last component are connected to its successor, e.g. C_0 is connected to C_1. A chain of N components is denoted as $\langle C_0, C_1, \ldots, C_{N-1} \rangle$. An PTTS chain is described as a list of tuples, where each tuple (d_i, o_i, f_i) specifies the parameters of the component C_i.

When two consecutive components trigger at the same instant we define that production always precedes consumption. A trivial change in one of the equations that is used to derive the latency distribution, which will be introduced later, could be applied in order to let consumption precede production. An example of a PTTS chain is shown in Figure 5.2. The analysis algorithm presented in Section 5.6 derives the probability distribution that characterizes the end-to-end latency of a chain of components. With the same algorithm we determine the probability that a container is overwritten before it is read, i.e. the probability that data loss occurs.

5.4 Basic Idea

In this section we present the basic idea behind our end-to-end latency distribution analysis algorithm, using an example. This analysis algorithm will be presented in detail in Section 5.6.

As an example we use the PTTS chain shown in Figure 5.2. The model in Figure 5.2 can be described with the chain $\langle (200\,\text{ms}, 0\,\text{ms}, f_0), (58\,\text{ms}, 30\,\text{ms}, f_1), (56\,\text{ms}, 8\,\text{ms}, f_2) \rangle$. In this example the ETPs of the latency parameters f_0, f_1, and f_2, in which $t \in \mathbb{N}_0$, are given in Equation (5.1), Equation (5.2) and Equation (5.3) respectively. These functions define a discrete homogeneous PMF over the given intervals and describe the ETP.

$$f_0(t) = \begin{cases} \frac{1}{100}, & t \in [50, 149] \\ 0, & \text{otherwise} \end{cases} \tag{5.1}$$

$$f_1(t) = \begin{cases} \frac{1}{30}, & t \in [25, 54] \\ 0, & \text{otherwise} \end{cases} \tag{5.2}$$

$$f_2(t) = \begin{cases} 1, & t = 5 \\ 0, & \text{otherwise} \end{cases} \tag{5.3}$$

A system described by the PTTS model has a hyper-period, i.e. the time after which it is in an identical state. Each triggering of a component in a hyper-period has a unique set of offsets to the triggerings of other components. For example, the offset between the first triggering of C_0 and the subsequent triggering C_1 is 30 ms, but the offset between the second triggering of C_0 at $t = 200$ ms and the subsequent triggering of C_1 at $t = 204$ ms is 4 ms. The same triggering pattern is repeated in every subsequent hyper-period. Given the periods of the components we can derive the hyper-period of the system by calculating the Least Common Multiple (LCM) of all the components period as in Equation (5.4).

$$h = \operatorname*{lcm}_{0 \leq i < N} (d_i) \tag{5.4}$$

The hyper-period for the PTTSs model in Figure 5.2 is LCM(200 ms, 58 ms, 56 ms) = 40 600 ms. As a consequence, if there is a triggering of a component at time $t = 0$ ms then there are also triggerings with the same end-to-end latency distribution of the same component at time $t = o_0 + k \cdot 40\,600$ ms with $k \in \mathbb{N}$ and where o_0 is the initial offset of component C_0. As a consequence it is sufficient to consider the interval of a single hyper-period during analysis.

A part of the triggering pattern of this example is depicted in Figure 5.3.

One way to derive the end-to-end latency is to follow a container from the input of the PTTS model to the output of the PTTS model. We follow a container in the state space by stepwise expanding the state where the container is produced and follow it as it passes through the PTTS chain. For example, assume that the input port of C_0 is triggered at 0 ms. From the ETP f_0 we conclude that the latency introduced by component C_0 is between 50 ms and 149 ms. Suppose now that for this particular triggering the latency introduced by C_0 is 120 ms, then a container is

produced by component C_0 at t=120 ms. This container will stay in the input buffer of component C_1 till the input port of this component is triggered. Component C_1 has a period of 58 ms and the first triggering of its input port happens at t=30 ms because its offset is 30 ms. Therefore, after C_0 has produced a container at 120 ms, the third triggering of C_1 will consume the container, because
$\min\{x \in \mathbb{N}_0 | 30 + x \cdot 58 \geq 120\text{ms}\} = 2$. In a similar manner we can find when C_2 consumes the containers produced by C_1 and at which moments in time C_2 produces its output containers. This way we can simulate what the end-to-end latency is for the triggering of C_0 that originates at 0 ms and for one particular latency per component.

In order to obtain an impression of the end-to-end latency distribution by means of simulation, we would have to simulate all possible triggerings of all components for many possible latencies of the individual components. A problematic aspect of such an approach is that such a simulation takes long, because many cases need to be considered. Furthermore, no bounds are defined on how accurately the end-to-end latency distribution obtained by simulation approximates the exact latency distribution. Therefore, it can be preferable to derive the exact end-to-end latency distribution by means of an analytical approach instead of simulation. Such an analytical approach can exploit the fact that not all possible latencies of the components need to be considered but that an interval of possible latencies results in the same end-to-end latency.

The sampling moment of a component C_i is defined by its period and offset, d_i and o_i respectively. Given these sampling moments and the ETP of a component we can derive the intervals in which containers are produced by a component. These intervals are then further divided in subintervals based on the sampling moments of a consuming component. For each of these subintervals we can derive the probability that a container is produced. From this we can derive for combinations of triggering moments of two subsequent components in the PTTS model the probability that a specific triggering of the component that produces the container is sampled by a specific triggering of the component that consumes the container.

In Figure 5.3 the triggerings of the components are indicated by arrows ($\rightarrow$). The same figure shows with rectangles in which interval of time a container is produced by a specific triggering. Furthermore, we can derive from the rectangles which triggering of a subsequent component in the chain is potentially consuming the container. With each fraction of a rectangle, a probability is associated that defines the probability that a

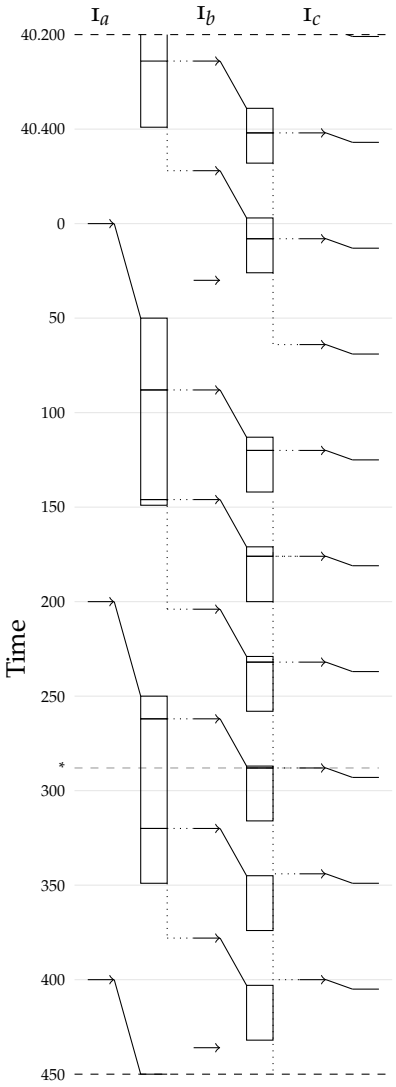


Figure 5.3: Part of the trace in one hyper-period of the PTTS in Figure 5.2. In this trace you can clearly see all the triggers for all three components. For each triggering a box is shown that depicts the range of the execution times associated with the components. The dotted lines lines represent which part of the execution times are read by the subsequent component. In the trace you can also see that due to the clock differences between the components that the time between the triggerings of subsequent components differs between each firing. Lastly we can see that the ending of the hyper-period coincides with the start, i.e. the state of the end of the hyper-period is equal to the state of the start of the hyper-period.

container is produced in the corresponding interval of time. Given these probabilities, we can derive the probability that a container produced by a triggering of a component, is consumed by a specific triggering of a consecutive component.

More precisely, the probabilities for our PTTS example are determined as follows. When C_0 activates at 0 ms we know from its ETP that the latency of this component lies between 50 ms and 149 ms. Given this information we can determine which triggerings of C_1 might sample the output of the first triggering of C_0. From the trace in Figure 5.3 we can conclude that the triggerings of C_1 at 88 ms, 146 ms, and 204 ms might sample the output of the first triggering of C_0. Furthermore, we can also calculate for each of these triggerings the probability that a particular triggering of C_1 will sample the output. This can be achieved by computing the sum of probabilities of C_0 for the interval that a specific triggering a C_1 consumes a container and that it is produced by a specific triggering of C_0. For example: the triggering of C_1 at 146 ms samples the output that is produced by C_0 that is triggered at 0 ms and produces it output between 89 ms and 146 ms. The probability that this triggering of C_1 consumes the container that is produced by C_0 is therefore: $\sum_{t=89}^{146} f_0(t) = \frac{58}{100}$.

After having calculated the probabilities for this component we have to do the same for the subsequent component in the chain. In this example we therefore perform three probability calculations by means of summation, one for each of triggering times at 88 ms, 146 ms and 204 ms. This analysis is repeated for each component in the chain till we reach the end of the chain. Furthermore, these steps are repeated for all triggering of C_0 in the hyper-period.

The information in the trace of Figure 5.3 together with information about the probabilities is stored in a so-called latency tree. Each node in the tree represents a triggering instant in one hyper-period of a component in the PTTS. The numbers in the node represent the index of the component in the chain and the triggering instant respectively. The value on the edges represent the probability that the two triggerings of different components are related, i.e. the triggering of a component that consumes the container that is produced as a result of the triggering of the other components. Lastly, the bottom leaves are a component that represents the end of processing of the last component. For example, the latency trees in Figure 5.4 are the results of the latency analysis of triggerings of C_0 at 0 ms and 200 ms.

The end-to-end latency distribution is derived from the latency tree. The

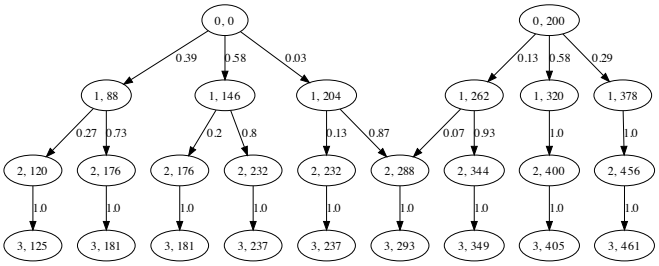


Figure 5.4: Latency tree of two triggerings of C_0 also shown graphically in Figure 5.3. The latency tree precisely shows the different ways the components can fire and their probability. Note that when a node in this tree has two or more parents there is a possibility for data loss.

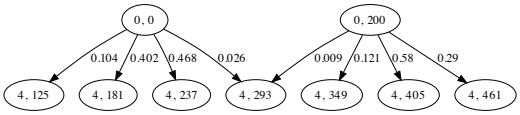


Figure 5.5: Flattened latency tree of two triggerings of C_0 from Figure 5.4. In this flattened latency tree all the intermediate node between the first and last components have been removed and their probabilities have also been merged.

end-to-end latency distribution is found by following all paths from the root till the leaves of the tree and multiplying the probabilities along those paths. These probabilities can be represented in a flattened version of the latency tree. Flattening will be elaborated in Section 5.6. For example, the flattened latency trees for the first two triggerings of C_0 are shown in Figure 5.5. Optionally, we could aggregate all paths in the tree in order to obtain the average end-to-end latency distribution of the complete system. Figure 5.6 shows the end-to-end latency distribution for the PTTS model in Figure 5.2.

The end-to-end latency distribution of the example, see Figure 5.6, has a Gaussian shape. That the end-to-end latency distribution can also have another shape is demonstrated by replacing the distribution ETP f_0 by the distribution ETP f_0', which is defined in Equation (5.5). With f_0' the end-to-end latency distribution changes, as shown in Figure 5.7.

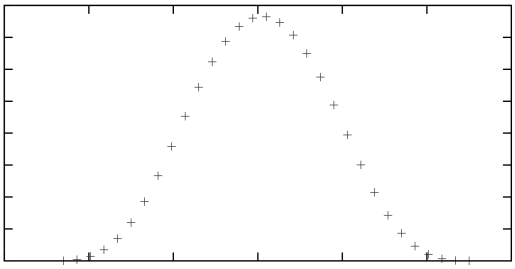


Figure 5.6: End-to-end latency distribution

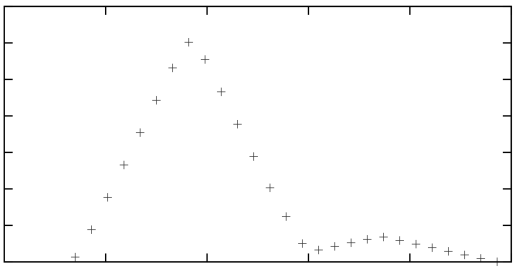


Figure 5.7: End-to-end latency distribution of the first example using $f_0^{'}$

$$f_0^{'}(t) = \begin{cases} \frac{1}{10}, & t \in [50, 58] \\ \frac{1}{10}, & t = 149 \\ 0, & \text{otherwise} \end{cases} \tag{5.5}$$

5.5 Container loss

The latency trees can also be used to derive the probability that a container is overwritten before it is read. This corresponds in an X-ray system with an image that is captured by the X-ray image sensor, which is not displayed. Because this can result in hiccups in the video sequence this is usually undesirable and may therefore only occur rarely.

That a container is overwritten before it is read can also occur in our running example. When we examine the latency trees for the triggerings at t=0 and t=200, we notice that they share two nodes with each other. The shared nodes represent the triggering of C_2 at $t = 288$ and the triggering of the last output node at $t = 293$. This means that, when the triggering at t=0 is slow and generates the path in the latency tree $\langle 0, 204, 288, 293\rangle$ and the subsequent triggering at $t = 200$ is fast and generates the path $\langle 200, 262, 288, 293\rangle$, that the container that is a result of the triggering at $t = 0$ is overwritten before it is read by C_2, i.e. the data in the container is lost. This behavior can also be concluded from the trace in Figure 5.3 where the data loss might occur at the dashed gray line at $t = 288$.

For the latency tree we can also calculate the probability that a container is overwritten before it is read. This probability in our running example is equal to the probability that the in the previous paragraph mentioned slow and fast trace happens after each other divided by the number of triggerings of C_0 in the hyper-period. This probability is equal to $\frac{(0.03 \cdot 0.87) \cdot (0.13 \cdot 0.07)}{203} = 1.17 \cdot 10^{-6}$. Because in this PTTS example the triggerings at $t = 0$ and $t = 200$ are the only triggerings that result in a potentially container loss, we conclude that $1.17 \cdot 10^{-6}$ is the probability that the data in a container is lost during a hyper-period of the system.

5.6 The Analysis Algorithm

In this section we present the analysis algorithm for the derivation of the end-to-end latency distribution based on the principles discussed in the previous section. First we will define how we construct a latency tree, and then we will flatten this latency tree to retrieve the distribution of the latency between the first and last component in the chain.

5.6.1 model definition

The latency tree is defined as follows. The tuple (i, t, S) denotes a node in the latency tree, where i represents the i^{th} component in the chain, t represents the instant at which component i is triggered and S the set of tuples that represents the children of this node. The tuple $s \in S$ that represents a child of a latency tree node is defined by the tuple (p, y), where p is the probability that component C_{i+1} consumes the output of C_i, and y is a recursive tuple $(i + 1, t', S')$ that represents the triggering of the subsequent component C_{i+1} that consumes a container from C_i at t'.

Before we present the derivation of the latency tree with recursive functions, we first define some helper functions.

Let the function $A(i)$ define the set of points in time that the i^{th} component in the PTTS model is triggered:

$$A(i) = \{k \mid n \in \mathbb{N}_0 \wedge k = n \cdot d_i + o_i\} \tag{5.6}$$

Let $u(i, t_a, t_b)$ define the function that calculates the probability that the triggering at t_a of component C_{i-1} produces the token that is consumed by C_i that is triggered at t_b:

$$u(i, t_a, t_b) = \sum_{t=t_b-d_i+1}^{t_b} f_{i-1}(t - t_a) \tag{5.7}$$

It is sufficient to sum the discrete probabilities from $t_b - d_i + 1$ because at $t_b - d_i$ the previous triggering of C_i takes place which consumes containers from C_{i-1}.

The function Q in Equation (5.8) computes a set of tuples, where each tuple was defined as (p, y) where y is the recursive tuple $(i+1, t, S)$. Each tuple corresponds to a triggering of component C_{i+1} that has a probability larger than o to sample a container produced by component C_i. This function accepts two parameters, i and t where i is the index of the component from which we want to derive the latency tree. The other parameter, t, represents the instant at which C_i is triggered. The function Q uses the function Z, which is defined in Equation (5.9) that will recursively generate the whole latency tree.

$$Q(i,t) = \{(u(i+1,t,a), (i+1,a,Z(i+1,a)) \\ \mid a \in A(i+1) \wedge u(i+1,t,a) > 0\} \tag{5.8}$$

The function Z is defined as follows:

$$Z(i,t) = \begin{cases} W(i,t) & \text{if } i+1 = N \\ Q(i,t) & \text{if } i+1 < N \end{cases} \tag{5.9}$$

The function Z calls the function Q that derives the time-triggered parts of the end-to-end latency distribution till the end of the chain is reached and subsequently the function W is called to add the part of the latency distribution that is not consumed by a consecutive timed-triggered

component. The function W in Equation (5.10) returns a set of tuples $(p, (i, t, S))$ for the last component in the chain of the PTTS model and adds the latency introduced by this component. It accepts the parameters i and s that represent the index of the component and the instant the component is triggered. The function W is defined as follows:

$$W(i,t) = \{(f_i(n), (i, t+n, \varnothing)) \mid n \in \mathbb{N}_0 \wedge f_i(n) > 0\} \tag{5.10}$$

Given the hyper-period h, the forest F of latency trees is calculated with:

$$F = \{(0, a, Z(0,a)) \mid a \in A(0) \wedge a < h\} \tag{5.11}$$

5.6.2 END-TO-END LATENCY DISTRIBUTION

The end-to-end latency distribution can be derived from a latency tree by flattening. The flattening step removes all the inner nodes in a latency tree and directly connects the root of a tree with its leaves. The probabilities on the path from root to the leaves are multiplied in order to derive the end-to-end probability. We do not directly derive the flattened tree because we need the intermediate nodes to detect data loss when data is overwritten before it is consumed.

In some cases there will be several leafs in a latency tree that represent the same triggering of the last component. This might happen because two components might have an ETP that is larger than the interval between triggerings of the subsequent component. In our example we see that we can have duplicate leafs in Figure 5.4, when the tree is flattened. The leafs with $t = 181$ and $t = 237$ will have duplicates, when the tree is flattened and should be aggregated. For each tree that has multiple leafs that correspond with the same triggering we aggregate the leafs and compute its probability by summing the probabilities of the original leafs that share the same instant.

The following function recursively flattens the tree by propagating the probability that a certain path is taken to the leaf node and then returns the list of leaf nodes with the derived probability that that leaf node is triggered. The argument p specifies the probability that the node n in the latency tree is triggered. The function recursively calls itself to generate the flattened subtree from node n and then concatenates those flattened subtrees and propagates them to the root node in order to obtain the flattened version of the complete latency tree.

$$g(p,n) = \begin{cases} \forall \\ (k,l) \in \{ \cup \forall_{(x,y) \in S_n} | g(x,y) \} | (p \times k, l) & \text{if } |S_n| > 0 \\ n & \text{if } |S_n| = 0 \end{cases} \tag{5.12}$$

The flattening step thus removes all the inner nodes in a latency tree and directly connects the root of a tree with its leafs. The probabilities on the path from root to the leafs are multiplied in order to derive the end-to-end probability.

5.6.3 TIME COMPLEXITY

In this subsection we derive an expression for the time-complexity of our analysis algorithm. This expression will be used in Section 5.7 to estimate the effect of a modification of the periods on the run-time of our analysis technique.

The time-complexity of our analysis algorithm is based on the number of latency trees that have to be derived and the time it takes to derive a latency tree. The number of latency trees that have to be derived is equal to the hyper-period divided by the period of the first component in the PTTS model. The time it takes to derive a latency tree is proportional to the depth, i.e. the number of components, and the branching factor in the latency tree, i.e. how many siblings each node has in the tree. The branching factor depends on the width of the ETPs, i.e. the interval between the best-case and worst-case latency, and the period of the subsequent component that reads the container that is produced after those ETPs. For example, the width of the ETP that is associated with component C_0, from the example in section Section 5.4, is $149\,\text{ms} - 50\,\text{ms} + 1\,\text{ms} = 100\,\text{ms}$. Component C_1 consumes containers from C_0 and has a period of 58 ms. This implies that each node from C_0 has between $\lfloor \frac{100}{58} \rfloor = 2$ and $\lceil \frac{100}{58} \rceil = 3$ siblings. In Figure 5.4 we can see that the first two triggerings of C_0 each have three siblings, and the branching factor for those two triggerings is therefore three. We therefore conclude that the time-complexity of our analysis algorithm can be expressed in the terms of the number of required operations with:

$$\check{\tau}(i) = \min\{t | f_i(t) > 0\} \tag{5.13}$$

$$\hat{\tau}(i) = \max\{t | f_i(t) > 0\} \tag{5.14}$$

$$\frac{1}{d_0} \cdot \operatorname*{lcm}_{0 \le i < N}(d_i) \cdot \prod_{i=0}^{N-2} \left\lceil \frac{\hat{\tau}(i) - \check{\tau}(i) + 1}{d_{i+1}} \right\rceil \tag{5.15}$$

	Components		
Configuration	C_0	C_1	C_2
a	(66.0, 0, f_0)	(16.0, 0, f_1)	(17.0, 0, f_2)
b	(66.6, 0, f_0)	(16.6, 0, f_1)	(16.7, 0, f_2)
c	(66.66, 0, f_0)	(16.66, 0, f_1)	(16.77, 0, f_2)

Table 5.1: Configurations for hyper-period reduction

Configuration	Latency trees	Analysis time (ms)
a	136	36
b	13861	3041
c	1388611	297620

Table 5.2: Run-time of algorithm on configuration from Table 5.1

5.7 Analysis Time Reduction

In this section we describe a technique to reduce the run-time of the analysis algorithm at the cost of accuracy. The technique is intended for the cases that the hyper-period is too long, which results in impractically long run-times.

From Equation (5.15) it follows that we can reduce the run-time of our analysis algorithm by reducing the hyper-period. In this section we examine how the modification of the periods of the components affects the run-time and accuracy of our analysis algorithm.

The hyper-period of a PTTS can be reduced by adapting the periods of the components. The effect of such a modification on the accuracy of the analysis result is hard to predict in general. However, for a number of realistic systems we have observed that the shape of the latency distribution remains intact. For example: consider a small system with three components where the parameters for the components are defined as in Table 5.1. The configurations *a*, *b* and *c* have an increasingly larger hyper-period, and as a results, an increasing larger number of latency trees has to be derived. We have run our analysis algorithm on the three configurations. In Table 5.2 the number of derived latency trees and the

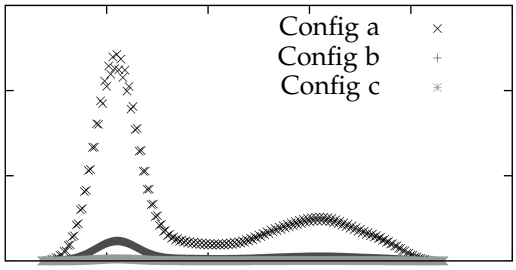


Figure 5.8: End-to-end latency distribution for configuration in Table 5.1

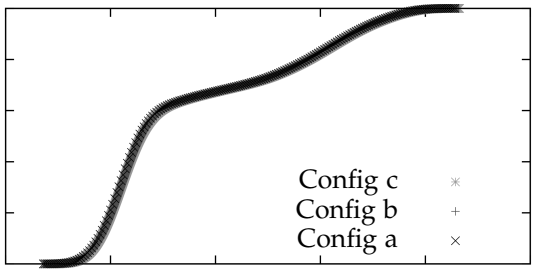


Figure 5.9: End-to-end cumulative latency distribution for configuration in Table 5.1

run-time of the analysis algorithm can be found. Furthermore, in Figure 5.8 we have depicted the end-to-end latency distribution. The shape of the end-to-end latency distribution is similar, but due to the decreased resolution of the PMF the amplitude of the configuration with the reduced hyper-period is higher. We also notice that, due to the dramatically decreasing hyper-period, the analysis time reduces drastically. In Figure 5.9 the cumulative end-to-end latency distribution is shown.

5.8 Case Study

In this section we study the influence of parameter changes on the latency distribution of an image processing chain using the analysis techniques

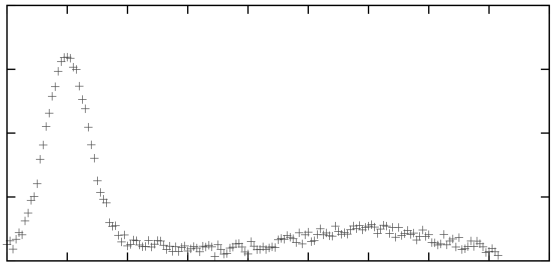


Figure 5.10: Definition of the ETP for f_0 that is used in the case study

presented in the previous sections. Furthermore, we compare the results obtained by simulation with the exact results obtained by analysis.

The image processing chain we consider in this section consists of an X-ray sensor (CCD), a network, an image processing subsystem, a GPU and a media wall, which also incorporates another GPU. The chain contains three time-triggered components namely the X-ray sensor and the two GPUs. The PTTS model is defined by the chain $\langle\, C_0, C_1, C_2 \,\rangle$.

	Components		
Configuration	C_0	C_1	C_2
d	(66.6, 0, f_0)	(16.6, 0, f_1)	(16.7, 0, f_2)
e	(66.6, 0, f_0)	(16.6, 0, f_1)	(16.6, 0, f_2)
f	(66.4, 0, f_0)	(16.6, 0, f_1)	(16.7, 0, f_2)

Table 5.3: Configurations for the model defined in Section 5.8

$$f_1(t) = \begin{cases} \frac{9}{10}, & t = 11 \\ \frac{1}{10}, & t = 12 \\ 0, & t \notin [11, 12] \end{cases} \tag{5.16}$$

$$f_2(t) = \begin{cases} 1, & t = 8 \\ 0, & t \neq 8 \end{cases} \tag{5.17}$$

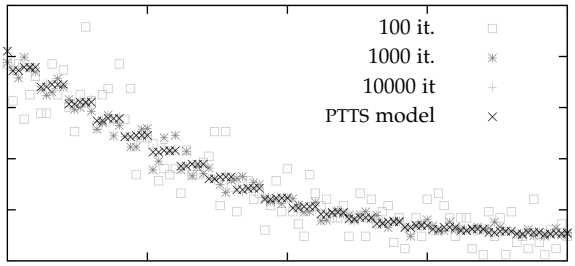


Figure 5.11: End-to-end latency distribution for configuration e

To study the effect of the parameters on the end-to-end latency distribution we consider the three system configurations of Table 5.3. The applied ETPs can be found in Figure 5.10, Equation (5.16) and Equation (5.17).

We compare the latency distribution, which is obtained by simulation, with the exact results. We have measured the end-to-end latency distribution by simulation for 100, 1000 and 10000 iterations. A comparison with the exact distribution is shown in Figure 5.11. It can be observed that the simulation results approximate the exact results accurately, when a large number of iterations is simulated. The number of iterations that results in a sufficiently accurate approximation is, however, not known in advance. In Table 5.4 we measured the time that our analytical algorithm took and the time that the simulations took. The table shows that the analytical approach is much faster than the simulation, even when the simulation only does 100 iterations.

The results for the configurations d, e and f can be found in Figure 5.13.

Analysis tool	Time (ms)
Analytical	30
Simulation (100 iterations)	98
Simulation (1000 iterations)	749
Simulation (10000 iterations)	11372

Table 5.4: Analysis time

Also the offsets can have a significant effect on the end-to-end latency distribution. We demonstrate this for configuration e from Table 5.3 in

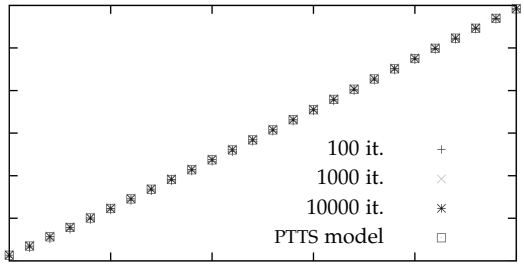


Figure 5.12: End-to-end latency CDF for config e

which we select different offsets for C_2. In this configuration the frequencies for the components C_1 and C_2 are derived from the same clock source. The obtained analytical results are shown in Figure 5.14. From this figure we can conclude that the offsets have a significant influence on the end-to-end latency. Another important observation that we can make from this figure is that the end-to-end latency distribution does not increase monotonically with regard to the varied offset. In this example we can see that each of three shown latency distributions perform differently. For example, the best case latency is only available in the latency distribution where the offset is 80. Furthermore, the worst case latency of the latency distribution with offset 80 is worse than the other two latency distributions. However, when we examine the cumulative probability at 160 ms, we see that the the probability of the distribution with offset 80 is much higher, which means that although the WCET of that distribution is higher, the chance is also also higher that the latency is under 160 ms.

5.9 Conclusion

In this chapter we presented the Probabilistic Time Triggered System model for analyzing the latency distribution of real-time systems that consist of a chain of subsystems with time-triggered interfaces and a probabilistic delay. This type of systems is found in for example the medical image processing domain, where the end-to-end latency distribution is an important design parameter. Furthermore, we have presented an analysis algorithm for the computation of the exact distribution of the end-to-end latency. The algorithm is exact, which guarantees that rare events such as data-races will be detected and that the algorithm does not introduce an overestimation or an underestimation of the end-to-end

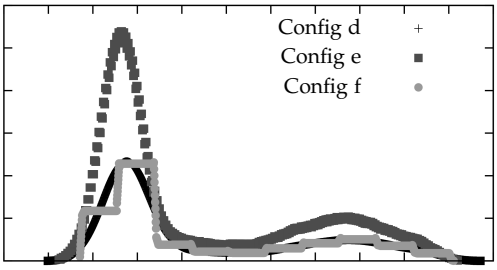


Figure 5.13: End-to-end latency distribution for different configurations of our case study

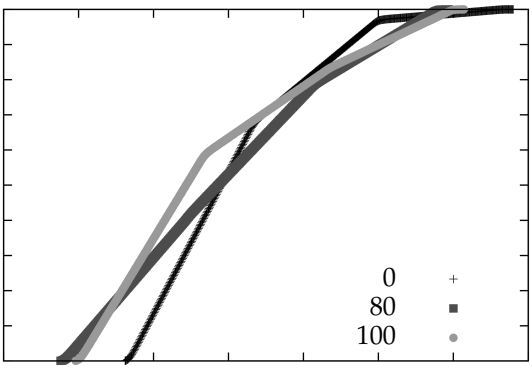


Figure 5.14: Cumulative end-to-end latency distribution for different configuration e with varied offset for C_1

latency distribution.

A useful feature of the presented analysis algorithm is that the size of the state-space can be computed, which allows the derivation of the time-complexity of our analysis algorithm. For the cases that the execution time is impractically high we present a technique that reduces the analysis time at the cost of accuracy.

We have compared the accuracy and run-time of our analysis approach

with results obtained by simulation of an X-ray system. The results indicate that the execution time of our analysis algorithm is lower, while the results are exact. Furthermore, we study the effects of parameters changes on the end-to-end latency distribution. The results show that parameter changes can have non monotonic effects on the latency distribution and that these effects can be substantial. Therefore, it can be concluded that the presented analysis algorithm is an attractive option for the exploration of different system configurations. With this exploration, system parameters can be found for which the latency distribution adheres to the temporal requirements.

CHAPTER 6

Conclusions

In this chapter we first summarize the results from the previous chapters and then draw some conclusions from them.

The outline of this chapter is as follows: In Section 6.1 we summarize the results of reducing jitter of a single application on a COTS system. Section 6.2 summarizes the results of how the cache hierarchy influences the performance of other applications and how to reduce that influence. The conclusions of the technique to analyze time triggered systems with asynchronous communication and phasing are summarized in Section 6.3. In Section 6.5 we present our contributions. In the last section we discuss some interesting directions for future work.

6.1 Jitter reduction on CMP-systems

In Chapter 3 we have presented several heuristics for the scheduling of static streaming applications on a general purpose multiprocessor system (one-to-one, many-to-one, many-to-one cache aware and many-to-one cache aware reduced). The scheduling heuristics are intended to reduce the variation in the execution times of the tasks and thereby the jitter of a streaming processing application. Furthermore, it is desirable that these heuristics reduce the end-to-end latency of the application. The scheduling heuristics have been evaluated using a quad core SMP Intel machine.

From the experimentally obtained results we observed that, if the scheduler in the OS is given the maximum scheduling freedom, the variation

of the end-to-end latency, i.e. the jitter, can be quite large. When the freedom of the OS scheduler is reduced by using at compile time computed static-order schedules, the jitter is reduced drastically. Our experimental results indicate that the jitter is reduced by roughly 90% by making use of the so-called fixed, predictable and reduced memory and task scheduling techniques. This technique maps tasks to cores in such a way that the locality of references is improved, scheduling overhead is minimized and synchronization points are reduced. Another interesting result is that once the influence of the OS scheduler has been removed, the other proposed techniques (see Chapter 3) hardly have an effect on the jitter anymore (but they do have an effect on latency). This was a surprising result because more influence of the cache on the jitter was expected.

Furthermore, from the experimentally obtained results we observe that the average latency is predominantly dependent on the amount of memory that is used at any point in time during the execution of the application. The reuse memory allocation technique decreases the amount of memory that is alive because it takes care that used memory allocations are reused within one iteration. In the case that the memory that is alive fits in the cache (i.e., good locality of reference), most of the memory accesses will result in cache hits so that the end-to-end latency as well as the jitter is reduced significantly.

In our experiments we found that, using cache aware scheduling techniques that reduce the memory footprint, also the average latency is reduced by roughly 60% compared to the case that the memory buffers do not fit into the cache.

The fixed, predictable and reduce scheduling heuristics further improved the locality of the memory accesses, such that more – ideally all – accessed data fits in the cache and that more data is closer to the core that accesses the data. As expected, this resulted in a further decrease of the end-to-end latency while the jitter remained roughly the same.

Given these observations we conclude that a reduction of the scheduling freedom of the operating system scheduler, by applying scheduling heuristics, can reduce the latency and jitter of stream processing applications significantly, and can therefore be a valuable technique for the design of e.g. medical image processing applications that are executed on general purpose multiprocessor systems.

6.2 Jitter reduction in the cache hierarchy

In Chapter 4 we showed how applications influence the performance of each other due to sharing of resources, e.g. memory ports in the cache hierarchy itself. We have shown that best effort applications can cause, due to bandwidth congestion in the cache hierarchy, a high amount of interference, which can result in a large increase of the latency of real-time applications on CMP architectures. This increase of latency makes it highly undesirable to execute best effort applications concurrently with real-time applications.

We therefore proposed a duty-cycling technique that can throttle (duty-cycle) best effort applications, when their bandwidth consumption causes too much interference with the real-time applications. The duty-cycling technique is implemented in the Linux operating system and uses performance counters available in the cores of the CMP to estimate the bandwidth usage of the best effort applications. The duty-cycling technique uses periodic sampling of these performance counters in order to estimate bandwidth usage. Furthermore, it uses a simple control mechanism in order to reduce bandwidth congestion.

From our experimental results we conclude that the proposed duty-cycling technique can significantly reduce the interference between best effort and real-time applications. For an industrial X-ray imaging application we showed that the latency increase that is caused by the interference can be controlled while maintaining some performance for best effort applications. Using synthetic benchmarks we showed that there are scenarios where the duty-cycling technique can reduce the latency degradation by 90%.

We therefore conclude that duty-cycling of best effort applications based on measured bandwidth consumption is an essential method to control the interference in CMP architectures that are used for the execution of real-time streaming applications in combination with best effort applications, such as the current interventional X-ray system.

6.3 Distribution Analysis for Probabilistic Time-Triggered Systems

In Chapter 5 we introduced the Probabilistic Time Triggered System (PTTS) model that can, given measured ETPs, analyze the end-to-end latency in a time-triggered stream processing system, under the assumption that the measured ETPs are correct. The PTTS model can be used for example to

model a medical image processing system. With this model we can efficiently compute the distribution of the end-to-end latency of the system, and show the consequences of using time-triggered interfaces that are not synchronized.

The components in this PTTS model are characterized by three parameters, frequency, offset and latency. The latency of a component is characterized by a Execution-Time Profile (ETP). The frequency describes at what rate the components are triggered and the offset the phase difference of the clock signals at time 0. The ETP is a Probability Mass Function (PMF) that can be obtained by measuring each component in isolation given a representative trace of input data. The components allow auto-concurrency internally and the produced data is stored in a container, where each production overwrites the previous stored container. This implies that under certain circumstances some data might be lost if it is overwritten before it is read.

Efficient computation of the latency distribution is enabled by the structure of the PTTS model. This structure makes it possible to consider only the triggerings of the components in one hyper period. The hyper period is defined as the period where the system returns to its initial state. This enabled us to derive a bound on the computational complexity of the presented analysis algorithm. Furthermore, the computational complexity of our algorithm is reduced by considering intervals of latency of the components during analysis. This differs from simulation of such a system where in a simulation run only one of the possible latencies that a component introduces is evaluated.

The computation time of the latency distribution might become unacceptably large if the hyper-period is extremely large. To handle these cases efficiently, we showed that for some practically relevant system configurations, the exact latency distribution of the end-to-end latency can be computed without visiting the complete state space of a part of the system. This reduces the state space that needs to be considered for the remaining part of the system and thus reduces the overall computation time.

We demonstrated the applicability of the PTTS model by exploring the consequences of different system configurations of an X-ray system. These configurations are modeled by changing the parameters in the PTTS model. This makes the consequences of implementation decisions explicit and allows the selection of an option that meets the temporal requirements of the system.

6.4 Overall

Firstly, we have devised and implemented techniques to run a single real-time image processing application on a dedicated system with a general purpose CMP. However, for several reasons such as reliability, logistics and cost it is undesirable to need a separate system for the real-time image processing. We have therefore also devised a technique that allows the integration of a real-time image processing application with several best-effort applications. So the real-time image processing application can now be run with high timing reproducibility and can even be integrated with some additional applications. This allows us to take a wider perspective and and examine the real-time image processing application in the wider context of the complete system. I.e., we can examine the end-to-end latency of the complete system of which the image processing application is only one part. With the demonstrated PTTS model we can model most of the COTS components that compromise the complete system and derive end-to-end latency distributions. This model and analysis techniques can then be used to explore the design alternatives and validate the system.

6.5 Contribution

In this section we summarize our contributions.

Contribution 1 We have identified the hardware and software components that contribute to the uncertainty of the performance of static image processing applications on commodity CMP systems.

Contribution 2 We describe several techniques to improve the reproducibility of a single real-time streaming image processing application on commodity CMP systems by adapting the scheduling of tasks and by adapting the memory allocation. Most importantly, we show that although the architecture that is used to execute the real-time applications is inherently unpredictable, we can still achieve high timing reproducibility in our use case. So on this unpredictable hardware architecture the software becomes responsible for maintaining the conditions in which the system as a whole remains predictable and the timing of the real-time applications reproducible. This makes the design and implementation of the system more complex, but we have also shown several techniques that can be used to create condition in which the system becomes more predictable.

Contribution 3 The techniques described in contributed 2 have been implemented in a streaming compiler. The streaming compiler can transform a high level description of a image processing chain into a real application. This application contains all the low level details on how to execute the image processing chain, including data parallelism instantiations, thread mapping, memory allocation, and synchronization. The compiler contains all the different techniques described in this thesis so it can be used to measure the performance and reproducibility of many image processing chains. Furthermore, a synthetic image processing chain generator has been implemented and used to verify the validity of the techniques on numerous different topologies. Using this streaming compiler it is also possible to tune the realized application for a certain system. For example, by changing the amount of threads that are instantiated which impact how the data parallelism is instantiated. This makes it possible to quickly evaluate new processors with a different amount of core or cache sizes.

Contribution 4 An approach was developed to reduce the interference, when running multiple applications on a single commodity CMP system by throttling tasks in order to reduce interference due to memory bandwidth contention. A reduction of the interference between applications is achieved by extending the scheduler of the Linux kernel with a budget enforcement mechanism.

Contribution 5 We have created a model that can be used to reduce the latency in a system by adapting the phase difference of the clock signals of time-triggered components. A computationally efficient probabilistic analysis approach for the derivation of the end-to-end latency distribution for real-time stream processing systems was developed to perform the required analysis. The systems considered are composed of subsystems with time-triggered interfaces.

Contribution 6 We implemented the model and analysis technique from contribution 5 in a tool. This analysis techniques together with the streaming compiler from contribution 3 allowed us to design and implement a soft real-time system that has acceptable performance characteristics in many practical situations.

6.6 Directions for future work

In this section we suggest interesting directions for future work for each of the topics addressed in this thesis.

6.6.1 JITTER REDUCTION ON CMP SYSTEMS

New micro architectures from Intel have features that can be used to improve timing reproducibility. For example, in the new Intel micro architectures a technique which is called CAT [13, 17], has been implemented. This technique allows the developer to partition the cache so that the applications can be mapped onto certain parts of the cache. This reduces evictions in the cache hierarchy and reduces jitter.

6.6.2 JITTER REDUCTION IN THE CACHE HIERARCHY

The presented throttling technique samples the performance counter at a fixed rate in order to estimate the actual bandwidth used in the cache hierarchy. Two improvements that can decrease overhead and improve best effort performance are the use of so-called event based bandwidth estimation and proportional suspend times.

The event based bandwidth estimation can be implemented by allowing the performance counters to make an interrupt request, when a certain threshold is exceeded. This reduces the number of times that the bandwidth is measured in applications with a low bandwidth usage.

The second improvement could be proportional suspend times instead of fixed suspend times. The amount of times the best effort application has exceeded bandwidth usage and by how much could be used to determine the length of the interval that a task should be suspended. This allows the duty-cycling technique to have a finer control over when and how much the best effort should be duty-cycled and thereby increasing best effort performance without violating real-time constraints.

6.7 OUTLOOK

The outlook of running soft real-time image processing applications on commodity CMP systems is good. Since the first multi-processors on commodity hardware became available, a large number of improvements have made them more usable for any kind of concurrent workloads, and not only for soft real-time image processing applications.

For example, for a long time most major OSes try to reduce thread migrations as much as possible, when scheduling threads on cores. This is desirable because the experiments in Chapter 3 showed that migration between cores increases latency and to a minor degree jitter.

On the hardware side there are even more improvements that could in-

crease overall performance and reduce jitter. The first multi processor systems implemented on commodity hardware were not even CMP systems, but just had multiple single core processors in different sockets. Subsequently, those processors were integrated in a single package, but still on separate dies. The main disadvantage of these solutions was that the bandwidth between the processors was relatively small and more resources such as the memory controller were shared. This resulted in large jitter, when applications were executed on multiple cores concurrently. Therefore, these systems were better suited for running multiple independent applications.

One important component that this solution lacked was a single coherent shared last level cache. The Intel Nehalem architecture was the first commodity CMP system that had a last level cache integrated with the cores on the same die. Furthermore a superior interconnect was used to connect the chip to other peripherals (for example memory controller) and other sockets (on multi socket systems). All these techniques were combined into a system that is better capable to correctly run soft real-time image processing applications as we have demonstrated in this thesis.

Recently some new techniques have been introduced that allow software designers to influence how the processors use the cache memory with a so-called CAT [13, 17]. These techniques will further reduce jitter and increase the number of applications with soft real-time requirements that can be run correctly on commodity CMP systems. Furthermore, the Intel Skylake family introduced changes inside the cache hierarchy. Due to an increasing core count, since the introduction of the three level cache hierarchy in the Nehalem architecture, the per core bandwidth steadily decreased. Intel therefore adopted a mesh architecture into the last level of the cache hierarchy in order to decrease this bottleneck [34]. It would be interesting to see how good the timing reproducibility of these new processors are. Due to a larger distance between the core of these new systems it is likely that the presented thread scheduling techniques can be extended to also take this distance into account in order to improve timing reproducibility.

Another interesting development is that companies such as Google are also starting to get interested in reducing the contention between applications on commodity CMP systems. In [31] techniques similar to the technique examined in Chapter 4 are evaluated in order to increase the workload of systems by increasing the number of best effort applications on a single system, although they are latency critical. So although the requirements are not as strict, they have similar problems as addressed

in this thesis, when implementing soft real-time image processing applications on commodity CMP systems.

The main advantage for the soft real-time community is that large processor manufactures suddenly have more customers that are interested in more than just best case performance. However, these new techniques do not automatically make a commodity CMP system more predictable. It just gives the software engineer a bigger toolbox to control the system and transform it into a system that can correctly run soft real-time applications.

In this thesis a number of software techniques have been introduced that improve the soft real-time reproducibility of applications executed on available CMP systems. We showed that the techniques can significantly improve the best effort performance so that these system become a viable option for medical image processing. We expect that a further improvement of the soft real-time reproducibility can be achieved by exploiting the above described, recently introduced hardware features in these systems. Furthermore, we expect that with these additions CMP systems will become an attractive option for a larger class of soft real-time applications.

APPENDIX A

Simple Streaming Compiler

In this chapter we will briefly introduce the Simple Streaming Compiling (SSC). The SSC is used to compile several of the experiments in this thesis. The SSC normally generates an instantiated description of the high level description that can be executed with an interpreter. The SSC can also output code that can be compiled in a real executable or into a Synchronous Dataflow Graph (SDFG) that can be analyzed with other analysis tools.

A.1 The compiler stages

The stages in the compiler have been described in Section 3.4.

A.2 Compiler options

The following listing gives an overview of the options of the SSC. The experiments in Chapter 3 and Chapter 4 used the tool the generate several realization of the same streaming application.

```
scc [OPTIONS] file

 COMPOSITION
      -gd  | --generatedeclaration
      -gw  | --generatewcet
      -cr  | --composeresolution w,h

 DECLARATION
      -dp  | --dataparallelism (none | some | full)
      -mdp | --maxdataparallelism number
      -gpp | --generatepreciscedataparallelism [experimental]

 INSTANTIATION
```

```
      -ts  | --threadschedule (simple | (fixed | predictable | reduced) number))
      -wt  | --weightedtokens true|false
      -ms  | --memoryschedule (none |
               naive | reuse | mesif | interference)
      -te  | --threadexecution
      -cd  | --coarsedepencies

RUNTIME
      -ab  | --addbarrier
      -ai  | --additerationsync
      -cm  | --coremask (core(, core)*)
      -ca  | --coreaffinity
      -ks  | --keepsynced
      -ki  | --keepsyncedinterval
      -kd  | --keepsynceddelta
      -dm  | --dynamicmemoryallocation (dynamic | reduced)
      -ss  | --sortsignals
      -st  | --sortsignalstwice

EXPORT
      -dg  | --declarationgraph file
      -ig  | --instantationgraph file
      -rg  | --runtimegraph file
      -xr  | --xmlsdfruntime
      -gg  | --groovegraph file
      -ni  | --normalizedinput file

CODE GENERATION
      -ht  | --hugetables
      -vo  | --verboseoutput
      -o   | --output file
      -od  | --outputdir dir
      -ic  | --iterationcount number
      -t   | --target (linux | helix | wcet | interpreter | info)

PERFORMANCE MEASUREMENTS
      -ta  | --inserttimersall
      -tt  | --threadtimers
      -td  | --typetimerdump
      -dt  | --dumptimer filename

 MISC
      -v   | --verbose
      -h   | --help

 ANALYSIS
      -pr  | --sdfruntime
      -sl  | --sdfas log filename
```

A.3 Example file

The following listing gives an example of streaming application and a possible realization of the SSC. The listing is divided in sections. The sections *composition* and *declaration* describe the stream application. In this example several simple image processing algorithms have been chained together to create a simple stream processing application. The section *instantiation* introduces parallelism. Lastly, the section *runtime* binds the parallelized tasks onto the hardware (threads and memory) and instantiates the necessary synchronization.

```
composition{
  container_declaration source {
    out OUT int mul [1, 1];
  };
```

```
  container_declaration sink {
    in IN int mul [1, 1];
  };

  container_declaration gain {
    in IN int mul [1, 1];
    out OUT int mul [1, 1];
  };

  container_declaration conv {
    in IN int mul [1, 1] [4, 4];
    out OUT int mul [1, 1];
  };

  container_declaration app {
  };

  container_bind source func_source NONE,
                 sink func_sink NONE,
                 conv func_conv FULL,

  container_composition app{
    container tgain gain index,
              tconv conv index,
              tsource source index,
              tsink sink index;

    tsource.out -> tgain.in;
    tgain.out -> tconv.in;
    tconv.out -> tsink.in;
  };

  compose app [256, 256];
}

declaration {
  function_declaration func_source NONE {
    out OUT int mul [1, 1];
  };

  function_declaration func_sink NONE {
    in IN int mul [1, 1];
  };

  function_declaration func_conv FULL {
    in IN int mul [1, 1] [4, 4];
    out OUT int mul [1, 1];
  };

  function_declaration func_gain FULL {
    in IN int mul [1, 1];
    out OUT int mul [1, 1];
  };

  function f6 func_gain [256, 256],
           f7 func_conv [256, 256],
           f8 func_source [256, 256],
           f9 func_sink [256, 256];

  f8.out -> f6.in;
  f6.out -> f7.in;
  f7.out -> f9.in;
}

instantiation{
  function_instantiation f6.0 [0, 0]:[256, 128], f6.1 [0, 128]:[256, 256],
                         f7.0 [0, 0]:[256, 128], f7.1 [0, 128]:[256, 256],
                         f8.0 [0, 0]:[256, 256], f9.0 [0, 0]:[256, 256];

  f8.0 => f6.0;
  f8.0 => f6.1;
```

```
  f6.0 => f7.0;
  f6.0 => f7.1;
  f6.1 => f7.0;
  f6.1 => f7.1;
  f7.0 => f9.0;
  f7.1 => f9.0;
}

runtime{
  timing func_source 1300, func_dup 1400, func_plus 560, func_gain 560,
         func_copy 560, func_avg 560, func_conv 100, func_sink 1;

  timing_system wt 1, sg 1;

  thread_schedule thread0 [f8.0, f6.0, f7.0, f9.0], thread1 [f6.1, f7.1];

  memory_mapping  m0 shared 0 262144,
                  m2 shared 262144 524288,
                  m4 shared 524288 786432;

  memory_binding m2 [f7.in, f6.out], m4 [f9.in, f7.out], m0 [f8.out, f6.in];

  memory_schedule ms2 [m2], ms4 [m4], ms0 [m0];

  thread_execution thread0 [W s1, E f8.0, S s3, W s5, E f6.0, S s7, W s8,
                            E f7.0, S s10, W s12, E f9.0, S s13],
                   thread1 [W s3, W s10, E f6.1, S s1, S s8, W s7, W s13,
                            E f7.1, S s5, S s12];

  signal ! s1, s3, ! s5, s7, s8, ! s10, s12, ! s13;
}
```

Acronyms

API	Application Programming Interface
ASIC	Application Specific Integrated Circuit
CCD	Charge-Coupled Device
CAT	Cache Allocation Technology
CMP	Chip Multi-Processor
COTS	Commercial Off The Shelf
CPU	Central Processing Unit
CSDF	Cyclo-Static Dataflow
DSP	Digital Signal Processor
DVI	Digital Visual Interface
ETP	Execution-Time Profile
FPGA	Field-Programmable Gate Array
FPS	Frames per second
FSB	Front-Side Bus
GPP	General Purpose Processor
GPU	Graphical Processing Unit
GUI	Graphical User Interface
HSDF	Homogeneous Synchronous Dataflow
HSDFG	Homogeneous Synchronous Dataflow Graph
LCM	Least Common Multiple

MMU	Memory Management Unit
OS	Operating System
PDF	Probability Density Function
PMF	Probability Mass Function
PTTS	Probabilistic Time Triggered System
QoS	Quality of Service
SADF	Scenario Aware Data Flow
SDFG	Synchronous Dataflow Graph
SDF	Synchronous Data Flow
SMP	Symmetric Multiprocessing
SMT	Simultaneous Multi-Threading
SOS	Static Order Schedule
SSC	Simple Streaming Compiling
TLB	Translation Lookaside Buffer
PTLTS	Probabilistic Timed Labeled Transition System
VPM	Virtual Private Machine
WCET	Worst Case Execution Time

Bibliography

[1] Rob Albers, Eric Suijs, and Peter H. N. de With. Optimization model for memory bandwidth usage in X-ray image enhancement. In *SPIE Electronic Imaging*, pages 6811–04, 2008.

[2] James H. Anderson, John M. Calandrino, and UmaMaheswari C. Devi. Real-time scheduling on multicore platforms. pages 179–190, 2006. doi: 10.1109/RTAS.2006.35. URL `http://portal.acm.org/citation.cfm?id=1128017.1128438`.

[3] G. Bernat, A. Colin, and S.M. Petters. WCET Analysis of Probabilistic Hard Real-time Systems. In *Real-Time Systems Symposium, 2002. RTSS 2002. 23rd IEEE*, pages 279–288, 2002. doi: 10.1109/REAL.2002.1181582.

[4] G. Butazzo and J. Stankovic. Red: Robust earliest deadline scheduling. In *Proceedings of The Third International Workshop on Responsive Computing Systems*, 1993.

[5] G. J. Chaitin. Register allocation & spilling via graph coloring. *SIGPLAN Not.*, 17:98–101, June 1982. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/872726.806984. URL `http://doi.acm.org/10.1145/872726.806984`.

[6] Samarjit Chakraborty, Tulika Mitra, Abhik Roychoudhury, and Lothar Thiele. Cache-aware timing analysis of streaming applications. *Real-Time Syst.*, 41:52–85, January 2009. ISSN 0922-6443. doi: 10.1007/s11241-008-9062-5. URL `http://portal.acm.org/citation.cfm?id=1485069.1485080`.

[7] J. Charles, P. Jassi, N.S. Ananth, A. Sadat, and A. Fedorova. Evaluation of the intel core i7 turbo boost feature. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 188–197, 2009. doi: 10.1109/IISWC.2009.5306782.

[8] Derek Chiou, Derek Chiouy, Larry Rudolph, Larry Rudolphy, Srinivas Devadas, Srinivas Devadasy, Boon S. Ang, and Boon S. Angz. Dynamic cache partitioning via columnization. 2000.

[9] Leonardo Dagum and Ramesh Menon. Openmp: an industry standard api for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998.

[10] M.G. Gouda, Y.W. Han, E.D. Jensen, W.D. Johson, and R.Y. Kain. Radar scheduling: Section 1, the scheduling problem. In *Distributed Data Processing Technology, Applications of DDP Technology to BMD: Architectures and Algorithms, volume IV, chapter 3*, 1977.

[11] Andreas Hansson, Kees Goossens, Marco Bekooij, and Jos Huisken. Compsoc: A template for composable and predictable multi-processor system on chips. *ACM Trans. Des. Autom. Electron. Syst.*, 14(1):2:1–2:24, January 2009. ISSN 1084-4309. doi: 10.1145/1455229.1455231. URL `http://doi.acm.org/10.1145/1455229.1455231`.

[12] Thomas A. Henzinger, Benjamin Horowitz, and Christoph Meyer Kirsch. Embedded Control Systems Development with Giotto. In *Proceedings of LCTES 2001*, pages 64–72. Press, 2001.

[13] Intel. Cache monitoring technology and cache allocation technology. URL `http://www.intel.com/content/www/us/en/communications/cache-monitoring-cache-allocation-technologies.html`. retrieved: 2016-12-22.

[14] Intel. Intel quickpath architecture, April 2011. URL `http://www.intel.com/technology/quickpath/whitepaper.pdf`.

[15] Intel. Smart cache, 2011. URL `http://www.intel.com`.

[16] Intel. Turbo boost, April 2011. URL `http://www.intel.com`.

[17] INTEL. Volume 3b: System programming guide, part 2. In *Intel 64 and IA-32 Architectures Software Developer's Manual*, 2016.

[18] Ravi Iyer. CQoS: a framework for enabling qos in shared caches of cmp platforms. In *Proceedings of the 18th annual international conference on Supercomputing*, ICS '04, pages 257–266, New York, NY, USA, 2004. ACM. ISBN 1-58113-839-3.

[19] Hari Kannan, Fei Guo, Li Zhao, Ramesh Illikkal, Ravi Iyer, Don Newell, Yan Solihin, and Christos Kozyrakis. From chaos to QoS: case studies in CMP resource management. *SIGARCH Comput. Archit. News*, 35(1): 21–30, March 2007. ISSN 0163-5964.

[20] D. Kartson, G. Balbo, S. Donatelli, G. Franceschinis, and Giuseppe Conte. *Modelling with Generalized Stochastic Petri Nets*. John Wiley & Sons, Inc., 1994. ISBN 0471930598.

[21] Seongbeom Kim, Dhruba Chandra, and Yan Solihin. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, PACT '04, pages 111–122, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2229-7.

[22] Seongbeom Kim, Dhruba Chandra, and Yan Solihin. Fair cache sharing and partitioning in a chip multiprocessor architecture. pages 111–122, 2004. doi: http://dx.doi.org/10.1109/PACT.2004.15. URL `http://dx.doi.org/10.1109/PACT.2004.15`.

[23] H. Kopetz. The Systematic Design of Large Real-time Systems or Interface Simplicity. In Michel BanÃćtre and PeterA. Lee, editors, *Hardware and Software Architectures for Fault Tolerance*, volume 774 of *Lecture Notes in Computer Science*, pages 250–262. Springer Berlin Heidelberg, 1994. ISBN 978-3-540-57767-6. doi: 10.1007/BFb0020039. URL `http://dx.doi.org/10.1007/BFb0020039`.

[24] H. Kopetz and R. Obermaisser. Temporal composability [real-time embedded systems]. *Computing Control Engineering Journal*, 13(4):156–162, Aug 2002. ISSN 0956-3385. doi: 10.1049/cce:20020401.

[25] Hermann Kopetz. Event-triggered versus time-triggered real-time systems. Research Report 8/1991, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-1, 1040 Vienna, Austria, 1991.

[26] D. Koufaty and D.T. Marr. Hyperthreading technology in the netburst microarchitecture. *Micro, IEEE*, 23(2):56 – 65, march-april 2003. ISSN 0272-1732. doi: 10.1109/MM.2003.1196115.

[27] E. A. Lee. Cyber physical systems: Design challenges. In *2008 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)*, pages 363–369, May 2008. doi: 10.1109/ISORC.2008.25.

[28] E. A. Lee and D. G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, Sept 1987. ISSN 0018-9219. doi: 10.1109/PROC.1987.13876.

[29] Edward A. Lee. Computing needs time. Technical Report UCB/EECS-2009-30, EECS Department, University of California, Berkeley, Feb 2009. URL `http://www2.eecs.berkeley.`

edu/Pubs/TechRpts/2009/EECS-2009-30.html. — See also the Published Version, Communications of the ACM, 52(5), pp. 70-79, May 2009 —.

[30] Edward Ashford Lee and David G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Trans. Comput.*, 36:24–35, January 1987. ISSN 0018-9340. doi: http://dx.doi.org/10.1109/TC.1987.5009446. URL http://dx.doi.org/10.1109/TC.1987.5009446.

[31] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. Heracles: Improving resource efficiency at scale. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, pages 450–462, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3402-0. doi: 10.1145/2749469.2749475. URL http://doi.acm.org/10.1145/2749469.2749475.

[32] Daniel Molka, Daniel Hackenberg, Robert Schone, and Matthias S. Muller. Memory performance and cache coherency effects on an Intel Nehalem multiprocessor system. In *Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques*, pages 261–270, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-0-7695-3771-9. doi: 10.1109/PACT.2009.22. URL http://portal.acm.org/citation.cfm?id=1636712.1637764.

[33] Ingo Molnar. Config_preempt_rt patch set. URL https://rt.wiki.kernel.org/index.php/CONFIG_PREEMPT_RT_Patch.

[34] David Mulnix. IntelÂő xeonÂő processor scalable family technical overview, 2017. URL https://software.intel.com/en-us/articles/intel-xeon-processor-scalable-family-technical-overview.

[35] K.J. Nesbit, M. Moreto, F.J. Cazorla, A. Ramirez, M. Valero, and J.E. Smith. Multicore resource management. *Micro, IEEE*, 28(3):6 –16, may-june 2008. ISSN 0272-1732.

[36] Dejan Novaković, Nedeljko Vasić, Stanko Novaković, Dejan Kostić, and Ricardo Bianchini. Deepdive: Transparently identifying and managing performance interference in virtualized environments. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, USENIX ATC'13, pages 219–230, Berkeley, CA, USA, 2013. USENIX Association. URL http://dl.acm.org/citation.cfm?id=2535461.2535489.

[37] T. M. Parks, J. L. Pino, and E. A. Lee. A comparison of synchronous and cycle-static dataflow. In *Conference Record of The Twenty-Ninth Asilomar Conference on Signals, Systems and Computers*, volume 1, pages 204–210 vol.1, Oct 1995. doi: 10.1109/ACSSC.1995.540541.

[38] R. Pellizzoni and H. Yun. Memory servers for multicore systems. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 1–12, April 2016. doi: 10.1109/RTAS.2016.7461339.

[39] Y. Pribadi, J. P. M. Voeten, and B. D. Theelen. Reducing Markov Chains for Performance Evaluation. In *In Proceedings of the 2nd workshop on Embedded Systems*, pages 173–179, 2001.

[40] Simon Schliecker, Mircea Negrean, and Rolf Ernst. Bounding the shared resource load for the performance analysis of multiprocessor systems. *Proc. of Design, Automation, and Test in Europe (DATE)*, March 2010.

[41] S. Sriram and S.S. Bhattacharyya. *Embedded Multiprocessors: Scheduling and Synchronization, Second Edition*. Signal Processing and Communications. CRC Press, 2009. ISBN 9781420048025. URL `https://books.google.nl/books?id=v13bnBCKJLEC`.

[42] H.S. Stone, J. Turek, and J.L. Wolf. Optimal partitioning of cache memory. *Computers, IEEE Transactions on*, 41(9):1054 –1068, sep 1992. ISSN 0018-9340.

[43] Vivy Suhendra and Tulika Mitra. Exploring locking & partitioning for predictable shared caches on multi-cores. In *Proceedings of the 45th annual Design Automation Conference*, DAC '08, pages 300–303, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-115-6. doi: http://doi.acm.org/10.1145/1391469.1391545. URL `http://doi.acm.org/10.1145/1391469.1391545`.

[44] B.D. Theelen, J.P.M. Voeten, and R.D.J. Kramer. Performance modelling of a network processor using POOSL. *Computer Networks*, 41 (5):667 – 684, 2003. ISSN 1389-1286. doi: http://dx.doi.org/10.1016/S1389-1286(02)00455-3. URL `http://www.sciencedirect.com/science/article/pii/S1389128602004553`.

[45] B.D. Theelen, M.C.W. Geilen, T. Basten, J.P.M. Voeten, S.V. Gheorghita, and S. Stuijk. A scenario-aware data flow model for combined long-run average and worst-case performance analysis. In *Fourth ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2006. MEMOCODE '06. Proceedings.*, pages 185–194, July 2006. doi: 10.1109/MEMCOD.2006.1695924.

[46] B.D. Theelen, O. Florescu, M.C.W. Geilen, J. Huang, P.H.A. van der Putten, and J.P.M. Voeten. Software/hardware engineering with the parallel object-oriented specification language. In *Formal Methods and Models for Codesign, 2007. MEMOCODE 2007. 5th IEEE/ACM International Conference on*, pages 139–148, May 2007. doi: 10.1109/MEMCOD.2007.371231.

[47] B.D. Theelen, O. Florescu, M.C.W. Geilen, J. Huang, P.H.A. van der Putten, and J.P.M. Voeten. Software/hardware engineering with the parallel object-oriented specification language. In *2007 5th IEEE/ACM International Conference on Formal Methods and Models for Codesign (MEMOCODE 2007)*, pages 139–148, May 2007. doi: 10.1109/MEMCOD.2007.371231.

[48] Stephan Thesing. *Safe and Precise WCET Determination by Abstract Interpretation of Pipeline Models*. PhD thesis, Universität des Saarlandes, Postfach 151141, 66041 Saarbrücken, 2004. URL `http://scidok.sulb.uni-saarland.de/volltexte/2005/466`.

[49] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. Simultaneous multithreading: maximizing on-chip parallelism. In *ISCA '95: Proceedings of the 22nd annual international symposium on Computer architecture*, pages 392–403, New York, NY, USA, 1995. ACM. ISBN 0-89791-698-0. doi: http://doi.acm.org/10.1145/223982.224449.

[50] Ubuntu. Ubuntu, July 2011. URL `http://www.ubuntu.com`.

[51] Leo J. van Bokhoven. *Constructive Tool Design for Formal Languages: From Semantics to Executing Models*. PhD thesis, Eindhoven University of Technology, 2002.

[52] Ben Verghese, Anoop Gupta, and Mendel Rosenblum. Performance isolation: sharing and isolation in shared-memory multiprocessors. In *Proceedings of the eighth international conference on Architectural support for programming languages and operating systems*, ASPLOS-VIII, pages 181–192, New York, NY, USA, 1998. ACM. ISBN 1-58113-107-0.

[53] Jeroen Voeten, Marc Geilen, Leo Van Bokhoven, Piet Van Der Putten, and Mario Stevens. A probabilistic real-time calculus for performance evaluation, 1999.

[54] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The worst-case execution time problem—overview of methods and survey of tools. *ACM*

Transactions on Embedded Computing Systems (TECS), 7(3), 2008. doi: 10.1145/1347375.1347389.

[55] Reinhard Wilhelm, Daniel Grund, Jan Reineke, Marc Schlickling, Markus Pister, and Christian Ferdinand. Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems. *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, 28:966–978, July 2009. ISSN 0278-0070. doi: 10.1109/TCAD.2009.2013287. URL `http://portal.acm.org/citation.cfm?id=1669804.1669808`.

[56] Jun Yan and Wei Zhang. WCET analysis for multi-core processors with shared l2 instruction caches. *Proceedings of the 2008 IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 80–89, 2008. doi: 10.1109/RTAS.2008.6. URL `http://portal.acm.org/citation.cfm?id=1440456.1440579`.

List of publications

Refereed

[MW:1] M. Westmijze, M.J.G. Bekooij, G.J.M. Smit, and M. Schrijver. Evaluation of scheduling heuristics for jitter reduction of real-time streaming applications on multi-core general purpose hardware. In *Embedded Systems for Real-Time Multimedia (ESTIMedia), 2011 9th IEEE Symposium on*, pages 140 –146, oct. 2011.

[MW:2] M. Westmijze, Marco Jan Gerrit Bekooij, and Gerardus Johannes Maria Smit. Interference control by best-effort process duty-cycling in chip multi-processor systems for real-time medical image processing. In *The Fifth International Conference on Resource Intensive Applications and Services, INTENSIVE 2013*. International Academy, Research, and Industry Association, 2013.

[MW:3] M. Westmijze, M. J. G. Bekooij, and G. J. M. Smit. Efficient end-to-end latency distribution analysis for probabilistic time-triggered systems. In *2014 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIV)*, pages 290–298, July 2014.

[MW:4] Flavius Gruian and Mark Westmijze. Investigating hardware micro-instruction folding in a java embedded processor. In *Java Technologies for Real-Time and Embedded Systems*, pages 102–108, 2010.

[MW:5] Flavius Gruian and Mark Westmijze. Vhdl vs. bluespec system verilog: a case study on a java embedded architecture. In *Proceedings of the 2008 ACM symposium on Applied computing*, pages 1492–1497, 2008.

[MW:6] Flavius Gruian and Mark Westmijze. Bluejep: a flexible and high-performance java embedded processor. In *Java Technologies for Real-Time and Embedded Systems*, pages 222–229, 2007.

[MW:7] Flavius Gruian and Mark Westmijze. Bluejamm: A bluespec embedded java architecture with memory management. In *Symbolic and Numeric Algorithms for Scientific Computing, 2007*, pages 459–466, 2007.

Non-refereed

[MW:8] P. T. Wolkotte, J. H. Rutgers, P. K. F. Hölzenspies, M. Westmijze, R. Blumink, and G. J. M. Smit. An automated design-flow for FPGA-based sequential simulation. In *Proceedings of the 19th Annual Workshop on Circuits, Systems and Signal Processing (ProRISC), Veldhoven, The Netherlands*, number 2008/14935/STW, pages 126–132, Utrecht, November 2008. Technology Foundation STW.

Dankwoord

Allereerst wil ik beginnen om Marco Bekooij te bedanken. Marco was mijn wekelijkse begeleiding op de vakgroep en is mijn promotor. Zonder Marco zijn begeleiding na mijn vertrek uit de vakgroep, zou dit proefschrift er na al deze jaren niet gekomen zijn. Tijdens mijn periode als promovendus heb ik vaak geworsteld met de positie van mijn onderzoek tussen de harde theoretische wereld van de universiteit en de praktische wereld vanuit de industrie. Samen met Marco hebben we getracht deze werelden te combineren en daar is dit promotieonderzoek het resultaat van.

Naast de begeleiding van van Marco hebben ook Gerard Smit en Marc Schrijver vanuit Universiteit Twente en Philips Healthcare meegeholpen met het tot stand brengen van mijn onderzoek en thesis. Elk met hun eigen blik, vanuit de universiteit en industrie. Onmisbaar waren natuurlijk ook de secretaresses. Marlous, Nicole en Thelma, bedankt voor alle hulp.

Natuurlijk wil ook ook graag mijn oud collega's van de universiteit bedanken met in het bijzonder Philips Hölzenspies en Rinse Wester. Philip als kamergenoot en oud huisgenoot, en later ook Rinse als kamergenoot.

Ook verschillende collega's bij SVI en Thales wil bedanken voor het helpen mij te blijven motiveren om de thesis af te ronden. Het is erg leuk om nu in de industrie bezig te zijn en ook om te zien hoe het werk uit de wetenschap en uit deze thesis toepasbaar zijn in de praktijk.

Bob en Olav, bedankt dat jullie mijn paranimfen wilden zijn.

Ook mijn ouders wil ik bedanken voor de vele dagen oppassen op eerst Vesper en daarna ook Gustav er bij, zodat ik op dat moment kon werken aan deze thesis. Vesper en Gustav, het is fantastisch om te zien hoe jullie opgroeien.

In het bijzonder wil ik Astrid van Lier bedanken. Ook dankzij haar had ik de tijd en motivatie om de thesis af te ronden.

Mark Westmijze
Hilversum, juni 2018