Predictable embedded multiprocessor architecture for streaming applications

Predictable Embedded Multiprocessor Architecture for Streaming Applications

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de
Technische Universiteit Eindhoven, op gezag van
de rector magnificus, prof.dr.ir. C.J. van Duijn, voor
een commissie aangewezen door het College voor
Promoties in het openbaar te verdedigen
op maandag 15 juni 2009 om 16.00 uur

door

Arnold Joannes Maria Moonen

geboren te Weert

Dit proefschrift is goedgekeurd door de promotoren:

prof.dr.ir. R.H.J.M. Otten

en

prof.dr. H. Corporaal

Cover design: Seph Rademakers

Printed by: Universiteitsdrukkerij Technische Universiteit Eindhoven

A catalogue record is available from the Eindhoven University of Technology Library

ISBN: 978-90-386-1811-1

Abstract

Predictable Embedded Multiprocessor Architecture for Streaming Applications

The focus of this thesis is on embedded media systems that execute applications from the application domain *car infotainment*. These applications, which we refer to as jobs, typically fall in the class of streaming, i.e. they process on a stream of data. The jobs are executed on heterogeneous multiprocessor platforms, for performance and power efficiency reasons. Most of these jobs have firm real-time requirements, like throughput and end-to-end latency. Car-infotainment systems become increasingly more complex, due to an increase in the supported number of jobs and an increase of resource sharing. Therefore, it is hard to verify, for each job, that the real-time requirements are satisfied. To reduce the verification effort, we elaborate on an architecture for a *predictable system* from which we can verify, at design time, that the job's throughput and end-to-end latency requirements are satisfied.

This thesis introduces a network-based multiprocessor system that is predictable. This is achieved by starting with an architecture where processors have private local memories and execute tasks in a static order, so that the uncertainty in the temporal behaviour is minimised. As an interconnect, we use a network that supports guaranteed communication services so that it is guaranteed that data is delivered in time. The architecture is extended with shared local memories, run-time scheduling of tasks, and a memory hierarchy.

Dataflow modelling and analysis techniques are used for verification, because they allow cyclic data dependencies that influence the job's performance. Shown is how to construct a dataflow model from a job that is mapped onto our predictable multiprocessor platforms. This dataflow model takes into account: computation of tasks, communication between tasks, buffer capacities, and scheduling of shared resources. The job's throughput and end-to-end latency bounds are derived from a self-timed execution of the dataflow graph, by making use of existing dataflow-analysis techniques. It is shown that the derived bounds are tight, e.g. for our channel equaliser job, the accuracy of the derived throughput bound is within 10.1%. Furthermore, it is shown that the dataflow modelling and analysis techniques can be used despite the use of shared memories, run-time scheduling of tasks, and caches.

Acknowledgments

Writing this chapter means that my thesis formally comes to an end. This chapter allows me to show my gratitude to all people that directly or indirectly contributed to or supported my research.

Support for my research was provided by NXP semiconductors, who gave me the facility to conduct my research and supported my PhD financially from 2004 until 2008. In this period, I divided my working time between NXP Research in Eindhoven (System-On-Chip Architectures and Infrastructure group), NXP semiconductors in Nijmegen (Business-Line Car Infotainment), and Eindhoven University of Technology (Electronic Systems group). I have been very lucky to have, on one hand, the facilities of an excellent research environment, and, on the other hand, the possibility to put research ideas on trial within a development team.

I would like to thank Jef van Meerbergen, from Eindhoven University of Technology, for giving me the opportunity to work on the challenging research topic of predictable embedded multiprocessor architectures. I am very grateful for his encouragement and guidance throughout my PhD journey.

I also want to thank Ralph Otten for accepting to be promotor at my PhD defense and for his support as a group leader of the Electronic Systems group. My thanks also extends to Henk Corporaal as being my second promotor and for his valuable feedback during our discussions at the University. Furthermore, I owe much gratitude to all members of my PhD committee for reading my thesis, giving useful feedback and for participating in my defence session.

I am indebted to my supervisors Marco Bekooij, from NXP research, and René van den Berg, from NXP semiconductors, for their outstanding support, advice, and guidance in my daily work. With Marco, I had many brainstorm sessions and a lot of in-depth discussions that were very useful and from which I have learned a lot. I was greatly inspired by René who is a system architect within Business-Line Car Infotainment, and who participated in our technical discussions and gave feedback to my work. Besides being good supervisors, Marco and René are pleasant people to work with.

Within the System-On-Chip Architectures and Infrastructure group in Eindhoven, my research was part of the Hijdra project. Unfortunately, I cannot mention everyone who helped me, but I would like to mention the Hijdra members for the valuable discussions and for getting the opportunity to work in such a team of enthusiastic and talented researchers.

I also want to thank all people with whom I have worked within the Business-Line Car Infotainment, for creating a nice working environment, for supporting my research, and for giving constructive feedback. I really enjoyed being part of the soccer team *FC BION* for the sportive relaxation after work.

My thanks also extend to my colleagues at the Electronic Systems group, for the nice working environment. A special thanks goes out to the members of the PreMaDoNa project for the technical discussions and for providing valuable feedback on my research.

Finally, I wish to thank my family and friends. My parents have always believed in me and helped me to reach my goals. The love and encouragement of my girlfriend Esther allowed me to finish this journey. Finally, I would like to dedicate this work to my lost father, Wiel Moonen, who left us too soon.

Arno Moonen
April 2009

Contents

Chapter 1

Introduction

1.1 Car-infotainment domain

The target application domain, in this thesis, is the application domain from business-line *car-infotainment solutions* at NXP semiconductors. This business line has more than seventeen years of experience in designing digital signal processing chips for car radios.

A car is a uniquely challenging environment that combines entertainment (e.g. audio and video) with information (e.g. weather, news, and traffic information). We refer to this combination as *car infotainment*. The main application areas are: radio, audio, video, and navigation.

In the past, radio was only analog terrestrial radio reception, like Amplitude Modulation (AM), Frequency Modulation (FM), and Weather Band (WB). Currently, digital satellite radio and digital terrestrial radio are emerging rapidly. An example of digital satellite radio is Satellite Digital Audio Radio Service (SDARS). SDARS is a popular type of digital radio in the United States of America (USA) and it is operated by XM-Radio and Sirius. Examples of digital terrestrial radio are Digital Radio Mondiale (DRM), Digital Audio Broadcast (DAB), and Hybrid Digital (HD) radio.

Examples of audio processing are playback compressed and uncompressed audio, streaming audio from a portable media player, audio post processing for enhanced audio quality, and encoding audio to a storage device. Current car radios support various kinds of interfaces for connecting the car radio to portable media players, mobile phones, and storage devices. Furthermore, the number of supported compression formats is increasing rapidly. Examples of currently used compression formats are *MPEG-1 audio layer 3* (more commonly referred to as MP3) and Microsoft's *Windows Media Audio* (WMA).

Currently, video processing is mainly video playback for rear-seat entertainment. The video source can be a mass storage device (e.g. DVD or hard disk) or digital radio (e.g. SDARS). Finally, there is navigation processing with, for example, road access services and eSafety. In this thesis, our focus will be mainly on application areas radio and audio, because these areas are the main focus of business-line car-

Figure 1.1: Multi-path reception is just one of the issues that has to be addressed.

infotainment solutions at NXP semiconductors. The individual functions in an application are referred to as jobs, like an MP3 decoder job and FM-radio demodulation job. Note that most of the jobs process on streams of data, like radio, audio, or video streams.

The car is a living room on wheels, which is a uniquely challenging environment. A car is a moving object driving at speeds up to 200 $^{km}/_{h}$ (causing doppler effect), autonomously following a radio station, in a continuous changing environment with temperature variation, high voltage/current peeks, multi-path reception (as depicted in Fig. 1.1), and background noise. At the same time, the user expects a high quality audio. Therefore, a number of quality enhancement features are added, like *RDS-follow-me* for autonomously switching frequencies for following a radio station, channel equalising for cancelling distortion coming from multi-path reception, two reception antennaes enabling an improved reception with *phase diversity*, and noise reduction and echo cancellation algorithms for making hands-free phone calls.

The development of the car-infotainment market is not uniform across different regions in the world. For example, terrestrial digital radio and (particularly) satellite digital radio are growing strongly in the USA, while terrestrial digital radio is emerging in Europe. In the Japanese market, video is already widespread (often integrated into the head units) and navigation is well advanced, both in terms of adoption and technology (three dimensional view and integrated hard-disc devices). The European market has the strongest adoption of *Bluetooth* (BT) for connecting personal devices to their car infotainment and the main focus in Asia today are low-end car radios. Therefore, platforms should be flexible and programmable so that they can cover multiple regions.

Next to these technical challenges, the car environment has also non-technical challenges. For example, car manufacturers aim for zero field returns. Therefore, the system should have zero bugs and it should handle abnormal conditions well. No system is totally bug free, but a robust system will not lock up, cause damage to data, or let the user wait forever. Field returns are typically expensive and they harm the brand's image. That is why car manufacturers require from their suppliers that they are automotive qualified. To pass the automotive qualification, the system is exposed to extensive field tests and it should behave according to its specified behaviour during these tests. New platforms need to be algorithmic upward compatible with existing platforms, because current algorithms are already proven in the field and algorithmic changes need again extensive field tests, which are time consuming and

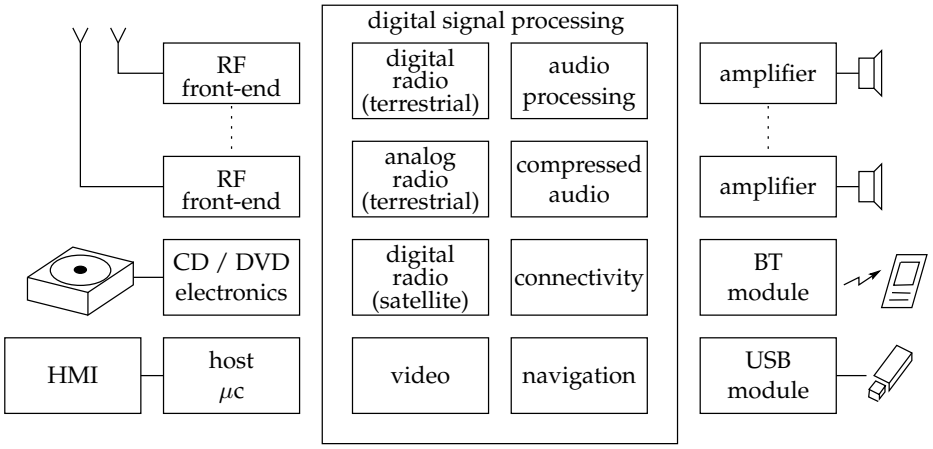

Figure 1.2: Simplified car-infotainment system.

costly. Therefore, they increase the development risks for new platforms.

There are some important differences between the life cycles in the automotive, consumer electronics, and personal computer domain, like time for introduction of new applications, and life time of applications and devices. For automotive products the planning and development of a new device takes about three years (two years for design and one year for validation and automotive qualification) and the life cycle is about eight to ten years. For products in consumer electronic, the planning and development time is six to nine months and the life cycle is about one and a half year. For the personal computer world this is even faster (several months). To survive the automotive life cycle, car-infotainment platforms should support flexibility and upgrade possibilities. For example, in October 2001 Apple launched its portable media-player *iPod* that became a hype in the following years. Car radios that are sold in 2003 contain digital signal processing platforms that are developed at least three years earlier. Therefore, they had to be flexible in such a way that the iPod could be connected to a car radio. Furthermore, when looking at the development of the car radio of today, can a *digital right management* solution survive an entire car life cycle?

A simplified block diagram of a car-infotainment system is depicted in Fig. 1.2. The application areas, such as analog terrestrial radio, digital terrestrial radio, digital satellite radio, audio processing, compressed audio, video, and navigation, have been considered as largely distinct in building car-infotainment systems. But future systems will be increasingly characterised by convergence of these application areas. The main digital signal processing platforms of business-line *car-infotainment solutions* at NXP semiconductors, support a nucleus of infotainment functionality. An infotainment nucleus consists of common and stable jobs from the areas radio, audio, video, as well as navigation. One chip or chip-set, including software, will be the heart of an integrated infotainment system, where cost, quality and application life time are balanced. Standard interfaces, to interconnect multiple chips, enable high-end application implementations for early market penetration, because we are never sure what jobs we can expect in future generations (e.g. speech recog-

generation	supported areas				supported modes		
	analog terrestrial radio	audio processing	compressed audio	digital terrestrial radio	single media	dual media	triple media
one	•	•			•		
two	•	•			•	•	
three	•	•	•		•	•	
four	•	•	•	•	•	•	•

Table 1.1: Overview of current infotainment-nucleus generations.

nition [22], audio spotlight [66], or anti sound [96]). Furthermore, it enables prototyping to capture requirements. An overview of the four generation digital signal processing platforms is shown in Table 1.1. Currently (generation four), the number of supported media streams is three, one for front-seat audio and two for rear-seat audio. Future generations will support even more than three streams. For example, for ripping audio streams to a hard disk (e.g. streams received from multiple radio stations). Furthermore, we expect that future infotainment-nucleus generations will increase to integrate additional jobs from the areas digital satellite radio, video and navigation, in a single chip. The increase in the number of jobs, increase in the number of active media streams, and increase in quality enhancement algorithms contribute to an rapid increase of possible set of simultaneously active jobs. This rapid increase will result in an increase of complexity and the demand for new design and verification methods to come to a robust and cost-efficient system implementation.

A short summary of characteristics in the car-infotainment domain is given below:

- Most jobs fall in the class of streaming, i.e. they process on a stream of data.
- Increasing demand for flexibility, e.g. supporting multiple digital-radio standards and multiple compression formats.
- The supported number of jobs and number of media streams is increasing.
- The possible number of simultaneously active jobs is increasing rapidly.

1.2 Reactive and real-time systems

The streaming jobs in the car-infotainment domain, typically need to react on their environment within certain timing constraints. Reactive systems [41, 6] have to react to an environment which cannot wait. Reactive systems maintain a permanent

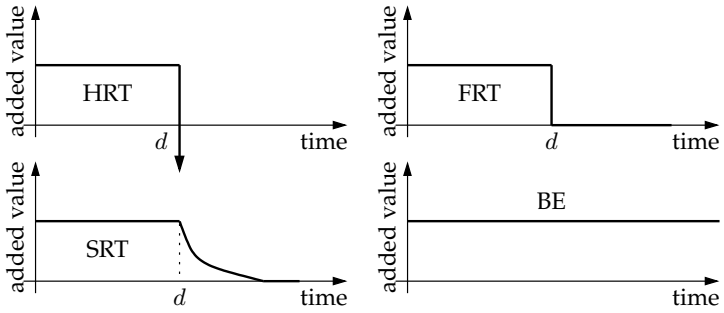

Figure 1.3: Example of jobs performance contribution function for different types of requirements.

interaction with their environment and have externally defined timing constraints. Radio reception is an example where the system should keep up with its environment, because the radio transmitter cannot be held up.

In [14], computing systems that must react within precise timing constraints to events in its environment, are called real-time systems. As a consequence, the correct behaviour of these systems depends not only on the value of the computation but also on the time at which the results are produced. Streaming media applications typically have such *real-time constraints*, i.e. deadlines. By contrast, non-real-time or *best-effort* (BE) systems are systems for which there are no deadlines, even if fast response or high performance is desired. Depending on the consequences that may occur when missing a deadline, real-time requirements are usually distinguished in three classes: (i) *hard real-time* (HRT), (ii) *soft real-time* (SRT), and (iii) *firm real-time* (FRT) [14].

(i) A job has hard real-time requirements if missing its deadline may cause catastrophic consequences on the environment (e.g. may result in the loss of life or in large damage). Therefore, in case of a hard real-time job, it is not allowed to miss any deadline. A hard real-time job contributes to its performance if it completes its function within its deadline d, as depicted in Fig. 1.3. Completing its function after its deadline, would jeopardise the behaviour of the system, therefore, its quality contribution can be seen as minus infinity. Systems that typically have hard real-time constraints, due to the potentially severe outcome of missing a deadline, are safety critical systems.

(ii) A job has soft real-time requirements if meeting its deadline is desirable for performance reasons, but missing its deadline does not cause serious damage to the environment and does not jeopardise correct system behaviour. If a soft real-time job completes its function after its deadline, it still contributes to the quality, but this contribution may decrease over time. Soft real-time jobs are typically those used where there is a need to keep a number of results up to date with changing situations. An example of a job with soft real-time constraints is navigation, dropping frames while displaying a map is hardly noticeable by a user as long as the deadline misses are sporadic.

(iii) A job has firm real-time requirements if missing its deadline results in no contri-

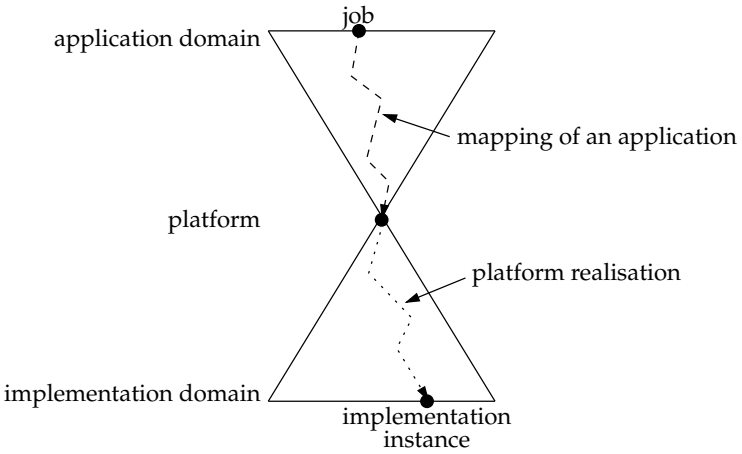

Figure 1.4: Platform-based design paradigm.

bution to its performance, but missing its deadline does not cause serious damage to the environment and does not jeopardise correct system behaviour. Audio and radio processing are examples of jobs with firm real-time requirements. Missing a deadline, when processing an audio stream, will result in a steep quality degradation. For example, missing deadlines at a digital-to-analog converter can cause hiccups in the audio, or missing a deadline in the digital radio path can result in loss of synchronisation. In both examples there are severe quality losses, but there are no catastrophic consequences on the environments. In this thesis, we will focus on jobs with firm real-time requirements, because our focus is up to infotainment-nucleus generation four, which exclude video and navigation jobs.

1.3 Platform-based design

The growing complexity of current and future embedded systems leads to a demand for new design paradigms. The car-infotainment domain consists of an increasing number of jobs, as described in Section 1.1. Implementing these jobs in a system-on-chip that meets the functional and non-functional constrains, is a large design problem. Platform-based design [24, 47] is an example of a design paradigm in which the design complexity is split into two, by specifying a platform specification in the middle as depicted in Fig. 1.4. The platform is targeting multiple jobs from a specific domain. The platform specification allows software engineers to map their jobs to the platform at the same time as the hardware engineers realise an implementation instance that meets the platform specification. Furthermore, a platform design increases the reuse between different products from the same application domain. This also reduces the non-recurrent engineering cost and time-to-market in developing a new product.

A system platform consists of a high-level architecture specification for hardware as well as software. For this high-level architecture, services are defined that can be

allocated to a job, which is mapped onto the platform. Mapping a job to the platform is challenging for meeting its real-time constraints like throughput and end-to-end latency. At the bottom in Fig. 1.4, there are possible implementations of the platform that comply to the high-level architecture specification and that are able to deliver the specified services. Realisation of a platform implementation is challenging for meeting cost constraints in terms of silicon area and energy consumption. In this thesis, we propose an architecture and design rules for building a platform that is optimised for infotainment-nucleus generation four and beyond.

1.3.1 Existing platforms

This section evaluates existing platforms within the car-infotainment domain and it elaborates on the platform trends. From a consumer perspective, we divide current NXP's car-infotainment platforms into four generations, as described in Table 1.1.

The design of the first generation car-infotainment chip SAA7701, started in the early nineties. It supported analog terrestrial-radio reception and audio post-processing functionality. The radio input signal was coming from a tuner chip and the radio signal was demodulated by a dedicated hardware Intellectual Property (IP) module. Beside the radio input, the chip had capability for two analog and two digital input signals. This chip integrated one Digital Signal Processor (DSP) (which belongs to the EPICS family [72]) that was able to process one audio stream at a time. The main task of the processor was audio post processing. Derivatives of the SAA7701 included a hardware accelerator for extra audio features, like equalisation, and they supported up to five analog and three digital input signals. As the chips were implemented in smaller process technologies, the processors were able to execute with higher clock frequencies and supporting more features with an increasing sound quality. The processor has a local on-chip memory that is used by only the processor.

The second generation car-infotainment platform supports processing single and dual media stream, for independent front-seat and rear-seat audio. The first chip that supported this was SAA7706, which was also the first car-infotainment system that integrated two DSPs instead of one. The chip SAA7724 was the first chip capable of processing two independent analog terrestrial-radio streams. It integrated three special purpose processors, for demodulating the two radio input streams, together with two DSPs, for sample-rate conversion and audio post processing. The chip SAF7730 [71] is also capable of processing two radio input signals and it consists of five DSPs and five hardware accelerators. Three DSPs in combination with the five hardware accelerators are used to demodulate two radio streams. The two remaining processors are used for sample-rate conversion and audio post processing. Next to dual-radio processing, the SAF7730 is also capable of single-radio reception with phase diversity, which is an advanced algorithm for improved radio reception. The interconnect of the chip SAF7730 is as follows. There are hardwired point-to-point connections between a processor and hardware IP modules like accelerators or peripherals. The interconnect between the processors is implemented with circular buffers that are stored in dual-ported memories. Next to these inter-processor communication memories, the processors have local memories for storing instructions, coefficients and data.

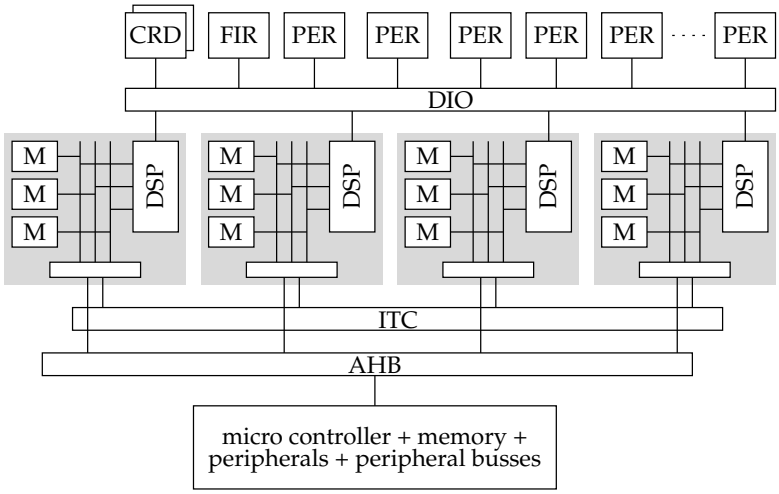

Figure 1.5: Hardware architecture generation three (SAF7780).

In the first two generations, the digital signal processing chips are controlled and monitored by a host micro controller (μc) that is connected to these chips. The micro controller can program the chip's parameters, like control parameters and filter coefficients. Such a micro controller is integrated in the third generation car-radio chip SAF7780 [88, 8]. Furthermore, it supports analog terrestrial-radio reception, playback compressed audio (MP3 and WMA), and connectivity to portable devices in different user modes, like single-media versus dual-media audio. Although it supports dual media, it can demodulate only one radio-input stream at the same time. This chip is composed of four DSPs, one micro controller (which can be used as host), three hardware accelerators (one Finite-Impulse-Response (FIR) filter and two Coordinate-Rotation-Digital computer (CRD) accelerators), and a number of input and output peripherals (PER), as depicted in Fig. 1.5. The radio demodulation is performed by two DSPs in combination with the three hardware accelerators. The two other DSPs perform sample-rate conversion, compressed-audio decoding, and audio post processing. As interconnect, the platform uses a multi-layer bus (AHB) in the micro-controller subsystem, a crossbar switch (ITC) between the DSPs, and a crossbar switch (DIO) between the DSPs and the accelerators/peripherals. The DSPs make use of local memories (M) for instructions, coefficients and data. In contrast with generation one and two, these processors make use of shared local memories instead of private local memories, i.e. a processor can also write to the local memory of another processor.

The fourth generation car-infotainment platform is currently under development and it will support analog as well as digital terrestrial-radio reception and decoding as well as encoding compressed audio (various formats). Compared to previous generations, this platform will support processing up to three independent media streams, one for front-seat and two for rear-seat audio. The platform will include a number of DSPs, a Very-Long-Instruction-Word (VLIW) processor, a micro controller, and a number of hardware accelerators and peripherals. The interconnect of

the chip will consist of a number of multi-layer busses that are connected via bridges. Such an interconnect is a first step towards a network-on-chip. The memory architecture is as follows. There will be an off-chip memory because the memory footprints (e.g. for the digital radio jobs) are considered to be too expensive to store on-chip. The processors will have local shared memories as well as caches to hide the large latencies in accessing the off-chip memory.

1.3.2 Platform trends

By investigating existing car-radio platforms, we observed the following trends. The first generation platforms contain only one processing core. Next generation platforms contain multiple processing cores. For cost and power-efficiency reasons, we see that heterogeneous multiprocessor systems are used that combine various types of processors with configurable hardware accelerators. The trend of an increase in the number of processing cores is expected to continue in the future. It is also visible in other application domains. For example in the CELL processor from IBM, which combines one PowerPC core with eight synergetic processors [46]. Another example is coming from Intel. Instead of still increasing the clock frequency, Intel shifts its strategy to increasing the number of processors. Currently, they shift from single-core to multi-core systems, and later on they expect to shift from multi-core towards many-core systems, which consist of more than hundred processing cores [10].

The integration of different types of cores into a working system is a major challenge. Currently multiple busses and custom interconnects (point-to-point, cross-bar switches) are used and they are interconnect to each other via bridges. However, with an increasing number of cores, designed in technologies with decreasing dimension, they do not sufficiently address hardware problems (deep sub-micron VLSI design) and software problems (application programming). Networks-on-chip tackle these problems and, therefore, are a better answer to the integration challenges. From a hardware perspective, they structure the top level wires in a chip, and facilitate modular design [74]. Structured wiring results in predictable electrical parameters, such as crosstalk. Network interconnects are *segmented* and *multi-hop*. The advantage of segments is that only those segments are activated that are actually used in the communication, so only those segments dissipate power. Multi-hop is needed because the transport delay from source to destination can become longer than the clock period. From a software perspective, networks can reduce the programming effort by defining proper transport-level services. The bottleneck in single-processor architectures is computation, whereas the bottleneck in multiprocessor architectures shifts from computation towards communication. Getting the right data at the right place at the right time will dominate the architecture. Networks that offer *guaranteed-communication services* make systems easier to program, easier to design [34], and more robust.

The first two generation car-infotainment platforms have only processors with private local memories. Although there is a C-compiler available, most of the code is written in assembly for performance and legacy reasons. Software algorithms that are written in assembly code are typically more efficient in terms of required processor cycles and memory usage compared to compiled C-code. The disadvantages of assembly code are that it is much harder to write, read, and maintain compared to

C-code and it is only applicable for the targeted processor. The trend is that processors become faster, larger, and more complex. Furthermore, software tasks become also larger and more complex in contrast to the small tasks as in the first platform generations. Therefore, the trend is larger memory footprints and more variation in the temporal behaviour. From small tasks in combination with predictable memory access latencies, we are able to derive conservatively-estimated upper bound on the execution times of these tasks, as we will describe later. For such a system, worst-case design is achievable because conservatively-estimated bounds (e.g. on the required number of processor cycles) are not far from the typical case. In current and future generation platforms, the local memories are shared, the code will be off-loaded to an off-chip memory, and caches will be introduced to prevent that every memory access will receive a large memory access latency. In such a memory hierarchy, the access latency from a processor to the external memory can vary, because of an unknown state of the cache, possible contention in the communication infrastructure, and possible contention at the external memory. Therefore, the distance between the typical-case and worst-case memory access latency will increase. The uncertainty of the required number of processor cycles increases and worst-case design can become too expensive, if the worst case is an order of magnitude different from the typical case. This increase of uncertainty makes it a challenge to build cost-efficient platforms with worst-case design methodologies.

1.3.3 Design-time versus run-time mapping

A platform consists of a high-level architecture for which platform services (or resources) are defined that can be allocated to a job. Mapping a job is defined as allocating resources to a job and finding scheduling settings in case of resource sharing. The main challenge that is discussed in this thesis, is mapping a job so that the job's real-time requirements are met.

There are trade-offs between run-time and design-time mapping of a job. Design-time mapping has a few advantages. First of all, the scope is typically larger than in case of run-time mapping, i.e. with design-time mapping the designer can make decisions based on knowledge over a large number of iterations, e.g. gathered via profiling during simulation, via static code analysis, or via knowledge from the application domain. Furthermore, design-time mapping can be more complex than run-time mapping, because computation is not time critical. The main advantage of run-time mapping is a more precise knowledge about the system load. For example, it is known in which scenario [30] or mode the job is executed and it is known which resources are actually used. However, at run-time the scenario or resource usage can change rapidly. Furthermore, computing a specific mapping will occupy a processor, takes time, and consumes energy. Therefore, run-time mapping algorithms should be simple.

In this thesis, we combine design-time and run-time mapping in the following way. Every possible set of simultaneously activated jobs, which we refer to as a use case, is investigated separately at design time. Each job in a use case is mapped to our platform, one by one. Tasks from a job are bound to the processing tiles. The required resource budgets and scheduler settings are computed so that the real-time requirements are met. At run-time, the jobs are started and stopped by loading a

predefined mapping.

1.3.4 Verification of real-time constraints

To verify that the job's real-time requirements are met, we need a performance analysis technique to analyse the temporal behaviour of a job. We identify two categories for existing analysis techniques, namely (i) simulation and (ii) exhaustive analysis.

(i) Extensive simulation is often used for analysing the temporal behaviour of a job and to verify that its throughput and end-to-end latency requirements are satisfied. The simulation tools accompanying the modelling language SystemC [44] and POOSL [85], for example, are used to simulate transaction level models [21]. Transaction level models trade-off accuracy for running time. However simulation can be performed at different levels of abstraction, simulation of all operation modes for a large set of input stimuli is time consuming. From cycle-accurate simulation, we can derive an optimistic estimate on the minimum throughput, because the throughput observed during simulation is only valid for the given set of input stimuli. A larger set of input stimuli can increase the accuracy, however, we are unable to guarantee that throughput requirements are met for all possible input stimuli and starting states.

Assuming that the parameters in a model (e.g. worst-case execution times) are conservative, proper analysis techniques, such as exhaustive analysis, are able to guarantee that no deadlines are missed, i.e. can guarantee a minimum throughput and maximum end-to-end latency.

(ii) Exhaustive analysis techniques are based on min-plus algebra [11] or max-plus algebra [3]. Network calculus [17], real-time calculus [53, 87], and event-models [45] have their roots in the min-plus algebra [5]. These analysis techniques bound the data traffic between tasks with (piece-wise) linear bounds. The analysis is based on the assumption that the bounds for each pair of communicating tasks can be given the same average slope. The slope of a bound can be adapted by, for example, changing the settings of run-time schedulers or by regulating arrival of data by introducing traffic shapers. An important drawback of this analysis approach is that it has problems with cyclic data dependencies that affect the temporal behaviour. The reason is that the linear bounds on the traffic are considered an input of the problem and not an outcome of the analysis. More precisely, the linear bounds are derived for each task in isolation. For acyclic graphs the end-to-end temporal behaviour can be computed given these bounds. However, for cyclic graphs incorrect results can be obtained because cyclic dependencies can affect these bounds and, therefore, can influence the job's temporal behaviour. Cyclic dependencies can be caused by functional dependencies (e.g. in the case of feedback loops), schedule dependencies (e.g. in the case of static-order scheduling), and back-pressure (to prevent buffer overflow). In our car-infotainment system, we have such cyclic dependencies. Therefore, we should be able to cope with them.

Dataflow-analysis techniques [51, 76, 91] have their roots in max-plus algebra [5]. Multiple dataflow models are described in literature, each with a different trade-off between expressivity and analysability. The best well known dataflow models are Single Rate DataFlow (SRDF) [9], Multi-Rate DataFlow (MRDF) [9, 50], Cyclo-Static

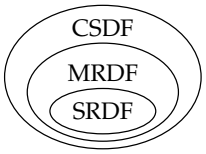

Figure 1.6: Ordering of dataflow models according to their expressivity.

DataFlow (CSDF) [9, 64], Boolean DataFlow (BDF) [13], and Dynamic DataFlow (DDF) [63]. Marked graphs [16] and Weighted Marked Graphs, which are a subclass of timed Petri Net theory, have the same expressiveness as SRDF and MRDF graphs, respectively. SRDF and MRDF are also known as Homogenous Synchronous DataFlow (HSDF) [76] and Synchronous DataFlow (SDF) [50], respectively. MRDF and CSDF graphs can be transformed into equivalent SRDF graphs, but the number of actors of the equivalent SRDF graphs can be large. Therefore, for a one-to-one relation between tasks in the implementation and actors in the model, Fig. 1.6 shows the ordering of SRDF, MRDF, and CSDF models according to their expressivity [9]. These models do not support data dependent input and output behaviour of tasks, as is supported in the by BDF and DDF models. But in these models, throughput and end-to-end latency cannot be derived for an arbitrary graph. Some generalisations of dataflow models have been proposed, i.e. techniques to allow input data dependent input and output behaviour of tasks [93, 84, 7], and that maintain the full potential for analysis. Another generalisation on dataflow models is scenario awareness, which is referred to as a Scenario-Aware DataFlow (SADF) model [86]. This model uses a dataflow model to represent a specific scenario and it uses a stochastic approach to model the order in which scenarios occur. The underlying model is a Markov chain that can be analysed using exhaustive or simulation-based techniques.

An important advantage of dataflow-analysis techniques [76, 28] is that it allows cyclic data dependencies that influence the temporal behaviour. Therefore, also back-pressure is supported by the dataflow model. This allows execution-time estimates of tasks, in case conservatively-estimated upper bounds are not available, as will be described in Chapter 8.

1.4 Problem definition

A car-infotainment system consists of a hardware platform on which the application software is executed. The application consists of a number of jobs that process on streams of data. For performance and power efficiency reasons, these jobs are executed on a heterogeneous multiprocessor architecture. Furthermore, the jobs have real-time constraints, like throughput and end-to-end latency. To reduce the verification effort, the job as well as the hardware platform should have a predictable temporal behaviour. This allows the designer to verify, at design time, that the job's real-time requirements are met. In the literature, also composable systems [49, 5, 37] are proposed to reduce the verification effort in integrating multiple jobs to a hard-

ware platform. In a composable system, the temporal behaviour of one job cannot be affected by another job while they are both executed on the same platform. Therefore, in a composable system the job's real-time requirements can be verified in isolation. The isolation of temporal behaviour is realised with fixed resource budgets for jobs. In a predictable system, in contrast with a composable system, jobs have a minimum resource budget and not a fixed resource budget. A composable system is especially useful in case jobs (e.g. soft real-time jobs) can have an overload. In a composable system, this overload cannot affect the performance of other jobs (e.g. firm real-time jobs), because the temporal behaviour of one job cannot affect the temporal behaviour of another job. Therefore, composability will ease the system's performance-verification effort, because an overload of a job will only affect the performance of the misbehaving job. Predictable and composable architectures are orthogonal to each other and they can be combined in one system, as is shown in [37]. In this thesis, we only focus on a predictable system on which firm real-time jobs are executed. A predictable system is necessary to be able to guarantee that the job's real-time requirements are met.

Definition of a predictable system:

Definition 1 (Predictable system)**.** A system is predictable if we can verify at design time whether temporal constraints are satisfied for the respective condition to hold.

The temporal constraints are expressed as repetitive deadlines at source or sink tasks, as will be described in next chapter. On the one hand, deadlines can be derived from a task's throughput constraint or end-to-end latency constraint. On the other hand, if we can guarantee an upper bound on end-to-end latency that is lower than the end-to-end latency constraint, and if we can guarantee a lower bound on throughput that is higher than the required throughput, then it is sure that no deadlines will be missed. Therefore, we should be able to derive an upper bound on end-to-end latency and a lower bound on throughput for every job. Furthermore, these bounds should be tight to enable a cost-efficient implementation of the system.

For a job with hard real-time requirements, no data can be lost. When mapping a job with hard real-time requirements on a hardware platform, the condition of a predictable system is as follows:

Definition 2 (Condition for hard real-time)**.** A hard real-time system must satisfy the temporal constraints for any input stream and any initial state of the system.

A system with hard real-time constrains must keep up with its environment in any circumstance. Therefore, for a hard real-time system, we must derive a conservatively-estimated upper bound on end-to-end latency and a conservatively-estimated lower bound on throughput for every job. These bounds must hold for any possible set of input stimuli and for any possible initial state of the hardware platform (e.g. initial state of a time-division-multiplex scheduler or cache).

For a job with firm real-time requirements, no data should be lost and the system should keep up with its environment. Not keeping up with the system environment does not jeopardise correct system behaviour, because it has firm real-time requirements instead of hard real-time requirements. When mapping a job with firm real-time requirements on a hardware platform, the condition of a predictable system is as follows:

Definition 3 (Condition for firm real-time)**.** A firm real-time system must satisfy the temporal constraints for a set of input streams and any initial state of the system. Furthermore, a firm real-time system must have a fall-back mechanism to recover from deadline misses.

For a firm real-time system with a specific set of input stimuli, we can derive an upper bound on end-to-end latency and a lower bound on throughput for every job. Notice that the derived bounds are optimistic estimates, because they are not guaranteed for every possible set of input stimuli. If the job exceeds our optimistically-estimated bounds on end-to-end latency or throughput, it can miss a deadline. When the set of input stimuli is representative, we are confident that the system exceeds a deadline only sporadically. It is not catastrophic if the job sporadically misses a deadline due to firm real-time requirements. Not keeping up with the system environment must not jeopardise correct system behaviour, so it must have a fall-back mechanism in case of a deadline miss. An example of a fall-back mechanism is reusing a previous computed audio sample.

When mapping a job with soft real-time requirements on a hardware platform, the condition of a predictable system is as follows:

Definition 4 (Condition for Soft real-time)**.** A soft real-time system has a target for its average behaviour but does not have temporal constraints. Furthermore, a soft real-time system must have a fall-back mechanism to recover from deadline misses.

For jobs with soft real-time requirements it is desirable that the system keeps up with its environment, but missing a deadline does not cause serious damage to the environment since there is a fall-back mechanism. In a soft real-time system, it is allowed to derive an estimated end-to-end latency and estimated throughput instead of optimistically-estimated bounds as in case of firm real-time. Therefore, the target temporal behaviour is typically average case. In this thesis, we only focus on hard and firm real-time systems and we do not investigate systems with soft real-time requirements. Furthermore, all active jobs are equally important.

We now arrive at the main problem statement of this thesis:

Problem statement. Develop a multiprocessor architecture for a predictable firm real-time system that is optimised for infotainment-nucleus generation four. Furthermore, show that a job, which is mapped on the architecture, can be represented in a dataflow model, so that existing dataflow-analysis techniques can be used to verify that the system satisfies the real-time constraints.

In next section, we elaborate on our approach in dealing with this problem.

1.5 Approach

A system consists of multiple jobs executed on a hardware platform. In order to come to a predictable system, we need (i) jobs from which we are able to reason about the temporal behaviour, (ii) a platform architecture from which we are able to

reason about the temporal behaviour, and (iii) a model of a job that is mapped on the platform in order to reason about the temporal behaviour of the total system.

(i) When a job complies to our model of computation, we are able to reason about its temporal behaviour at design time. Chapter 2 formulates the characteristics of such a job. A job can be represented in a task graph, where tasks are represented by nodes and inter-task communication channels are represented by edges. The main characteristics in order to bound the temporal behaviour are bounded execution times, bounded production behaviour, and bounded consumption behaviour of tasks. The bounds on execution times should be conservatively estimated in case of hard real-time requirements, or they can be optimistically estimated in case of firm real-time requirements.

(ii) We are able to reason about the temporal behaviour of the following platform architectures. First, we start by defining a multiprocessor architecture with limited resource sharing, to limit the uncertainty in the temporal behaviour. Predictable memory-access latencies are achieved with a local private memory for each processor, i.e. a processor only accesses its local memory and this memory is not accessed by another processor. Furthermore, each processor executes only tasks from the same job and these tasks are executed in a static order. As an interconnect between the processors, we make use of a network-on-chip that supports network connections with guaranteed communication services. For such a system, we are able to derive conservatively-estimated bounds on throughput and end-to-end latency for each job. The architecture has limitations in terms of the supported sizes for memory footprint and data containers. Furthermore, the supported number of communication channels and the supported buffer capacities for inter-tile communication, are fixed at design time. This architecture is only applicable for jobs coming from infotainment-nucleus generation one and two, because these jobs make use of sample-based processing.

Next, this multiprocessor architecture is extended with shared local memories between processors. This allows us to communicate via circular buffers that are stored in the local memory of a processor. The main advantages of these circular buffers is a cost-efficient implementation of large buffers and that the buffer capacities are programmable at run time. The number of supported communication channels is also programmable at run time. Furthermore, the use of circular buffers enables checking of available space or data, which in its turn enables the use of run-time scheduling of tasks that belong to different jobs. This sharing of memory and processor resources will increase the uncertainty in the temporal behaviour of the system. However, conservatively-estimated bounds on throughput and end-to-end latency can still be derived. The architecture has still the limitation in the supported memory footprint, because tasks are assumed to fit in the local memories of the processors. Therefore, this architecture is only applicable for jobs up to infotainment-nucleus generation three.

Finally, the multiprocessor architecture is extended with an off-chip memory that is shared between the processors. This is required in case the memory footprint of tasks is considered to be too large to store on-chip. The access latencies to an off-chip memory are larger than the access latencies of local memories. Therefore, processors will use level one caches to hide these larger access latencies. This thesis does not elaborate on explicitly (software controlled) pre-fetching of the task's program code

and working data set into a local memory, but it is seen as future work. The introduction of caches will introduce additional uncertainty in the temporal behaviour of the system. In practice, for such an architecture, optimistically-estimated bounds on throughput and end-to-end latency are used instead of conservatively-estimated bounds. This architecture is applicable for jobs from infotainment-nucleus generation four.

(iii) In order to reason about the temporal behaviour, we require a model that accurately represents the temporal behaviour of a job that is executed on the platform. For such a model, we make use of dataflow models. First, the job's task graph will be transformed into a dataflow graph. Next, the job will be mapped to the platform and after every mapping step additional constraints are added to this dataflow model. For every extension in the architecture, we need to represent the temporal behaviour in a dataflow model. Finally, the job's real-time requirements are verified by making use of existing dataflow-analysis techniques. Furthermore, buffer capacities and scheduler settings can be derived for given throughput and end-to-end latency constraints.

1.6 Contributions

This thesis makes several contributions to develop a predictable and cost-efficient system for streaming jobs.

Main contribution of this thesis

- Introduction of a network-based multiprocessor system that is predictable. This is achieved by starting with an architecture where processors have private local memories and execute tasks in a static order, so that the uncertainty in the temporal behaviour is minimised. This architecture is extended with shared local memories, run-time scheduling of tasks, and a memory hierarchy. After each extension, it is shown that the temporal behaviour can still be modelled in a dataflow model and, hence, we are still able to verify that the job's throughput and end-to-end latency requirements are met.

Contributions in Part I of this thesis

- We show that tasks, which are executed in a static order, can be represented with one actor despite the absence of the firing rule in the implementation. An algorithm is introduced for generating a CSDF graph that models tasks that are executed on processors with static-order schedules. Earlier versions of this work were published in [58, 57].

- For an industrial case study, we compare the Æthereal network-on-chip with the traditional interconnects for infotainment-nucleus generation three. For this generation, we conclude that it is feasible to replace the traditional interconnects by an Æthereal network and still meet the communication requirements. We conclude that the network-area cost is mainly determined by the number of connections (translating to a number of buffers) and the network

topology (affecting the number of routers, the slot table and the sizes of the buffers). Earlier versions of this work were published in [60, 55].

- For an industrial case study, we investigate the tightness of a conservatively-estimated lower bound on the throughput for a job mapped onto our multiprocessor platform. The conservatively-estimated throughput bound is computed from a dataflow model and it is compared with the optimistically-estimated throughput bound that is measured with cycle-accurate simulation. The difference is only 10.1% for our job, which is a channel equaliser for FM radio. The difference is small because of tight conservatively-estimated bounds on execution times (each processor has a private local memory) and communication latencies (due to guaranteed-throughput services and a small slot table in the network). Finally, we identify three causes for the difference between a throughput bound that is computed from a dataflow model and a throughput bound that is measured with cycle-accurate simulation. An earlier version of this work was published in [57].

Contributions in Part II of this thesis

- For address-based communication, we introduced a formula to compute an upper bound on the number of processor stall cycles, which can be translated in a lower bound on the processor utilisation. For our industrial case study, which is an MP3 decoder, we have shown that the bound on the processor utilisation has an accuracy of at least 6 %. Furthermore, in case of sharing of a network connection between multiple communication channels, it is shown that larger network-interface buffers can lead to larger buffer requirements for the circular buffers in the memory. An earlier version of this work was published in [56].

- We proposed a novel cache-aware mapping technique that reduces the number of instruction and data cache misses for streaming jobs that are mapped onto a multiprocessor system. This technique is based on a technique [73] that executes tasks multiple times in a loop before executing another task. It is shown that it is only beneficial if the individual tasks fit in the instruction and data cache, and the set of tasks, which are executed on a processor, do not fit simultaneously. We use a dataflow model for representing an application that is mapped onto a multiprocessor with a specific number of successive task executions. From this dataflow model we derived the maximum number of successive task executions by making use of existing dataflow-analysis techniques. For our industrial case study, which is a Digital-Radio-Mondiale receiver, we reduce the number of cache misses by a factor 4.2. This work was published in [58].

1.7 Thesis outline

This thesis is divided into two parts. In part I, we introduce design rules for a predictable multiprocessor system-on-chip. In part II, these concepts are extended to-

wards a multiprocessor architecture for jobs up to infotainment-nucleus generation four.

Part I: In the next chapter, we first formulate a streaming job and describe its characteristics. For these streaming jobs we define, in Chapter 3, a scalable heterogeneous multiprocessor architecture that consists of tiles which communicate via a network-on-chip. In Chapter 4, we describe dataflow modelling and analysis for verifying the real-time requirements of our streaming jobs. This chapter also describes how such a model can be constructed, so that it captures the temporal behaviour of a job mapped onto a multiprocessor platform. In Chapter 5, the network cost is investigated in terms of area and latency for a number of automatically generated network instances and this is compared to traditional interconnects from car-infotainment platform SAF7780. Chapter 6 evaluates the practical use and tightness of dataflow modelling and analysis for our channel-equaliser case study.

Part II: In Chapter 7, we extend the multiprocessor architecture with shared on-chip memories and run-time scheduling of tasks. For these extensions, it will be shown that the real-time constraints can still be verified by dataflow modelling and analysis. In Chapter 8, the multiprocessor architecture is extended with a shared off-chip memory. It introduces a cache-aware mapping technique for streaming applications, because an efficient use of the memory hierarchy is important for current and future multiprocessor systems. Finally, Chapter 9 concludes this thesis and gives recommendations for future work.

Part I: Design rules for a predictable multiprocessor architecture

Chapter 2

Streaming application domain

The jobs in the infotainment nucleus, typically process streams of input data and generate streams of output data. Furthermore, they have real-time constraints, for example caused by a periodic source (e.g. analog-to-digital converter), periodic sink (e.g. digital-to-analog converter), or both periodic source and sink. These jobs are called streaming jobs in this thesis. The characteristics of streaming jobs are described in the following section. In Section 2.2, we elaborate on the real-time requirements of these jobs. Jobs with sample-rate conversion are described in Section 2.3. Finally, Section 2.4 gives some examples of streaming jobs in the car-infotainment domain.

2.1 Characteristics of streaming

Streaming jobs are common in the embedded domain and they encompass a broad spectrum of applications, including media encoding and playback. Every possible set of simultaneously activated jobs is called a use case. Jobs can be started and stopped by the user. The user can start or stop jobs while others continue. Furthermore, the mode of a job can be changed by the user. Muting an audio stream or changing its volume are examples of mode changes where switching between use cases is not necessary. A switch between use cases is a dynamic process, which must be handled at run time. The number of use cases is increasing rapidly, because of the increasing number of supported jobs. For example, if there are N jobs and each job can be active or inactive, then the number of use cases is theoretically 2^N. Obviously, jobs can only be started if enough resources are available.

Streaming jobs are characterised by concurrent computation processes, which we refer to as tasks, that process potentially infinite sequences of data provided by the environment. A task represents a function transformation that has one or more inputs and outputs. Furthermore, tasks execute independently and they can have state that contains all the information necessary to execute. They are repeatedly executed and thus have explicit start and finish times. After a task started its execution, it will continue to execute until it is finished. In case of run-time scheduling, tasks can also

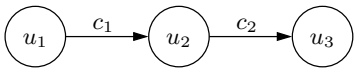

Figure 2.1: Example of a streaming job represented as a task graph.

be interrupted, as we will see in Part II of this thesis. Once a task finished its execution, it can start to execute its next iteration. Tasks are executed by, for example, a processor or hardware accelerator. A scheduler repetitively enables tasks to start executing. Of course, when the user stops the job, the tasks are not enabled anymore.

For each task, it is made explicit which data is private to a task (state) and which data is shared between tasks (communication). A task has random access to its private data. Shared data is communicated from one task to another task via a communication channel. Containers are used for the synchronisation between tasks, i.e. tasks communicate containers via First-In First-Out (FIFO) buffers. A fixed amount of data can be stored in a container and they can be full or empty. Containers are also useful for memory management, namely for allocating and releasing space in a memory. Examples of containers are audio stereo samples, MP3 frames, video pixels, video lines, or video frames. A task can have random access within a container, despite the FIFO synchronisation between containers. The number of containers that can be stored in a communication channel is fixed, i.e. the FIFO buffer capacity is fixed. During one execution of a task, it consumes a number of full data containers from its input channels and it produces a number of containers to its output channels.

It is natural to express a streaming job as a graph, where the nodes represent independent tasks and the edges represent communication channels between these tasks. Such a task graph $H = (U, C)$ consists of a finite set of tasks U and a finite set of communication channels C. Fig. 2.1 shows an example of a task graph that represents a streaming job that consists of three tasks and two communication channels.

A communication channel $c_k = (u_i, u_j)$ connects an output of task u_i to an input of task u_j with $u_i, u_j \in U$ and $c_k \in C$. Task u_i produces data containers on this channel and tasks u_j consumes data containers from this channel. The maximum number of containers that can be stored in a channel c_k is bounded and it is denoted with $d(c_k)$.

After a task started its execution, it will continue until it is finished. The start time of the l th execution of task u_i is denoted by $s(u_i, l)$. The time that task u_i finished its l th execution, is denoted by $f(u_i, l)$. After one execution of task u_i, the number of data containers that are produced on channel c_k is denoted by $\mu(u_i, c_k) \in \mathbb{N}^+$. The number of data containers that task u_j consumes from channel c_k is denoted by $\lambda(u_i, c_k) \in \mathbb{N}^+$. In this thesis, we assume that the number of consumed and produced containers are known at design time and cyclo static.

Every task has an execution time that is defined as:

Definition 5 (Execution time)**.** The execution time of task u_i is defined as the difference between the time this task started its execution and the time this task finished its execution, i.e. $f(u_i, l) - s(u_i, l)$, assuming that sufficient filled containers are available at all its inputs, sufficient empty containers are available at all its outputs, and this task is the only task executed on a processor and the processor is the only master

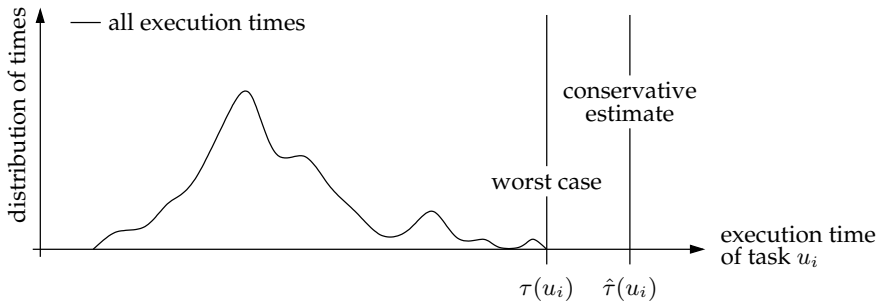

Figure 2.2: The definition of worst-case executions time and a conservatively estimated upper bound on the execution time.

that is accessing the memories.

This execution time does not include the time a task has to wait for input data and output space. It does also not include the time a task has to wait before it is scheduled and it does not include the interference time caused by interrupting tasks. Furthermore, execution time does not include processor stall time caused by arbitration at shared memories and cache misses. Therefore, task execution times do not include dependencies between resource and tasks.

For a predictable system we can derive bounds on throughput and latency for every active job. In order to derive these bounds on throughput and end-to-end latency, we need upper bounds on execution times. Conservatively-estimated execution-time upper bounds can be computed with static-program analysis techniques [15]. Unfortunately, it is not always possible to obtain upper bounds on execution times of tasks [95]. This is only possible if we use a restricted form of programming, which guarantees that tasks always terminate, i.e. recursion and loops are only allowed if the iteration counts are explicitly bounded. A task typically shows a certain variation of execution times, e.g. depending on the input data. The maximum of all possible execution times is referred to as the *worst-case execution time*, as depicted in Fig. 2.2. The worst-case execution time of task u_i is denoted by $\tau(u_i)$.

Conservatively-estimated upper bounds on the execution time of a task can be computed by methods that consider all possible execution times of the task. These methods use abstraction of the task to make timing analysis of the task feasible. Abstraction loses information, so the computed upper bound usually overestimates the exact worst-case execution time. A conservatively-estimated upper bound represents the worst-case guarantee that the method or tool can give. How much is lost depends both on the methods used for timing analysis and on overall system properties, such as the hardware architecture and characteristics of the software. In part I of this thesis, we make use of conservatively-estimated upper bounds on execution times. The conservatively-estimated upper bound on the execution times of a task u_i, is denoted by $\hat{\tau}(u_i)$.

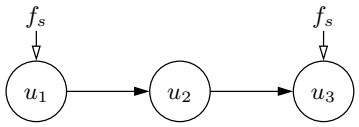

Figure 2.3: Example of a job where the source (u_1) and sink (u_3) task execute strict-periodic with sample frequency f_s.

2.2 Job's real-time constraints

Jobs in the infotainment nucleus have real-time constraints, as introduced in Section 1.2. Each job is composed of communicating tasks and it is represented in a task graph, as described in previous section. In this section, we elaborate on the job's real-time constraints in terms of throughput and end-to-end latency.

The jobs in the infotainment nucleus contain typically a source or sink task. These source or sink tasks have often repetitive deadlines, e.g. periodic or cyclo-static deadlines. An example of a sink task with strict-periodic deadlines is a digital-to-analog converter, because it takes an input sample every clock edge and this clock has a fixed sample frequency. Similar holds for an analog-to-digital converter which is a source task with strict-periodic deadlines. In this thesis, we assume that source and sink tasks produce and consume a fixed amount of containers per execution and that they have strict-periodic deadlines. Therefore, we define throughput constraint as follows:

Definition 6 (Throughput constraint)**.** A throughput constraint is specified for the number of containers per time interval at the output channel of a source task or at the input channel of a sink task.

For example in case of a digital-to-analog or analog-to-digital converter, the number of containers per second are specified by the sample frequencies of these converters, because they consume and produce one container per execution. Figure 2.3 depicts an example of a job with a strict-periodic source and sink task that execute f_s times per second. The strict-periodic executions of the source and sink tasks are indicated with the open arrows in Fig. 2.3. The sample frequencies are specified by the numbers next to these arrows.

Next to a throughput constraint, a job can have an additional end-to-end latency constraint. Most streaming jobs can tolerate additional end-to-end latency, therefore, the throughput constraint is typically more critical than the end-to-end latency constraint. However, the German quality management system for the automobile industry *Verband der Automobilindustrie* [20], specified, for hands-free terminals, the maximum delay from mouth reference point to output of the speech codec in a mobile phone. This results in an end-to-end latency constraint of 30 milliseconds between the microphone in the hands-free terminal and the Bluetooth device that is connected to the mobile phone.

The specification of end-to-end latency can be difficult in case there is no direct relation between containers at a source task and corresponding containers at a sink task.

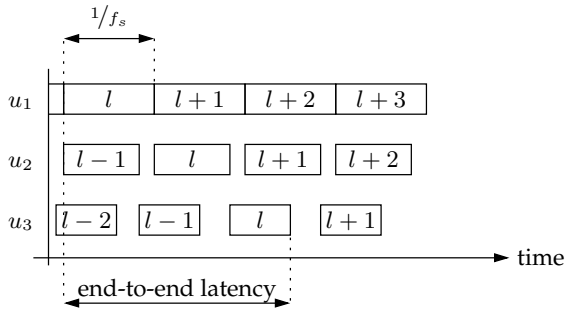

Figure 2.4: Definition of end-to-end latency $L(u_1, u_3)$ for a job with a source (u_1) and sink (u_3) task that both execute strict periodic with sample frequency f_s.

For our case studies in this thesis, we specify end-to-end latency between a strictly-periodic source and sink task that are executed with the same sample frequency. Therefore, in this thesis, end-to-end latency constraint is defined as follows:

Definition 7 (End-to-end latency constraint)**.** End-to-end latency constraint $L(u_i, u_j)$ is defined as an upper bound on the time between the source task u_i started its l'th execution and the time the sink task u_j finished its l'th execution, that means Eq (2.1) must hold.

$$f(u_i, l) - s(u_j, l) \leq L(u_i, u_j) \tag{2.1}$$

Therefore, end-to-end latency is the total time data takes to ripple through the chain of tasks. The end-to-end latency of the job in Fig. 2.3 is made visible in Fig. 2.4. For this job a possible schedule for task u_1 through u_3 is depicted assuming that the tasks consume and produce one container per execution. The source and sink tasks u_1 and u_3 are executed with the sample frequency f_s and the sample period is $1/f_s$. Now the end-to-end latency is the time between task u_1 started to produce the l'th container and task u_3 finished consuming the l'th container, as depicted in Fig. 2.4.

2.3 Sample-rate conversion

Sample-rate conversion is required when the sample frequency of the source task is different from the sample frequency of the sink task. Sample-rate conversion can be implicit in case of a so called multi-rate graph where tasks consume a number of containers and produce a different number of containers per execution of the task. It can also be explicit, so that it is specified by a sample-rate converter task in the task graph. Sample-rate conversion can be synchronous or asynchronous (also known as fixed or flexible sample-rate conversion [23]). Synchronous sample-rate conversion can be used when there is only one source or sink task that has a throughput constraint, e.g. in case of MP3 playback from compact-disc, as we will see in Section 2.4.1. It can also be used when there is a fixed rate between the sample-rate frequencies of the source and sink tasks, i.e. at design time there exists a $k \in \mathbb{Q} \wedge k > 0$ (i.e. k is a positive rational number) so that $f_{\text{source}} = k \cdot f_{\text{sink}}$. In this case, the consumed

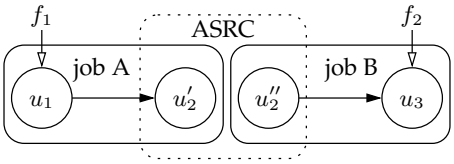

Figure 2.5: An asynchronous sample-rate converter can be seen as a composition of two tasks. Task u_2' is the sink of job A and task u_2'' is the source of job B.

number of containers related to the produced number of containers is predefined and known at design time. Therefore, synchronous sample-rate conversion can be represented with one task.

Asynchronous sample-rate conversion is required when the source and sink tasks have independent sample frequencies. In this case, the relation between the consumed number of containers and the produced number of containers, is not known at design time. Asynchronous sample rate conversion works as follows. Next to the media samples it receives also time stamps when input samples arrived at the source and when output samples should have been produced at the sink. From the samples and the received time stamps, the sample-rate converter is able to reconstruct an over-sampled media stream. The output stream is then derived from the over-sampled media stream and the time stamps on which the output should have been produced. The implementation of an asynchronous sample-rate converter consists of two tasks that share their state data. One task will consume containers from the input stream and the number of executions of this task matches with the sample frequency of the source task. The other task produces containers to the output stream and the number of executions of this task matches with the sample frequency of the sink task. Therefore, an asynchronous sample-rate converter is represented with two tasks (u_2' and u_2'' in Fig. 2.5) that are executed on the same processor. More precisely, the asynchronous sample-rate converter will split the original job into two jobs, as depicted in Fig. 2.5. The sample-rate converter task u_2' is a sink task for job A, and task u_2'' is a source task for job B. The throughput constraint of job A is f_1 executions per second of task u_1 and the throughput constraint of job B is f_2 executions per second of task u_3. The tasks u_2' and u_2'' are both executed on the same processor and they are scheduled using a run-time scheduler, which will be described in part II of this thesis.

2.4 Jobs and use cases in the infotainment nucleus

In this section, we describe the infotainment nucleus of two generations car-radio platforms, namely generation three and four. Generation three is based on NXP's car-radio chip SAF7780 [88, 8], which went in production in 2007. For this generation, we will do a quantitative comparison between our proposed network-based architecture and the existing architecture with traditional interconnects in Chapter 5. The development of car-radio generation four started in 2007. Although the development of generation four is still in progress, we describe a few jobs for the purpose

of our case studies in part II of this thesis.

2.4.1 Generation three

The platform SAF7780 is, among other things, capable of terrestrial analog-radio reception, compressed audio playback and hands-free voice with acoustic echo cancellation, possibly in different use cases like single versus dual media sound. Next to the radio and audio jobs, the user-interface software of a customer is executed on the micro controller that is integrated in the chip. In this thesis, our focus is on the radio and audio jobs and not on the user-interface software, because that is dependent of the car-radio set maker.

The platform SAF7780 supports the following jobs: (i) analog terrestrial-radio reception, (ii) playback compressed audio, (iii) read one or more audio streams from input peripherals, (iv) acoustic echo cancellation for hands-free voice, and (v) audio post processing. An example of a supported use case is depicted in Fig. 2.6. All these jobs are equally important, i.e. all jobs have the same priority.

(i) The analog terrestrial-radio reception job comes in several flavours, like Amplitude Modulated (AM) radio, Frequency Modulated (FM) radio, and Weather Band (WB) radio. The input of the radio job is first sampled at 41.6 MHz by an analog-to-digital converter (ADC), and then down-sampled to a sample frequency of 325 kHz by a primary decimation chain (PDC). The radio processing is performed by the radio demodulation tasks RD1 and RD2 in combination with CRD and FIR hardware accelerator tasks. The radio job has an Automatic Gain Control (AGC) that controls the amplitude of the analog radio signal at the input. The automatic gain control algorithm requires a feedback signal from task RD1 to the tuner chip. The tuner chip is designed with analog hardware, therefore this signal is converted with a digital-to-analog converter (AGC-DAC). The received Radio Data Service (RDS) messages are decoded by the radio data service decoding task RDS-dec and they are sent via a UART peripheral to the display in a car. The radio processing implicit down converts the 325 kHz radio stream with a factor eight, into an audio stream with a sample rate of 40.625 kHz. An asynchronous sample-rate converter task (ASRC) converts the 40.625 kHz audio stream into a 44.1 kHz audio stream that matches the digital-to-analog converter task DAC.

(ii) The playback compressed-audio job receives its input data from a compact disc or a portable storage device. Two compression formats are supported, namely *MPEG-1 audio layer 3* (more commonly referred to as MP3) and *windows media audio* (which is developed by Microsoft). The compressed-audio playback job is depicted in Fig. 2.6 and it is composed of a Block Reader (BR) task, Source Decoder (SD) task, and a synchronous sample-rate converter (SSRC) task. The task BR is responsible for reading the input data from a compact disc (CD) or portable memory device. This data is transferred into bursts of 512 words to task SD, which will decode the compressed audio stream. The source decoder supports input bit-rates up to 320 kbit/s. At the output of task SD, the container size is one stereo sample of two words and the available sample frequencies are 32, 44.1 and 48 kHz. The output stream will be converted by a sample rate converter so that it will match the 44.1 kHz sample frequency of the digital-to-analog converters.

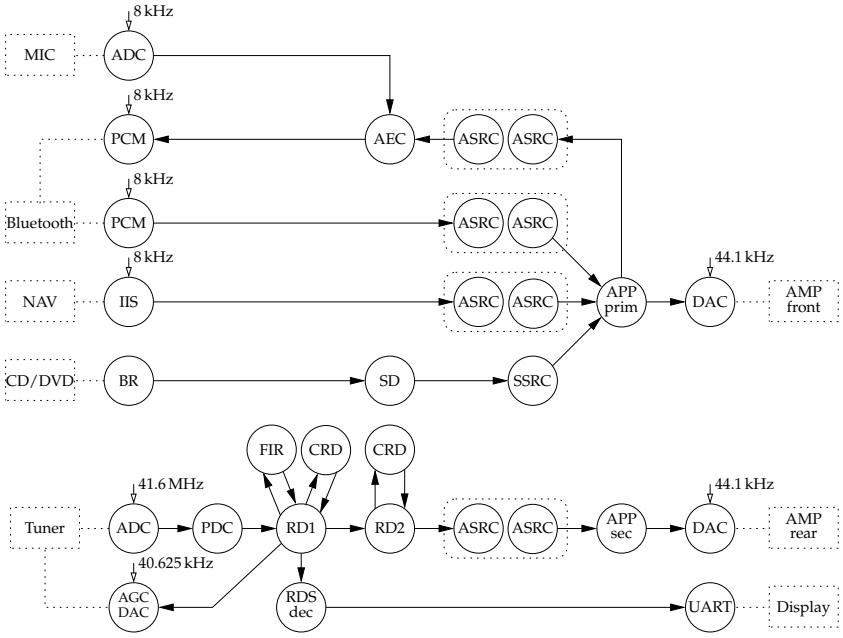

Figure 2.6: Task graph of a dual-media use case supported by car-radio generation three.

(iii) Additional to the analog terrestrial-radio and compressed audio stream, the system can have one or more audio input streams from several input peripherals. In generation three, there are two analog-to-digital converter input peripherals, four Inter-IC Sound (IIS) digital input peripherals, one Puls-Code Modulation (PCM) input peripheral, and one Sony/Philips Digital Interconnect Format (SPDIF) input peripheral. The external input device can vary from a navigation system (NAV), hands-free phone, up to a compact-disc changer, or Digital Video Disc (DVD) player. Typically, digital input streams are converted to a sample frequency of 44.1 kHz. The container sizes of the input streams are one word for mono audio samples and two words for stereo audio samples.

(iv) Acoustic Echo Cancellation (AEC) is used in a use case where a driver makes a hands-free phone call while the driver is listening to background music. It is used to improve voice quality on a telephone call by removing echo from voice communication and to cancel background sound from the loudspeakers. The background audio is removed by *subtracting* it from the signal that is recorded by the microphone of the hands-free terminal. The mobile phone is connected to a Bluetooth module, which in its turn is connected to the puls-code modulated peripheral, as depicted in Fig. 2.6. The microphone (MIC) signal, which is an analog signal, is sampled by an analog-to-digital converter. Background-audio substraction is done by task AEC. The audio stream that is sent to the loudspeaker, is subtracted from the microphone input stream before it is sent to the mobile phone. The voice-audio stream is mono audio with a sample frequency of 8 kHz and a container size of one word. The end-to-end latency constraint from task ADC to task PCM is 30 milliseconds, which is

derived from the specification of the Verband der Automobilindustrie [20].

(v) Finally, the audio post-processing job is responsible for, for example, audio enhancement, equalisation, blending input streams, and changing the audio volume, fader and balance. Generation three supports two modes, namely single media (only primary) and dual media (primary and secondary). In case of dual media, there are different streams for front-seat and rear-seat audio, as depicted in Fig. 2.6. The Audio Post Processing (APP) of the primary and secondary audio streams are performed by task APP-prim and APP-sec, respectively. The audio streams have a sample frequency of 44.1 kHz and container sizes vary between one word for mono audio and two words for stereo audio. Audio streams can be sent to several output peripherals. In generation three, there are two stereo digital-to-analog converter peripherals, one puls-code modulation peripheral, and three inter-IC sound output peripherals. Examples of external devices are amplifiers (AMP) for front-seat speakers, rear-seat speakers, centre speaker, sub-woofer speaker, and headphones.

2.4.2 Generation four

The platform SAF7780 can already connect a CD/DVD changer, personal navigation device, portable storage device and portable media player, for example, via a Universal Serial Bus (USB) or Bluetooth module. Connectivity is becoming more popular in infotainment-nucleus generation four, for example, the platform should be able to connect a hard-disc memory device (for mass storage) and a WiFi module (for synchronisation with a home server). Furthermore, the number of supported jobs and the number of use cases are increased. The main differences in functionality between generation three and four are: advanced audio post processing with a sample rate up to 96 kHz, playback compressed audio with the support for various formats with and without digital right management, encoding audio streams for ripping audio streams to a hard disc, and digital terrestrial-radio reception. There are a number of digital radio-transmission standards and different standards are common in different regions.

For the purpose of our case studies, we make use of two jobs from generation four. These jobs are: (i) channel equalisation for FM radio and (ii) Digital Radio Mondiale [78].

(i) Channel equalisation reduces multi-path distortion in an FM signal, as was illustrated in Fig. 1.1. This distortion is caused by houses, cars, and hills that reflect FM signals and these reflections cause variations in the magnitude and phase of the signal. The user can add the channel equalisation in front of an FM-radio demodulation job for improved radio reception. The input of the channel equaliser is coming from task PDC and the output is going to task RD1. The task graph of our channel equaliser job will be described in Section 6. This job should keep up the output stream of task PDC, which has a sample frequency of 325 kHz.

(ii) Digital radio Mondiale [78] is a technology designed to work over the bands currently used for AM broadcast. This technology is based on in-band on-channel, which is a method of transmitting digital radio and analog radio broadcast signals simultaneously on the same frequency. By utilising additional digital subcarriers or sidebands, digital information is piggybacked on a normal AM signal, thus avoid-

ing any complicated extra frequency allocation issues. Digital Radio Mondiale can fit more channels than AM radio, at higher quality, into a given amount of bandwidth, using various compression and decompression techniques. It has been designed especially to use portions of older AM transmitter facilities such as antennas, avoiding major new investments. Furthermore, it is robust against the fading and interference which often plagues conventional broadcasting on these frequency ranges. The task graph of our Digital-Radio-Mondiale job will be described in Chapter 8. This job should keep up with the periodic input task ADC, which is triggered with a sample frequency of 48 kHz. The main difference between our analog and digital radio job is that the digital radio job requires a larger memory footprint than the analog radio job.

Chapter 3

Multiprocessor architecture for streaming applications

Existing multiprocessors can be easy to program, but they are, typically, hard to tune. It can be time consuming before every job satisfies its real-time constraints. The end-to-end timing behaviour depends, among others, on the interaction between multiple local arbiters that can be mutually dependent. Every arbitration level will, typically, increase the uncertainty in the temporal behaviour at design time.

In this chapter we describe our multiprocessor architecture with limited resource sharing. The architecture is optimised for applications that belong to the class of streaming, from which the characteristics are described in previous chapter. First, in Section 3.1, we describe the requirements for a multiprocessor architecture that has a predictable temporal behaviour. In Section 3.2, we introduce a tiled multiprocessor architecture that is built on top of a network-on-chip. Section 3.3 describes the communication and synchronisation between tasks that are executed on the multiprocessor architecture. Finally, we summarise this chapter with concluding remarks in Section 3.5.

3.1 Requirements for a predictable architecture

In this thesis, we build a multiprocessor system that has a predictable temporal behaviour so that we can reason about the temporal behaviour of the jobs when they are mapped to the system at design time. For a predictable system, we must be able to derive a conservatively-estimated lower bound on throughput and a conservatively-estimated upper bound on end-to-end latency for each active job, as described in Section 1.4. Therefore, the architecture should enable the derivation of a conservatively-estimated upper bound on the execution time of a task. This results in a requirement for predictable memory-access latencies for every processor in our multiprocessor architecture. Furthermore, the architecture should support a conservatively-estimated upper bound on the communication latency for the communication between two tasks. This requires a communication infrastructure

with guaranteed communication services. When sharing resources like processors and communication resources, the arbiter should guarantee a minimum resource budget and it should bound the interference, otherwise we are unable to derive conservatively-estimated bounds on execution times and communication latencies. Finally, the derived bounds should be tight, so that we can come to a cost-efficient implementation.

In part I of this thesis, we focus on an architecture that minimises the uncertainty in the temporal behaviour. That means, an architecture that enables the derivation of tight upper bounds on execution times of tasks and tight upper bounds on communication latencies. This architecture can support most of the streaming jobs from infotainment-nucleus generation three. In Part II, we will describe architectural extensions to be able to support all jobs from infotainment-nucleus generation three and four.

3.2 Multiprocessor architecture

Embedded media applications demand for an increasing performance without increasing the area and power dissipation. For the performance and power efficiency reasons, such media applications are typically executed on heterogeneous platforms that contain different types of processing cores, like micro controllers, digital signal processors, and application-specific hardware accelerators. This allows a task to be executed on the processor that is most efficient in terms of performance and power dissipation. Furthermore, a system contains peripherals for the communication with the system environment and memories for storing the job's program code and data.

On-chip memories will be distributed, that means attached close to the processing cores, again for reasons of performance and power dissipation. This leads to the formation of clusters, which we refer to as tiles, that consist of processing cores with local memory. Each tile can have one or more memories, which we refer to as *local memories*. The processing core in a tile has low-latency memory accesses to its local memory, which enables a high processor utilisation and a high performance. A tile can also contain a peripheral for the communication with the system environment.

The integration of our heterogeneous tiles into a working system is a major challenge. The bottleneck in such multiprocessor architectures shifts from computation towards communication. Getting the right data at the right place at the right time will dominate the architecture. Currently busses and custom interconnects (point-to-point, crossbar switches) are often used, but with an increasing number of tiles designed in technologies with decreasing dimension, they do not sufficiently address hardware problems (deep sub-micron VLSI design) and software problems (application programming). Networks-on-chip tackle these problems and therefore are a better answer to the integration challenges.

First, *hardware problems*: networks help to answer some basic *deep sub-micron questions* because they structure the top level wires in a chip, and facilitate modular design [74]. Structured wiring results in predictable electrical parameters, like crosstalk. Network interconnects are *segmented* and *multi-hop*. The advantage of segments is that only those segments are activated that are actually used in the communication.

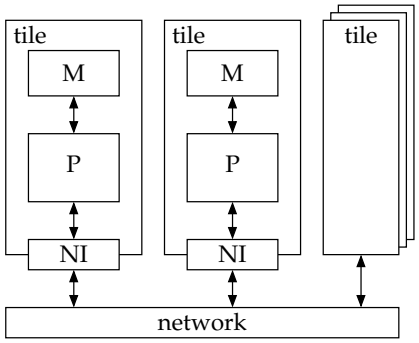

Figure 3.1: Tiled-multiprocessor architecture where tiles communicate via a network-on-chip.

So only those segments dissipate power. In addition, multiple segments can be active simultaneously, enhancing throughput (or reducing wiring cost). Multi-hop is needed because the transport delay from source to destination can become longer than the clock period.

Second, *software problems*: to reduce the programming effort proper transport-level services have to be defined. In particular, networks that offer *guaranteed-communication services* make systems on chip more robust, easier to design [34] and easier to program with a much lower non-recurring engineering cost. Networks also provide concurrency, i.e. several transactions can be dealt with simultaneously.

To come to a scalable architecture, the inter-tile communication infrastructure should be scalable. Furthermore, each tile should execute at its own desirable clock frequency, i.e. globally there is asynchronous communication and locally inside a tile there is synchronous communication. Tiles are very much autonomous, i.e. they run independent from other tiles and from the communication infrastructure. Similar multiprocessor architectures can be found in literature. For example, the author of [18] describes a generic scalable multiprocessor architecture that consists of a collection of essentially complete computers (tiles in our case), including one or more processors and memory, communicating through a general-purpose high-performance scalable interconnect.

In part I, we assume the multiprocessor architecture that is depicted in Fig. 3.1. A tile consists of a processing core (P), a memory (M) and a network interface (NI). The processing core is the only master that is accessing its own local memory. Therefore, the processing core cannot access a memory in another tile. The processing core has predictable memory-access latencies, because it is the only master that is accessing its own local memory. This enables us to derive tight conservatively-estimated upper bounds on execution times of tasks, as is required for a predictable architecture. The disadvantages of this memory architecture are: each processing core has a fixed amount of available memory space and all program code, program data, and state of each task must fit in the on-chip local memory of a processor. Therefore, this architecture can become expensive in terms of area cost and the flexibility is limited. In part II of this thesis, we will extend our multiprocessor architecture with shared lo-

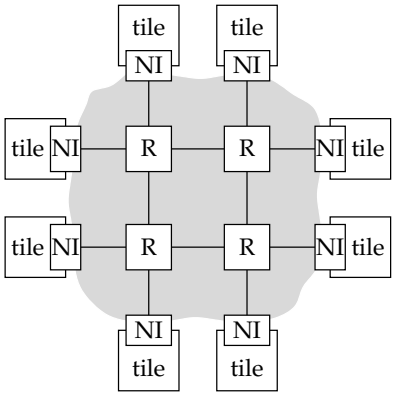

Figure 3.2: Network is composed of Network Interfaces (NI) and Routers (R) that are combined in a scalable fashion.

cal memories and a shared off-chip memory to make the multiprocessor architecture more flexible and more cost efficient.

3.2.1 Æthereal network-on-chip

In our multiprocessor, tiles communicate via the Æthereal network-on-chip [33]. The Æthereal network is modular because it is built with only two parameterisable components, network interfaces [70] and routers [68], that are combined in a scalable fashion to form the complete communication infrastructure, as depicted in Fig. 3.2. Each tile is connected to a network interface, which translates the tile's communication protocol to network-internal packet-based protocol. Furthermore, they implement clock-domain crossings, so that each tile can run at its own desired clock frequency. Each network interface is connected to a router via a link. The routers are interconnected by links and the router network can have a generic, but predefined, topology.

Communication channels between tasks can be mapped to network connections, in case the tasks are executed on processors from different tiles. The definition of a network connection is as follows:

Definition 8 (Network connection)**.** A network connection is a point-to-point connection between two network interfaces. Each network connection contains a set of network-interface buffers that are dedicated to the network connection. Data that is sent to the buffer at one network interface will be delivered, by the network, to the buffer in the other network interface.

Figure 3.3 depicts a network connection that transfers data from an output buffer in one network interface to an input buffer in another network interface. For each network connection, flow control is used to prevent data loss caused by buffer overflow and to prevent deadlock. Flow control is implemented using credits [83] that are sent back from destination to source [70] after data is consumed from the buffer

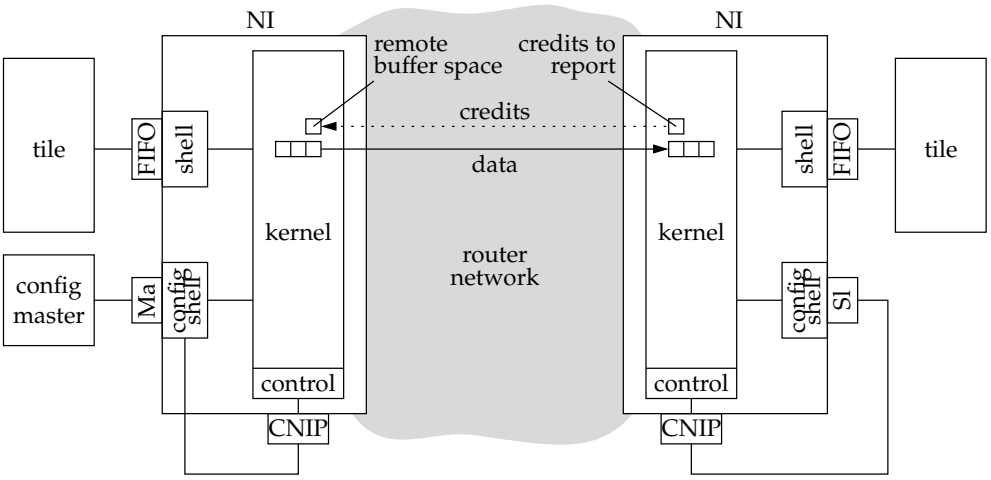

Figure 3.3: Network-interface architecture and network-connection implementation.

at the destination. The credit flow is depicted with the dotted arrow in Fig. 3.3 and it works as follows. For each network connection, there is a counter (remote buffer space) tracking the empty buffer space of the destination network-interface buffer. This counter is initialised with the destination buffer capacity. When data is send from source network-interface buffer, the counter is decremented. When data is consumed at the destination network-interface buffer, credits are produced in a counter (credits to report) to indicate that more empty space is available. These credits are send to the source network interface (dashed line in Fig. 3.3) to be added to the counter remote buffer space.

The network-interface buffers have three purposes. First, they implement clock boundaries between the tiles and the network. Second, they decouple and isolate tile communication behaviour from network behaviour. That is, data bursts from the tile are buffered to fit the network's schedule, and vice versa. Third, they hide the round-trip latency of flow-control credits for the request and response channel.

The Æthereal network offers two types of network connections (or service classes): *guaranteed throughput*, and *best effort* [33]. To guarantee bandwidth and latency, resources such as links are allocated to network connections [25]. Both, guaranteed-throughput and best-effort network connections use source routing, i.e. the path to the destination is decided a priori and it is known by the source network interface. The source network interface is configured with this path. Data is sent from one network interface to another using packets and wormhole routing, which has a low buffering cost. In case of wormhole routing, the packet contains a header with the destination. The transmission from the source to the destination is done through a sequence of routers. When packets arrives at an intermediate router for forwarding, the router examines the header, sets up a circuit to the next router, and then transfers the packet. A packet is broken in smaller pieces, called flits. Flits of a single packet may occupy multiple consecutive routers and links, like a worm. One flit is defined as three words of 32 bits. Every router contains guaranteed-throughput

input buffers consisting of one flit and best-effort buffers of eight flits. Guaranteed-throughput input buffers require only one flit, as guaranteed-throughput packets never stall in the router network. This is accomplished by globally time-division-multiplex scheduling packet injection from the network interfaces to the routers, so that packets never use the same link at the same time (thus avoiding contention). The pipelined virtual connections that are implemented this way, have a guaranteed bandwidth (roughly, the number of slots reserved for the connection) and bounded latency. The time-division-multiplex slot allocation is an optimisation problem per use case. Guaranteed-throughput connections are used for transferring data containers over communication channels, so that we can derive conservatively-estimated upper bounds on communication latency. Best-effort connections use slots that have not been reserved, or have not been used by guaranteed-throughput packets. Best-effort packets are scheduled dynamically at run time, and their behaviour (bandwidth and latency) is therefore dependent on the guaranteed-throughput packets. Furthermore, packets may occupy multiple routers causing dependencies between routers. Therefore, it is hard to predict the behaviour (bandwidth and latency) of best-effort packets. Data that is sent on best-effort network connections is guaranteed to arrive at the destination (due to flow control), but without minimum bandwidth and maximum latency bounds. Best-effort network connections can be used for programming the network and tiles, in case the job start-up times are not time critical.

From a tile's perspective, network communication can be divided into address-less and address-based communication. In case of address-less communication, a tile is connected to a network interface via a hardware FIFO interface port. Master (Ma) and slave (Sl) ports are used to connect a tile to a network interface in case of address-based communication. These ports are connected to a *network-interface kernel* via *network-interface shells*, as shown in Fig. 3.3. A network-interface shell converts transaction requests of a particular IP protocol (e.g. DTL Peer-to-Peer Streaming Data [65]), into transport-layer messages. For address-based communication, transaction requests contain a command, address, and potentially some data (in case of write accesses). As commands we distinguish read, acknowledge write, and posted write accesses. In case of accessing a remote memory with read or acknowledge write, the processor waits until it receives the read data or an acknowledge that the data is written. For posted write accesses, the transaction is stored in the network-interface buffer and the processor does not have to wait for an acknowledge, but it can continue doing useful work while the transaction is transported over the network and the data is written at the destination. The kernel converts these generic transport-layer messages into network-layer guaranteed-throughput or best-effort packets, this conversion is referred to as packetisation. It is also responsible for depacketisation, which is the conversion of the guaranteed-throughput or best-effort packets into generic transport-layer messages.

In part I of this thesis, we focus on address-less communication, because it matches the streaming model of computation. An example of address-less communication is DTL Peer-to-Peer Streaming Data [65]. In address-less communication, only data is sent over network connections. No addresses need to be sent additional to the data, because these connections are configured a priori, in such a way that the source, destination and the path are known. Each network-interface buffer can be addressed

directly by the processing core in a tile. Address-based communication will be explored in part II of this thesis.

The network must be programmed with the appropriate configuration at run time. The configuration consists of allocating slots to network connections and the routing path from source to destination. In this thesis, we solve the configuration for each use case at design time, resulting in predefined configurations. At run time these configurations can be programmed (or loaded) into the network by a configuration master, e.g. the host micro controller. Programming the network is done by programming the configuration for all connections in every network interface via address-based communication (DTL Memory-Mapped Input Output [65]). The network interfaces can be programmed via a so called Configuration Network-Interface Port (CNIP). This configuration port is looped back to a network-interface slave port, as depicted at the bottom right in Fig. 3.3. Therefore, the network is configured using itself and no separate control interconnect is required [33].

3.3 Communication and synchronisation between tasks

Streaming jobs from the infotainment nucleus will be mapped onto our multiprocessor architecture. A streaming job consists of tasks and communication channels, as described in Section 2.1. Tasks are executed by the processors in tiles and communication channels are implemented as intra-tile communication or inter-tile communication. To illustrate the mapping step, we map a producer consumer job to the multiprocessor platform in Fig. 3.1.

When two communicating task are executed on the same processor, then the communication channel between these tasks is implemented with intra-tile communication. Intra-tile communication is implemented with a circular buffer [26] that is located in the local memory, as shown by the communication channel c_1 between task u_1 and u_2 in Fig. 3.4. The buffer management is done at the level of containers and the synchronisation between containers is FIFO based. A processor has random access within a container, because it is stored in the local memory of the processor. A circular buffer consists of a buffer administration and of a memory region for storing the data containers. The buffer administration consists of a *base-address*, *size*, *Read-Pointer* (RP), and *Write-Pointer* (WP). The memory region where the data containers are stored, is defined by the administration values base-address and size. The read-pointer is pointing to the memory location where the first full container is stored. The memory location of the first empty container is defined by the write-pointer. A task only reads or writes from a buffer if there are, respectively, full or empty containers available. A task can check for the number of available full and empty containers in the buffer, by investigating the buffer's administration values. After the consuming task has read a full container, it updates the read-pointer so that it points to the next full container in the buffer. After the producing task has filled an empty container, it updates the write-pointer. Once a pointer reaches a memory location that is outside the buffer's memory region, the pointer value wraps around and will point to the beginning of the buffer again. Therefore, this buffer is called a circular buffer. The protocol for intra-tile communication will be described in six steps. The steps are also depicted in Fig. 3.4 and are as follows:

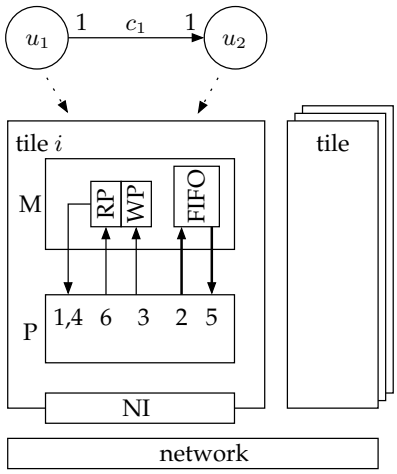

Figure 3.4: Streaming communication, between two tasks mapped to the same tile, via a circular buffer located in the local memory of a processor.

1. The processor, on which task u_1 is executed, reads the administration values from its local memory (i.e. read and write pointers) to see if there is space available to store a data container.
2. In case there is space available, task u_1 produces output data and the processor stores it in the circular buffer that is located in its local memory.
3. After the data container is written, the processor updates the write pointer in the buffer administration.
4. The processor, which also executes task u_2, reads the circular-buffer's administration values from its local memory (i.e. read and write pointers) to see if there is a data container available in the circular buffer.
5. In case there is a data container available, the processor reads the data container from its local memory.
6. After the processor has finished reading the data, it updates the read pointer in the buffer administration.

When two communicating task are executed on two different processors, then the communication channel between these tasks is implemented with inter-tile communication. An example of inter-tile communication, is the implementation of the communication channel c_1 between task u_1 and u_2 in Fig. 3.5. This communication is implemented with address-less communication over a network connection between tile i and tile j. Address-less communication requires the allocation of a network connection for every inter-tile communication channel to distinguish data between communication channels, as no address information is sent along with the data. Network interfaces contain input and output buffers that are implemented in hardware. Data containers are stored in network-interface buffers and buffer accesses are word

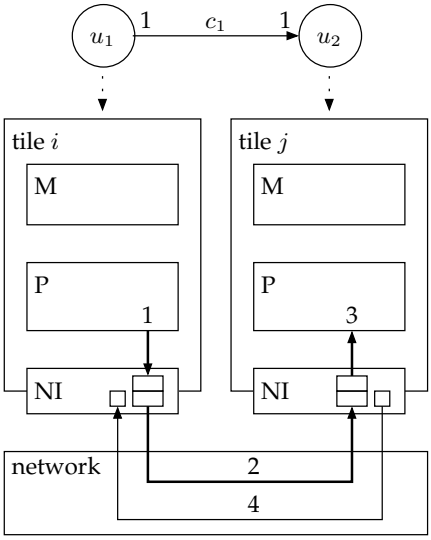

Figure 3.5: Address-less communication between two tiles in the multiprocessor.

by word and in sequential order. A processor can write directly to an output buffer by accessing the tail of this FIFO buffer. A processor can read from an input buffer by accessing the head of this FIFO buffer. The processor is blocked if it writes to a full network-interface buffer, or if it reads from an empty network-interface buffer. The protocol of inter-tile communication is described with four steps that are depicted in Fig. 3.5. In spite of the fact that we describe the steps sequentially, the processors and the network will perform these steps concurrently. Each step is implemented with a blocking semantics so that the whole path shows back-pressure behaviour. The four steps are as follows:

1. The processor, on which task u_1 is executed, produces an output container and stores it in the network-interface buffer in tile i.
2. The network will transfer the data from the network-interface buffer in tile i to the network-interface buffer in tile j.
3. The processor, on which task u_2 is executed, reads the data from the input buffer that is located in the network interface of tile j.
4. After the data has been read from the input buffer, the network will sent credits back to the network interface in tile i to report that there is space available in the network-interface buffer which is located in tile j.

The communication latency of address-less communication over a network connection, from a source network-interface port to a destination network-interface port, is defined as follows:

Definition 9 (Address-less communication latency)**.** The address-less communication latency of a container that is sent over a network connection, from a source

network-interface port to a destination network-interface port, is defined as the time between the moment that this container is accepted by the source network interface and the moment when this container can be read from the destination network interface, assuming that no other data is pending in the network connection.

Communication latency is composed of *waiting latency* and *network latency*. Waiting latency is defined as the difference in time at which the container is arrived in the source network-interface buffer and the time it has been scheduled for packetisation. This time depends on the distances between two allocated slots in the slot table.

Network latency is a consequence of latency in the network-interface shells, network-interface kernels, clock-domain crossings, routers, arbitration, and end-to-end flow control. A network-interface shell introduces two to four cycles latency (e.g. depending on address-less or address-based communication). Between one and three clock-cycles latency is introduced by the network-interface kernels (as data needs to be aligned to a three word flit boundary) [70]. The clock-domain crossings, between source and destination, introduce two clock-cycles latency at the destination clock. Three clock-cycles latency are introduced per router. Per slot-table rotation a predefined number of words can be sent over the network. Therefore, time-division-multiplex scheduling causes additional latency, namely a predefined number of slot-table rotations that are necessary for transferring a message. The round-trip latency of end-to-end flow-control credits can be neglected if it is hidden by sufficiently large network-interface buffers [25].

Notice that, at design time, the communication latency cannot be bounded from above, for a container that is sent via a best-effort network connection. Because in the slot table no slots are allocated to best-effort network connections and the connection may be starved.

3.4 Static-order scheduling of tasks

In case multiple tasks are executed by the same processor (e.g. tasks u_1 and u_2 in Fig. 3.4), a scheduling mechanism determines the execution order of these tasks. In this thesis, such a scheduling mechanism is divided into *design-time scheduling* and *run-time scheduling*. Part I of this thesis considers design-time scheduling of tasks and run-time scheduling of tasks is considered in part II of this thesis. Examples of design-time schedules are fully-static schedules [49] and static-order schedules [76]. In a fully-static schedule, the start times of tasks are predefined. An advantage is a low synchronisation overhead. The disadvantage is that a global notion of time is required. This can be problematic when each tile has its own clock frequency. In static-order schedules, on the contrary, a global notion of time is not required. It only requires an execution order of the tasks that are executed by a single processor. Therefore, we make use of static-order schedules. That means processors can only execute tasks from the same job or from jobs which are triggered by the same (or a derived) sample frequency. For each processor, the static-order schedule is determined in such a way that all throughput and end-to-end latency requirements are met, as we will describe in next chapter.

3.5 Concluding remarks

In this chapter, we described our first architecture that limits the uncertainty in the temporal behaviour to enable tight bounds on throughput and end-to-end latency.

The multiprocessor architecture has processing tiles that contain local memories and the processor is the only master who is accessing its local memory. Furthermore, it makes use of local synchronisation, which means that all input and output containers of a task are stored locally (i.e. stored in local memories and network-interface buffers). The transport from one tile to another is implemented as a separate step, i.e. a copy action over the network using the guaranteed communication services of the network. This way computation and communication are separated. Tasks are executed on processors by means of static-order schedules. So that the execution order of tasks, that are executed on the same processor, is known at design time.

The advantage of this architecture is that we can derive tight conservatively-estimated bounds on execution times and communication latencies. This enables us to derive tight bounds on throughput and end-to-end latency. There are also some limitations. First, the task's program code and the complete working data set must fit in the local memory of a tile and, in case of inter-tile communication, the network-interface buffers must be able to contain a complete data container. So containers must be rather small. Second, it is a hardware driven approach which lacks flexibility. For example, the number of supported network connections is coupled to the number of hardware buffers in the network interfaces. Third, data containers need to be stored at both sides (source and destination) which can have a negative impact on area and end-to-end latency. As a consequence, the target domain is sample-based processing, like analog terrestrial radio.

Chapter 4

Analysing real-time performance

Performance-analysis techniques are used to analyse the temporal behaviour of jobs, in order to verify that its real-time requirements, like throughput and end-to-end latency, are met. In this chapter, we motivate dataflow-analysis techniques to verify the real-time requirements of every job running on our multiprocessor platform.

First, in Section 4.1, we describe the use of dataflow graphs for modelling a job that is mapped to the multiprocessor platform. Next, Section 4.2 introduces the semantics of dataflow graphs that will be used throughout this thesis. Section 4.3 describes how we construct a dataflow model from our implementation, in case tasks are executed in a static-order schedule. Existing dataflow analysis techniques are described in Section 4.4. Finally, we summarise with concluding remarks in Section 4.5.

4.1 Modelling a job that is mapped to the platform

In this thesis, we focus on performance analysis at design time. Figure 4.1 depicts the relation between mapping a job on a platform, the job's dataflow model representation, and the job's real-time performance analysis.

A job is specified by a task graph, as described in Chapter 2. A task is typically specified in a sequential programming language, like C. The job's task graph as well as the tasks are assumed to be given in this thesis. Jobs can contain loops (cycles in the task graph) that influence the temporal behaviour of a job.

The tile-based multiprocessor architecture is described in Chapter 3. A streaming job is mapped to the multiprocessor platform. For performance reasons, a job is mapped to multiple tiles and therefore tasks can execute in parallel. The temporal behaviour of a job's implementation depends also on the mapping of the job to the multiprocessor platform. A mapping of a job consists of the binding of tasks to processors, static-order scheduling of tasks on a processor, configuration of network connections, and buffer sizing.

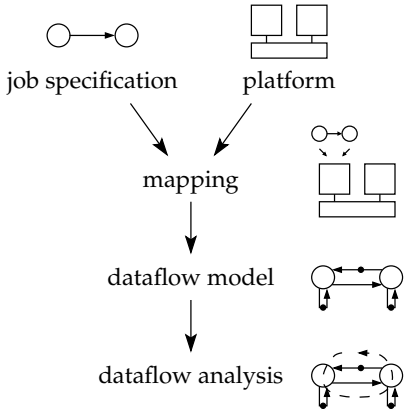

Figure 4.1: The mapping of a job on an architecture, the dataflow model that represents the job's implementation, and the dataflow analysis to derive the job's real-time performance.

The analysis technique should account for (cyclic) data dependencies between tasks, because they can affect the temporal behaviour. Cyclic dependencies can be caused by functional dependencies (e.g. in the case of feedback loops), schedule dependencies (e.g. in the case of static-order scheduling), and back-pressure (to prevent buffer overflow). We make use of dataflow modelling and analysis techniques, because they can take into account cyclic dependencies that influence the temporal behaviour. The streaming job that is mapped to the multiprocessor architecture is modelled in a dataflow graph. That means, it models the execution of tasks on processors, the static-order scheduling of tasks, the intra-tile as well as inter-tile communication between tasks, and the buffer capacities of circular buffers and network-interface buffers. The dataflow-model semantics will be described in Section 4.2 and the construction of a dataflow model will be described in Section 4.3.

By analysing the job's dataflow model with existing dataflow-analysis techniques, we can derive conservatively-estimated bounds on the job's throughput and end-to-end latency. The derived bounds should be tight, because a large deviation can result in a significantly over-dimensioned system. Existing dataflow-analysis techniques will be described in Section 4.4.

4.2 Dataflow model preliminaries

In this section, we describe the semantics of a dataflow graph [50]. A dataflow graph $G = (V, E)$ is a directed graph that consists of a finite set of actors V, and a finite set of directed edges $E = \{(v_i, v_j)|v_i, v_j \in V\}$. An actor represents a quantum of work, e.g. a function with an input and output. Actors synchronise by communicating tokens over edges that represent FIFO channels. A token is used to represent a container in which a fixed amount of data can be stored. The number of initial tokens on an edge $e \in E$ is denoted with $\zeta(e)$. An initial token is depicted as a black dot on an

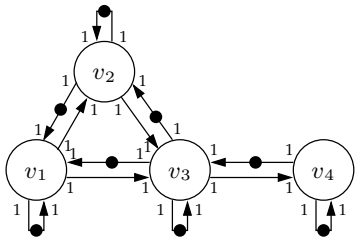

Figure 4.2: An example of a strongly connected dataflow graph G.

edge. In case there are more than one initial tokens, the number of initial tokens $\zeta(e)$ is depicted next to the black dot. An actor is enabled to fire when the *firing rule* is evaluated as true, i.e. the number of tokens that will be consumed is available on each input edge. The number of tokens consumed by actor v_i is determined by the edge $e \in E$ and equals $\gamma(e)$ tokens. When actor v_i finishes, it produces the specified number of tokens on each output edge $e = (v_i, v_j)$. The number of tokens produced is denoted by $\pi(e)$. If the number of tokens consumed ($\gamma(e)$) or produced ($\pi(e)$) are not depicted in the picture of a dataflow graph, then we assume them equal to one. This is done in order to make the graph more readable. An actor in the dataflow model supports auto-concurrency because an actor is per definition stateless. In the implementation tasks contain state and they are executed by a single processing core, therefore, they can only be started again after completing the previous execution. This is modelled in the dataflow graph with a self cycle $e = (v_i, v_i)$ with one initial token. In part I of this thesis, we assume that each actor has a self cycle with one initial token (which excludes auto concurrency). In part II, we will distinguish actors with and without a self cycle to be able to represent run-time scheduling of tasks in a dataflow model.

Dataflow graphs are extended with a notion of time, i.e. actors are annotated with an execution time [76]. The execution time $\rho(v_i)$ is the difference between the finish and the start time of actor v_i. The specified number of tokens is consumed in an atomic action from all input edges when the actor is started. When an actor v_i finishes, it produces the specified number of tokens on each output edge in an atomic action. Therefore, this consumption and production of tokens does not take time.

There are a number of classes of dataflow models described in literature, as already mentioned in Section 1.3.4. The main difference is in the expressiveness of an actor, as depicted in Fig. 4.3. In an Single-Rate DataFlow (SRDF) graph [50], which is a subclass of Multi-Rate DataFlow (MRDF) graphs [50], every actor consumes one token from every input edge and it produces one token to every output edge, i.e. $\gamma(e) = 1$ and $\pi(e) = 1$. As all production and consumption rates are equal to one, the repetition rate [76] (i.e. the relative firing frequency) equals one for every actor in the SRDF graph. This means that every actor is executed as often as any other actor in the SRDF graph. Actor v_1 in Fig. 4.3, is an example of an actor that consumes and produces one token per firing. Any consistent MRDF graph can be converted to an equivalent SRDF graph, by using the conversion algorithm in [76]. In an MRDF graph, the number of tokens consumed and produced by an actor equals, respec-

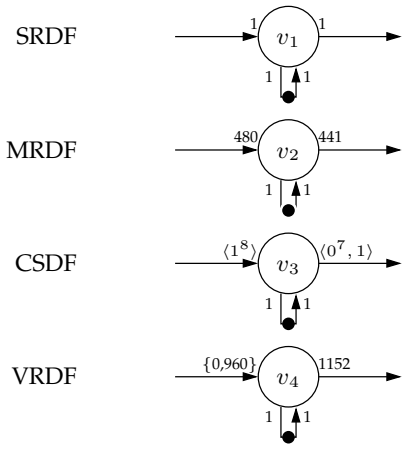

Figure 4.3: Actors from different classes of dataflow models.

tively, $\gamma(e)$ and $\pi(e)$, with $\gamma(e), \pi(e) \in \mathbb{N}$. In Fig. 4.3, actor v_2 is an example of an MRDF actor that consumes 480 tokens and produces 441 tokens per firing.

A Cyclo-Static DataFlow (CSDF) [9, 64] actor $v_i \in V$ has $\theta(v_i)$ distinct phases of execution and transitions from phase to phase in a cyclic fashion. The phase f of CSDF actor v_i in firing k is $f = ((k-1)\%\theta(v_i)) + 1$, where $k \geq 1$, $1 \leq f \leq \theta(v_i)$, and $x\%y$ stands for x modulo y with the result the same sign as the divisor y. The execution time $\rho(v_i, f)$ is the difference between the finish and the start time of phase f of actor v_i. The number of tokens consumed by an CSDF actor is determined by the edge $e \in E$ and the current phase f of the actor and therefore equals $\gamma(e, f)$ tokens. The number of tokens produced in a phase will be denoted by $\pi(e, f)$. Actor v_4 in Fig. 4.3 is an example of a CSDF actor. The notation $\langle 1^8 \rangle$ and $\langle 0^7, 1 \rangle$ is equivalent to, respectively, $\langle 1, 1, 1, 1, 1, 1, 1, 1 \rangle$ and $\langle 0, 0, 0, 0, 0, 0, 0, 1 \rangle$, which means the number of tokens consumed and produced in each phase of the task. If the number of tokens consumed ($\gamma(e, f)$) or produced ($\pi(e, f)$) are not depicted in the picture of a dataflow graph, then we assume them equal to one for every phase f.

For a Variable Rate DataFlow (VRDF) [93] actor, the number of tokens that are consumed on an edge $e \in E$, in a particular firing, is a value taken from $\gamma(e)$, where $\gamma(e)$ has a minimum and maximum token consumption quanta. For example in Fig. 4.3, the minimum and maximum consumed number of tokens is, respectively, 0 and 960 tokens per firing of actor v_4. Similar can hold for the token production quantum in that firing on an edge e, which is a value taken from $\pi(e)$, where $\pi(e)$ has a minimum and maximum token consumption quanta. For example, actor v_4 has a fixed number of produced tokens per firing, namely 1152.

In this thesis, we make use of the CSDF model. The CSDF model will be used to model the job that is executed on the multiprocessor platform. The throughput and end-to-end latency are derived via a so called self-timed execution of the CSDF graph, which is defined as follows:

Definition 10 (Self-timed execution)**.** In a self-timed execution of a dataflow graph,

actors fire as soon as they are enabled.

Furthermore, we say that a CSDF graph maintains a FIFO ordering of tokens, if each actor either has a constant execution time, or has a self cycle with one token. This is because queues by definition maintain FIFO ordering of tokens, which means that tokens cannot overtake each other in such a CSDF graph. An important property is that self-timed execution of a strongly connected CSDF graph that maintains a FIFO ordering of tokens is monotonic in time, which is defined as follows [94].

Definition 11. A CSDF graph executes monotonically in time if a decrease in execution time or start time of any firing k of any actor v_i cannot lead to a later enabling of any firing l of any actor v_j.

If a CSDF graph G maintains FIFO ordering of tokens, the self-timed execution of G is monotonic [94]. This is because a decrease in execution time or start time can only lead to earlier token production times, and therefore only to an earlier actor enabling.

4.3 Dataflow model construction

In this section, we describe the construction of a dataflow graph that models a job which is mapped to our multiprocessor platform. This dataflow model takes into account: computation of tasks, communication between these tasks, arbitration at shared resources, and FIFO buffer capacities, because all these properties influence the job's temporal behaviour.

A streaming job is composed of tasks that communicate via communication channels, as described in Section 2.1. All tasks are bound to tiles in the multiprocessor platform and these tasks are executed on a processor or hardware accelerator. In the implementation, tasks read data from their input channels and produce data to their output channels during their execution. In the dataflow model, actors are enabled when the firing rule is evaluated as true. That means, actors are enabled if sufficient tokens are available at every input edge. As we will show in Section 4.3.1, when tasks are executed in a static-order schedule, each task can still be modelled with one actor despite the absence of the firing rule in the implementation. Therefore, tasks do not have to wait for sufficient full and empty containers, but they can already start executing after the previous task, in the static-order schedule, is finished. The task's conservatively-estimated upper bound on the execution time is taken as the execution time of the actor that represents this task.

Tasks communicate data containers over communication channels. These containers are represented by tokens that are communicated between actors in the dataflow model. A communication channel in the implementation has a bounded capacity, whereas the capacity of an edge, in the dataflow graph, is by definition unlimited. Therefore, a FIFO buffer with a bounded capacity is modelled, in the dataflow model, with two edges in opposite direction (a forward and backward edge). The availability of full containers in the FIFO buffer corresponds with the presence of tokens on the forward edge. If a task consumes a full container, it creates space in a FIFO buffer, which corresponds to the production of a token on the backward

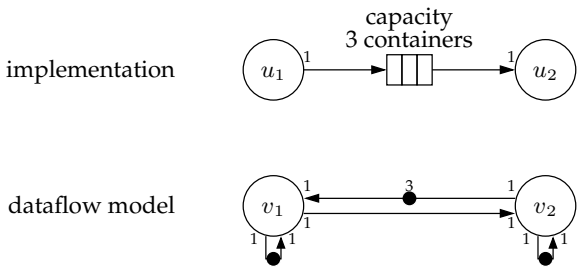

Figure 4.4: Dataflow model representation of a FIFO buffer with a bounded capacity.

edge. In other words, the production of tokens on the backward edge represents the production of empty containers in the FIFO buffer of the implementation. The number of initial tokens on both edges represents the buffer capacity in the number of containers. An example of a dataflow model of a bounded FIFO buffer is shown in Fig. 4.4 where task u_1 and u_2 are represented by actor v_1 and v_2. The bounded FIFO buffer is represented by the forward and backward edge and the FIFO buffer capacity of three containers is modelled by the three initial tokens located on the backward edge (v_2, v_1).

The communication between two tasks is either intra-tile or inter-tile communication, as described in Section 3.3. Intra-tile communication is the communication between tasks that are executed on the same processor. This communication is implemented via a circular buffer in the local memory of the processor. The circular buffer has a bounded capacity, therefore, it is modelled with a forward edge, backward edge, and a number of initial tokens to represent the buffer capacity. More advanced intra-tile communication implementations can be expressed in a CSDF graph by relaxing the tight relation between tokens and containers [19], but this is beyond the scope of this thesis.

Inter-tile communication is the communication between tasks that are executed on processors of different tiles. This communication is implemented with address-less communication over a network connection from the network interface in one tile to the network interface in another tile. Inter-tile communication can be modelled with one or more actors in the dataflow model. In [54, 59, 40] a detailed dataflow model of an Æthereal network connection is introduced, which models the routers, network interfaces, network-interface buffers, time-division-multiplex scheduling, and flow control. In this thesis, we model the communication latency with one actor for simplicity reasons. For example, actor v_3 in Fig. 4.5 models the communication latency, as defined by Definition 9. This is a suitable model in case the network-connection bandwidth allocation is sufficiently large, so that the jobs throughput is only affected by the communication latency. The execution time of actor v_3 is equal to this communication latency. Task u_1 and u_2 are represented by actor v_1 and v_2. The two network-interface buffers are modelled with the forward and backward edges between actor v_1 and v_3, and between actor v_3 and v_2. The buffer capacities are modelled with the number of initial tokens that are placed on the edges (v_3, v_1) and (v_2, v_3).

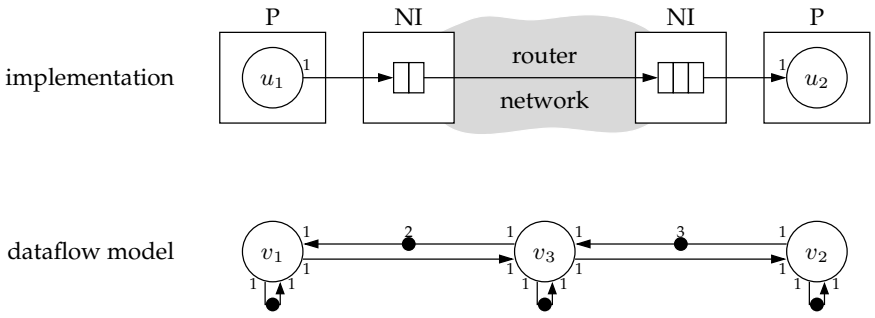

Figure 4.5: Dataflow model representation of inter-tile communication implemented with address-less communication over a network connection.

Multiple tasks can be executed on the same processor. Therefore, a scheduler determines which task is executed next by means of a predefined scheduling mechanism. Such a scheduling mechanism can be divided into compile-time scheduling (e.g. static-order scheduling) and run-time scheduling (e.g. round robin and time division multiplex), as described in Section 3.3. By modelling these schedule mechanism in the dataflow model, the throughput and end-to-end latency can be found by analysing the self-timed executed dataflow graph. In part I of this thesis, tasks are executed using a static-order scheduling mechanism. Actors are enabled to fire when sufficient tokens are available on every input edge. Therefore, the static-order execution of tasks can be modelled with additional edges between these tasks. These edges form a cycle and we place one initial token on this cycle, so that the actors execute sequentially. The initial token is placed at the input of the actor that represents the first task in the static-order schedule to make sure that this actor can start to fire. An algorithm to model generic static-order schedules into a CSDF graph, will be introduced in Section 4.3.2. A dataflow model of a run-time scheduler that belongs to the latency-rate server class [77], will be described in part II of this thesis.

4.3.1 Absence of the firing rule in the implementation

In this section, we will show that if tasks are executed in a static-order schedule, they do not need to implement the firing rule in the implementation. Therefore, the task's C-code can be modelled without modification, which reduces the modelling effort.

A CSDF actor is enabled to fire when the firing rule is evaluated as true, i.e. the number of tokens that will be consumed is available on each input edge. The minimum throughput can still be computed from a CSDF model, despite the absence of the firing rule in the implementation. Intuitively, tasks can already start consuming containers before the corresponding actor is enabled (firing rule is not present in a task) and earlier consumption of containers cannot result in later corresponding token production times (temporal monotonic). Therefore, the CSDF model is a conservative representation of the implementation. This will be described in detail below.

We use a generic example to show that the CSDF model is a conservative represen-

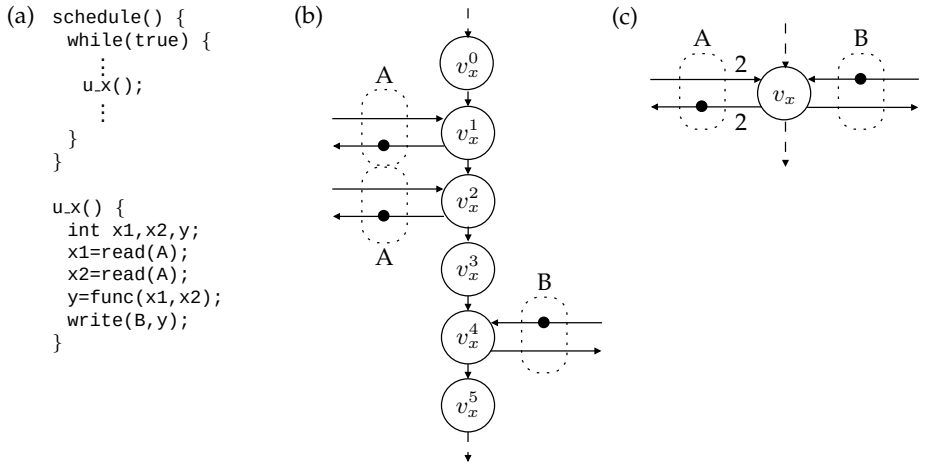

Figure 4.6: (a) Pseudo code of static-order schedule and task u_x, (b) intermediate CSDF model of a specific trace from task u_x, and (c) actor v_x that is modelling task u_x.

tation of the implementation. A task u_x is executed on a processor p in a static-order schedule S_p. The pseudo codes of the static-order schedule and task u_x are shown in Fig. 4.6a. Task u_x reads two words of input data from FIFO buffer A, computes the output data, and writes the output data to FIFO buffer B. During the statements `x1=read(A);` and `x2=read(A);` the processor is stalled if no data is available in FIFO buffer A and it continues again if there is data. During the statement `write(B,y);` the processor is stalled if FIFO buffer B is full and it continues again if space is available.

Task u_x will be modelled with actor v_x that consists of a set of edges E_b modelling the FIFO buffers A and B, and of the set of edges E_s modelling the scheduling dependencies. FIFO buffers A and B are modelled with two edges in opposite direction, as depicted in Fig. 4.6c. The initial tokens represent the FIFO buffer capacities. Actor v_x is enabled if: two tokens are available at the input edge that represents the communication of full containers in FIFO buffer A, one token is available at the input edge that represents the communication of empty containers in FIFO buffer B, and one token is available on the dashed input edge that models the static-order schedule. When actor v_x finishes its execution, two tokens are produced on the output edge that represents the communication of empty containers in FIFO buffer A, one token is produced at the output edge that represents the communication of full containers in FIFO buffer B, and one token is produced on the dashed output edge that models the static-order schedule.

To show that actor v_x is a conservative representation of task u_x, we make use of an intermediate CSDF model, which is shown in Fig. 4.6b. The intermediate model has a one-to-one relation with a specific trace of task u_x. A trace is composed of read statements, write statements, and non-blocking code segments. A non-blocking code segment is a code segment that, after it is started, can always finish without having to wait for additional input data or output space. That means, in our case, a non-blocking code segment is a code segment between read and write statements. A trace

of task u_x can be seen as a row of CSDF actors v_x^i with $0 \leq i < w$, in such a way that every non-blocking code segment, read statement and write statement is represented as a separate actor. For example, in Fig. 4.6b we see a CSDF model of the trace from task u_x with $w = 6$. The read and write statements are represented by the actors v_x^1, v_x^2 and v_x^4, and the non-blocking code segments are represented by actors v_x^0, v_x^3, and v_x^5. The dependencies between the read statements, write statements, and non-blocking code statements are modelled with the vertical dashed edges in Fig. 4.6b, i.e. modelling the order within a specific trace. The intermediate model has a one-to-one relation with the specific trace of task u_x, because in the implementation the processor stalls during read and write statements until data or space is available, whereas in the model the actors v_x^1, v_x^2 and v_x^4 are enabled to fire when sufficient tokens are available at every input edge.

We will show that actors v_x^0 through v_x^5 in Fig. 4.6b can be modelled by actor v_x in Fig. 4.6c, so that the production behaviour of actor v_x is conservative with respect to the production behaviour of actors v_x^0 through v_x^5. To show this, we make use of the following terminology. For actor v_x in Fig. 4.6c, we define I_x and O_x to be respectively the set of input and output edges representing the FIFO buffers in the implementation, with $I_x, O_x \subset E_b$. We further define $a_c(m, j_m)$ to be the arrival time at the input $m \in I_x$ of the j_m-th token and $a_c(n, j_n)$ to be the arrival time on the output $n \in O_x$ of the j_n-th token, both of actor v_x. The vertical dashed edges in Fig. 4.6c are part of the set of edges E_s, representing the static-order schedule S_p. We define $a_c(p_x, j)$ to be the arrival time of the j-th token on this dashed input edge and we define $a_c(q_x, j)$ to be the arrival time of the j-th token on this dashed output edge, with $p_x, q_x \in E_s$. For actor v_x^i, we define I_x^i and O_x^i to be respectively the set of input and output edges representing the FIFO buffers in the implementation, with $I_x^i \subset I_x$ and $O_x^i \subset O_x$. We define $a_b(m, j_m)$ to be the arrival time at the input $m \in I_x^i$ of the j_m-th token and $a_b(n, j_n)$ to be the arrival time on the output $n \in O_x^i$ of the j_n-th token, both of actor v_x^i. The vertical dashed edges in Fig. 4.6b model the execution order of the read statements, write statements, and non-blocking code segments. The time when the j-th token arrives at this vertical dashed input edge of actor v_x^i is defined as $a_b(p_x^i, j)$ and the time when the j-th token arrives at this vertical dashed output edge is defined as $a_b(q_x^i, j)$.

In general, the production behaviour of actor v_x is conservative with respect to the production behaviour of actors v_x^0 through v_x^{w-1} if the following holds.

Theorem 1. *If the worst-case arrival times of tokens at the input are conservative (i.e. condition* (4.1) *holds) and the execution time of actor v_x is at least the sum of every actor that models the trace of task u_x (i.e. condition* (4.2) *holds), then the worst-case arrival times of tokens at the output are conservative (i.e. Eq.* (4.3) *holds).*

$$\begin{aligned} \forall m \in I_x, a_b(m, j_m) \leq a_c(m, j_m) \wedge \\ a_b(p_x^0, j) \leq a_c(p_x, j) \end{aligned} \tag{4.1}$$

$$\sum_{i=0}^{w-1} \big(\rho(v_x^i, f)\big) \leq \rho(v_x, f) \tag{4.2}$$

$$\begin{aligned} \forall n \in O_x, a_b(n, j_n) \leq a_c(n, j_n) \wedge \\ a_b(q_x^{w-1}, j) \leq a_c(q_x, j) \end{aligned} \tag{4.3}$$

Intuitively, actors v_x^i can consume and produce the tokens only earlier than actor v_x, because an actor v_x^i is enabled if there are sufficient number of tokens available on a subset of all inputs of actor v_x (i.e. $I_x^i \subset I_x$) and the execution time of actor v_x is conservative (i.e. Eq. (4.2) holds).

Proof of Theorem 1. Let y_c be the maximum arrival time of the tokens at all inputs $m \in I_x$ of actor v_x that enable actor v_x (as defined by Eq. (4.4)), then the arrival time of the j_n-th token of all outputs $n \in O_x \cap \{q_x\}$ of actor v_x is given by Eq. (4.5).

$$y_c = \max_{m \in I_x} \left(a_c(m, j_m)\right) \tag{4.4}$$

$$a_c(n, j_n) = \max\left(y_c, a_c(p_x, j)\right) + \rho(v_x, f) \tag{4.5}$$

Let y_b^i be the maximum arrival time of the tokens at all inputs $m \in I_x^i$ of actor v_x^i that enable actor v_x^i (as defined by Eq. (4.6)), then the arrival time of the j_n-th token of all outputs $n \in O_x^i \cap \{q_x^i\}$ of actor v_x^i is given by (4.7).

$$y_b^i = \max_{m \in I_x^i} \left(a_b(m, j_m)\right) \tag{4.6}$$

$$a_b(n, j_n) = \max\left(y_b^i, a_b(p_x^i, j)\right) + \rho(v_x^i, f) \tag{4.7}$$

In Fig. 4.6b, the ordering between code segments is modelled with the vertical dashed edges between actors v_x^{i-1} and v_x^i, therefore, we know the following.

$$a_b(p_x^i, j) = a_b(q_x^{i-1}, j) \text{ , for } 1 \leq i < w \tag{4.8}$$

After substituting Eq. (4.8) in Eq. (4.7) for output $n = q_x^i$ we get the following recurrence relation with $1 \leq i < w$.

$$\begin{aligned} a_b(q_x^0, j) &= \max\left(y^0, a_b(p_x^0, j)\right) + \rho(v_x^0, f) \\ a_b(q_x^i, j) &= \max\left(y^i, a_b(q_x^{i-1}, j)\right) + \rho(v_x^i, f) \end{aligned} \tag{4.9}$$

By applying the relation $\max(r, s + t) \leq \max(r, s) + t$ iteratively on Eq. (4.9) we can rewrite it into.

$$y_b = \max_{i=0}^{w-1} \left(y_b^i\right) \tag{4.10}$$

$$a_b(q_x^{w-1}, j) \leq \max\left(y_b, a_b(p_x^0, j)\right) + \Sigma_{i=0}^{w-1} \rho(v_x^i, f) \tag{4.11}$$

The right-hand side of Eq. (4.11) is smaller than $a_c(q_x, j)$ from Eq. (4.5) if Eq. (4.1) holds (i.e. $y_b \leq y_c$ and $a_b(p_x^0, j) \leq a_b(p_x, j)$) and Eq. (4.2) holds. Furthermore, actor v_x^{w-1} is the last actor in the trace, so we know that all outputs $n \in O_x$ have an arrival time $a_b(n, j_n) \leq a_b(q_x^{w-1}, j)$. Therefore, we conclude that the production behaviour of actor v_x is conservative with respect to the production behaviour of actors v_x^0 through v_x^{w-1} for all outputs $n \in O_x \cap \{q_x\}$. We further conclude that actor v_x is conservative for all possible traces of task u_x and it is independent of the order in which data is consumed and produced, when $\rho(v_x, f)$ is the worst-case execution time of task u_x for every phase f. □

Each FIFO buffer in the implementation results in two edges in opposite direction in the CSDF graph, therefore, our model is a strongly connected CSDF graph. In Section 4.2, we have shown that the self-timed execution of a strongly connected CSDF

graph that maintains FIFO ordering, is monotonic in time. Since we know that the self-timed execution of the CSDF graph has a monotonic temporal behaviour and every actor has a conservative temporal behaviour compared with the corresponding task, we arrive at the following conclusion. If during static-order execution, the first execution of the first task in the schedule is not enabled later than the corresponding actor, then producing containers before the task finishes or a shorter execution time of a task cannot result in a later production of containers than the production of the corresponding tokens.

4.3.2 Modelling static-order schedules

In a static-order schedule the order in which tasks execute is fixed at design time. In [75], it is shown how a static-order scheduling of an SRDF graph can be modelled into another SRDF graph such that, in the self-timed execution, actors are fired in the order specified by the static-order schedule. This is done by adding edges to the original SRDF graph. These edges form a cycle that enforces the ordering in which the actor firings should occur. It is shown in [82], that in general a static-order schedule cannot be modelled in a MRDF graph without adding additional actors. This is, for example, possible by first converting the MRDF graph into an equivalent SRDF graph and then applying the method from [75]. The disadvantage is that this can lead to an exponential increase in the number of actors in the graph.

In this section we take a different approach. We model static-order schedules in a CSDF graph instead of a SRDF graph, therefore, we prevent that the number of actors increase exponentially. Instead of increasing the number of actors, the number of phases are increased. Furthermore, additional edges are added between the actors to enforce the ordering in which the actor firings should occur, according to the static-order schedule. One initial token is placed on the cycle that represents the static-order schedule. The initial token is placed before the actor that represents the first task in a static-order schedule. After modelling the static-order execution of tasks, existing dataflow-analysis techniques can be applied to check whether the static-order schedule is deadlock free and to compute the throughput. Note, that we use a static-order schedule for jobs where the tasks consume and produce a predefined number of containers from the input and output channels, i.e. the jobs can be represented in CSDF graphs. For these jobs we can derive, at design time, static-order schedules that are deadlock free.

In this section we use the following terminology. Each task is executed on a processor p and each processor p is executing tasks in a static-order schedule. A static-order schedule can be denoted by $S_p = (s_0, s_1, ..., s_{N-1})$, with task s_i executed on processor p. Furthermore, in this chapter, we assume that every task u_i is represented by actor v_i.

Now we introduce a method to extend a CSDF graph that represents a job, into a CSDF graph that represents a job from which the tasks are executed on processors using static-order schedules. The input of our algorithm is: a CSDF graph $G = (V, E)$ representing a job, and a specified mapping that consists of a binding of actors to processors and a static-order schedule S_p for each processor p. The output is a CSDF graph $G' = (V', E')$ that models the job with the specified mapping of tasks.

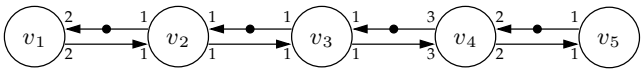

Figure 4.7: CSDF graph G representing a streaming job.

The algorithm is illustrated with an example of a streaming job that consists of five tasks. Tasks u_1 and u_2 are bound to processor $p1$ and task u_3 through u_5 are bound to processor $p2$. Depending on the processor and its clock frequency, each task will have a certain conservatively-estimated upper bound on its execution time. In our example, these upper bounds equal T time units and every task has only one phase. Task switching cost (e.g. due to cache misses) is assumed to be C time units for every task switch. The task switching cost will be taken into account in part II of this thesis. Furthermore, processor $p1$ and $p2$ use the static-order schedules $S_{p1} = (u_1, u_2, u_2)$ and $S_{p2} = (u_3, u_3, u_4, u_5, u_3, u_5)$, respectively. The streaming job is represented by the CSDF model that is depicted in Fig. 4.7. Each actor v_i represents task u_i. The execution time $\rho(v_i, f) = T$ time units for every actor v_i with $1 \leq i \leq 5$ and phase $f = 1$, because $\theta(v_i) = 1$.

For our algorithm we use the following terminology. The number of occurrences of task u_i in schedule S_p equals $\Omega(u_i, S_p)$, with $0 \leq \Omega(u_i, S_p) \leq N$. For example, the number of occurrences of task u_1 and u_2 in schedule S_{p1} equal $\Omega(u_1, S_{p1}) = 1$ and $\Omega(u_2, S_{p1}) = 2$, respectively. Furthermore, for a certain schedule S_p, the l'th occurrence of task u_i is at position $\phi(l, u_i, S_p)$, with $1 \leq l \leq \Omega(u_i, S_p)$ and $0 \leq \phi(l, u_i, S_p) < N$. For example, the third occurrence of task u_3 in schedule S_{p2} is at position $\phi(3, u_3, S_{p2}) = 4$.

The new graph G' is constructed by (i) creating the new set of actors V' and (ii) creating the new set of edges E'.

(i) The new set of actors V' consists of an equal number of actors as in set V. Each actor $v'_i \in V'$ of graph G' is representing actor $v_i \in V$ of the original graph G. A task u_i is modelled with actor v_i and the cyclo-static behaviour of the task is represented by the $\theta(v_i)$ distinct phases of actor v_i. Actor v_i is translated into actor v'_i that also models the cyclo-static behaviour of the static-order schedule. Task u_i occurs $\Omega(u_i, S_p)$ times in schedule S_p. The different positions of task u_i in schedule S_p, are modelled with different phases of Actor v'_i. Therefore, the number of phases $\theta(v'_i)$ of actor v'_i is equal to the least common multiple (lcm) of the number of occurrences of task u_i in schedule S_p and the number of phases of actor v_i, i.e. $\theta(v'_i) = \text{lcm}(\Omega(u_i, S_p), \theta(v_i))$. The modelling of the different positions in the schedule has also the advantage that task switching cost can be modelled. With the number of phases $\theta(v'_i)$ we can express the cyclo-static behaviour of the static-order schedule as well as the cyclo-static behaviour of the task. For phase f' of actor v'_i, its firing is at position q in the static-order schedule S_p, where the position q is can be compute as follows:

$$q = \phi(((f' - 1)\%\Omega(u_i, S_p)) + 1, u_i, S_p) \tag{4.12}$$

Equation (4.12) is further described in Appendix A. The execution time of actor v'_i can be calculated from the execution time of actor v_i and the actor switching overhead cost C_i. The execution time of actor v'_i in phase f' is computed by Eq. (4.13) with

actor	$\theta(v'_i)$	$\rho(v'_i)$
v'_1	1	$\langle T+C \rangle$
v'_2	2	$\langle T+C, T \rangle$
v'_3	3	$\langle T+C, T, T+C \rangle$
v'_4	1	$\langle T+C \rangle$
v'_5	2	$\langle T+C, T+C \rangle$

Table 4.1: The number of phases $\theta(v'_i)$ and the worst-case execution times $\rho(v'_i)$ for actor v'_1 through v'_5.

$1 \leq f' \leq \theta(v'_i)$. We only have to account for the task-switching overhead cost if the previous task in a schedule $S_p = (s_0, s_1, ..., s_{N-1})$ is different from the current task, i.e. only if there is a task switch. There is a task switch when $s_j = u_i \wedge s_{(j-1)\%N} \neq u_i$. Per definition $s_q = u_i$. Therefore, the execution time of actor v'_i in phase f' is computed as follows:

$$\rho(v'_i, f') = \begin{cases} \rho(v_i, ((f'-1)\%\theta(v_i))+1) & \text{if } s_{(q-1)\%N} = u_i \\ \rho(v_i, ((f'-1)\%\theta(v_i))+1) + C_i & \text{if } s_{(q-1)\%N} \neq u_i \end{cases} \tag{4.13}$$

Table 4.1 gives an overview of the number of phases and the execution times of the actors v'_1 through v'_5 in our example.

(ii) The new set of edges E' consists of the set of edges E'_b modelling the FIFO buffers (with forward and backward edges) and a set of edges E'_s modelling the scheduling dependencies, i.e. $E' = E'_b \cup E'_s$. The set of edges E'_b of graph G' consists of an equal number of edges as in set E of the original graph G. Each edge $e'_b \in E'_b$ is representing edge $e \in E$. The number of tokens consumed ($\gamma(e'_b, f')$) and produced ($\pi(e'_b, f')$) by actor v'_i on edge $e'_b \in E'_b$ equals, respectively, Eq. (4.14) and Eq. (4.15) for every phase f', with $1 \leq f' \leq \theta(v'_i)$.

$$\gamma(e'_b, f') = \gamma(e, ((f'-1)\%\theta(v_i))+1) \tag{4.14}$$
$$\pi(e'_b, f') = \pi(e, ((f'-1)\%\theta(v_i))+1) \tag{4.15}$$

The static-order schedule of tasks is modelled with the set of edges E'_s. The static-order schedule $S_p = (s_0, s_1, ..., s_{N-1})$ represents the execution order of tasks, which are represented by actors. In the CSDF graph G', there is an edge $e \in E'_s$ from the actor that represents task s_i to the actor that represents task $s_{(i+1)\%N}$, where task $s_i \neq s_{(i+1)\%N}$, for $0 \leq i < N$. For our example, every edge $e'_s \in E'_s$ is depicted with a dashed arrow in Fig. 4.8. One initial token is added on every input edge $e'_s \in E'_s$ of the actor that represents task s_0 of every schedule S_p to take care that these actors can start to fire. For our example in Fig. 4.8, there are initial tokens added on the dashed edges (v'_2, v'_1) and (v'_5, v'_3), so that actor v'_1 and v'_3 can start to fire. The number of tokens that are consumed by actor v'_i in phase f' on edge $e'_s = (v'_j, v'_i) \in E'_s$, is computed by Eq. (4.16). If task u'_j is executed before task u'_i in schedule S_p, then actor v'_i consumes one token from the edge $e'_s = (v'_j, v'_i)$, else no tokens are

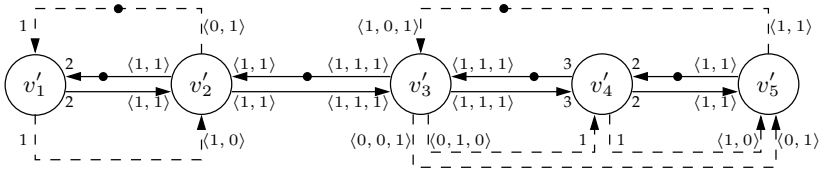

Figure 4.8: CSDF graph G' modelling the job in Fig. 4.7 with static-order schedules $S_{p1} = (v_1, v_2, v_2)$ and $S_{p2} = (v_3, v_3, v_4, v_5, v_3, v_5)$.

consumed.

$$\gamma(e'_s, f') = \begin{cases} 0 & \text{if } s_{(q-1)\%N} \neq u_j \\ 1 & \text{if } s_{(q-1)\%N} = u_j \end{cases} \tag{4.16}$$

The number of tokens that is produced by actor v'_i in phase f on edge $e'_s = (v'_i, v'_j) \in E'_s$, is computed by Eq. (4.17). If task u'_j is executed after task u'_i in schedule S_p, then actor v'_i produces one token on the edge $e'_s = (v'_i, v'_j)$, or else no tokens are produced.

$$\pi(e'_s, f) = \begin{cases} 0 & \text{if } s_{(q+1)\%N} \neq u_j \\ 1 & \text{if } s_{(q+1)\%N} = u_j \end{cases} \tag{4.17}$$

For our example in Fig. 4.8, the number of consumed and produced tokens are depicted at the head and tail of each edge.

4.4 Dataflow analysis techniques

A streaming job is executed on a platform and this implementation is modelled in a dataflow graph. By analysing this graph with dataflow-analysis techniques, we derive the job's real-time behaviour. In this section, we describe such techniques that are able to derive throughput, end-to-end latency, and buffer capacities from a dataflow graph.

Many existing dataflow-analysis techniques are based on maximum-cycle-mean analysis. This analysis is done typically on a SRDF graph where every actor consumes and produces per firing one token from every input and output edge, respectively. The maximum cycle mean is related to the inverse of the job's maximum achievable throughput, i.e. it is related to the number of produced and consumed containers per time interval for a source or sink task. This maximum achievable throughput is reached during a self-timed execution of the dataflow graph. The Maximum Cycle Mean MCM(G) for a SRDF graph G is defined by [76] as:

Definition 12.

$$\text{MCM}(G) = \max_{\text{simple cycle } o \text{ in } G} \left(\frac{\sum_{v \text{ is on } o} \rho(v)}{\sum_{e \text{ is on } o} \zeta(e)} \right) \tag{4.18}$$

where a simple cycle is defined as a cycle with no repeated actors (aside from the start/end actor). CSDF graphs can be transformed into equivalent SRDF graphs [9], from which the maximum cycle mean can be computed with Eq. (4.18). The number of phases of each actor in combination with the number of tokens consumed and produced in each phase, determines the number of actors in the equivalent SRDF graph. Therefore, a CSDF graph with a few actors can result in a large equivalent SRDF graph with many actors and edges, resulting in a much larger number of cycles than in the original CSDF graph. So, computing the maximum cycle mean can be computation intensive for such an equivalent SRDF graph. The end-to-end latency can also be computed from the maximum cycle mean, after translating the end-to-end latency requirement into a throughput requirement [61].

In [28] an alternative technique is proposed for deriving the maximum achievable throughput of MRDF graphs. This technique is based on explicit state-space exploration and, unlike the previous techniques, it can work directly on an MRDF graph, avoiding a possible explosion of the equivalent SRDF graph. It is shown by [28] that for every consistent and strongly-connected self-timed executed MRDF graph, the state-space consists of a transient phase, followed by a periodic phase. The maximum cycle mean and maximum achievable throughput can be derived by examining the periodic phase. The derivation of an upper-bound on the end-to-end latency, can also be derived by examining the periodic phase [29]. This technique is also applied on CSDF graphs [81]. The disadvantage is that it can take a long time before the periodic phase is detected, due to a potentially long transient phase. Long transient phases can occur if a consuming actor is slightly slower than a producing actor and there is a backward edge (from the consuming actor to the producing actor) with a large number of initial tokens. Although the transient state can be long, this approach is in practice typically faster than first transforming a MRDF graph into a SRDF graph [28, 81]

FIFO buffer capacities are taken into account in our dataflow models. The maximum cycle mean can only be computed after specifying all buffer capacities in the dataflow model. Backtracking or heuristics can be used in defining the initial buffer capacities, before computing the maximum cycle mean. An exact technique for determining all trade-offs (Pareto points) between the throughput and buffer capacities, is described in [79, 81]. This technique is based on iteratively computing the maximum cycle mean via explicit state-space exploration and it is implemented in the tool SDF^3 [80]. In case the runtime of this tool becomes problematic, an approximation technique can be used. Such an approximation technique with a low computational complexity is described in [91] and it is included in a tool called Hebe. The approximation technique relies on the fact that container arrival times can be bounded from above, because the corresponding CSDF graph has a monotonic temporal behaviour [91]. This means that we can construct a conservative schedule that satisfies the temporal constraints to derive buffer capacities. The production times of this schedule are conservative compared to the container production times when executing the CSDF graph. This means that buffer capacities, derived with the conservative schedule, are sufficiently large. Therefore, this approximation technique trades off runtime for accuracy.

4.5 Concluding remarks

At design time we need to guarantee that each job will meet its real-time requirements like throughput and end-to-end latency. This thesis uses dataflow-analysis techniques, because they allow cyclic data dependencies that influence the job's temporal behaviour. This chapter described existing dataflow-analysis techniques that compute throughput, end-to-end latency, and FIFO buffer capacities from these dataflow models. Furthermore, we have shown how to construct a CSDF model from a job that is mapped onto a predictable multiprocessor platform. After each mapping step, additional constraints are added to the CSDF model. The final CSDF model takes into account: computation of tasks, communication between these tasks, FIFO buffer capacities, and static-order scheduling of tasks. We introduced an algorithm for generating a CSDF model of tasks that are executed on a processor in a static-order schedule. Furthermore, it is shown that these tasks, which are executed in a static order, can be represented with one actor despite the absence of the firing rule in the task's implementation.

Chapter 5

Case study: comparison of Æthereal network and interconnects in SAF7780

In the automotive domain the platform area and system performance are important. Therefore, when introducing a network-on-chip, the question becomes: *What is the impact on area and performance?* This isn't easy to quantify. In [52] a general (artificial) design example is used. In this thesis, we start from a real-life application to compute the Æthereal-network cost for various network configurations and to compare the network cost with the interconnects in SAF7780 (car-infotainment platform generation three). An earlier version of this work was published in [60, 55].

First, in Section 5.1, we describe our reference design that is used for infotainment-nucleus generation three. Our design flow that is used for dimensioning and generating the Æthereal network-on-chip, which will be described in Section 5.2. In Section 5.3, the design flow is used to generate multiple network instances for comparing network costs in terms of area and latency. Finally, we summarise this chapter with concluding remarks in Section 5.4.

5.1 Car-infotainment generation three

For our case study, we make use of the communication requirements from infotainment-nucleus generation three, which is described in Section 2.4.1. First, we describe the reference design SAF7780 in Section 5.1.1 and subsequently we elaborate on the communication requirements in Section 5.1.2.

5.1.1 Reference design

Our reference design is the platform SAF7780 [88, 8], which is already introduced in Section 1.3.1. This design contains four DSP tiles (tile 0 till 3), one Finite-Impulse-

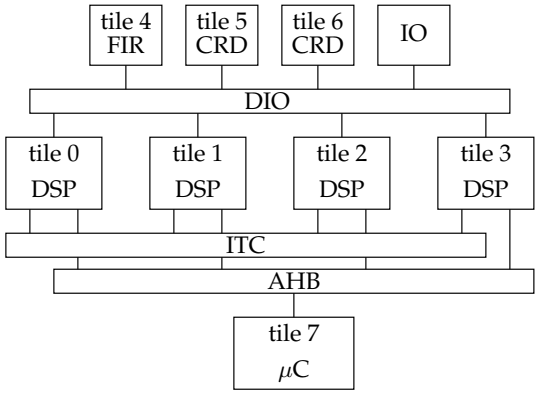

Figure 5.1: The communication infrastructure of our reference design.

Response (FIR) filter (tile 4), two Coordinate-Rotation-Digital (CRD) computing hardware modules (tile 5 and 6), one micro-controller (μC) subsystem (tile 7), and a number of input and output peripherals, as depicted in Fig. 5.1. In this figure, the peripherals are represented by the input/output (IO) box to keep the figure simple.

The communication infrastructures between these tiles are also shown in Fig. 5.1. The micro-controller subsystem makes use of a multi-layer bus (AHB) and this bus is also used to communicate between the micro controller and DSP tiles. The DSP tiles communicate via a fully connected crossbar switch, which we refer to as the Inter-Time Communication (ITC) interconnect. Furthermore, the DSP tiles can read from and write to the registers of the input/output peripherals and hardware accelerators, by making use of a crossbar switch, which we refer to as the Digital Input/Output (DIO) interconnect.

5.1.2 Communication requirements

To dimension our network, we make use of the communication requirements from infotainment-nucleus generation three, which is described in Section 2.4.1. The mapping of tasks to tiles is done similarly as in SAF7780, to get a fair comparison. The communication requirements as a number of inter-tile communication channels are shown in Fig. 5.2. Communication channels 1 through 18 have a peripheral as a source or sink task. In Fig. 5.2, these peripherals are represented by the input/output (IO) box to keep the figure simple. Each peripheral and each processing tile is connected to network-interface ports, as described by the multiprocessor architecture template in Section 3.2. Furthermore, each network interface can contain multiple network-interface ports, as described by the network architecture in Section 3.2.1.

The inter-tile communication channels are divided into two classes:

- *Data channels (1-29)*: streaming communication channels that are represented by the edges in the task graphs.
- *Programming channels (30-33)*: connections used only at application startup to

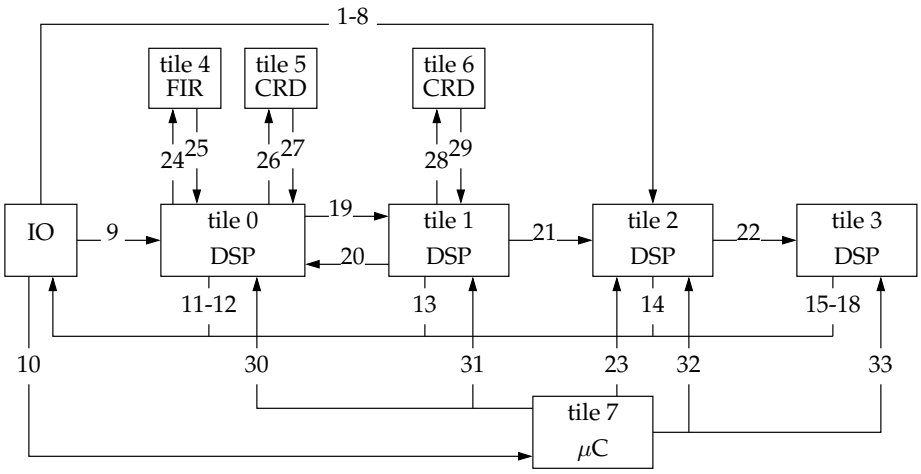

Figure 5.2: Application requirements after mapping the audio application.

load the program memories and control registers of tiles and hardware IP modules.

For dimensioning the network we need: (i) communication-bandwidth requirements, and (ii) communication-latency requirements for both classes of communication.

(i) *Communication-bandwidth requirements*: for data channels, these requirements are derived from a container throughput (which is related to the sample frequency of a periodic source or sink) and the container sizes. Audio streams have typically low communication-bandwidth requirements and most of the communication bursts (container sizes) are small. The sample frequency is 8 kHz for speech (e.g. telephone and navigation), between 40 and 48 kHz for audio, and 325 kHz for a terrestrial analog radio signal. Container sizes vary from one word for a mono sample, two words for a stereo sample, up to 512 words for the input frame of the MP3 decoder task. The average communication-bandwidth requirements for communication channels are between 40 $^{\text{KByte}}/_{\text{sec}}$, for compressed audio, and 2 $^{\text{MByte}}/_{\text{sec}}$, for terrestrial analog radio. The required average communication bandwidth is typically higher between a processor and a hardware accelerator. For example, for an analog-radio demodulation job, the average communication-bandwidth requirement is approximately 44 $^{\text{MByte}}/_{\text{sec}}$ between the DSP processor and CRD hardware accelerator. Such a communication-bandwidth requirement can be accommodated by the network. The amount of bandwidth assigned to programming channels affects only the startup time of a job, which is not time critical. Therefore, these communication-bandwidth requirements are relaxed.

(ii) *Communication-latency requirements*: These requirements are caused by both: total time containers take to ripple through the task graph, and the job's throughput requirements if a task graph contains loops due to feedback or control. The time it takes before data is rippled through the task graph is referred to as end-to-end latency. For most streaming jobs, end-to-end latency is not critical, as described in Section 2.2. The communication-latency requirements can also be caused by a job's

throughput requirement, in case its task graph contains feedback loops. For example, if such a feedback loop contains inter-tile communication while it determines the lower bound on a job's throughput, then an increase in communication latency will increase the loop time and decreases this lower bound on the job's throughput. Therefore, such feedback loops limit the possibility of pipelining and algorithmic transformations are needed to increase the performance of these jobs. The analog terrestrial-radio demodulation and channel-equalisation jobs, for example, contain adaptive filters with such feedback loops. In case of channel equalisation, new filter coefficients are calculated and updated every sample, as we will see in the next chapter. The filter coefficients are computed on the DSP processor in cooperation with the CRD hardware accelerator. Therefore, the round-trip latency between the DSP and CRD can limit the lower bound on the job's throughput. The round-trip latency is composed of communication latency and computation latency. The SAF7780 is implemented in 0.18 μm technology. The DSP processor and CRD hardware accelerator share the same clock, running at a clock frequency of 125 MHz. The DSP processor has one-cycle access to the input and output registers of the CRD hardware accelerator. Therefore, the round-trip latency is determined by the computation latency of the task executed on the CRD. The computation latency of the CRD is 36 clock cycles at a clock frequency of 125 MHz. That means, the round-trip latency is equal to $^{36}/_{125 \cdot 10^6}$ which is 288 ns. The algorithms of the analog terrestrial-radio jobs are potentially upwards compatible if the round-trip latency is not increased after replacing the traditional interconnects with a network.

5.2 Design flow and tools

In this thesis, we use a design flow with a number of tools for generating our multiprocessor platform. In our design flow, most of our tools communicate using XML formats. The design flow is composed of the following steps: (i) deriving inter-tile communication requirements, (ii) generating a network instance that meets the communication requirements, and (iii) computing sufficiently large network-interface buffer capacities.

(i) Inter-tile communication requirements are derived manually in our case study. Each job consists of a number of tasks and is represented as a task graph. Accesses to shared variables (communication) are made explicit in the task graph. Characteristics, like number of consumed and produced containers per task execution, and upper bounds on container sizes are derived from the application domain and from static-program analysis. In general, it is not always possible to obtain these characteristics, but in our case tasks have bounded loop iterations. Bounds on communication requirements are derived from the job's throughput requirement and the task graph for each job. Tasks are mapped to tiles, so that we can derive bounds on the inter-tile communication requirements (minimum throughput and maximum latency). The communication requirements are specified per use case, because multiple jobs can be active per use case. The communication requirements that are used in our case-study, are already described in the previous section. These requirements are stored in the *communication.xml* file, as depicted in Fig. 5.3.

(ii) The network is generated with the automated tool chain of Æthereal [32]. To

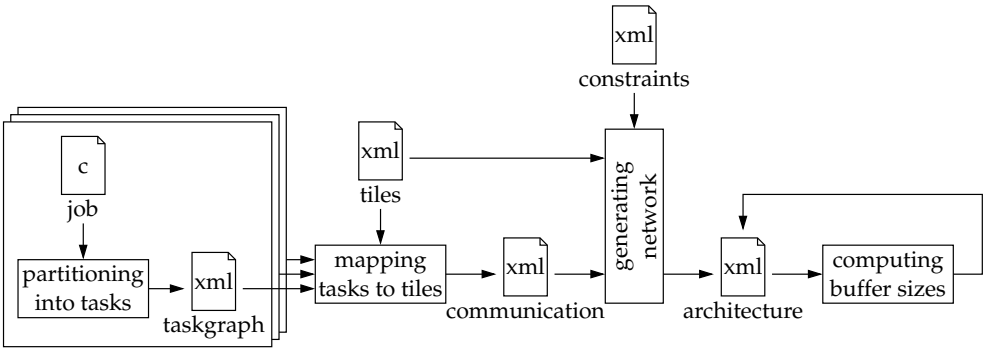

Figure 5.3: Design flow for generating a multiprocessor platform.

generate a network instance, we feed the network generation tool [38]: constraints of the network as a whole (*constraints.xml*), specification of the tiles (*tiles.xml*), and specification of the communication requirements of network connections (*communication.xml*). The file *constraints.xml* contains, for example, topology template (mesh, in our case), clock frequency of the network (500 MHz, in our case), and link width (32 bits, in our case). The tiles are specified in the file *tiles.xml*. The file *communication.xml* specifies the minimum communication bandwidth, maximum communication latency, and maximum burst size for each network connection. With these three XML files as an input, the network generation tool is able to automatically map tiles to network interfaces, compute routing paths, and compute a slot allocation for the slot tables in each network interface [38]. The network architecture is written in the file *architecture.xml*.

(iii) Sufficiently large network-interface buffer capacities are computed for all guaranteed-throughput connections. The best-effort connections have network-interface buffers with a fixed size, which is specified upfront. A guaranteed-throughput connection is represented in a dataflow model of a network connection [59]. From such a dataflow model, we can derive sufficient large network-interface buffers for a given throughput constraint [39]. For the case study in this chapter, we make use of the dataflow-analysis technique [91] for computing network-interface buffer capacities of guaranteed-throughput connections. Section 4.4 described this dataflow-analysis technique.

5.2.1 Estimating the network area

In this chapter, we compare the cell area between different network instances and between a network instance and a traditional interconnect. Therefore, we have to estimate the network area for the generated network instances. The network area is composed of: (i) network-interfaces area and (ii) routers area.

(i) *Network-interfaces area*: depends on the number of network interfaces, the number of connections and the total buffer capacity. As mentioned before, the buffers decouple the tile communication behaviour from the network behaviour. A larger

transaction burst size means more bursty traffic, hence a larger buffer is required to decouple the tile and the network. The buffers must also be sufficiently large to enable the hiding of the round-trip latency of end-to-end flow-control credits. Therefore, the size of network-interface buffers depends on the connection's transaction burst size and round-trip latency, which in its turn depends on the network topology, the binding of tiles to network-interface ports, the routing, the number of slots in the slot table, and the slot allocation. These parameters are mutually dependent. The slot-table size and slot allocation are determined by the usage of the network links. Time-division-multiplex scheduling serves two purposes. First, to allocate and enforce different bandwidths to different connections. Second, to avoid contention, as described before. Contention occurs within the router network, but also at the links between routers and network interfaces. Especially the latter depends very much on the mapping of tiles to network-interface ports. If many connections use the same network-interface-router link, a large slot table is required. The slot table can also be large in case there is a connection which requires significantly more bandwidth than another connection and if it is required to divide the bandwidth in a fine granularity. The contention that occurs within the router network, depends mostly on the topology. A star topology, for example, funnels all connections to a single bottleneck, and requires a large slot table. A highly connected topology has less contention because links are less used, and because alternative paths may be available to route around congested areas. Thus, the slot-table size, and the slot allocation are determined by the quality of the mapping, routing, and slot allocation algorithms.

(ii) *Routers area*: depends on the number of routers, their degree (number of inputs and outputs) and the type of router, because the guaranteed-throughput and best-effort buffers in the routers have a fixed size. The number of routers and their degree is determined by the topology. There are two types of routers, one supports both guaranteed-throughput and best-effort connections (GT+BE), and the other supports only guaranteed-throughput connections (GT-only). The GT+BE router contains guaranteed-throughput and best-effort buffers whereas the GT-only router contains only guaranteed-throughput buffers. Furthermore, the GT+BE router requires additional hardware for a run-time scheduler to schedule best-effort packets. Therefore, the area cost of a GT-only router is smaller than a GT+BE router. For example, a 6x6 GT+BE router occupies 0.175 mm^2 and a 6x6 GT-only router occupies 0.033 mm^2 [33] in 0.13 μm process technology.

Reducing the area of the network requires a trade-off between minimising the number of routers and network interfaces, and minimising contention. In [31] the area for a single network-interface and single router, are estimated by Eq. (5.1) and Eq. (5.2), respectively, assuming GT+BE routers, 500 MHz operation, testable, with worst-case military back-annotated lay-out timing, in Philips's 0.13 μm process technology. In these equations p denotes the number of ports, c denotes the number of connections per port, q denotes the average buffer depth in the number of words, and a denotes the router degree. For the network-interface and router buffers, Eq. (5.1) and Eq. (5.2) count for hardware ripple-through FIFO buffers [90] that are faster and smaller than flip-flop-based FIFO buffers.

$$A_{\mathrm{NI}}(p,c,q) = (19.6pc + 0.72pcq + 4.8) \cdot 10^{-3}\ \mathrm{mm}^2 \quad (5.1)$$

$$A_{\mathrm{R}}(a) = (0.808a^2 + 23a) \cdot 10^{-3}\ \mathrm{mm}^2 \quad (5.2)$$

The network tool chain can generate RTL VHDL for gate-level synthesis [32] to derive more accurate area-cost estimates, but this is beyond the scope of this thesis.

5.3 Comparison design-space exploration and reference design

The network cost is now investigated in terms of network cell area and communication latency.

5.3.1 Network cell area

In this section, we assess the impact of (i) the network topology, (ii) the use of GT+BE versus GT-only routers, and (iii) the number of connections in the design.

(i) Network topology: to explore the impact of network topology on network area we implemented different networks without further optimisations. Table 5.1 contains the estimated area results. The cell areas of the network-interfaces and routers are estimated by the automated tool chain, using Eq. (5.1) and Eq. (5.2), respectively. The total estimated network cell area is shown in the row labelled *total est. cell area*. The designs gen3_1-gen3_4 contain all mesh topologies, but with different sizes, from a 1x1 up to a 2x2 mesh. Design gen3_1 and gen3_2 contain six network interfaces and design gen3_3 and gen3_4 contain eight network interfaces. All designs are based on 29 guaranteed-throughput and four best-effort connections, as described in the previous section. The connections have low communication-bandwidth requirements that can be accommodated by the network. The host micro controller (μC) uses only one connection to program the network, therefore, occupying only one slot in the slot table. Recall that each network connection occupies four network-interface buffers and that each network interface has an additional configuration port that occupies two buffers. Therefore, a network instance with six network interfaces consists of 144 buffers and one with eight network interfaces consist of 148 buffers. The buffer sizes of guaranteed-throughput connections are computed by the automated design flow and buffers of size eight words are used for the best-effort connections.

Recall that the slot-table size is affected by the number of connections that occupy the link between a network interface and router (depending mainly on the mapping), and the contention on links in the routing network (depending on routing and slot allocation). The design gen3_1 consists of a 1x1 mesh and has only network interface-router contention, leading to a slot-table size of 14 slots. The design gen3_2 consists of a 1x2 mesh and additionally suffers from contention in the network, but the slot table contains still 14 slots due to the network interface-router contentions that are dominant. The network interface-router contention is reduced in the designs gen3_3 and gen3_4, because they contain eight network interfaces instead of six. The 2x2 mesh offers more freedom to the routing algorithm, but for both designs the slot-table sizes are equal to 12 slots due to the network interface-router contentions that are dominant. Although the slot-table size impacts the network-interface buffering cost of high-bandwidth connections, the large number of low-bandwidth connections minimises the impact on the network-interface area. The difference in the number of

design	gen3_1	gen3_2	gen3_3	gen3_4
mesh	1x1	1x2	1x2	2x2
# routers (and degree)	1(6)	2(4)	2(5)	4(4)
# network interfaces	6	6	8	8
slot-table size (slots)	14	14	12	12
# buffers	144	144	148	148
avg. buffer size (words)	4.5	4.5	4.6	4.6
est. cell area NI (mm^2)	1.91	1.91	1.98	1.98
est. cell area R (mm^2)	0.17	0.21	0.27	0.42
total est. cell area (mm^2)	2.07	2.12	2.25	2.40

Table 5.1: Effects of topology scaling on network area.

design	gen3_1gt	gen3_2gt	gen3_3gt	gen3_3gt
mesh	1x1	1x2	1x2	2x2
# routers and degree	1(6)	2(4)	2(5)	4(4)
# network interfaces	6	6	8	8
slot-table size	14	14	12	12
# buffers	144	144	148	148
avg. buffer size (words)	4.1	4.1	4.1	4.1
est. cell area NI (mm^2)	1.86	1.86	1.93	1.93
est. cell area R (mm^2)	0.03	0.04	0.06	0.09
total est. cell area (mm^2)	1.89	1.90	1.98	2.02
difference with BE+GT	-9 %	-10 %	-12 %	-16 %

Table 5.2: Network cell area estimation of GT-only optimisation.

network interfaces and the difference in the router area has the most impact on the total network area.

(ii) GT+BE versus GT-only routers: the area of the network can be reduced by using GT-only routers, because the 6x6 GT+BE router occupies 0.175 mm^2, and a 6x6 GT-only router 0.033 mm^2 [33] (roughly decreased by a factor five). The four programming connections, which were best-effort connections, are converted to guaranteed-throughput connections. As a result, the unified mapping, routing and slot-allocation algorithm finds a different tile-to-network interface mapping so that the slot-table size does not increase, as shown in Table 5.2. Formerly, buffer capacities of eight words were used for best-effort connections. Now all buffer sizes are computed by the automated design flow. Therefore, for all the designs, the average buffer sizes are reduced slightly, because the programming connections have small bursts and low bandwidth requirements. The reduction in buffer capacities has a small impact on the network-interface area. The router areas are reduced with roughly a factor five. Therefore, the total estimated network cell area is reduced between 9 % and 16 %, as shown in Table 5.2. Furthermore, the former best-effort connections now have a guaranteed throughput and a bounded communication latency.

The previous designs demonstrate that the network cell area is mainly determined

design	gen3_3	gen3_3gt	gen3_3gt_opt
mesh	1x2	1x2	1x2
# routers and degree	2(5)	2(5)	2(5)
# network interfaces	8	8	8
slot-table size	12	12	8
# buffers	148	148	104
avg. buffer size (words)	4.6	4.1	4.6
est. cell area NI (mm^2)	1.98	1.93	1.40
est. cell area R (mm^2)	0.27	0.06	0.06
total est. cell area (mm^2)	2.25	1.98	1.46
difference with gen3_3	ref	-12 %	-35 %

Table 5.3: Area results of the 1x2 mesh networks with eight network interfaces.

by the number of connections (i.e. number of buffers). When there are also high-bandwidth connections, the network area is also affected by the contention in the network (affecting the slot-table size and buffer capacities). We have illustrated that converting best-effort connections to guaranteed-throughput connections reduces the network area. The slot-table size can be reduced even further by making use of a technique called channel trees [35] that enables slot sharing between low-latency connections. In our designs, sharing slots has a small impact on the network area, because our buffers are already very small. The number of network-interface buffers could be reduced by reconfiguring network connections between use cases [36], instead of dimensioning the network for the union of use cases. In our case study, reconfiguration could save only a few buffers at the network interface that is connected to DSP tile 2, because some audio streams from input peripherals are mutual exclusive.

(iii) number of connections: the following designs use specific optimisations that are design dependent, unlike the previous trade-offs that could all be automatically generated by the design flow. In design gen3_3gt_opt the number of guaranteed-throughput connections is reduced from 33 to 22 for determining the impact on the number of connections. This reduction of connections in the network is achieved by sharing the low-bandwidth connections from and to peripherals, by means of combining them in one tile, similarly as tile IO in Fig. 5.2. The communication to the peripherals can, for example, be implemented as address-based communication. The number of network-interface buffers is reduced from 148 to 104, as shown in Table 5.3. Although the slot-table size is reduced (from 12 to 8), the average buffer depth is slightly increased (from 4.1 to 4.6), because we remove mainly small buffers. The total estimated network area is 35 % lower than the original design gen3_3, and it is 26 % lower than the design gen3_3gt. When comparing the design gen3_3gt_opt with the traditional interconnect in SAF7780, the area increase is only a few percent on total chip area.

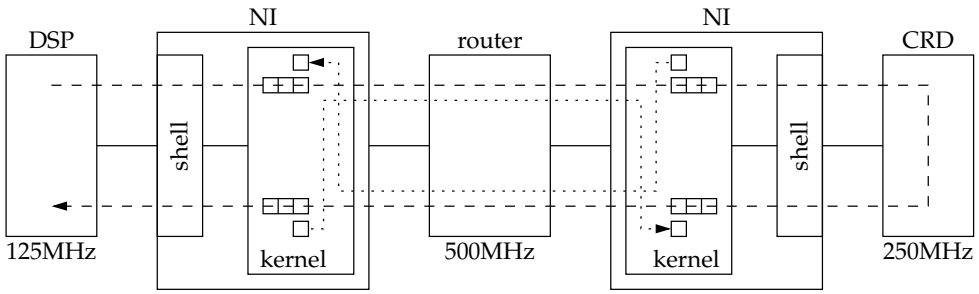

Figure 5.4: Round-trip latency from DSP to CRD and back to the DSP.

5.3.2 Network communication latency

Analog terrestrial-radio demodulation and channel-equalisation jobs contain feedback loops that lead to tight latency constraints in the communication between a DSP processor and CRD hardware accelerator. In the SAF7780 design, this round-trip latency is 288 ns, as described in Section 5.1.2. In a network-based architecture, the DSP and CRD are attached to two different network interfaces. Clock domain crossings in network-interface kernels enable the CRD and the network to process at a higher clock frequency than the 125 MHz clock frequency of the DSP processor. As mentioned before, the round-trip latency is composed of interconnect latency and computation latency. A higher CRD clock frequency results in a lower computation latency (execution time) and, therefore, more relaxed constraint for the communication latency.

The round-trip latency from the DSP to the CRD and back to the DSP is illustrated with the dashed arrow in Fig. 5.4. For the purpose of analysing the round-trip latency we assume an implementation in 0.13 μm technology. The clock frequency of the EPICS is assumed to be 125 MHz, which is the same as in the SAF7780. The network can run at a clock frequency of 500 MHz in 0.13 μm technology. In this technology it is expected that the CRD can run at a clock frequency of 250 MHz. There is a connection from the DSP to the CRD and there is a connection from the CRD to the DSP. Both connections are configured as guaranteed-throughput connections. The end-to-end flow control credits of one connection are piggy-backed on messages send over the other connection, as illustrated with the dotted arrows in Fig. 5.4.

The CRD is running at a clock frequency of 250 MHz, therefore, the task executed on the CRD has a computation latency (execution time) of 144 ns (36 clock cycles at 250 MHz, i.e. $^{36}/_{250 \cdot 10^6}$). The communication latency depends on the length of the message and the allocation of slots in the slot table. The CRD reads four words from the input connection and writes two words to the output connection. In the case address-less inter-tile communication is used, no extra data (e.g. control and address) is added to a message. Low latency results can be achieved by reserving multiple slots spread over the slot table, so that the distance between two allocated slots is small. For example, reserving one slot out of every two consecutive slots results in a 50 % bandwidth allocation and an upper bound on the network-interface latency of 108 ns (including the scheduling latency). The latency introduced by the

clock-domain boundaries and by the router is 32 ns and 12 ns, respectively. Therefore, the total round-trip latency is $144 + 108 + 32 + 12 = 296$ ns. This round-trip latency is 2.7 % higher than the round-trip latency in the SAF7780 (which is 288 ns).

5.4 Concluding remarks

To investigate the impact on area and performance, we presented an interconnect comparison for the communication requirements of infotainment-nucleus generation three. We conclude that it is feasible to replace the traditional interconnects in the platform SAF7780 by an Æthereal network and still meet the communication bandwidth and latency requirements.

The experiments in this thesis have shown that it is worthwhile to trade-off guaranteed-throughput and best-effort connections (using BE connections and GT+BE routers, or only GT connections and GT-only routers). For our 2x2 mesh network, this lead to an area reduction of 16%. The network designs demonstrate that the network-area cost is mainly determined by the number of connections (translating to a number of buffers) and the network topology (affecting the number of routers, the slot table and the sizes of the buffers). The large number of low-bandwidth peripheral connections need special attention. Essentially it is their number rather than their low-bandwidth that causes most cost. After reducing the number of low-bandwidth peripheral connections from 33 to 22 (e.g. by using address-based communication instead of address-less communication), the area is reduced by 35 % (for the 1x2 mesh with eight network interfaces). The network is competitive in terms of area with the current dedicated interconnects in the platform SAF7780, i.e. the area increase was only a few percent on the total chip area.

Communication-latency requirements are typically harder to meet than communication-bandwidth requirements. In case of the Æthereal network, the communication latency is reduced by over-allocating slots and to spread them over the slot table, which is at the cost of bandwidth efficiency. Over-allocating slots and to spreading them over the slot table, reduces the distance between two allocated slots (reducing the time before a container is scheduled) and it reduces the number of slot-table rotations. Furthermore, the tiles are able to run at its own clock frequency, which allows lower execution times of tasks and more relaxed communication requirements. The round-trip latency between the DSP and CRD is only 2.7 % higher than the round-trip latency in the SAF7780. From a predictability perspective, the guaranteed communication services, offered by the Æthereal network [33], are a step forward in mastering the programming effort (i.e. we can guarantee that data is delivered in order and in time). Furthermore, it is shown that a predictable design does not have, per definition, a large increase in cost.

Chapter 6

Case study: analysing real-time performance of a channel equaliser

For the case study in the previous chapter, we generated multiple network instances for given communication requirements. In this chapter, we describe an industrial case study in which we map a channel-equaliser job to a platform instance. For this mapping, we compute a conservatively-estimated lower bound on the channel-equaliser's throughput, by means of dataflow modelling and analysis techniques, as described in Chapter 4. Furthermore, we compared this conservatively-estimated lower bound with an optimistically-estimated lower bound that is measured with a cycle-accurate simulator. This allows us to reason about the accuracy of the conservatively-estimated lower bound on the throughput. This case study is also published in [57].

The outline of this chapter is as follows. First, in Section 6.1, we introduce the channel-equaliser job and the platform on which this job is executed. Section 6.2 models the channel equaliser in a dataflow model and derives a conservatively-estimated lower bound on the throughput. In Section 6.3, this lower bound is compared with the throughput measured in a cycle-accurate simulation environment of the implementation. Section 6.4 elaborates on the comparison by identifying three causes for the differences between the computed and measured throughput. Finally, we conclude our case study in Section 6.5.

6.1 Channel equaliser implementation

Channel equalisation is used to reduce multipath distortion in an FM signal, as is illustrated by Fig. 6.1. Houses, cars, and hills reflect FM signals and these reflections cause variations in the magnitude and phase of the FM signal. Multipath can be described as a complex digital transversal filter $C(Z)$ because there is a delay between

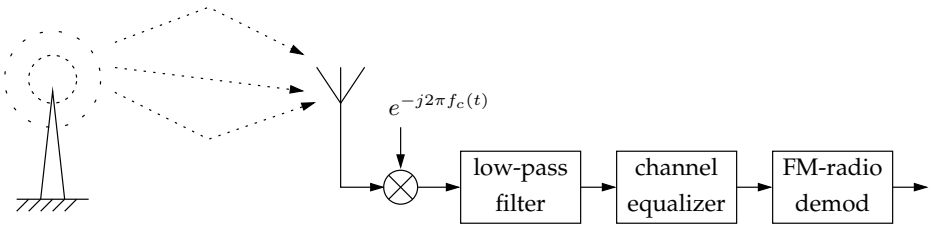

Figure 6.1: Channel equalisation is used to compensate for multipath distortion in an FM-signal.

different paths and each path has a different phase and magnitude.

$$C(Z) = a + b \cdot Z^{-\Delta_1} + c \cdot Z^{-\Delta_2} + d \cdot Z^{-\Delta_3} + ... \tag{6.1}$$

To correct for multipath distortion, a complex digital filter can be made which approximates the inverse of the multipath filter. The channel equaliser should be adaptive in car radios, because the channel characteristics vary over time.

The channel-equaliser job is specified in a sequential (executable) C-code, which we refer to as the reference code. The reference code is manually rewritten, so that task level parallelism is explicit. After partitioning the job into tasks, each task is represented with its own executable C-code that is derived from the reference code. Accesses to shared variables (communication) are made explicit in the C-code of each task. The tasks and the communication between tasks is represented by a task graph that represents the channel-equaliser job. Characteristics, like container sizes, number of consumed and produced containers per task execution, are derived from static code analysis.

The job's tasks are executed on the multiprocessor platform that is depicted in Fig. 6.2. This platform consists of hardware accelerator (ACC) tiles, peripheral (PER) tiles, and digital-signal-processor (DSP) tiles. The DSP tiles contain an EPICS processor [72] which has a dual-Harvard architecture with three memories (M), one memory for instructions and two memories for data. The EPICS processor comes with a tool suit that consists of a C-compiler and a worst-case execution-time analysis tool. In our case-study, this tool is used for deriving, from the compiled C-code, a conservatively-estimated upper bound on the execution time of a task. The tiles communicate via the Æthereal network. The design flow, which is described in Section 5.2, is also able to generate a SystemC [44] simulator of the generated multiprocessor platform. The simulator is automatically generated from the SystemC models of tiles, network interfaces and routers. The network interfaces and routers make use of flit-accurate models in SystemC, because the network packets have a granularity of a flit (three words). The tiles make use of cycle-accurate [21] models in SystemC. The DSP tile makes use of an *instruction set simulator* to model the EPICS processor. The processor tiles contain also local memories, the size and the initial content of these memories are configured at the start of the simulation.

The channel-equaliser job is mapped to the hatched tiles in Fig. 6.2. The analog-to-digital converter (ADC) peripheral is the input of our channel equaliser. We make use of one CRD and one FIR hardware accelerator for performance and cost reasons.

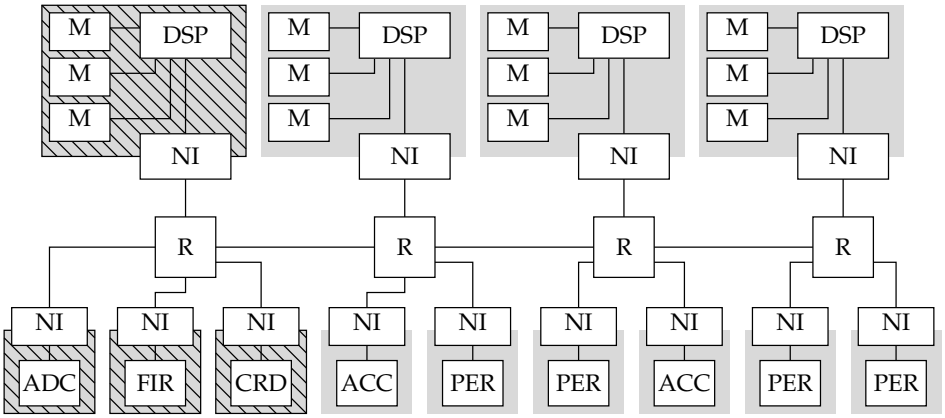

Figure 6.2: Heterogeneous multiprocessor system.

Furthermore, the channel-equaliser job makes use of one DSP processor. The output of the channel equaliser is sent to the input of the FM-radio demodulation job, which is mapped on the remaining (not hatched) tiles in Fig. 6.2. The channel equaliser's real-time performance can be derived independent from the demodulation job, because the DSP processors have private local memories and the network connections have guaranteed-throughput services, as described in Chapter 3.

6.2 Performance analysis via a dataflow model

For our case study, we are interested in the throughput of the channel equaliser job. In this section, first we model the channel equaliser in a CSDF graph and, subsequently, we compute the maximum achievable throughput of the CSDF graph. The computed throughput represents a conservatively-estimated lower bound on the actual throughput in the implementation.

The task graph of the channel-equaliser job is modelled in the CSDF graph that is shown in Fig. 6.3. Actor *adc* is modelling the strictly-periodic analog-to-digital convertor and actor *rad* is modelling the strictly-periodic input of the FM-radio receiver. The production and consumption rates that are equal to one in every phase, are not depicted in Fig. 6.3 in order to keep the figure conveniently. The production and consumption rates specified by the symbol y, are equal to $\langle 1, 0, 0, 0, 0, 0, 0, 0 \rangle$. Therefore, the number of firings of actor *absx* and *log* is eight times lower than the number of firings of the other actors.

The channel-equaliser job is mapped onto our multiprocessor platform. After each mapping decision we add constraints to the CSDF graph and compute an estimated throughput with maximum-cycle-mean analysis, allowing early feedback and short design iterations. Once all mapping steps are taken into account in the CSDF model, the computed throughput is a conservatively-estimate lower bound compared to the actual throughput in the implementation. The channel-equaliser mapping is com-

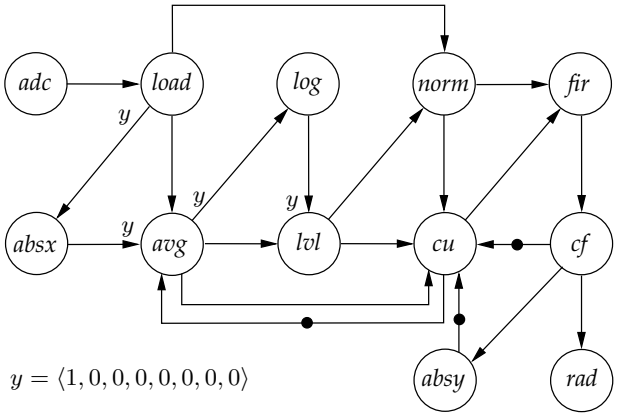

Figure 6.3: CSDF graph modelling of the channel equaliser job.

posed of three steps: (i) binding tasks to tiles, (ii) mapping inter-task communication channels, and (iii) determining static-order schedules.

(i) Binding tasks to tiles: Our application consists of twelve tasks. The actors *adc* and *rad* are strictly periodic with the period $1/f_s$, where f_s is the sample frequency. We offload the DSP by binding task *fir* to tile FIR and tasks *absx*, *absy*, and *log* to tile CRD. One execution of the CRD hardware accelerator takes 34 clock-cycles and the CRD is executed with a clock frequency of 250 MHz. Therefore, the execution time is 144 ns for the tasks *absx*, *absy*, and *log*, as shown in Table 6.1. The remaining tasks are executed on the DSP tile. The execution times can be bound from above, because all data depended loops can be bounded. The notation 8x224, in Table 6.1, is a short-hand notation for a cyclo-static execution time of 224,224,224,224,224,224,224,224 with eight phases. The conservatively-estimated upper bounds on the execution times are annotated to the execution times of the corresponding actors in the CSDF graph in Fig. 6.3. For early feedback, we already compute the maximum cycle mean of the current CSDF graph. This maximum cycle mean is 49248 ns. During this maximum-cycle-mean period, 8 samples are read from actor *adc* and 8 samples are written to actor *rad*. This results in an estimated throughput of $f_s = 8/(49248 \cdot 10^{-9})$ =162 kHz. This estimate does not include the impact of communication latencies and static-order scheduling of tasks.

(ii) Mapping inter-task communication channels: The next step is to set up point-to-point network connections for inter-task communication and to model these network connections in the CSDF graph. Our binding of tasks to tiles requires five connections in the network, namely from tile ADC to DSP, from DSP to FIR, from FIR to DSP, from DSP to CRD, from CRD to DSP, and from DSP to the input of the radio demodulation. The remaining inter-task communication channels are implemented with circular buffers in the local private memory of the DSP processor. The tool that comes with the network [38] is able to generate a configuration for the network, in such a way that all network connections have a guaranteed-throughput service. We compute an upper bound on the communication latency for each point-to-point connection. Notice that multiple communication channels use one single network connection. For

Task u_x	Tile	$\hat{\tau}(u_x)$ [ns]
adc	ADC	$1/f_s$
load	DSP	8x224
absx	CRD	144
avg	DSP	704, 7x416
log	CRD	144
lvl	DSP	440, 7x24
norm	DSP	328
cu	DSP	4944, 7648, 6x4944
fir	FIR	144
cf	DSP	328
absy	CRD	144
rad	-	$1/f_s$

Table 6.1: The binding of tasks to tiles and their corresponding cyclo-static conservatively-estimated upper bounds on the execution times.

example the channels (*norm*,*fir*) and (*cu*,*fir*) are both mapped on the connection from DSP to FIR tile. Sharing a network connection can result in a higher communication latency. However, in our implementation we know that their communication is mutually exclusive, because of the static-order schedule of tasks. In Fig. 6.4 the communication latencies are represented by actors *c1* through *c11*. The FIFO buffers in the network interfaces are modelled with a forward and backward edge and the number of initial tokens on the backward edge are representing the FIFO buffer capacities. In our multiprocessor platform the capacity of each network-interface buffer is 32 words, which is sufficiently large. For the current CSDF graph, in which the computation of tasks and communication latencies are modelled, the maximum cycle mean is 51940 ns and the estimated throughput is $f_s = {}^{8}/_{(51940 \cdot 10^{-9})}$ =154 kHz.

(iii) Determining static-order schedules: Six tasks are executed on the DSP tile and three tasks are executed on the CRD tile. The static-order schedule on the DSP processor is (*cu*, *load*, *avg*, *lvl*, *cf*, *norm*). We came to this static-order schedule by optimising the processor utilisation, i.e. task *fir* can be executed in parallel with the tasks that are executed on the DSP processor. In other words, in this static-order schedule, the processor does not have to wait until task *fir* finished its execution. Additional edges are added to the CSDF graph, so that the static-order schedule dependencies are modelled, as shown in Fig. 6.5. The preamble to this fixed order schedule is the execution of tasks (*load*, *absx*, *avg*, *log*, *lvl*, *norm*). The preamble is modelled by the placement of the initial tokens in the CSDF graph. Strictly speaking the tile CRD contains a first-come first-serve scheduling mechanism, but the tasks *absx*, *absy*, and *log* are mutual exclusive due to the static-order schedule of the DSP processor. Therefore, no additional edges are added to these tasks. In the final CSDF graph that models the computation of tasks, communication latencies, and static-order schedule of tasks, the maximum cycle mean is 54616 ns. Therefore, the throughput estimate is $f_s = {}^{8}/_{(54616 \cdot 10^{-9})}$ =146.4 kHz. This throughput is a conservatively-estimated lower bound on the throughput of the channel equaliser, because the temporal behaviour of the CSDF graph is conservative with respect to the temporal behaviour of the

actor	communication channel	container size [words]	communication latency [ns]
c1	(*adc*,*load*)	2	66
c2	(*load*,*absx*)	4	114
c3	(*absx*,*avg*)	4	114
c4	(*avg*,*log*)	4	114
c5	(*log*,*lvl*)	4	114
c6	(*cf*,*absy*)	4	114
c7	(*absy*,*cu*)	4	114
c8	(*norm*,*fir*)	3	66
c9	(*cu*,*fir*)	17	162
c10	(*fir*,*cf*)	4	114
c11	(*cf*,*rad*)	2	66

Table 6.2: Actor *c1* through *c11* model the communication latency of a container, with a certain size, that is sent over a communication channel.

implementation.

6.3 Performance comparison with simulation

The throughput of the channel-equaliser implementation is measured by means of cycle-accurate simulation. From this measurement, we can derive an optimistically-estimated lower bound on the throughput that we can compare with the conservatively-estimated lower bound on the throughput, which is derived in previous section.

To be able to measure the channel-equaliser's throughput, we need an implementation of the platform and we need to run the channel-equaliser software on this platform. It takes a significant effort to build such a system implementation. Furthermore, if we conclude that the implementation does not meet its requirements, we need to change the platform or the software. These changes can take a large effort, resulting in time-consuming design iterations.

For this thesis, we took this effort and built a cycle-accurate SystemC implementation, so that we can analyse the tightness of the conservatively-estimated lower bound on the throughput, which is computed in previous section. This throughput is derived from the maximum cycle mean, which is 54616 ns. During one maximum-cycle-mean period actor *adc* produced eight tokens and actor *rad* consumes eight tokens. Let $a_i(m, j)$ be the arrival time of container j at the input channel m of the radio demodulation task in the implementation. The execution period P in the implementation is defined as $P = a_i(m, j) - a_i(m, j-8)$. With cycle-accurate simulation and after assuring that the task ADC and radio demodulation do not determine the throughput, we have measured a maximum P of 49080 ns, an average P of 48609 ns and a minimum P of 48366 ns. The difference between the measured maximum period P and the computed maximum-cycle-mean period is $54616 - 49080$ =5536 ns, which is 10.1 % compared to the maximum cycle mean. We don't know the actual

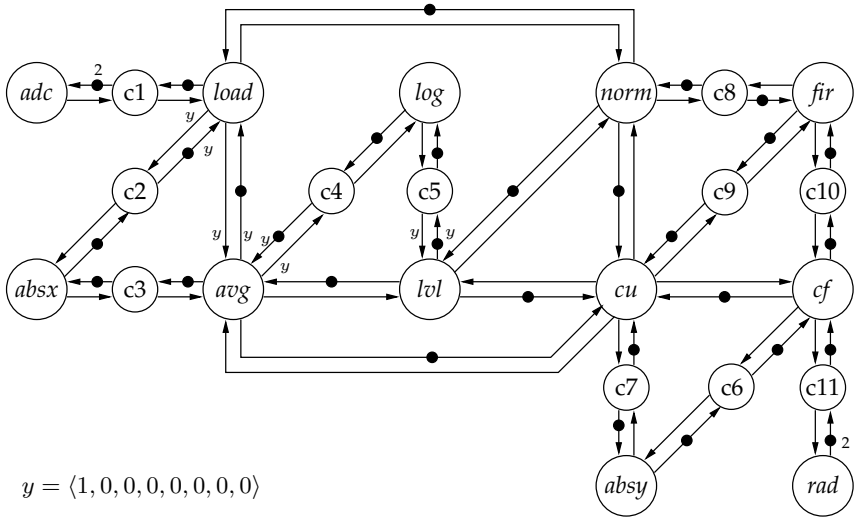

Figure 6.4: CSDF graph in which the inter-task communication is modelled.

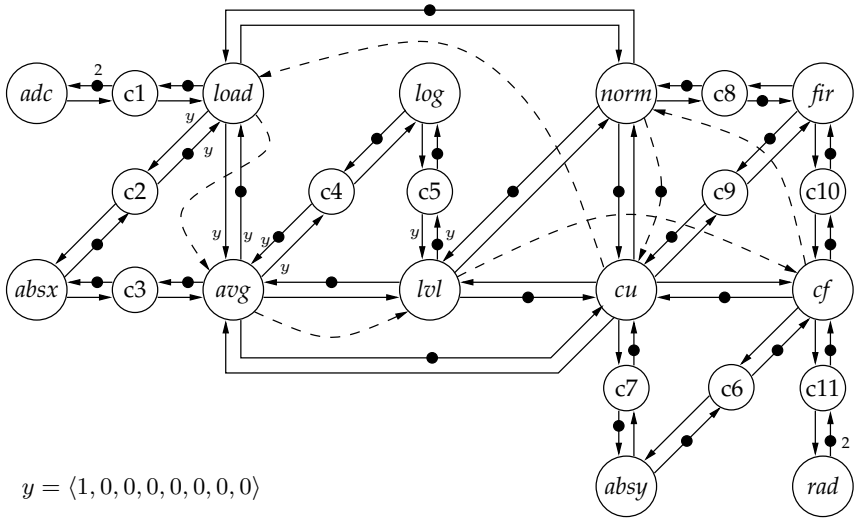

Figure 6.5: CSDF graph in which communication latencies and static-order schedule of tasks are modelled.

minimum throughput of the channel equaliser, but it is sure that the difference between the conservatively-estimated lower bound on the throughput and the actual minimum throughput is less than 10.1 %.

6.4 Sources of inaccuracy

In this section, we identify three causes for the difference between the computed and measured lower bound on the throughput. These causes are: (i) overestimated upper bounds on execution times, (ii) earlier production and consumption of data, and (iii) unknown state of the time-division-multiplex schedule in the network. A trace gives a good impression about the contribution of these sources on the total inaccuracy. A 10 μs trace from the DSP, CRD and FIR tile is shown in Fig. 6.6 and Fig. 6.7 for the CSDF graph and implementation, respectively.

(i) Overestimated upper bounds on execution times: The execution time of an actor is a compile-time conservatively-estimated upper bound on the execution time. Conservatively-estimated upper bounds on execution-times (worst-case execution times) have been actively investigated in the real-time system design community [95]. Variation on the execution time is, for example, a consequence of conditional branches, data dependent loops, and varying memory access latencies. For our channel-equaliser implementation, all tasks executed on the processor are part of the critical cycle that determines the maximum cycle mean. Therefore, variation in the execution times of tasks is affecting the throughput linearly. The sum of conservatively-estimated upper bounds on execution times is 53520 ns in one maximum-cycle-mean period. The maximum of the sum of measured execution times is 48700 ns in one period P. The difference between the sum of upper bounds on execution times and measured execution times is $53520 - 48700 = 4820$ ns, which is 8.8 % of the maximum-cycle-mean period. In our case study, this cause has the biggest impact on the difference between the computed and measured throughput, because it is affecting the throughput linearly and the maximum difference is 10.1 %. The conservatively-estimated upper bounds on execution times are tight (within 8.8 %), because the DSP processor has local memories that are not shared with other processors. A processor with a shared local memory or with a cached memory would potentially increase the inaccuracy in conservatively-estimating the upper bounds on execution times.

(ii) Earlier production and consumption of data: In a static-order schedule, a task can consume and produce containers earlier than an actor can consume and produce the corresponding tokens, because a task is started after the previous task in the static-order schedule is finished whereas an actor is enabled if sufficient tokens are available on every input edge (firing rule), as explained in Section 4.3.1. In our case study, the processor is not stalled between execution of tasks *load* and *avg* (as depicted in Fig. 6.7), whereas the processor is idle between execution of actors *load* and *avg* (as depicted in Fig. 6.6). The impact of this cause is at most 994 ns on one maximum-cycle-mean period. This 944 ns is 1.8 % of the maximum cycle mean, therefore, this cause has a small impact on the lower bound on the throughput. The impact of this cause depends on the critical cycle, which in its turn depends, among other things, on the topology of the CSDF graph. Furthermore, when modelling the implementation in a CSDF graph, a trade-off is made between accuracy and complexity of the

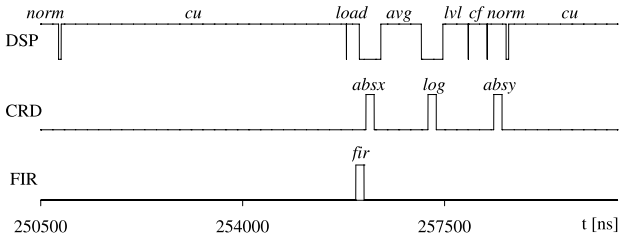

Figure 6.6: A 10 μs trace computed from the CSDF model.

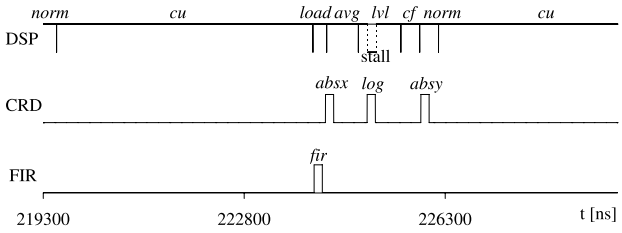

Figure 6.7: A 10 μs trace measured with cycle-accurate simulation.

model. On one hand, an increase in the number of phases of an actor enables a more accurate modelling of input and output behaviour. On the other hand, a higher number of phases results in a more complex model, an increase of the modelling effort, and an increase in runtime to analyse the model.

(iii) Unknown state of the time-division-multiplex schedulers: Even when we run our system with identical input data multiple times, the temporal behaviour of each run can vary. This variation can be caused by a run-time scheduler that has to grant permission in accessing a shared resource, while its behaviour (e.g. state) is not known at design time. In our multiprocessor architecture the network uses time-division-multiplex scheduling. In calculating an upper bound on communication-latency, we assume the worst-case initial state of the slot table. The maximum impact of the unknown state of this slot table is investigated by comparing the original CSDF model with a model that takes the best-case initial state of the slot table into account instead of the worst case. The difference is 544 ns between the maximum cycle means of these two CSDF graphs. Therefore the impact of this cause is at most 1 % on the maximum cycle mean of the original CSDF graph. This impact is small because the slot table in the network is small (only 8 slots). A larger slot table would potentially increase the impact of this cause on the maximum cycle mean.

The difference is 10.1 % between the conservatively-estimated and the optimistically-estimated lower bound on the throughput. This is caused by the combination of the above three causes. In our case study, the first cause effects the throughput linearly, therefore, this cause has an impact of 8.8 % on the throughput. The impact of the second and third causes are not independent, but we know that both causes have together an impact of 1.3 % on the throughput.

6.5 Concluding remarks

In this chapter, we mapped a channel-equaliser job onto our multiprocessor platform. A conservatively-estimated lower bound on the channel-equaliser's throughput is derived by means of dataflow modelling and analysis. An optimistically-estimated lower bound on the throughput is derived with cycle-accurate simulation of the implementation in SystemC. The comparison allowed us to reason about the tightness of the dataflow model compared to the implementation. For our channel-equaliser case study, there is only 10.1 % difference between the conservatively-estimated and optimistically-estimated lower bound on the throughput. The difference is small because of tight conservatively-estimated upper bound on execution times (each processor has a private local memory) and tight upper bounds on the communication latencies (due to a small slot table in the network). With our predictable architecture in combination with the dataflow modelling and analysis techniques, we are able to give tight estimates on the lower bound on the throughput, so that we can come to a cost efficient implementation.

Part II: Multiprocessor architecture extensions

Chapter 7

Shared memory architecture and remote write accesses

In infotainment-nucleus generation one and two the container sizes are small, e.g. one word for a mono sample and two words for a stereo sample. However, in generation three and four the container sizes exceed the network-interface buffer capacities, so the multiprocessor architecture of Chapter 3 does not suffice. Furthermore, the maximum number of supported communication channels is fixed at design time. Therefore, in this section, we will extend the multiprocessor architecture in such a way that the number of communication channels and their capacities are programmable at run time.

The outline of this chapter is as follows. First, we describe an architecture where tiles communicate via shared memories. In such an implementation processors can suffer from stall cycles when accessing a shared memory. An upper bound on the number of processor stall cycles can be derived, as will be described in Section 7.2. Section 7.3 shows that the multiprocessor architecture enables the use of run-time scheduling mechanisms. Therefore, multiple tasks from multiple jobs can be executed on the same processor. Section 7.4 describes how we can construct a dataflow model of a job that is mapped to the architecture with shared memories and run-time schedulers. The dataflow's temporal behaviour is conservative with respect to the temporal behaviour of the implementation, so that we can derive guarantees on the job's throughput and end-to-end latency by making use of existing dataflow-analysis techniques. The presented techniques are applied on our MP3 case study, in Section 7.5. Finally, we conclude in Section 7.6.

7.1 Inter-tile communication via a shared memory

In Chapter 3, we described a multiprocessor architecture where the processing tiles contain local memories, the tile's processor is the only master who is accessing its local memory, and tiles communicate via network connections and make use of address-less communication. The main advantage of this architecture is predictabil-

ity due to limited resource sharing, i.e. local private memories and a network connection for every communication channel.

The communication between tiles was implemented with address-less communication. In this section, we describe an architecture where processors can access their local memory as well as the memories in other tiles via address-based communication. Therefore, this architecture allows inter-tile communication via shared local memories where buffers are implemented in software. The main advantage of communication via a shared memory is flexibility, but this is at the cost of an increase of uncertainty in the temporal behaviour.

7.1.1 Address-less versus address-based communication

An example of address-less communication is Device Transaction Level (DTL) Peer-to-Peer Streaming Data [65]. In case of address-less communication, only data is transferred over a network connection and no additional information (e.g. address) is sent along with the data. This has the advantages that it requires less communication bandwidth than in case when addresses are sent along with the data. However, it requires the allocation of a network connection for every inter-tile communication channel to distinguish data between communication channels, as no additional information is sent next to the data. The number of supported buffers per network interface is chosen at design time. Therefore, the supported number of inter-tile communication channels between two tiles is fixed at design time. The data containers that are sent over an inter-tile communication channel are stored in network-interface buffers. These buffers are implemented in hardware and their buffer capacities are chosen at design time. Therefore, the maximum number of data containers that can be stored is fixed at design time. One of the platform requirements, which is introduced in Chapter 1, is flexibility. An architecture that supports a fixed number of inter-tile communication channels and where the communication channels can store a fixed amount of data containers has a limited flexibility.

There are three important reasons why the system designer wants to store data containers in memory instead of hardware buffers. First, the data containers produced by the processor can exceed the capacity of a hardware buffer. Second, it is desirable that the buffer capacity can be changed by adapting the software, because the required buffer capacity is job dependent. Third, storing data containers in memory is cheaper, in terms of silicon area, than storing them in hardware buffers, because the local memories are already present in a tile.

In contrast with the previous architecture, this one supports address-based communication. When inter-tile communication is implemented with address-based communication, the data containers are stored in a shared memory. Both the producing and consuming processors can access this memory via address-based communication. Between the processors we create a circular buffer [26] to solve the synchronisation between the producing and consuming processor.

Examples of address-based communication are DTL's Memory-Mapped Input Output (MMIO) and Memory-Mapped Block Data (MMBD) [65]. With address-based communication, a processor is able to access the memory in its own tile as well as the memory in another tile. Of course there should be a network connection between

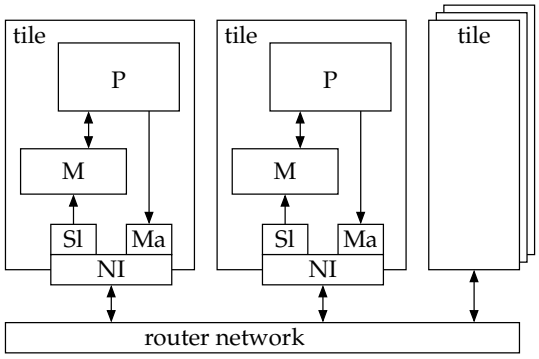

Figure 7.1: Multiprocessor architecture with shared memories.

these tiles. In order to access the memory, a memory address from the memory location has to be sent along with the data. In case of MMIO, a memory address is sent along with every data word. In case of MMBD, one memory address is sent along with every block of data. A block can contain a number of data words and this number is defined in a command that is also sent along with the data. An example of streaming communication implemented with address-based communication is the Sea-of-DSP architecture that is presented in [88]. This architecture is used for the platform SAF7780 (the platform of infotainment-nucleus generation three). This platform contains an Inter-Tile-Communication (ITC) crossbar switch between the DSP tiles. Each DSP can access its local memory and it can write to the memory in another DSP tile, via address-based communication.

With address-based communication additional information is sent with to the data, like the command and address. Therefore, in contrast to address-less communication, a higher communication bandwidth is required to transfer the same amount of data. However, up to infotainment-nucleus generation four, the inter-tile communication bandwidth requirements can be accommodated by our network. An advantage is that the number of inter-tile communication channels is not limited by the number of supported network connections, because the circular buffers and the buffer administrations are stored in memory. Of course there should be sufficient memory space available to store all the circular buffers and buffer administrations. Furthermore, the required number of network connections can be lower than in case of address-less communication, because the data containers from multiple communication channels can be transferred via the same network connection.

Note that address-based communication increases the uncertainty in temporal behaviour of a job, because of arbitration at local memories and sharing of network connections between communication channels.

7.1.2 Implementation of inter-tile communication

The architecture that is considered in this chapter, is illustrated in Fig 7.1. It is a tiled architecture where tiles communicate via a network. Compared to the architecture

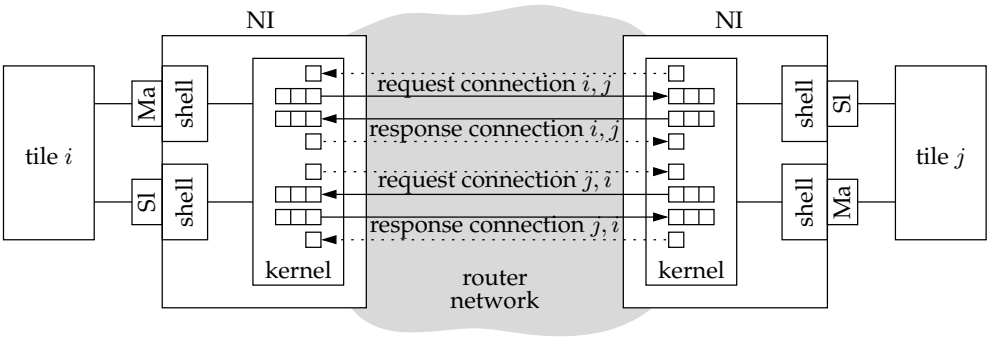

Figure 7.2: Network-interface architecture and network-connection implementation.

in Chapter 3, we now consider shared local memories instead of private local memories. Therefore, a processor can access its own local memory as well as the memories in another tile, as long as there is a network connection between these tiles. A processor has low access latency to its own local memory and a higher access latency to a remote memory.

A tile is connected to a network interface via network-interface ports that can be either master (Ma) or slave (Sl). Master and slave ports are connected to a *network-interface kernel* via *network-interface shells*, as shown in Fig. 7.2.

Transaction requests, like read and write, are issued via the master port. The IP module that is connected to the slave port will execute these requests. Typically, a tile is connected to both a master as well as a slave port. The master port allows the tile to sent data to other tiles, and the slave port allows other tiles to sent data to this tile. A shell converts the transaction requests of a particular IP protocol (e.g. DTL [65] or AXI [2]), into transport-layer messages. The transport-layer message contains the transaction's command (cmd), address (addr) and data information, as depicted in Fig. 7.3. In traditional address-based interconnects (e.g. busses) the processor can address each IP module. In our architecture, the shells transparently deliver transport-layer messages to network connections for backward compatibility. Naturally, there should be a network connection between the processor tile and the tiles of each IP module that the processor wants to address. The shell knows to which network connection the messages should be delivered based on the address that is specified in the transaction. The kernel converts the generic transport-layer messages into network-layer guaranteed-throughput or best-effort packets, this conversion is referred to as packetisation. These packets contain payload (i.e. transport-layer messages) with additional headers that contain the routing path information, as is described in Section 3.2.1.

An address-based network connection is composed of a request and response connection for every master-slave port pair, with two network-interface buffers in each connection, as shown in Fig 7.2. For each connection in the network, flow control is used to prevent data loss caused by buffer overflow and to prevent deadlock. Flow control is implemented using credits, similar as in case of address-less communication that is described in Section 3.2.1. The command, address, and data (in case

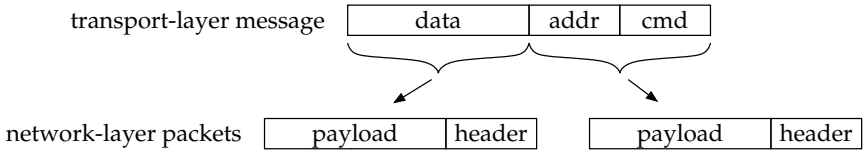

Figure 7.3: Example of a transport-layer message that is transmitted by two network-layer packets.

of a write command) are sent over one connection and the read data (in case of a read command) is sent back over another connection. Therefore, two address-based network connections are required to implement an inter-tile communication channel between two tasks, as we will describe later in this section. One connection from tile i to tile j and one connection from tile j to tile i, as depicted in Fig 7.2. However, these two address-based network connections can be used to transport the data from multiple inter-tile communication channels between tasks that are executed on tile i and tile j.

In this chapter, processors make use of posted-write transactions to send data to a memory in another tile. In case of posted write, the transaction is stored in the network-interface buffer and the processor does not have to wait for an acknowledge, but it can continue doing useful work while the transaction is transported over the network and the data is written at the destination. If there are posted write transactions pending in the network and the network-interface buffers are full, then the processor suffers from stall cycles until the posted write transaction is accepted by the network-interface shell.

In address-based communication, data containers are stored in circular buffers [26] that are located in memories. The implementation of a circular buffer requires four administration registers, namely *base address*, *size*, *write pointer* (WP), and *read pointer* (RP). Before a task can write data into the buffer, it has to verify that an empty container is available in this buffer. If a task wants to read data from the buffer, it has to verify that a full container is available in this buffer. This verification is done by examining the buffer's administration registers. The task that writes data into a circular buffer, updates the write pointer after it completes writing the data in a container. The task that reads data from a circular buffer, updates the read pointer after it completes reading the data from a container. This implementation guarantees memory consistency, because the read pointer is updated by only one task and the write pointer is updated by only one task. Furthermore, data containers are produced and consumed in a FIFO order. A task can have random access within a container, despite the FIFO order between containers. We duplicate the administration registers so that they are stored in the local memory of both processers. Therefore, verification of space and data in a circular buffer, can be done by fetching the administration values from local memory. Updating the read and write pointers has to be done by writing the new pointer value to the administration in a local and remote memory, as will be described below.

A general producer-consumer job is taken as an example in explaining the implementation of the streaming-communication protocol. This protocol is build on top

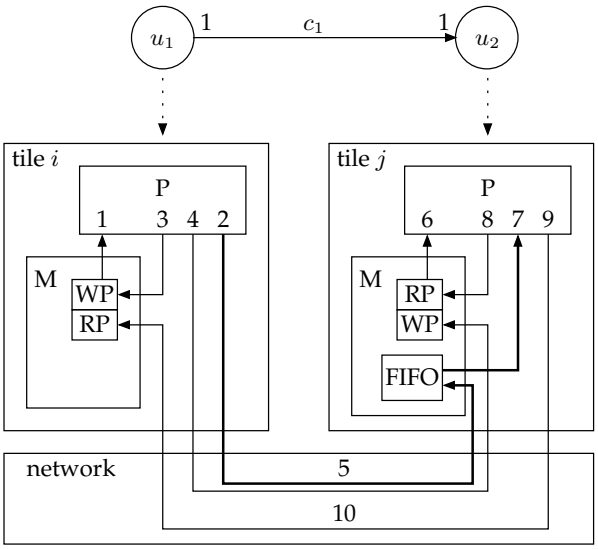

Figure 7.4: Inter-tile communication implemented via shared memories.

of the network services, therefore, it is a higher level protocol compared to the network protocol. Task u_1 is executed on the processor in tile i and task u_2 is executed on the processor in tile j, as depicted in Fig. 7.4. Task u_1 produces containers of data that are stored in the circular buffer (FIFO) that is located in the memory of tile j. These containers are stored in the local memory of the processor in tile j, so that task u_2 has low access latency when consuming them. The processor in tile i has a higher access latency to the memory in tile j, but it generates posted write accesses. Therefore, it can continue processing task u_1 while its output data is transferred over the network and it is written into the memory of tile j. Suppose that the data containers would be stored in the memory of tile i instead of tile j. Then task u_2 would have a higher access latency in consuming its input data and after each read access it has to wait until data returns. Therefore, storing the data containers in the local memory of the processor on which the consuming task is executed, is the preferred solution.

The protocol of streaming communication between task u_1 and u_2 will be described in ten steps. These steps are also depicted in Fig 7.4. In spite of the fact that we describe the steps sequentially, the processors and the network will perform these steps concurrently. In case the circular buffer is able to store two or more data containers, the processor in tile i is able to store data containers at the same time as the processor in tile j can read data containers that were already written. The ten steps are as follows:

1. The processor, on which task u_1 is executed, reads the circular-buffer's administration values from its local memory (i.e. read and write pointers) to see if there is space available to store a data container.

2. In case there is space available, task u_1 produces output data and the processor

stores it in the circular buffer by generating posted write transactions to the memory in tile j.

3. After the posted write transactions are accepted by the network interface in tile i, the processor updates the write pointer in the buffer administration that is located in its local memory.
4. Finally, the processor in tile i updates the write pointer in the buffer administration that is located in the memory of tile j. This is done by generating a posted write transactions to this memory.
5. The output data and write pointer will be transferred from tile i to tile j by the network. The network-interface shell in tile j will do the actual writing of the output data and write pointer to the memory in tile j.
6. The processor, on which task u_2 is executed, reads the circular-buffer's administration values from its local memory (i.e. read and write pointers) to see if there is a data container available in the circular buffer.
7. In case there is a data container available, the processor in tile j reads the data container from its local memory.
8. After the processor has finished reading the data, the processor in tile j updates the read pointer in the buffer administration that is located in its local memory.
9. Finally, the processor in tile j updates the read pointer in the buffer administration that is located in the memory of tile i. This is done by generating a posted write transactions to this memory.
10. The read pointer will be transferred from tile j to tile i by the network. The network-interface shell in tile i will do the actual writing of the read pointer to the memory in tile i.

In case of a remote memory access, the network-interface shell in a tile will do the actual memory access. Therefore, in a tile, an arbiter has to grant the processor or the network-interface shell access to the memory. As a result the processor can suffer from stall cycles when accessing its local memory. The maximum number of processor stall cycles during the execution of a task can be limited by selecting an appropriate arbitration scheme. The arbitration scheme of the arbiter in the tile must have three characteristics. First, a low latency for the local memory accesses of the processor, because a lower latency results in fewer stall cycles for the processor. Second, a guaranteed throughput for the network-interface shell to access the memory, because we should be able to derive an upper bound on the communication latency. Third, it must be simple and cost efficient, so that it can be realised in dedicated hardware. Hosseine-Khayat and Bovopoulos [43] proposed a bus arbitration scheme that satisfies these three requirements. The arbitration has a period, which is called the *service-cycle time*. Each service cycle is divided into a fixed number of *time slots*. A portion of the time slots is reserved for the network-interface shell slave port to store the incoming data into the shared local memory. This ensures that memory bandwidth for incoming data is guaranteed. In this chapter, the reserved time for incoming data is one time slot. In which slot the slave port can access the memory depends on the memory accesses requested by the processor, but it is guaranteed that it can access the memory during one time slot within the service-cycle time.

7.2 Upper bound on processor stall cycles

A processor can suffer from stall cycles when accessing a shared memory during the execution of a task. The execution time of a task is defined by Definition 5 in Chapter 2.1. This execution time does not include processor stall cycles caused by accessing a shared local memory. In order to verify that end-to-end performance requirements are met, we need to derive an upper bound on the number of processor stall cycles during the execution of a task.

When a processor is writing to a circular buffer in a remote memory, it can experience stall cycles due to occupied network-interface buffers and a limited bandwidth allocation in accessing a remote memory. Predicting the number of processor stall cycles can be difficult, because this depends on the traffic pattern generated by the processor (which is often input data dependent) and the availability of space in the network-interface buffers (which depends on allocated bandwidth and the state of the network). However, it is possible to derive an upper bound on the number of processor stall cycles, because in our multiprocessor system we use guaranteed-throughput services in the network and at the memory-port arbiters.

Apart from accessing a remote memory, the processor can also suffer from stall cycles when it accesses its own local memory. This can happen if both the processor and the network-interface shell access the local memory at the same time. During one execution of a task the worst-case number of memory accesses from the network-interface shell to the local memory can be large, due to three reasons. First, the actual execution time of a producing task can be smaller than the worst-case execution time. In this case the producing task can execute a number of times during one execution of a consuming task, i.e. if there is sufficient space in the FIFO buffer between the producing and consuming task. Second, the number of containers that are produced by the producing task, can be large compared to the number of containers that are consumed by the consuming task. Third, a number of tasks can be mapped onto the same processor and a large container of one task can arrive during the execution of another smaller task. However, it is possible to derive an upper bound on the number of processor stall cycles, because the processor has a guaranteed throughput to its local memory.

Predicting a *tight* upper bound on the number of processor stall cycles is difficult. Therefore, it is desirable that the multiprocessor architecture enables the derivation of a tight upper bound on the processor stall cycles. Since a too conservative upper bound can result in a significantly over-dimensioned system.

7.2.1 Processor stall cycles due to remote write accesses

In address-based communication, a processor generates posted write transactions to a memory in another tile. If the processor issues such a transaction and the network interface does not immediately accepts it, then the processor experiences stall cycles.

To derive an upper bound on the number of processor stall cycles, we use the following terminology. The maximum time until the network interface accepts a posted write transaction, is M processor cycles, with $M \in \mathbb{N}$. Note that M depends on the

allocated bandwidth to the network connection from a source to destination tile and in the destination tile on the allocated bandwidth from the network connection to the memory. Furthermore, $\varrho(u_x)$ is defined as a conservatively-estimated upper bound on the number of posted write accesses of task u_x.

In this thesis, we assume that it takes one processor cycle for the processor to access its local memory, assuming that the processor is the only master who is accessing this memory. This memory-access latency is already taken into account in the conservatively-estimated execution time of a task $\hat{\tau}(u_x)$. Therefore, one remote-write access results in at most $(M-1)$ number of processor stall cycles. An upper bound on the number of processor stall cycles $\sigma_1(u_x)$, during one execution of task u_x, can be computed as follows:

$$\sigma_1(u_x) = (M-1) \cdot \varrho(u_x) \tag{7.1}$$

From Eq. (7.1) it follows that the upper bound on the number of processor stall cycles depends on the number of posted write accesses to a remote memory. The upper bound on the number of stall cycles can be expressed with the task's ratio between communication and computation, to see the impact on the number of stall cycles. We define the communication-computation ratio $\rho(u_x)$ as the number of posted write accesses to a remote memory divided by the upper bound on the execution time for a task u_x. In our architecture the communication-computation ratio of task u_x is given by:

$$\rho(u_x) = \frac{\varrho(u_x)}{\hat{\tau}(u_x)}\ ,\ 0 \leq \rho(u_x) \leq 1 \tag{7.2}$$

The value of the communication-computation ratio $\rho(u_x)$ is zero if every cycle on the processor is spent on computation and $\rho(u_x)$ is one if every processor cycle is spent on communication. Equation (7.2) can be substituted in Eq. (7.1). Therefore, the upper bound on the number of processor stall cycles due to remote write accesses equals:

$$\sigma_1(u_x) = (M-1) \cdot \hat{\tau}(u_x) \cdot \rho(u_x) \tag{7.3}$$

From Eq. (7.3), we conclude that the number of processor stall cycles is large if the communication-computation ratio $\rho(u_x)$ of task u_x is large.

The upper bound from Eq. (7.3) can be reduced to zero, by adding a communication assist [18] next to the processor [56]. A communication assist is an autonomous DMA controller that offloads the processing core with communication tasks, like sending posted write transaction to the network interface. Instead of the processor, the communication assist is stalled when sending posted write transactions. A tile with a communication assist allows a high bandwidth allocation for the processor to access its local memory, while it only requires an average bandwidth allocation in the network. Therefore, it decouples computation and communication and still allows flexible allocation of buffers in the local memory of a processor. However, in infotainment-nucleus generation three and four, the communication-computation ratios of a task are typically small (e.g. 0.5 % for an MP3 decoder [56]). Therefore, the impact of a communication assist on the processor performance is also small (e.g. 4 % for the MP3 decoder [56]).

7.2.2 Processor stall cycles due to local memory sharing

A processor can also suffer from stall cycles when accessing its local memory, because other processors can also access this memory. An upper bound on the number of stall cycles can be derived from the number of memory accesses from the processor and network-interface shell within a time interval. However, during one execution of a task, the number of memory accesses from the network-interface shell to the local memory can be large, as described at the beginning of Section 7.2. Therefore, in deriving an upper bound on the number of processor stall cycles, we assume that the network-interface shell wants to access the local memory. As a result of this assumption, we will derive a conservatively-estimated upper bound on the number of processor stall cycles.

The maximum number of processor stall cycles during the execution of a task is bounded as a result of the chosen arbitration scheme [43] at the memory port. In this chapter, the reserved time for the network-interface shell to access the memory is one time slot. One time slot is equal to one processor cycle and the service-cycle time equals to N processor cycles, with $N \in \mathbb{N}$. If the service-cycle time N equals four, it is guaranteed that the network-interface shell can access the memory at least once every four processor cycles. In which slot it can access the memory depends on the memory accesses requested by the processor.

Given this arbitration scheme, we can derive conservatively-estimated upper bounds on the number of processor stall cycles at multiple levels of abstraction. The more information there is available about the memory-access behaviour of a processor, the more accurate upper bound we can derive for the processor stall cycles. In this section, we distinguish three levels of abstraction: (i) we only know the worst-case execution time of a task, (ii) we know an upper bound on the worst-case number of local memory accesses of a task, and (iii) we know the memory-access burst lengths of the individual bursts of a task. For all these abstraction levels we assume an unknown number of memory accesses from the network-interface shell. Now we will derive conservatively-estimated upper bounds on the number of processor stall cycles for these levels of abstraction:

Abstraction level (i): The worst-case execution time of a task is known and the worst-case number of local memory accesses are unknown. Therefore, we assume that the processor is accessing its local memory every cycle, in order to derive a conservatively-estimated upper bound. Given the memory-port arbitration scheme, the processor can access the memory at least $(N-1)$ times within the service-cycle time N. When the processor wants to accesses the memory more than $(N-1)$ times, it can suffer from stall cycles, as depicted in Fig. 7.5. In deriving a conservatively-estimated upper bound, we assuming that the network-interface is backlogged. This means that the network interface has a number of memory-access requests pending in its network-interface buffers. In computing a conservatively-estimated upper bound on the processor stall cycles, we assume that the processor has a stall cycle at its first memory access, because the processor can already have spend its budget in accessing its local memory. After the processor has suffered from the stall cycle, it will receive again budget to access the memory $(N-1)$ times. If the number of memory accesses are below or equal to $(N-1)$, then there are no additional stall cycles for the processor. If the number of memory accesses are above $(N-1)$, then

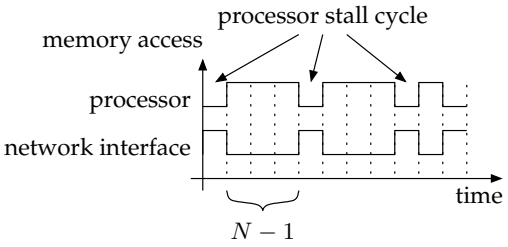

Figure 7.5: Processor stall cycles due to arbitration at the memory port.

the processor will suffer from a stall cycle every $(N-1)$ memory accesses. Therefore, the conservatively-estimated upper bound on the number of processor stall cycles $\sigma_2(u_x)$ during one execution of task u_x, is given by:

$$\sigma_2(u_x) \leq 1 + \left\lfloor \frac{\tau(u_x) - 1}{N-1} \right\rfloor \tag{7.4}$$

Assume that the worst-case execution time $\tau(u_x)$ equals $N-1$, then there is only a processor stall cycle during the first memory access. This matches with the result from Eq. (7.4), because the result of the floor function is zero. When the worst-case execution time $\tau(u_x)$ equals N, then there are two processor stall cycles (one during the first memory access and one at the Nth memory access). Again, this matches with the result from Eq. (7.4).

Abstraction level (ii): If an upper bound on the number of local memory accesses of a task is known, we can derive a tighter upper bound on the number of processor stall cycles. We define $\alpha(u_x)$ as the conservatively-estimated upper bound on the number of local memory accesses of the processor during one execution of task u_x. Given the arbitration scheme, the processor can access the memory $(N-1)$ times within the service-cycle time N. Compared to the previous computed bound, we now have a more accurate estimate on the number of memory accesses and do not need to assume that the processor accesses its memory every cycle. However, it is unknown when the processor accesses its local memory. Therefore, we assume that the processor will access the memory in one burst of $\alpha(u_x)$ memory accesses, since this will result in the highest number of processor stall cycles. So the conservatively-estimated upper bound on the number of processor stall cycles $\sigma_2(u_x)$ for task u_x, is given by:

$$\sigma_2(u_x) \leq 1 + \left\lfloor \frac{\alpha(u_x) - 1}{N-1} \right\rfloor \tag{7.5}$$

The difference between the upper bounds on the number of processor stall cycles of Eq. (7.4) and Eq. (7.5), depends on the difference between $\alpha(u_x)$ and $\tau(u_x)$. When a processor generates a local memory access almost every cycle (i.e. $\alpha(u_x)$ is close to $\tau(u_x)$), then the difference between Eq. (7.4) and (7.5) is small. When the processor does not often access its local memory (i.e. $\alpha(u_x)$ is close to zero), then there is a significant difference between Eq. (7.4) and (7.5). In general, the $\alpha(u_x)$ is lower than $\tau(u_x)$. In [42], they measured the average number of load and store instructions for an Intel 80x86 processor and a TMS320C540x digital signal processor. After executing a collection of integer programs (SPECint92), the average number of instructions

that access the memory, were 34 % and 76 % for the 80x86 and TMS320C540, respectively. Therefore, it is worthwhile to investigate an upper bound on the number of local memory accesses during the execution of a task.

Abstraction level (iii): We can increase the accuracy on the number of processor stall cycles even further, by taking into account the memory-access bursts in which the processor accesses its local memory. Notice that this is not possible in general, because the memory-access bursts can be input data dependent. To compute an upper bound on the number of processor stall cycles, we define b as the length of a memory-access burst and B_x as the set of memory-access bursts that are generated by the processor when executing task u_x. Now the upper bound on the number of processor stall cycles can be computed as follows:

$$\sigma_2(u_x) \leq 1 + \sum_{b \in B_x} \left\lfloor \frac{b-1}{N-1} \right\rfloor \tag{7.6}$$

This upper bound on the number of stall cycles is a summation of Eq. (7.5) for each burst b, where one stall cycle is added for taking into account that the processor does not have any budget when it starts to execute task u_x. The difference between the upper bounds on the number of processor stall cycles of Eq. (7.5) and Eq. (7.6), depends on the number of bursts (i.e. size of B_x) and the size of the individual bursts (i.e. size of $b \in B_x$). When the individual bursts b are smaller than $(N-1)$, the difference between Eq. (7.5) and (7.6) is large. When there are only a few bursts (i.e. B_x is small) and the individual bursts $b \in B_x$ are larger than $(N-1)$, then the difference is smaller.

Dependent on where the bottleneck is in the system, more effort can be spend on making the conservatively-estimated upper bounds more tight.

7.3 Run-time scheduling of task executions

Run-time schedulers are essential in case multiple tasks run at independent rates while they are executed on the same processor, or in case jobs can be started and stopped at run-time while they share a processor with another job that is already running. Typically, tasks from different jobs are running at independent rates. In case of variable consumption or production behaviour of containers (i.e. if the job is represented in a VRDF graph [93]), then tasks from the same job can also run at independent rates. Furthermore, for run-time scheduling, we do not need to store a schedule for each use case, as in case of static-order scheduling.

An example of independent rates between jobs is when the sample rate of one job is locked to an external input while the sample rate of another job is locked to an internal clock. In this case, at design time, it is unknown what the relation is between the number of firings of a task from one job compared to the number of firings of a task from the other job. Only at run time we know if a task is allowed to execute, i.e. at run time we can check whether a task is enabled. Therefore, in run-time schedules, a task executes only when it is enabled to execute. A task is enabled if sufficient data is available at each input and sufficient space is available at each output. Once a task is enabled, it is guaranteed that it can finish after it is started. Circular buffers

support checking for sufficient input data and output space by examining the buffer administrations of the input and output buffers. This allows a scheduler to check whether a task is enabled to execute. Therefore, our architecture supports the use of run-time scheduling of tasks.

A run-time scheduler is predictable if we can derive an upper bound between the time a tasks is enabled to execute and the time this task finished its execution. Therefore, for a predictable run-time scheduler, we should be able to bound the interference caused by scheduling other tasks before or while scheduling the task of interest. We defined response time as follows:

Definition 13. The response time of a task is the time between the task is enabled to executed and the time it finished its execution. A task is enabled if sufficient data is available at each input and sufficient space is available at each output.

Examples of predictable run-time scheduling mechanisms are: round robin and time division multiplex. In case of round robin, which is a non-preemptive scheduling mechanism, each task, in the round-robin list, must have a bounded execution time to be able to derive an upper bound on the response time for a task. If the execution time of a task exceeds its execution-time upper bound, all tasks, in the round-robin list, potentially exceed their response-time upper bound. The advantage of a round-robin scheduler is that it is work conserving, because if one task is unable to execute, the resource budgets can be used by another task. In case of a time division multiplex, which is a preemptive scheduling mechanism, for each task a fixed amount of resource budgets are allocated. Therefore, if the execution time of a task exceeds its execution-time bound, the temporal behaviour of another task is not affected, because these tasks are preempted. It is also non work conserving between tasks, because if resources are not used by one task, these resources cannot be used by other tasks and they are waisted. However, time-division-multiplex scheduling is work conserving within a task, because if a task finished its execution before it finished its allocated budget, then it can already start executing its next execution. Naturally the task should be enabled before it can start executing again.

All schedulers that are predictable (i.e. that are starvation-free so that we can derive upper bounds on the response time) belong to the latency-rate server class. Latency-rate servers [77] were originally proposed as a modelling paradigm to model the effect of scheduling on traffic passing through a chain of heterogeneous routers in a packet-switched network. The behaviour of a latency-rate server is determined by two parameters: the *latency* and *allocated rate* [77]. A *busy period* is defined in [77] as a maximum interval of time (s,t] during which the server is never idle. The latency Θ and rate ρ parameters define an envelope to bound the minimum service offered during a busy period, as illustrated in Fig. 7.6. The latency Θ of an latency-rate server may be seen as the worst-case delay which is seen by the first container of the busy period. After this latency, the offered service is guaranteed to be at a constant rate ρ. The round-robin and time-division-multiplex schedulers fall in the class of latency-rate servers, this is shown for round robin in [77] and for time division multiplex in [92].

The parameters Θ and ρ depend on the run-time scheduling mechanism, the allocated processor resources, and the resource requirements of a task. A processor can suffer from stall cycles due to sharing its local memory or when writing to a remote

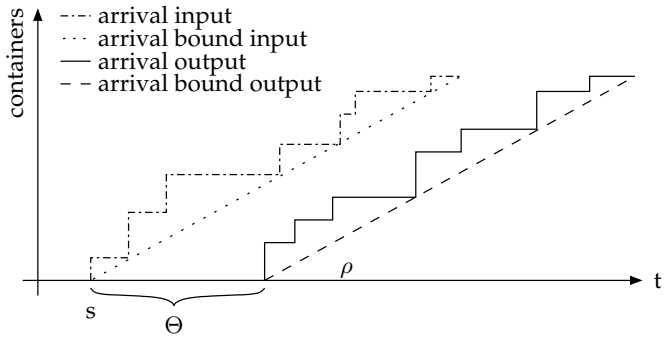

Figure 7.6: Definition of Θ and ρ.

memory, as described in previous section. Therefore, the upper bound on the execution time of a task as well as the upper bound on the number of processor stall cycles are taken into account in computing the latency Θ and rate ρ parameters for representing the run-time scheduling of the task.

7.4 Dataflow model construction

Previous sections introduced two extensions to our predictable multiprocessor architecture, namely inter-tile communication via a shared memory and run-time scheduling of tasks on a processor. In this section, we elaborate on the representation of these extensions in a dataflow model, so that we can use existing dataflow-analysis techniques to compute a lower bound on the throughput and an upper bound on the end-to-end latency of a job. First, we describe the dataflow model representation of tasks that are scheduled using a run-time scheduling mechanism. Second, we model inter-tile communication latencies with additional actors in the dataflow model and we will show how to compute upper bounds on these communication latencies.

7.4.1 Modelling run-time scheduling of tasks

In [54, 4, 94] it is shown that the effects of run-time schedulers, like round-robin and time division multiples, can be taken into account in a dataflow model. This result is refined and generalised in [92] were it is shown that all schedulers that belong to the latency-rate server [77] class can be applied in combination with dataflow analysis.

In [92] it is proven that the dataflow construct in Fig. 7.7 models a task u_x that executes on a latency-rate server with latency Θ_x and allocated rate ρ_x. The temporal behaviour of the latency-rate server is captured by this dataflow model. Note that, in this model, we distinguish actors with and without a self edge. When the execution time of actor v_y is equal to Θ_x and the execution time of actor v_z is equal to $1/\rho_x$, then it is proven in [92] that the arrival times of tokens in the dataflow model are conservative with respect to the arrival times of the corresponding containers in the

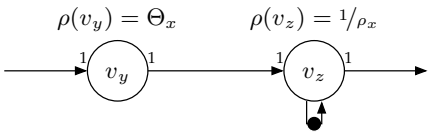

Figure 7.7: Dataflow model representation of a latency-rate server with latency Θ_x and allocated rate ρ_x.

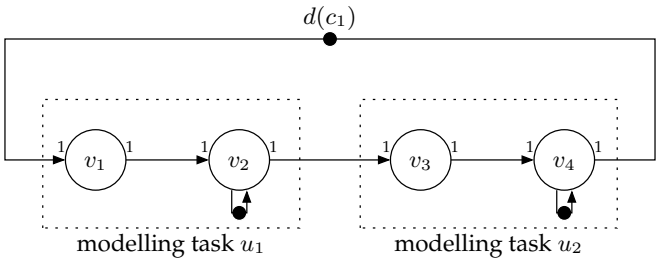

Figure 7.8: Dataflow model of a producer-consumer job from which both tasks are modelled with a latency-rate server component.

implementation.

Traditionally, every task in the implementation is modelled with one actor in the dataflow model. This is similar as we did in Chapter 4, when modelling tasks that are execute in a static-order schedule. However, in modelling run-time schedules that fall in the class latency-rate server, a task is more accurately modelled with a component that consists of two actors. It is shown in [92, 54] that the execution of one task can be represented by a component which can be a collection of dataflow actors, as long as the temporal behaviour of the component is conservative with respect to the temporal behaviour of the task. Therefore, for the dataflow graph of a producer-consumer job in Fig. 7.8, the tasks u_1 and u_2 are represented with a latency-rate server component.

In this section, we assume that both tasks u_1 and u_2 are executed on the same processor. Therefore, the communication between the tasks is via a circular buffer that is located in the local memory of the processor. This buffer is represented with the forward edge (v_2, v_3) and backward edge (v_4, v_1) in the dataflow model, similar as in Chapter 4.3. The number of initial tokens on edge (v_4, v_1) in Fig. 7.9, represent the number of empty data containers in the circular buffer between task u_1 and u_2. That means the capacity $d(c_1)$ of the communication channel $c_1 = (u_1, u_2)$

7.4.2 Modelling inter-tile communication

In this section, we assume that task u_1 and u_2 are executed on different processors and they communicate via a shared memory, as depicted in Fig. 7.4. The processor on which task u_1 is executed, will sent posted write transactions to the memory in tile j. When the posted write transactions are accepted by the network interface,

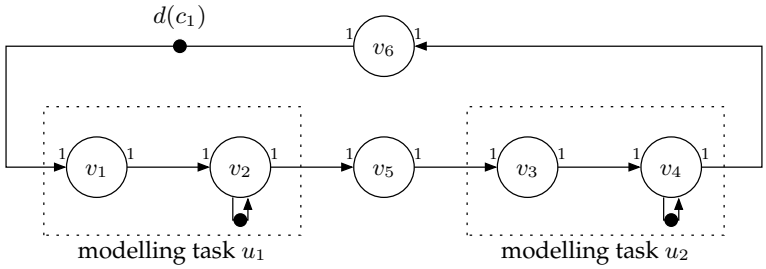

Figure 7.9: Dataflow model of a producer-consumer job from which both tasks are mapped to separate tiles.

the processor can already continue with executing its tasks. However, when the network interface has accepted the posted write transaction, the data and write pointer update are not yet arrived at the memory in tile j because it takes time for the network to deliver this data to the memory. A similar reasoning holds for task u_2. After updating the read pointer in the memory of tile i, it takes time for the network to deliver this data to the memory. These communication latencies have to be taken into account in the dataflow model in order to make the temporal behaviour of the dataflow model conservative with respect to the temporal behaviour of the implementation.

Fig. 7.9 shows a dataflow model of the producer-consumer job from which both tasks are mapped onto separate tiles and the communication between the tasks is realised via address-based communication over the network. Tasks u_1 and u_2 are executed on processors that make use of a run-time scheduling mechanism that falls in the class of a latency-rate server. Therefore, these tasks are both modelled with the dataflow model representation of a latency-rate server, as described in previous section. Compared to the previous section, tasks u_1 and u_2 are mapped to different tiles instead of one tile. When task u_1 has written its output data and updated the write pointer in the remote memory, the data and write pointer are written in the memory of tile j after a certain communication latency. This communication latency is modelled with actor v_5, where the execution time of this actor is equal to the communication latency between tile i and tile j. When task u_2 has read its input data from its local memory, it will update the read pointer in its local memory and in the memory of tile i. It takes a certain communication latency before the read pointer is written in the memory. This communication latency is modelled with actor v_6, where the execution time of this actor is equal to the communication latency between tile j and tile i. The data containers that are transferred between task u_1 and u_2 are stored in a circular buffer that is located in the memory of tile j. The buffer capacity is modelled with the initial tokens on edge (v_6, v_1) in Fig. 7.9. The number of initial tokens represents the number of empty data containers in the circular buffer. The initial tokens are located on this edge because task u_1 can derive the status of the circular buffer at the start of the job, by investigating the buffer's administration from which a copy is stored in its local memory.

Upper bounds on communication latencies can be computed, because of guaranteed-bandwidth allocations of network connections, guaranteed-bandwidth allocations

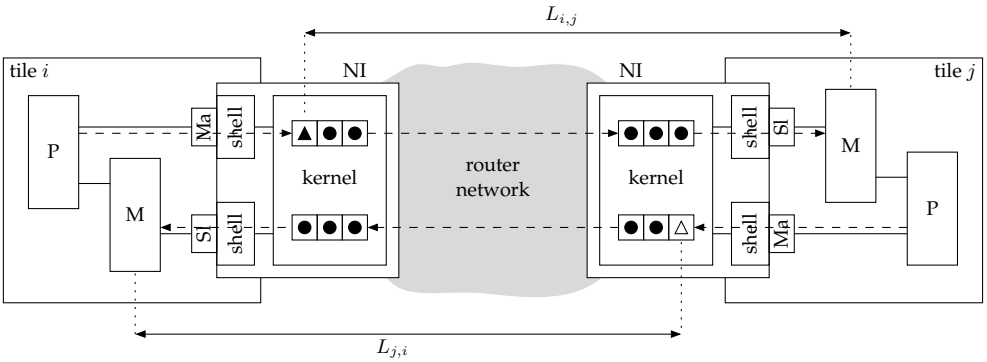

Figure 7.10: Communication latency for address-based communication.

for network-interface shells to access their memories, and bounded network-interface buffer capacities. The bounds can be computed for a given configuration of the network and memory-port arbiters. In computing the communication-latency upper bounds, we assume that multiple tasks are executed on a processor. Therefore, there can exist multiple communication channels that have to be implemented between two tiles. In address-based communication, these communication channels are implemented with the same set of network connections between these two tiles. Over the network connections, data containers, write pointers, and read pointers are transferred. Furthermore, the processors use run-time scheduling mechanisms to execute their tasks. Therefore, the execution order of tasks is unknown at design time. If the execution order is unknown, it can be, especially in case of preemptive scheduling, that one task stores output containers in the memory of another tile just before another task wants to store output containers in the same remote memory. Therefore, the network connection can be occupied, i.e. the network-interface buffers can be full when writing to a remote memory. Thus we assume that the network-interface buffers are already full when we derive an upper bound on the communication latency. Therefore, the temporal behaviour of the model is conservative with respect to the temporal behaviour of the implementation so that the upper bound holds for every possible use case. Of course more effort can be put in deriving a tighter upper bound, in case end-to-end latency is not tight enough. In case of meeting the job's throughput requirement, we can increase the buffer capacity of the circular buffer, as we will describe later.

When writing to a remote memory, the time until the network interface accepts a posted write transaction is taken into account in computing the processor stall cycles, as described in Section 7.2.1. Therefore, communication latency, for writing a read and write pointer update in a remote memory, is defined as follows:

Definition 14 (Pointer-update communication latency)**.** Pointer-update communication latency is defined as the time between the pointer update is accepted by the network-interface and it is written in the memory at the destination tile.

In the dataflow model of Fig. 7.9, the execution time of actor v_5 equals to the upper bound on the communication latency it takes for the write pointer to be written in the memory of tile j. The execution time of actor v_6 equals to the upper bound on the communication latency it takes for the read pointer to be written in the memory of tile i. These communication latencies are depicted by $L_{i,j}$ and $L_{j,i}$ in Fig. 7.10. Communication latency depends on the amount of posted-write transactions that are pending in a network connection, i.e. in the network-interface buffers or in the router network. Therefore, larger network-interface buffer capacities will increase the possibly pending number of posted-write transactions in the network connection, because multiple inter-task communication channels can make use of the same network connections. In computing an upper bound on the communication latency, we assume that the network-interface buffers are full with posted-write transactions, as described earlier. Therefore, larger network-interface buffer capacities will increase our upper bound on the communication latency. The higher communication latencies will increase the cycle mean on the cycle from actors v_1 through actor v_6 in the dataflow graph of Fig. 7.9. This increase of cycle mean can result in a larger maximum cycle mean of the dataflow graph. From Eq. (4.18) we know that a larger maximum cycle mean can be compensated by increasing the number of initial tokens on the edge (v_6, v_1). A higher number of initial tokens represents a larger capacity of the circular buffer in the memory of tile j. Therefore, we conclude the following: *If multiple communication channels are implemented with address-based communication and they make use of the same network connections, then the increase of the network-interface buffer capacities can result in larger required buffer capacities for the circular buffers.*

The dataflow-analysis technique [91] makes use of the property that a too conservative buffer requirement cannot result in a decrease of a job's minimum throughput. Notice that this holds for the buffer capacities of circular buffers and it does not hold for the network-interface buffers, in case of address based communication. The reason is that the network-interface buffers are used for transferring messages from multiple communication channels. This problem does not occur in case of address-less communication, which is used in part I of this thesis. Because, in address-less communication, data containers are stored in these network-interface buffers and these network-interface buffers are dedicated to only one communication channel.

7.5 Case study: MP3 playback

In this section, we introduce an industrial case study to show that we can model a job that is executed on a platform which makes use of a shared memory architecture and run-time scheduling between tasks.

An MP3-playback job contains, among others, an MP3-decoder and sample-rate converter task. The compressed-audio input of the MP3-decoder task is left out in this case-study, because the task consumes a variable number of tokens per execution, which is beyond the scope of this thesis. In [93] it is shown that tasks with a variable number of token consumption and production behaviour can be modelled in a Variable Rate DataFlow (VRDF) graph. This is shown in case a task is represented by one actor in a dataflow model.

The MP3-playback job makes use of an asynchronous sample-rate conversion, which

Task	MP3	SRC
Execution time $\check{\tau}(u_x)$	467899 cc	791 cc
Number of local memory accesses $\alpha(u_x)$	112898 cc	485 cc
Number of remote write accesses $\varrho(u_x)$	0	3

Table 7.1: Overview of the parameters of the MP3-decoder and sample-rate converter tasks.

converts the 48 kHz audio stream into a 44.1 kHz audio stream that matches with the digital-to-analog converter at the output. The MP3-decoder and sample-rate converter tasks are both executed on an EPICS processor, by making use of a run-time scheduling mechanism. The EPICS processor is executing with a clock frequency of 125 MHz. The MP3-decoder task executes in the main loop on this processor and the sample-rate converter task executes in an interrupt service routine. A timer periodically generates interrupts, so that the sample-rate converter task executes periodically. The execution time and number of local and remote memory accesses per execution of the MP3-decoder and sample rate converter tasks, are depicted in Table 7.1. For our case study, the execution times and number of local memory accesses are derived via cycle-accurate simulation. Therefore, the numbers in the table are optimistically-estimated upper bounds, which will be described in the next chapter.

First, we compute upper bounds on the number of processor stall cycles for executing the MP3-decoder and a sample-rate converter tasks. The bound of the MP3-decoder task is also compared with the measured stall cycles during cycle-accurate simulation. Subsequently, we show how we can derive the latency Θ and rate ρ parameters for the MP3-decoder task, since this task is scheduled using a run-time scheduling mechanism.

7.5.1 Upper bounds on processor stall cycles

The EPICS processor can suffer from stall cycles when accessing its local memory, because we make use of a shared local memory. Therefore, we derive upper bounds on processor stall cycles, when executing the MP3-decoder and sample-rate converter tasks.

First, we compute upper bounds on the number of processor stall cycles while executing the sample-rate converter task u_{SRC}. This task generates posted write transactions to store three words of data in the memory of another tile, two words of data for the stereo sample and one word of data for the write-pointer update, since we communicate via address-based communication as described in Section 7.1.

So the upper bound on the number of processor stall cycles, caused by the remote write accesses, is computed with Eq. 7.1 and equals:

$$\sigma_1(u_{\text{SRC}}) = (M-1) \cdot 3 \tag{7.7}$$

The processor can also suffer from stall cycles by accessing its shared local memory. The upper bound on the number of processor stall cycles $\sigma_2(u_{\text{SRC}})$, caused by accessing the local memory, is computed with Eq. (7.4), (7.5) and (7.6) for different values

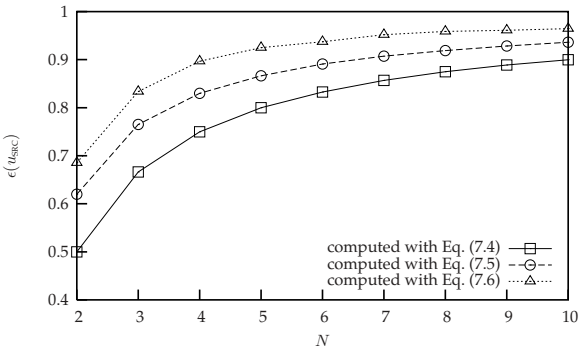

Figure 7.11: Estimated bounds of metric $\epsilon(u_{\text{SRC}})$ computed for the sample-rate converter task.

of N. The execution time $\check{\tau}(u_{\text{SRC}})$, the number of memory accesses $\alpha(u_{\text{SRC}})$, and the set of bursts B_{SRC} are all derived from profiling the sample-rate converter task u_{SRC} during cycle-accurate simulation.

To compare the upper bounds on the number of processor stall cycles ($\sigma_1(u_x) + \sigma_2(u_x)$), we define the metric $\epsilon(u_x)$ that makes the number of processor stall cycles relative to the execution time. Therefore, the reader will get a better feeling about the impact on performance by using the different levels of abstraction in computing $\sigma_2(u_x)$. For task u_x the metric $\epsilon(u_x)$ is defined as follows:

$$\epsilon(u_x) = \frac{\tau(u_x)}{\tau(u_x) + \sigma_1(u_x) + \sigma_2(u_x)} \ , \ 0 < \epsilon(u_x) \leq 1 \tag{7.8}$$

With the computed upper bounds on the number of processor stall cycles, we compute lower bounds on $\epsilon(u_{\text{SRC}})$ with Eq. (7.8). The lower bounds are plotted in Fig. 7.11 for different values of N and assuming that $M = 10$.

The tightness of abstraction level (ii) (i.e. Eq. (7.5)) is investigated with profiling the MP3-decoder task u_{MP3} during cycle-accurate simulation. The MP3-decoder task u_{MP3} is executed on an EPICS processor while another tile keeps sending posted write accesses to the local memory of the EPICS processor, so that the network-interface slave port keeps writing in this local memory. In the simulator, we measure the number of processor stall cycles for different values of N while executing the MP3-decoder task. From the computed and measured number of processor stall cycles, we compute the metric $\epsilon(u_{\text{MP3}})$ with Eq. 7.8 for each value N, which are depicted in Fig. 7.12. An indication for the accuracy of the computed bound is the difference between the computed and measured metric $\epsilon(u_{\text{MP3}})$. From Fig. 7.12, we conclude that this difference is less than 6 %. Furthermore, the measured number of processor stall cycles is already zero (i.e. $\epsilon(u_{\text{MP3}}) = 1$) given a service-cycle time N of seven clock cycles. Therefore, it seems that the bursts from the processor to the memory are at most six processor cycles for our input stream. Typically, sufficient memory bandwidth is available for the network-interface slave port to access the local memory. For example, the tasks u_{MP3} and u_{SRC} access the data memory, respectively, 24 % and 54 % of the time, as can be seen from the numbers in Table 7.1.

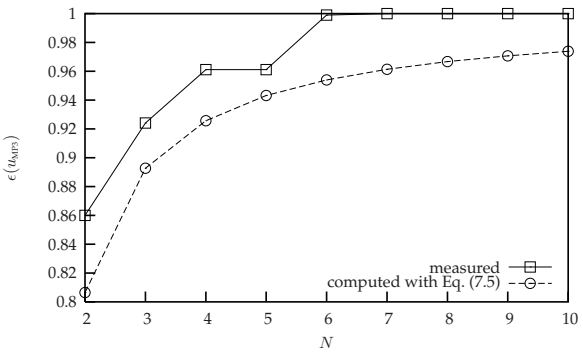

Figure 7.12: Comparison between a conservatively-estimated and measured metric $\epsilon(u_{\text{MP3}})$ for the MP3-decoder task.

7.5.2 Latency-rate server representation of the MP3-decoder

The next step is to derive a latency-rate server representation for the MP3-decoder task, which is executed on an EPICS processor using a run-time scheduling mechanism. The MP3-decoder task executes in the main loop on this processor and the sample-rate converter task executes in an interrupt service routine. The output stream of the MP3-decoder task has a sample rate of 48 kHz. The output stream of the sample-rate converter task has a sample rate of 44.1 kHz. The interrupt service routine will be triggered with the highest sample-rate frequency, which is 48 kHz. This results in an interrupt once every 20.8 ns. An upper bound on the interrupt time is the sum of the upper bound on execution time and processor stall cycles of the sample-rate converter task. Assuming that $M = 10$ and $N = 10$, the upper bound on the interrupt time is 875 cc, which is 7 ns on the EPICS processor running at 125 MHz. Therefore, the MP3-decoder task can occupy the processor at least 13.8 ns per period of 20.8 ns. Assuming that $M = 10$ and $N = 10$, the sum of upper bounds on execution time and processor stall cycles of the MP3-decoder task is 480454 cc or 3844 ns. Therefore, the maximum allocated rate parameter ρ_{MP3} can be computed as follows:

$$1/\rho_{\text{MP3}} = 7 \cdot \frac{3844}{13.8} + 3844 = 5794\,\text{ns} \tag{7.9}$$

This means that, in a busy period, the MP3 decoder produces at least one MP3 frame per 5794 ns.

Next to the parameter ρ_{MP3} we have to derive the parameter Θ_{MP3}. The latency parameter is derived from the worst-case response time of the MP3-decoder task. The response time is the time between the MP3-decoder task is enabled and the time it has finished its execution. Once the MP3-decoder task is enabled, the additional latency (to the 5794 ns) is at most 7 ns, which occurs if there is an interrupt service routine just after the task is enabled. Therefore, the parameter Θ_{MP3} equals 7 ns.

7.6 Concluding remarks

This chapter described a multiprocessor with a shared memory architecture that supports the use of address-based communication, so that processors can access their local memories and the memory in another tile. This architecture increases the flexibility, because the number of communication channels and buffer capacities can be adapted by changing the software.

A processor can suffer from stall cycles when accessing a shared memory (local or remote), but the number of stall cycles can be bounded. The conservatively-estimated upper bound on these stall cycles depends on the allocated bandwidth between the processor and the shared memory. In case of address-based communication, network connections can transfer data from multiple communication channels. But the communication latency can still be bounded, due to guaranteed bandwidth services and bounded network-interface buffer capacities. Data containers are stored in circular buffers that are located in the local memories of processors. Circular buffers support checking for the available number of full and empty containers in a buffer. This allows us to use run-time scheduling mechanisms, which enables us to execute tasks from multiple jobs on the same processor.

The uncertainty in temporal behaviour is increased at design time, because a processor can suffer from stall cycles, multiple communication channels can share network connections, and the tasks execution order is determined at run time. However, we can construct a dataflow graph to model a job that is mapped to a multiprocessor platform that makes use of a shared memory architecture and run-time scheduling of tasks. The in this chapter presented techniques are demonstrated by means of an MP3-playback job. The job's throughput and end-to-end latency constraints can be verified from the dataflow model by making use of existing dataflow-analysis techniques.

Compared to our first architecture, data containers can exceed the size of network-interface buffers and processors can execute tasks from multiple jobs. Therefore, the target domain is block-based processing, e.g. a MP3-decoder task that processes frames of data.

Chapter 8

Cache-based multiprocessor architecture

For infotainment-nucleus generation four, an external memory is required because the memory footprint of the jobs is considered too expensive to store in an on-chip memory. An external memory that is shared between processors is a bottleneck in current systems. Efficient use of the memory hierarchy is critical for achieving high performance in a multiprocessor system-on-chip.

In Section 8.1, we first describe our multiprocessor architecture that contains a shared external memory and a number of caches. This architecture increases the uncertainty in the temporal behaviour, e.g. due to unknown number of cache misses and unknown cache-miss penalties. Therefore, Section 8.2 investigates the use of optimistically-estimated bounds instead of conservatively-estimated bounds on execution times and processor stall cycles. A selection of cache-miss reduction techniques are described in Section 8.3. In section 8.4, the number of cache misses are trade-off against end-to-end latency and memory usage, by making use of a cache-aware mapping technique that is based on *execution scaling* [73]. The instruction and data locality are improved by executing a task multiple times before moving to the next task in a schedule. In Section 8.5, we apply this cache-aware mapping technique to our industrial application, which is a Digital-Radio-Mondiale receiver. Finally, we conclude this chapter in Section 8.6.

8.1 Multiprocessor architecture with external memory

External memory is required when the memory footprint of the software is too expensive to store in an on-chip memory. Therefore, our tile-based multiprocessor architecture is extended with an off-chip SDRAM, which we refer to as the external memory (M). The external memory is connected to a memory controller (CTRL), as depicted in Fig. 8.1. Multiple processors can access the memory, therefore, it is a shared external memory. A processor (P) can have a level-one cache ($), as depicted in Fig. 8.1. Once a copy of an external-memory location is stored in the cache, the

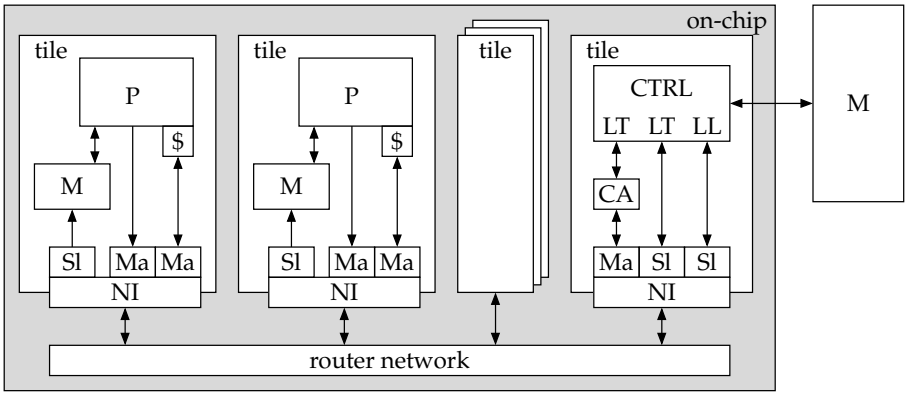

Figure 8.1: Multiprocessor architecture with external memory and caches.

processor has a low latency in accessing this memory location. If there is no copy of this memory location present in the cache, a new cache block will be fetched from the external memory so that a copy of the memory location is stored in the cache. The processor suffers from stall cycles until the cache block is fetched from the external memory and it is stored in the level one cache. This fetching of a new cache line is referred to as a cache miss.

The memory-access latency, to the shared external memory, is larger compared to the on-chip local memories. This is, among others, a result of arbitration at the external-memory port. For example, when two or more processors want to access the memory at the same time, an arbiter in the memory controller will grant access to one processor. The accesses from the other processors will be postponed until the current processor finishes its access. In our architecture, the SDRAM controller distinguishes two classes of memory accesses: *low latency* (LL) and *latency tolerant* (LT) [1]. The advantage of splitting latency-tolerant from low-latency memory accesses is more scheduling freedom. The latency for the class low-latency accesses can be reduced by postponing the latency-tolerant accesses. Fetching new cache blocks falls in the class of low-latency accesses because its latency has a direct impact on the number of processor stall cycles. Pre-fetching data containers from the external to the local memory belongs to the class of latency-tolerant accesses, because some extra latency has little or no affect on the system performance.

The processor tiles have also local memories (M) which can be accessed directly by the processors, we refer to these memories as uncached memories. Inter-task communication between two processors is either with address-less communication via dedicated network connections (as described in Chapter 3.2) or with address-based communication via circular buffers stored in an uncached local memory (as described in Chapter 7.1). The buffers are stored in uncached memory regions because streaming input data is only read once and streaming output data is only produced once. If these buffers would be stored in cached memory regions, the accesses to these buffers would always result in cache misses.

In case data containers are too large to store in local memory, then the buffer must

be stored in the external memory. The circular buffer and its administration can be stored in a cached memory region if *streaming consistency* [12] is applied to ensure cache coherency and memory consistency. This approach requires caches to have means to invalidate and flush cache blocks in an address range. Such functionality is present in, for example, ARM11 and TriMedia [12].

When a buffer does not fit in a local memory, but the individual containers do fit, then this buffer can be distributed between the local and external memory. In this case, the buffer is composed of two circular buffers, one (small) buffer that is located in uncached local memory, and one larger buffer that is located in an uncached memory region of the external memory. The producing processor will generate posted write transactions to the circular buffer in the external memory. In the SDRAM-controller tile there is a communication assist [18], which is an autonomous DMA controller that is introduced to prefetch data containers from the circular buffer in the external memory to the circular memory in the local memory. Finally, the consuming processor can read the data containers from the circular buffer that is located in its local memory. This inter-processor communication does not have the disadvantage of cache misses and it does allow storing of large number of containers in the external memory.

Key for this architecture is that we do not communicate the input and output data via the cache (preventing cache misses), that we use latency-tolerant memory accesses, and that we can distribute a FIFO buffer between local and external memory.

8.1.1 Inter-tile communication via external memory

A general producer-consumer example is used to explain the implementation of streaming communication via a buffer that is distributed between local and external memory. Task u_1 is executed on the processor in tile i and task u_2 is executed on the processor in tile j, as depicted in Fig. 8.2. The program code and state of task u_1 and u_2 are stored in a cached memory region of the external memory. Task u_1 produces containers of data that are stored in the circular buffer (FIFO1) that is located in an uncached memory region of the external memory. The communication assist transfers these containers to the circular buffer (FIFO2) that is located in the memory of tile j, so that task u_2 has low access latency when consuming the data containers.

The implementation of streaming communication between task u_1 and u_2 will be described in the following steps. These steps are also depicted in Fig 8.2. The container communications are depicted with the thick lines in this figure.

1. The processor, on which task u_1 is executed, reads the FIFO1's administration values from its local memory (i.e. read and write pointers) to see if there is space available to store a data container.

2. In case there is space available, task u_1 produces output data and the processor stores it in FIFO1 by generating posted write transactions to the external memory. These posted write transactions are latency-tolerant external-memory accesses.

3. After the posted write transactions are accepted by the network interface in

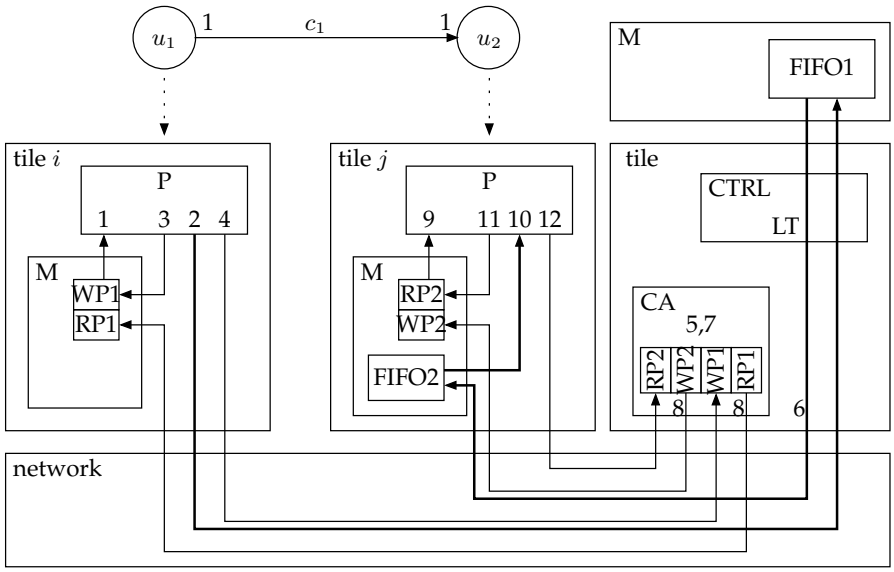

Figure 8.2: Implementation of streaming communication via a buffer that is distributed between local and external memory.

tile i, the processor updates the write pointer in the buffer administration that is located in its local memory.

4. Finally, the processor in tile i updates the write pointer in the buffer administration that is located at the communication assist. This is done by generating a posted write transaction to the memory-controller tile.

5. The communication assist reads the administration values from FIFO1 and FIFO2 to see if there is a data container available in FIFO1 and space available in FIFO2.

6. In case a container can be transferred from FIFO1 to FIFO2, the communication assist reads the data container from the external memory and stores it in FIFO2 by generating posted write transactions to tile j. The external-memory accesses of the communication assist are latency-tolerant memory accesses, because communication-assist stall cycles are not costly.

7. After the communication assist has finished transferring the data, the communication assist updates the read pointer of FIFO1 and the write pointer of FIFO2 from the buffer administrations that are located in the memory-controller tile.

8. Finally, the communication assist updates the read pointer in the buffer administration that is located in the memory of tile i and the write pointer in the buffer administration that is located in the memory of tile j. This is done by generating a posted write transactions to this memory.

9. The processor, on which task u_2 is executed, reads FIFO2's administration values from its local memory (i.e. read and write pointers) to see if there is a data

container available in FIFO2.

10. In case there is a data container available, the processor in tile j reads the data container from its local memory.

11. After the processor has finished reading the data, the processor in tile j updates the read pointer in the buffer administration that is located in its local memory.

12. Finally, the processor in tile j updates the read pointer in the buffer administration that is located in the memory-controller tile. This is done by generating a posted write transaction to the memory-controller tile.

8.2 Optimistically-estimated versus conservatively-estimated bounds

The execution time of a task is defined by Definition 5 in Chapter 2.1. This execution time does not include processor stall cycles due to cache misses. In general, upper bounds on the execution times and upper bounds on processor stall cycles are needed to show that real-time requirements are satisfied.

It is not always possible to obtain upper bounds on execution times of tasks [95]. This is only possible if we use a restricted form of programming, which guarantees that tasks always terminate, i.e. iteration is only allowed if iteration counts of loops are explicitly bounded. A reliable conservatively-estimated execution-time bound of a task can potentially be given if the worst-case input for a task is known. Unfortunately, often the worst-case input is not known or hard to derive. Furthermore, if a worst-case input could be extracted, it is still unclear how often this input would occur in practice.

The literature on timing analysis is not always clear on making a distinction between worst-case execution times and estimates for them [95]. Figure 8.3 depicts the number of occurrences of an execution time of task u_i as function of the execution time, for the set *observed execution times* and *all execution times*. A task typically shows a certain variation of its execution time depending on the input data, e.g. due to input data dependent conditional branches and loops. The distribution of times for set *all execution times* is shown as the upper curve in Fig. 8.3. The longest execution time is called the worst-case execution time $\tau(u_i)$. The execution time depends on the path that is taken at conditional branches in the task's program code. Typically, there is a large number of possible paths and the number of paths can even depend on the input data. Therefore, it can be problematic to exhaustively explore all possible execution paths and in consequence of that it is hard to determine the exact worst-case execution time. Furthermore, it can also be difficult to derive a conservatively-estimated execution-time bound $\hat{\tau}(u_i)$ that is tight. Today, in many parts of the industry, the common method to estimate execution-time bounds is to measure the execution time for a subset of the possible input stimuli [95]. This determines the set *observed execution times* from which an optimistically-estimated upper bound $\check{\tau}(u_i)$ is derived, as shown in Fig. 8.3. This optimistic estimate will, in general, underestimate the worst-case execution time.

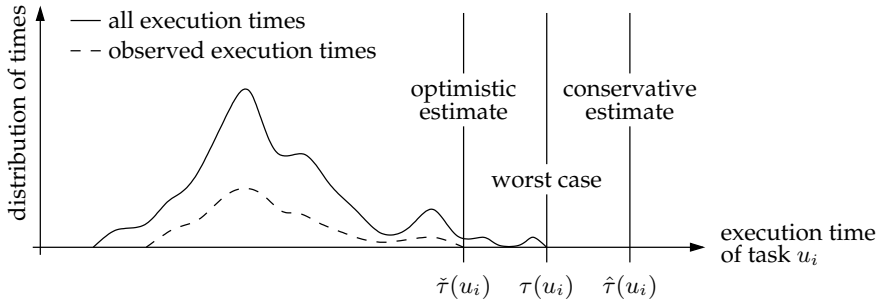

Figure 8.3: The definition of worst-case execution time and an optimistically-estimated and conservatively-estimated upper bound on the worst-case execution time.

It is also difficult to derive tight conservatively-estimated upper bounds on the number of processor stall cycles. Processors are stalled in case of a cache miss and continue to execute after a new cache block is fetched from the external memory. In order to verify that the real-time requirements are met, we need to derive an upper bound on the number of processor stall cycles during the execution of a task. The number of processor stall cycles is determined by the number of cache misses and the cache miss penalty [42]. A cache miss penalty is hard to predict because it depends on the communication latency in the network, arbitration at the external memory port, and the access time to the external memory. The communication latency depends on the allocated bandwidth in the network, the state of the slot table, and the amount of data that has to be communicated. The contention at the external memory port depends on the number of requests, the burst size of each request, and the distances between individual requests. The memory access time depends on the number of cache blocks that need to be read and written. In case of a data cache miss, a cache block can be modified which means that the current cache block has to be written back to the external memory before receiving the new cache block. In case of accessing double data words it is possible that two (instead of one) cache blocks have to be fetched, i.e. if a data word is split over two consecutive cache blocks. Therefore, it is hard to derive a tight conservatively-estimated upper bound on the number of processor stall cycles. Furthermore, there can be a large difference between the typical number of processor stall cycles and a conservatively-estimated upper bound on them. Therefore, in the industry, often optimistically-estimated upper bounds on the processor stall cycles are used in a multiprocessor system with external memory and caches.

With the use of optimistically-estimated upper bounds on execution times and processor stall cycles, we are not able to guarantee that real-time requirements are met for all input stimuli. Therefore, optimistically-estimated bounds are not safe to be used for the design of hard real-time constraint systems. However, for a specific set of input stimuli, we are able to derive an optimistically-estimated upper-bound on execution times and an optimistically-estimated number of cache misses. When the external memory controller is able to guarantee upper bounds on the external memory accesses, then we are able to guarantee that real-time requirements are met for

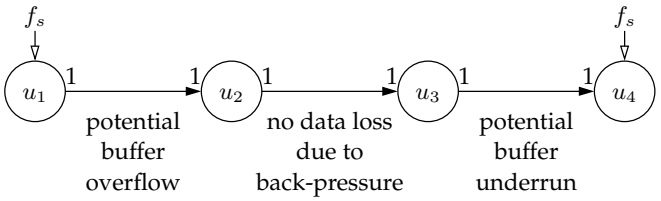

Figure 8.4: Optimistically-estimated execution-time bounds can lead to overflow at the input buffer and underrun at the output buffer.

our set of input stimuli. In the implementation the task should be allowed to finish its execution even when the execution time exceeds the optimistically-estimated execution-time bound. When other tasks have a lower execution time than expected or when input and output buffers still have space and data available, respectively, then hopefully no deadlines will be missed. Once our set of input stimuli is representative, we are confident that the real-time constraints are not violated too often. However, a deadline miss can occur for other input stimuli. Such a deadline miss will not cause serious damage to the environment and will not jeopardise correct system behaviour, because our jobs have firm real-time requirements instead of hard real-time requirements. However, a fall-back mechanism is required which must be activated in case a deadline miss does occur. An example of a fall-back mechanism is to reuse a previous audio sample or display a previous video frame. Once the deadline misses are sporadic, typically, they are hardly noticeable by the user.

A task only consumes a data container from its input buffer, in case a full container is available in this buffer. Furthermore, a task only produces a data container to its output buffer, if there is an empty container available in this buffer. The fact that a task waits until sufficient data or space is available leads to an efficient mechanism to prevent buffer overflow. We refer to this mechanism as back-pressure. The use of back-pressure ensures that potentially non-conservative execution-time estimates result in either overflow at the input buffer or underrun at the output buffer, as depicted in Fig. 8.4. No data containers will be lost by communicating data over internal buffers, because such loss of data is difficult to handle by tasks. Note that in case of a composable architecture, a deadline miss of one job will have no affect on the performance of another job [37].

8.3 Cache-miss reduction techniques

As the gap between processor and memory performance is increasing [42], efficient use of the memory hierarchy is critical for achieving a high performance. Processors are stalled in case of a cache miss and continue to execute after a new cache block is fetched from the external memory.

The increasing number of processors and the increasing contention at the external-memory port contribute to an increase of the cache-miss penalty. A reduced number of cache misses can compensate for an increased miss penalty. Furthermore, it de-

creases the average number of low-latency memory accesses and thereby reduces the contention at the external memory and indirectly reduces the cache-miss penalty for other processors in the multiprocessor system. The reduction of cache misses and cache-miss penalty reduces the average number of processor stall cycles and increases the system performance. Minimising the number of cache misses is seen as an intermediate approach, because there is still low-latency communication between the external memory and cache. Future work is explicit pre-fetching the task's program code and working data set into a local memory, instead of pre-fetching with a cache, so that the external-memory communication becomes more latency tolerant. In the ideal situation, tasks are pre-fetched in a local memory and then they are executed from this memory. It is shown in [89] that the use of a local memory can have a positive impact on the estimated upper bound on the worst-case execution time, compared to the use of a cached memory.

There is a large body of literature on reducing the number of cache misses. First of all the cache parameters (e.g. cache line size, cache size, and associativity) have an impact of the number of cache misses [42]. Next, there are many compiler optimisation techniques for reducing the instruction and data cache misses [62]. The compiler can reduce the number of instruction cache misses by placing functions near to their callers in memory (assuming routines and callers are temporally close to each other), and by removing infrequently executed code (such as error handling) out of the main body of the code and straightening the code, so that in general, a higher fraction of the instructions fetched into the instruction cache are actually executed. For programs that manipulate large arrays of data, the number of data cache misses can be reduced by loop transformations. Examples of loop transformations are interchanging two nested loops, reversing the order in which loop's iterations are performed, and fusing two loop bodies together into one. Cache-miss reduction comes from a better use of the memory hierarchy. In this thesis, we apply a cache-optimisation technique *execution scaling* [73], which is a transformation that improves the use of the memory hierarchy by executing each task multiple times before moving to the next task in a schedule. Execution scaling is related to loop transformations that concentrate on optimising the use of data caches, but execution scaling is focussed on transforming the scheduling of tasks (main loop), whereas conventional compiler loop transformations are quite locally applied. That means, a compiler typically does not change the order in which tasks are executed, e.g. due to the dependencies between tasks.

In the context of Synchronous Data-Flow (SDF) graphs, which is a subset of CSDF graphs [9], there is a large body of literature on scheduling these graphs to optimise various metrics. The number of context-switches is minimised in [69]. First, they use a single appearance schedule in which each task appears once and is activated a minimum number of times. Second, they scale this schedule with constraints on end-to-end latency and memory usage. The focus is a single processor with local memory and the goal is to reduce context-switching overhead cost and maximise the degree of vector processing opportunity. The number of cache misses are minimised in [48, 73] in the context of a single processor. They store the input and output buffers in a cached memory, creating the problem that the input and output data eventually overflow the data cache, when task executions are scaled extensively.

This chapter focus on mapping of jobs onto a multiprocessor architecture instead

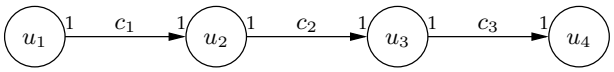

Figure 8.5: Streaming job.

of a single processor. The input and output buffers are stored in a memory region that is not cached, in contrast with [48, 73]. Therefore, input and output data cannot overflow the data cache, and execution scaling is only limited by end-to-end latency and memory-usage constraints. FIFO buffers can be distributed between the local and external memory allowing us to create large buffer capacities. Furthermore, we present a dataflow model from a job that is mapped onto a multiprocessor system with a certain execution scaling factor m. From this model, we compute the end-to-end latency and memory usage by making use of existing dataflow-analysis techniques.

8.4 Cache-aware mapping of streaming jobs

Often, streaming jobs are rich with parallelism and regular communication patterns, which can be exploited in our multiprocessor system. In this way tasks can be freely combined and split to improve the system behaviour. For example, two small tasks can be merged into one to reduce task's switching overhead cost, or one big task can be split into two to enable more task's level parallelism. Obviously, splitting a task into two can potentially lead to cyclic dependencies that can limit task level parallelism.

For the class of streaming jobs, we apply the cache-optimisation technique *execution scaling*, which is a transformation that improves instruction and data locality by executing each task multiple times before moving to the next task in a schedule. This cache-aware optimisation technique is based on [73], but we target a multiprocessor architecture instead of a single processor and use uncached local memories to store the input and output data of a task. This allows us to scale the execution extensively and still reduce data cache misses.

We map a general job onto a simplified multiprocessor to illustrate the trade-off between mapping of tasks to processors and the maximum allowed number of successive task executions. Mapping consists of binding tasks to processors and scheduling tasks on a processor.

The general application is depicted in Fig. 8.5 and it has a minimum throughput constraint of $1/2T$. The tasks u_1 through u_4 communicate via communication channel c_1 through c_3. The tasks are executed on two identical processors $p1$ and $p2$. The worst-case execution time of each task is T time units. On each processor p, we execute two tasks in a static-order schedule S_p. The static-order schedule $S_p^m = (u_i^m, u_j^m)$ represents m executions of task u_i followed by m executions of task u_j. Our goal is to find the mapping with the maximum execution scaling factor m that satisfies end-to-end latency and memory constraints.

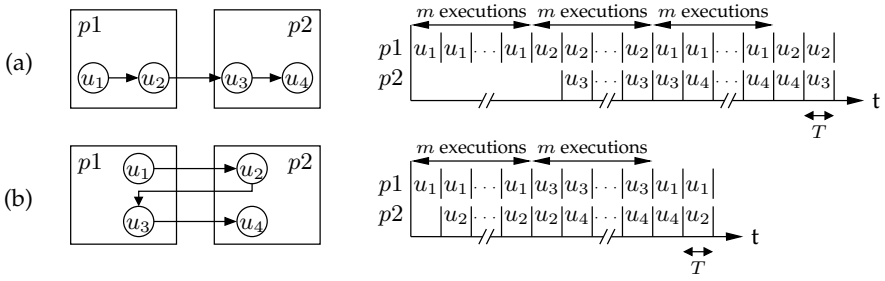

Figure 8.6: Two mapping options (a) and (b).

In Fig. 8.6, we show two mapping options that satisfy the minimum throughput constraint $1/2T$. In mapping option (a), we execute tasks u_1 and u_2 on processor $p1$ and we execute tasks u_3 and u_4 on processor $p2$. This mapping option requires FIFO buffer capacities of m, 2, and m data containers for communication channels c_1, c_2, and c_3, respectively. The end-to-end latency (from task u_1 until u_4) is equal to $(2m + 2) \cdot T$ time units. In mapping option (b), we execute tasks u_1 and u_3 on processor $p1$ and we execute tasks u_2 and u_4 on processor $p2$. This mapping option requires FIFO buffer capacities of 2, m, and 2 data containers for communication channels c_1, c_2, and c_3, respectively. The end-to-end latency is equal to $(m + 2) \cdot T$ time units.

In mapping option (a), the end-to-end latency and FIFO buffer capacities grow with a factor $2m$ whereas in mapping option (b) these grow with a factor m. Therefore, we conclude that mapping option (b) allows a higher value of m for the same end-to-end latency and memory constraint. Notice that mapping option (b) requires more inter-tile communication channels than mapping option (a) (three instead of one), but these inter-tile communication channels belong to the class of latency-tolerant communication whereas cache refills belong to the class of low-latency communication. A higher value of m results in fewer cache misses and less low-latency memory accesses. Therefore, the amount of low-latency external-memory accesses is lower in mapping option (b) than in mapping option (a). Since the external memory is typically a bottleneck in current and future platforms, mapping option (b) is a better mapping option than mapping option (a).

This example shows that the mapping of tasks to processors influences the maximum execution scaling factor m. We need tools to compute buffer capacities and end-to-end latencies for exploring different mapping options with different execution scaling factors.

8.4.1 Execution scaling

If a task is executed in a loop repeatedly, then, ideally the first iteration brings its code into the cache and subsequent iterations execute from the cache, rather than requiring it to be reloaded from memory each execution. Therefore, the first iteration may incur overhead for fetching the instructions into the cache, but subsequent iterations generally need not. Similarly, if a block of data is used repeatedly, it is

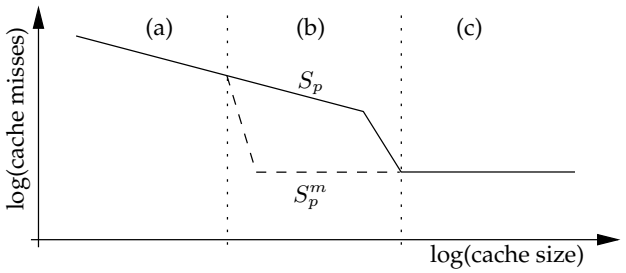

Figure 8.7: Cache misses as function of the cache size.

ideally fetched into the cache and accessed from there, again incurring the overhead of reading it from main memory only on its first use.

Disadvantages of execution scaling are an increase of end-to-end latency and an increase of FIFO buffer capacities. The end-to-end latency increases because we execute a task multiple times before moving to the next task, therefore, it takes more time before the data is rippled through the job's task graph. This problem is not often severe, as many streaming jobs can tolerate additional latency. The FIFO buffer capacity increases, because when executing a task multiple times, we need sufficient capacity to store the data communicated between the tasks. However, large FIFO buffers can be distributed between the local and external memory and data can be pre-fetched by a communication assist.

To explain execution scaling, we use the following terminology. Let U_p be a set of tasks executed on a processor p and S_p a static-order schedule with length N. The schedule is denoted by $S_p = (s_1, s_2, ..., s_N)$ with $s_i \in U_p$ and $1 \leq i \leq N$. Scaling the execution with factor m means that each task s_i is executed m times before moving to the next task in the static-order schedule. We refer to the new schedule by $S_p^m = (s_1^m, s_2^m, ..., s_N^m)$.

When the cache size is small compared to the size of the set of tasks U_p and the cache size q increases, then the number of cache misses decreases with $\sqrt{q_0/q}$, where q_0 is application dependent. If the cache size exceeds the size of the set of tasks U_p, then only compulsory misses [42] (cold start misses) remain, because in our architecture the input and output buffers are stored in the local memory that is not cached. When executing schedule S_p, the number of cache misses follow the line in Fig. 8.7 [27]. The compulsory misses are depicted by the flat line in Fig 8.7. The number of cache misses can be reduced by executing task s_i multiple times before moving to the next task $s_{(i+1)\%N}$ in the schedule S_p, for $1 \leq i \leq N$. After executing schedule S_p^m, the number of cache misses follow the dashed line in Fig. 8.7.

The impact of execution scaling on the number of cache misses for the cache size ranges (a), (b), and (c), in Fig. 8.7, are the following: (a) There is hardly any impact on the number of cache misses, none of the tasks $u_i \in U_p$ fit in the cache. (b) This has the largest impact on the number of cache misses. Individual tasks $u_i \in U_p$ fit in the cache while the set of tasks U_p does not fit. During the first execution of a task we see compulsory misses, because the task is being discarded from the cache when executing the other tasks in the schedule. During the following $m - 1$ executions,

generally the task can execute from the cache, because the program code and state data are already present in the cache. The average number of cache misses reduces when increasing the scaling factor m. (c) There is no impact on the number of cache misses. For both schedules S_p and S_p^m only compulsory misses remain, because the individual tasks $u_i \in U_p$ as well as the set of tasks U_p fit in the cache.

The more tasks that are executed on the processor (i.e. the larger set of tasks U_p), the larger the size of range (b). For example, if two tasks with the same size are executed on a processor, then for schedule S_p^m the flat line in Fig. 8.7 starts at half the cache size compared to schedule S_p. When four tasks with the same size are executed on the processor, then for schedule S_p^m the flat line starts at $1/4$ of the cache size compared to schedule S_p.

There are limitations to what extend we can increase the execution scaling factor m. First, it is limited by the constraints on end-to-end latency and memory usage. Second, if two tasks u_k and u_l are executed on one processor and there is a feedback loop (cycle in the task graph) between these tasks, then the maximum value of m is limited because of the cyclic dependency between tasks u_k and u_l. The latter can be solved by executing tasks u_k and u_l on separate processors, but the tasks have to wait for each other due to the cyclic dependency, affecting the processor performance.

The cache-miss model that is described in this section, holds for instruction and data cache misses, because in our architecture the input and output buffers are stored in uncached memory regions so that input and output data cannot overflow the data cache.

8.4.2 Computation of the execution scaling factor

In this section, we describe a flow for mapping a job onto a multiprocessor architecture. We assume that each task $u_i \in U_p$ fits in the instruction and data cache and that the set of tasks U_p does not fit. For this case we can apply execution scaling to minimise the number of cache misses, as described in Section 8.4.1. The job's throughput and its dataflow-graph representation are an input to our flow. Furthermore, we need the end-to-end latency and memory constraints as an input. Dataflow modelling and analysis techniques are used to derive the maximum execution scaling factor m.

The design flow, which maximises the execution scaling factor m, is depicted in Fig. 8.8. First, we bind the tasks of a job to the processors in our multiprocessor architecture. Second, we construct a static-order schedule S_p for each processor p. These schedules are used as initial schedules. In the first iteration the execution scaling factor m is initialised to one, which represents the original schedule without execution scaling. For the computed binding and static-order schedule S_p^m we can construct a CSDF graph, as described in Section 4.3.2. For computing buffer capacities and end-to-end latencies, we use the existing dataflow-analysis techniques [79, 91] that are described in Section 4.4. The required buffer capacities can be computed from the constructed dataflow model in combination with the given throughput constraint. The end-to-end latency is computed after computing the buffer capacities. We repeat this procedure for different execution scaling factors m until we find the maximum value of m that satisfies the end-to-end latency and memory constraints.

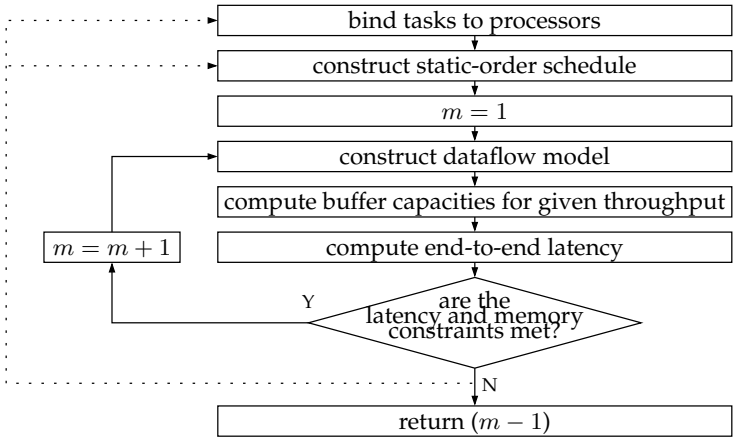

Figure 8.8: Flow to calculate the maximum execution scaling factor m.

Future extensions of the design flow are backtracking for different initial schedules and different bindings of tasks to processors. Furthermore, in this thesis we use only one execution scaling factor m for all processors. The exploration with different scaling factors for different processors and a more efficient heuristic for increasing factor m is also left for future work.

8.4.3 Example of execution scaling in a dataflow model

A job that is mapped onto a predictable multiprocessor and from which the tasks are executed in static-order schedules, can be modelled in a CSDF model, as described in Section 4.3.2. After applying the execution-scaling technique, the static-order schedules are scaled with scaling factor m. The result is still a static-order schedule, therefore, the job can still be modelled into a CSDF graph. The final model will be used in a design flow to minimise the number of cache misses by maximising the execution scaling factor m and still satisfy the end-to-end latency and memory constraints.

We take the job in Fig. 8.5 as an example to model execution scaling in a CSDF graph. We assume the binding and static-order schedule as defined by mapping option (b) in Fig. 8.6. Furthermore, we assume an execution scaling factor $m = 3$ and a task switching overhead cost C. Figure 8.9 shows the CSDF graph in which the schedules $S^3_{p1} = (u^3_1, u^3_3)$ and $S^3_{p2} = (u^3_2, u^3_4)$ are modelled. The number of phases of the actors v_1 through v_4 equal $\text{lcm}(1, 3) = 3$, as described in Section 4.3.2. The cyclo-static execution times of actor v_1 through v_4 are $\langle T + C, T, T\rangle$. The communication channels c_1 through c_3 are modelled with the forward and backward edges between the actors, which we refer to as the set E_b. The actors that model the communication latencies, are omitted for simplicity reasons. The numbers beside the black dots indicate the number of initial tokens that model the FIFO buffer capacities in the number of containers. The input and output rates on every edge $e_b \in E_b$ equal $\langle 1, 1, 1\rangle$. On the remaining edges E_s, which represent the scheduling dependencies, the input rates $\gamma(e_s)$ are $\langle 1, 0, 0\rangle$ and the output rates $\pi(e_s)$ are $\langle 0, 0, 1\rangle$. The initial tokens on

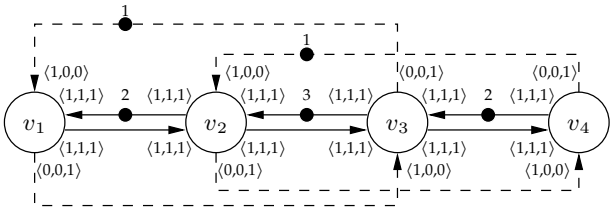

Figure 8.9: CSDF graph modelling the job in Fig. 8.5 with mapping option (b) in Fig. 8.6 and execution scaling factor $m = 3$.

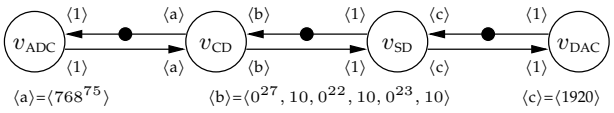

Figure 8.10: CSDF model of the digital radio receiver.

the edges (v_3, v_1) and (v_4, v_2) make sure that the actors v_1 and v_2 start firing, because tasks u_1 and u_2 are the first tasks to execute in the schedule S^3_{p1} and S^3_{p2}.

8.5 Case study: Digital-Radio-Mondiale receiver

In this section, we apply the cache miss reduction technique to our Digital-Radio-Mondiale [78] receiver from infotainment-nucleus generation four. We measure the impact of execution scaling on the number of cache misses for different cache sizes and for different values of execution scaling factor m. Finally, we compute, by means of our CSDF model, the maximum value m that still meets our end-to-end latency and memory constraints.

The Digital-Radio-Mondiale job is first partitioned into a number of tasks. The CSDF graph that represents our Digital-Radio-Mondiale job before mapping, is depicted in Fig. 8.10. The graph consists of four actors that model an Analog-to-Digital Converter (v_{ADC}), Channel Decoder (v_{CD}), Source Decoder (v_{SD}), and Digital-to-Analog Converter (v_{DAC}). The analog-to-digital and digital-to-analog converters are implemented as separate tiles in our multiprocessor system. The tasks u_{CD} and u_{SD} are executed on a TM2270 which belongs to the TriMedia family [67]. We refer to this processor as the Digital Signal Processor (DSP). An external memory is applied because the memory footprints of task u_{CD} and u_{SD} are considered too expensive to store in an on-chip memory. During our measurements, the static-order schedule on the DSP processor is $S^m_{\text{DSP}} = (u^m_{\text{CD}}, u^m_{\text{CD}}, u^m_{\text{CD}}, u^m_{\text{SD}}, u^m_{\text{CD}}, u^m_{\text{CD}}, u^m_{\text{SD}})$. We used the preamble $P_{\text{DSP}} = (u^{28}_{\text{CD}})$ before executing schedule S^m_{DSP}, so that there are ten containers on the communication channel from task u_{CD} to task u_{SD} and task u_{SD} is able to execute. This preamble is a result of the production rate $< 0^{27}, 10, 0^{22}, 10, 0^{23}, 10 >$ on the edge $(v_{\text{CD}}, v_{\text{SD}})$.

The multiprocessor platform that is described in Section 8.1, is built in a SystemC [44]

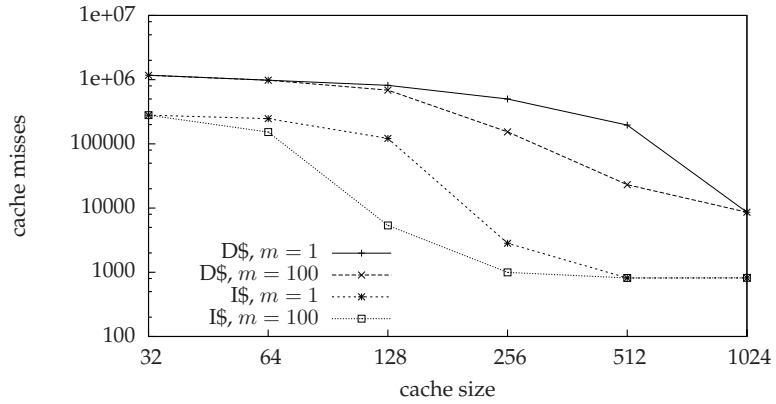

Figure 8.11: Instruction (I$) and data (D$) cache misses.

simulation environment. Tiles are modelled using cycle-accurately [21] models. The network makes use of flit-accurate models, since the network packets have a granularity of a flit (three words). The TriMedia tile makes use of an *instruction set simulator* to model the DSP. The size and the initial content of memories, cache sizes, and cache parameters for the instruction and data cache are configured at the start of the simulation. The presented number of cache misses and execution times are measured in this SystemC simulation environment.

For different instruction and data cache sizes, we measure the number of cache misses for an execution scaling factor $m = 1$ and $m = 100$, as shown in Fig. 8.11. Schedule S^1_{DSP} represents the original code without execution scaling and schedule S^{100}_{DSP} represents a mapping where the execution scaling of tasks is extensively. The number of cache misses is measured during hundred executions of schedule S^1_{DSP} and one execution of the schedule S^{100}_{DSP} so that the total number of task executions is equal in both cases. The cache misses in Fig. 8.11 follow the same pattern as the cache misses in Fig. 8.7. The impact of execution scaling on the number of cache misses is the largest for an instruction and data cache size of 128 KByte and 512 KByte, respectively. For these cache sizes, the numbers of cache misses are reduced by a factor 22.7 and 8.5 for the instruction and data cache, respectively. For smaller cache sizes the program code and private data of the individual tasks does not fit in the cache, hence execution scaling has a small or no impact on the number of cache misses. When the instruction and data cache sizes grow, both tasks u_{CD} and u_{SD} fit in the cache, therefore, the impact of execution scaling on the number of cache misses reduces again.

For an instruction and data cache size equal to 128 KByte and 512 KByte, respectively, we measured the impact of the execution scaling factor m on the number of cache misses, as shown by the plot in Fig. 8.12. The number of cache misses were measured during 500 executions of task u_{CD} and 200 executions of task u_{SD}. From this log-log plot we conclude that the number of instruction and data cache misses reduce when increasing the execution scaling factor m. Ideally, the measured points form a straight line in the log-log plot, because in the first iteration we observe cache misses

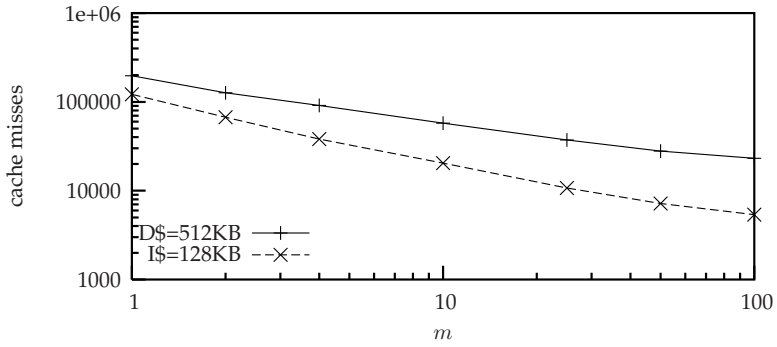

Figure 8.12: Cache misses versus scaling factor m.

but subsequent iterations we generally do not.

Finally, we compute the maximum value m that still meets our end-to-end latency and memory constraints. The throughput of our receiver is determined by the analog-to-digital and digital-to-analog converters, which have a sample rate of 48 kHz. Furthermore, the end-to-end latency should be within reasonable bounds. In this thesis we assume a maximum end-to-end latency of one second. The memory usage is not critical because the FIFO buffers can be distributed between the local and external memory. The end-to-end latency is defined as the difference between finishing the first execution of task u_{DAC} and starting the first execution of task u_{ADC}. An optimistically-estimated upper bound on the execution time plus processor stall cycles of task u_{CD} is $\langle 2896^{27}, 14071, 2896^{22}, 14071, 2896^{23}, 14071\rangle$ microseconds and an optimistically-estimated upper bound on the task switching cost C_{CD} is 631 microseconds. An optimistically-estimated upper bound on the execution time plus processor stall cycles of task u_{SD} is $\langle 2202\rangle$ microseconds and an optimistically-estimated upper bound on the task switching cost C_{SD} is 595 microseconds. These estimates are based on the DSP processor with a clock frequency of 300MHz, instruction cache of 128 KByte, data cache of 512 KByte, and assumed cache miss penalties of 100 and 150 DSP clock cycles for an instruction and data cache miss, respectively. The execution times of tasks u_{ADC} and u_{DAC} are equal to $1/48\,\text{kHz}$. With the design flow that is described in Section 8.4, we compute the buffer capacities and end-to-end latency for different execution scaling factors m. The end-to-end latencies are shown in Table 8.1. The presented latencies include the latency of the preamble P_{DSP}, which is 0.444 s. From Table 8.1, we conclude that execution scaling factor $m = 11$ still meets the end-to-end latency constraint. The impact of execution scaling factor $m = 11$ on the number of instruction and data cache misses is 6.4 and 3.4, respectively. The impact on the total number of cache misses (instruction plus data) is a factor 4.2.

The tools SDF^3 and Hebe, which are described in Section 4.4, compute the buffer capacities for the given throughput requirement of 48 kHz at the analog-to-digital and digital-to-analog converter. The tool SDF^3 uses an exact technique in computing buffer capacities, which is based on explicit state-base exploration. The tool Hebe uses an approximation technique in which sufficiently large buffer capacities are computed. The sum of the individual buffer capacities is plotted in Fig. 8.13 for

m	Latency [s]	m	Latency [s]
1	0.507	7	0.772
2	0.542	8	0.818
3	0.588	9	0.864
4	0.645	10	0.910
5	0.680	11	0.968
6	0.726	12	1.003

Table 8.1: End-to-end latency for different execution scaling factors m.

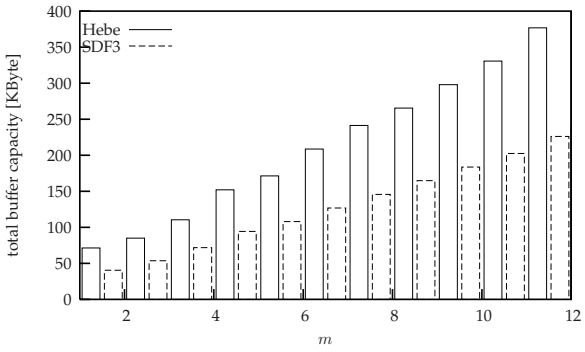

Figure 8.13: Total required FIFO-buffer capacities.

both the exact and approximation technique, and for different values of m.

The runtime to analyse a dataflow graph depends on the complexity of the graph. A measure of the complexity of a CSDF graph is the number of actors in the equivalent SRDF graph. Table 8.2 shows the number of equivalent SRDF actors for every execution scaling factor m and it shows the runtimes of the Hebe tool to analyse the CSDF graphs. For the SDF^3 tool, we were unable to measure the runtimes, because we used an experimental version of the tool that required manual interaction in generating the Pareto points between throughput and buffer capacities. Therefore, the total experiment took approximately four hours, which is higher than the runtimes observed with the Hebe tool. However, for our case study, the approximation technique is overestimating the buffer requirements with approximately 60 %. A similar trade-off has been seen in [39], where the approximation technique was compared with an exact technique that is based on maximum-cycle-mean analysis.

For an execution scaling factor $m = 11$, the input and output buffers are distributed between local and external memories. Therefore, additional bandwidth is required for the external memory to store and pre-fetch input and output data, resulting in estimated additional net-bandwidth of 2.3 $^{\text{MByte}}/_{\text{second}}$. However, these external memory accesses are latency tolerant. Therefore, they have a limited impact on the miss penalty of other processors in the multiprocessor system. The measured net bandwidth for instruction cache refills, data cache refills , and writing modified cache

m	Equivalent SRDF actors	runtime [seconds]
1,3,5	115305	0.19
2,6,10	230610	0.35
9	345915	0.54
4,12	461220	0.66
7	807135	1.13
8	922440	1.29
11	1268355	1.89

Table 8.2: Measured runtimes for computing the results with the dataflow-analysis tool Hebe.

blocks back to the external memory, equals 7.6 $^{\text{MByte}}/_{\text{second}}$ and 2.0 $^{\text{MByte}}/_{\text{second}}$ for an execution scaling factor $m = 1$ and $m = 11$, respectively. This is a low-latency bandwidth requirement reduction of a factor 3.9.

Finally, we compare the metric ϵ (as defined in Eq. (7.8)) for the case with and without execution scaling. This metric gives the number of processor stall cycles relative to the execution times. This is done for both tasks on the DSP processor, therefore, we refer to it as the processor utilisation. The reduction of the number of cache misses will reduce the number of processor stall cycles. Additionally, cache miss reduction to other processors in the multiprocessor system will potentially reduce the average cache miss penalty, due to a reduction of contention at the external memory port. When assuming cache-miss penalties of 100 and 150 DSP clock cycles for an instruction and data cache miss, respectively, then the processor utilisation is 88 % for the original code (i.e. execution scaling factor $m = 1$) and 97 % for an execution scaling factor $m = 11$. Therefore, the impact of execution scaling on the DSP processor utilisation is 9 %.

For the experiments in this thesis, we adapted the size of the instruction and data cache to show the impact of execution scaling on the number of cache misses. In general, if cache sizes are fixed, we can change the task granularity so that each task fits in the instruction and data cache, allowing us to use execution scaling to optimise for cache misses.

8.6 Concluding remarks

This chapter described a cache-based multiprocessor architecture with a shared external memory. The external memory enables the execution of tasks from which the memory footprint is considered to be too expensive to store on-chip. Compared to the previous multiprocessor architectures, the use of a shared external memory, instruction caches, and data caches increases the uncertainty in the temporal behaviour. Typically, at design time, it is hard to give conservatively-estimated upper bounds on execution times and processor stall cycles that are tight. Therefore, in the industry, often optimistically-estimated instead of conservatively-estimated upper bounds are used. With the use of optimistically-estimated bounds, we are unable to guarantee that throughput and end-to-end latency requirements are met for all

possible input stimuli. Therefore, a fall-back mechanism is required that need to be activated in case of a deadline miss.

Furthermore, this chapter proposed a novel cache-aware mapping technique that reduces the number of instruction and data cache misses for streaming jobs in a multiprocessor system. It is shown that executing tasks multiple times in a loop is effective if the individual tasks fit in the instruction and data cache, and the set of tasks executed on a processor do not fit simultaneously. We have described how to model a job mapped onto the multiprocessor and a specific execution scaling factor. With this model we derived the maximum number of successive task executions, by making use of existing dataflow-analysis techniques. For our industrial case study, which is a Digital-Radio-Mondiale receiver, we reduced the number of cache misses by a factor 4.2. The reduction of the number of cache misses and the reduction of contention at the external memory will improve the overall system performance. Future work is on optimising the memory hierarchy by explicitly pre-fetching the task's program code and working data set into a local memory, instead of pre-fetching with a cache.

Chapter 9

Concluding remarks

Car-infotainment platforms support a nucleus of common and stable jobs from the application areas: radio, audio, navigation and video. Jobs are, for performance and power efficiency reasons, executed on heterogeneous multiprocessor platforms. The number of supported jobs is increasing rapidly and the number of possible use cases is increasing exponentially with the number of jobs. Most of the jobs belong to the class of streaming and they have firm real-time requirements, like throughput and end-to-end latency.

To reduce the verification effort, we described an architecture for a predictable system from which we can verify, at design time, that the job's throughput and end-to-end latency requirements are satisfied. This thesis described a multiprocessor architecture for a firm real-time system that is optimised for infotainment-nucleus generation four. This generation includes jobs from the application areas radio and audio. Future generations will include jobs from the application areas navigation and video, but this is seen as future work. Figure 9.1 gives an overview of the architecture for infotainment-nucleus generation four. This architecture combines different flavours of tiles, so that it can execute all the jobs from infotainment-nucleus generation four while the uncertainty in temporal behaviour is reduced.

Tile i is based on the tile-based multiprocessor architecture that is described in Chapter 3. This architecture minimises resource sharing to come to a predictable temporal behaviour. Tiles communicate via a network-on-chip, because it offers a structural and scalable integration of our tiles into a working system. From a predictability perspective, the network's guaranteed communication services are a step forward in mastering the verification effort, i.e. we can guarantee that data is delivered in time. Processing tiles contain local memories and the processor is the only master who is accessing its local memory, so that memory-access latencies are predictable. The architecture makes use of local synchronisation, which means that all input and output containers of a task are stored locally (i.e. stored in local memories and network-interface buffers). The transport from one tile to another is implemented as a separate step, i.e. a copy action over the network using the services of the network. Finally, tasks are executed on processors by means of static-order schedules.

The advantage of this architecture is that we can derive, at design time, tight conser-

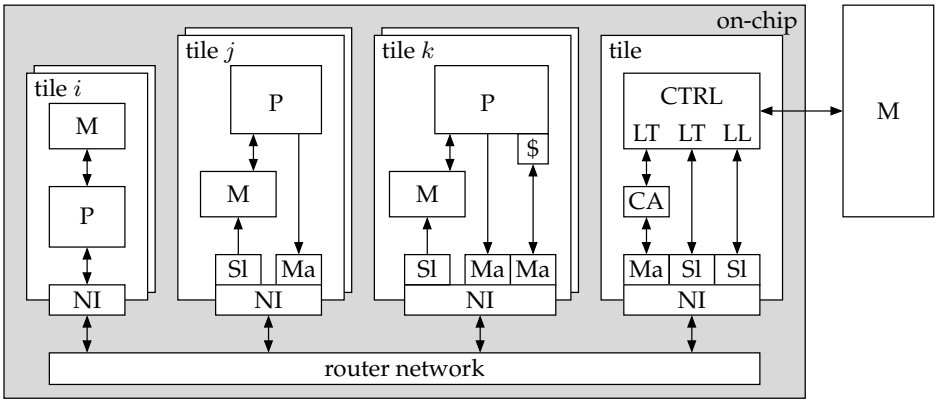

Figure 9.1: Predictable multiprocessor architecture for infotainment-nucleus generation four.

vatively-estimated upper bounds on execution times and communication latencies, and that the execution order of tasks is known. Therefore, the architecture enables the derivation of tight bounds on throughput and end-to-end latency. However, this architecture has also some limitations. First, static-order schedules cannot be used in case the execution order between tasks cannot be derived at design time, which is typically the case for tasks from multiple jobs. Second, the task's program code and the complete working data set must fit in the local memory of a tile and, in case of inter-tile communication, the buffer in a network interface must be able to contain a complete data container. So the size of tasks is limited and containers must be rather small. Third, it is a hardware driven approach which lacks flexibility. For example, the supported number of communication channels is coupled to the number of hardware buffers in network interfaces. As a consequence, for tile i in Fig. 9.1, the target domain is restricted to sample-based processing, like analog terrestrial radio and audio post-processing. Tile i can also contain a hardware accelerator or peripheral instead of a processor.

In Chapter 7, the multiprocessor architecture is extended with shared memories and address-based communication, so that processors can access their local memories and the memories in another tile. Data containers are stored in circular buffers that are located in a shared memory and they are implemented by software. A processor can suffer from stall cycles when accessing a shared memory while there is contention at the memory port. Furthermore, multiple communication channels can communicate data over the same network connection. The number of stall cycles and the communication latencies can be bounded due to guaranteed bandwidth services and bounded network-interface buffers. The circular buffers support checking for available number of full and empty containers in a buffer, which enables the use of run-time scheduling mechanisms. Therefore, processors can execute tasks from multiple jobs with uncorrelated throughput requirements.

The uncertainty in temporal behaviour is increased compared to the first architecture. Because a processor can suffer from stall cycles when accessing a shared memory, communication channels can influence each others communication latency, and

the execution order of tasks is unknown at design time. However, it increases the flexibility, because the number of communication channels and the buffer capacities can be adapted by changing the software. Therefore, for tile j in Fig. 9.1, the target domain is block-based processing. However, this architecture still requires that program code and data fits in the local memories of a processor. So the size of tasks is limited. Examples of tasks that can be executed on tile j are the MP3-decoder and acoustic-echo-cancellation tasks.

Chapter 8 extended the architecture with a shared external memory so that it can store tasks from which the memory footprint is considered to be too expensive to store on-chip. Level-one caches are used for instruction and data, to hide the large memory-access latencies in accessing the external memory. Processors suffer from stall cycles in case of a cache miss. The number of stall cycles depends on the number of cache misses and the cache-miss penalty. Compared to the previous multiprocessor architectures, the use of a shared external memory, instruction caches, and data caches again increases the uncertainty in the temporal behaviour. Typically, it is hard to give conservatively-estimated upper bounds on execution times and processor stall cycles that are tight (and useful). Therefore, in many parts of the industry, often optimistically-estimated instead of conservatively-estimated upper bounds are used. With the use of optimistically-estimated bounds, we are unable to guarantee that throughput and end-to-end latency requirements are met for all possible input stimuli. Therefore, a fall-back mechanism is required that need to be activated if a deadline miss occurs.

A predictable architecture does not, per definition, add a large increase in cost. It is shown in Chapter 5, after comparing the Æthereal network (which supports guaranteed throughput connections) with the traditional interconnect from platform SAF7780, that the chip area increased only a few percent and the round-trip latency between the DSP and CRD increased only 2.7 %. After mapping our channel equaliser job to the predictable architecture, the accuracy of the derived conservatively-estimated throughput bound is within 10.1 %. Furthermore, in Chapter 8, we described a cache-aware mapping technique to reduce the number of cache misses for streaming applications. This approach is seen as an intermediate step. Future work is optimising the memory hierarchy by explicitly pre-fetching the task's program code and working data set into a local memory, instead of pre-fetching with a cache.

At design time, we need to guarantee that each job will meet its real-time requirements like throughput and end-to-end latency, in order to guarantee a high quality for the user. This thesis uses dataflow modelling and analysis techniques, because they allow cyclic data dependencies that influence the job's performance. Cyclic data dependencies can come from feedback loops (i.e. cycles in a job's task graph), back-pressure due to bounded buffers, or scheduling dependencies (e.g. in case of static-order schedules). In this thesis, we limit us to jobs that can be modelled in a cyclo-static dataflow graph [9]. Modelling jobs with input data dependent container consumption and production behaviour, by making use of variable rate dataflow graphs [93], is seen as future work. We have shown how to construct a dataflow model from a job that is mapped onto our predictable multiprocessor platforms. It is shown that tasks, which are executed in a static order, can be represented with one actor despite the absence of the firing rule in the task's implementation. After each

mapping step, additional constraints are added to the dataflow model. The final dataflow model takes into account: computation of tasks, communication between tasks, buffer capacities, and the scheduling of shared resources. The job's throughput and end-to-end latency bounds can be derived from a self-timed execution of the dataflow graph, by making use of existing dataflow-analysis techniques.

The focus of this thesis is mapping of streaming jobs at design time. Adding a resource manager that can map streaming jobs at run time and one that can trade-off quality levels of jobs, is seen as a challenge for future work.

Appendix A

Modelling static-order schedules: Relation between phase f' and position q

This chapter is an extension on Section 4.3.2. It describes the relation between q and f', as introduced in Section 4.3.2.

A task u_i is modelled with actor v_i and the cyclo-static behaviour of the task is represented by the $\theta(v_i)$ distinct phases of actor v_i. Furthermore, task u_i occurs $\Omega(u_i, S_p)$ times in schedule S_p. The different positions of task u_i in schedule S_p, are modelled with different phases of Actor v'_i. The number of phases $\theta(v'_i)$ of actor v'_i is equal to the least common multiple (lcm) of the number of occurrences of task u_i in schedule S_p and the number of phases of actor v_i, i.e. $\theta(v'_i) = \text{lcm}(\Omega(u_i, S_p), \theta(v_i))$. Therefore, the number of phases $\theta(v'_i)$ is a multiple of the number of occurrences $\Omega(u_i, S_p)$, i.e. $\theta(v'_i) = l \cdot \Omega(u_i, S_p)$ with $l \in \mathbb{N}^+$ and $l \geq 1$. This is illustrated in the first column of Table A.1.

For phase f' of actor v'_i, which represents task u_i, its firing is at a certain position q in the static-order schedule S_p. The first firing of actor v'_i represents the first firing of task u_i in the static-order schedule S_p, therefore, it is at position $\phi(1, u_i, S_p)$. The kth firing of actor v'_i represents the kth firing of task u_i in the static-order schedule S_p. The static-order schedule is cyclo static and task u_i occurs $\Omega(u_i, S_p)$ times in the schedule. Therefore, the kth firing of task u_i is during the $\lfloor k/\Omega(u_i,S_p) \rfloor$ iteration of static-order schedule S_p. So it is at position:

$$\phi(((k-1)\%\Omega(u_i, S_p)) + 1, u_i, S_p) \tag{A.1}$$

For phase f' of actor v'_i, its firing is at a position q in the static-order schedule S_p. Now, we can compute position q as follows:

$$q = \phi(((f'-1)\%\Omega(u_i, S_p)) + 1, u_i, S_p) \tag{A.2}$$

The relation between phase f' and position q is also illustrated in Table A.1.

f'	$((f'-1)\%\Omega(u_i,S_p))+1$	q
1	1	$\phi(1,u_i,S_p)$
2	2	$\phi(2,u_i,S_p)$
$\vdots$	$\vdots$	$\vdots$
$\Omega(u_i,S_p)$	$\Omega(u_i,S_p)$	$\phi(\Omega(u_i,S_p),u_i,S_p)$
$\Omega(u_i,S_p)+1$	1	$\phi(1,u_i,S_p)$
$\vdots$	$\vdots$	$\vdots$
$2\cdot\Omega(u_i,S_p)$	$\Omega(u_i,S_p)$	$\phi(\Omega(u_i,S_p),u_i,S_p)$
$2\cdot\Omega(u_i,S_p)$+1	1	$\phi(1,u_i,S_p)$
$\vdots$	$\vdots$	$\vdots$
$l\cdot\Omega(u_i,S_p)$	$\Omega(u_i,S_p)$	$\phi(\Omega(u_i,S_p),u_i,S_p)$

Table A.1: Illustrating the relation between phase f' and position q.

Bibliography

[1] Benny Akesson, Kees Goossens, and Markus Ringhofer. Predator: A predictable SDRAM memory controller. In: *Proc. Int'l Conf. on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2007.

[2] ARM. *AMBA AXI Protocol Specification*, 2004.

[3] F. Bacelli, G. Cohen, G.J. Olsder, and J-P. Quadrat. *Synchronization and Linearity*. John Wiley & Sons, Inc., 1992.

[4] Marco Bekooij, Rob Hoes, Orlando Moreira, Peter Poplavko, Milan Pastrnak, Bart Mesman, Jan David Mol, Sander Stuijk, Valentin Gheorghita, and Jef van Meerbergen. Dataflow analysis for real-time embedded multiprocessor system design. In: *Dynamic and Robust Streaming in and between Connected Consumer-Electronic Devices*, volume 3, pp. 81–108. Springer, 2005.

[5] Marco Bekooij, Arno Moonen, and Jef van Meerbergen. Predictable and composable multiprocessor system design: a constructive approach. In: *Proc. Bits & Chips Symposium on Embedded Systems and Software*, 2007.

[6] Albert Benveniste and Gerard Berry. The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE*, 79 (9): pp. 1270–1282, 1991.

[7] B. Bhattacharya and SS Bhattacharyya. Parameterized dataflow modeling of DSP systems. In: *Proc. Int'l Conf. on Acoustics, Speech, and Signal Processing (ICASSP)*, 2000.

[8] H.S. Bhullar, R. van den Berg, J. Josten, and F. Zegers. Serving digital radio and audio processing requirements with sea-of-DSPs for automotive applications the philips way. In: *Proc. Int'l Conf. on Global Signal Processing (GSPx)*, 2004.

[9] G. Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete. Cyclo-static dataflow. *IEEE Transactions on signal processing*, 44 (2): pp. 397–408, February 1996.

[10] Shekhar Borkar, Pradeep Dubey, Kevin Kahn, David Kuck, Hans Mulder, Steve Pawlowski, and Justin Rattner. *Platform 2015: Intel Processor and Platform Evolution for the Next Decade*. Technical report, Intel, 2005.

[11] J.-Y. Le Boudec and P. Thiran. *Min-plus and Max-plus System Theory*, Chapter 4. Springer Berlin / Heidelberg, 2001.

[12] Jan Willem van den Brand and Marco Bekooij. Streaming consistency: a model for efficient MPSoC design. In: *Proc. Euromicro Symposium on Digital System Design (DSD)*, 2007.

[13] J.T. Buck. *Scheduling Dynamic Dataflow Graphs with Bounded Memory using the Token Flow Model*. PhD thesis, University of California at Berkeley, 1993.

[14] G.C. Buttazzo. *Hard Real-Time Computing Systems*. Kluwer Academic Publishers, 1997.

[15] K. Chen, S. Malik, and D.I. August. Retargetable static timing analysis for embedded software. In: *Proc. Int'l Symposium on System Synthesis (ISSS)*, pp. 39–44, 2001.

[16] F. Commoner, A. Holt, S. Even, and A. Pnueli. Marked directed graphs. *Journal of Computer and System Sciences*, 1971.

[17] R.L. Cruz. A calculus for netork delay, part ii: Network analysis. *IEEE Transactions on Information Theory*, 37 (1): pp. 132–141, January 1991.

[18] D.E. Culler, J.P. Singh, and A. Gupta. *Parallel computer architecture: a hardware/software approach*. Morgan Kaufmann Publishers, Inc., 1999.

[19] K. Denolf, M. Bekooij, J. Cockx, D. Verkest, and H. Corporaal. Exploiting the expressiveness of cyclo-static dataflow to model multimedia implementations. *EURASIP Journal on Advances in Signal Processing*, pp. 1–14, 2007.

[20] Verband der Automobilindustrie. VDA specification for car hands-free terminals. Draft, Version 1.5, Website: http://www.vdaqmc.de, December 2004.

[21] A. Donlin. Transaction level modeling: flows and use models. In: *Proc. Int'l Conf. on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2004.

[22] Royal Philips Electronics. Speech recognition systems. http://www.speechrecognition.philips.com.

[23] A.W.M. van den Enden. *Efficiency in multirate and complex digital signal processing*. PhD thesis, Eindhoven University of Technology, 2001.

[24] Alberto Ferrari and Alberto Sangiovanni-Vincentelli. System design: Traditional concepts and new paradigms. In: *Proc. Int'l Conf. on Computer Design (ICCD)*, pp. 2–12, 1999.

[25] Om Prakash Gangwal, Andrei Rădulescu, Kees Goossens, Santiago González Pestana, and Edwin Rijpkema. Building predictable systems on chip: An analysis of guaranteed communication in the Æthereal network on chip. In: Peter van der Stok (Ed.), *Dynamic and Robust Streaming In And Between Connected Consumer-Electronics Devices*, volume 3 of *Philips Research Book Series*, Chapter 1, pp. 1–36. Springer, 2005.

[26] O.P. Gangwal, A. Nieuwland, and P. Lippens. A scalable and flexible data synchronization scheme for embedded HW-SW shared-memory systems. In: *Proc. Int'l Symposium on System Synthesis (ISSS)*, pp. 1–6. ACM, 2001.

[27] J.D. Gee, M.D. Hill, D.N. Pnevmatikatos, and A.J. Smith. Cache performance of the SPEC92 benchmark suite. *Micro, IEEE*, Aug. 1993.

[28] A.H. Ghamarian, M.C.W. Geilen, S. Stuijk, T. Basten, A.J.M. Moonen, M.J.G. Bekooij, B.D. Theelen, and M.R. Mousavi. Throughput analysis of synchronous data flow graphs. In: *Proc. Int'l Conf. on Application of Concurrency to System Design (ACSD)*, 2006.

[29] A.H. Ghamarian, S. Stuijk, T. Basten, M.C.W. Geilen, and B.D. Theelen. Latency minimization for synchronous data flow graphs. In: *Proc. Euromicro Symposium on Digital System Design (DSD)*, 2007.

[30] Stefan Valentin Gheorghita. *Dealing with Dynamism in Embedded System Design: Application Scenarios*. PhD thesis, Eindhoven University of Technology, 2007.

[31] Santiago González Pestana, Edwin Rijpkema, Andrei Rădulescu, Kees Goossens, and Om Prakash Gangwal. Cost-performance trade-offs in networks on chip: A simulation-based approach. In: *Proc. Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pp. 764–769. IEEE Computer Society, Washington, DC, USA, February 2004.

[32] Kees Goossens, John Dielissen, Om Prakash Gangwal, Santiago González Pestana, Andrei Rădulescu, and Edwin Rijpkema. A design flow for application-specific networks on chip with guaranteed performance to accelerate SOC design and verification. In: *Proc. Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pp. 1182–1187. IEEE Computer Society, Washington, DC, USA, March 2005.

[33] Kees Goossens, John Dielissen, and Andrei Rădulescu. The Æthereal network on chip: Concepts, architectures, and implementations. *IEEE Design and Test of Computers*, 22 (5): pp. 414–421, Sept-Oct 2005.

[34] Kees Goossens, John Dielissen, Jef van Meerbergen, Peter Poplavko, Andrei Rădulescu, Edwin Rijpkema, Erwin Waterlander, and Paul Wielage. Guaranteeing the quality of services in networks on chip. In: Axel Jantsch and Hannu Tenhunen (Eds.), *Networks on Chip*, Chapter 4, pp. 61–82. Kluwer Academic Publishers, Hingham, MA, USA, 2003.

[35] Andreas Hansson, Martijn Coenen, and Kees Goossens. Channel trees: Reducing latency by sharing time slots in time-multiplexed networks on chip. In: *Proc. Int'l Conf. on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2007.

[36] Andreas Hansson, Martijn Coenen, and Kees Goossens. Undisrupted quality-of-service during reconfiguration of multiple applications in networks on chip. In: *Proc. Design, Automation and Test in Europe Conference and Exhibition (DATE)*, 2007.

[37] Andreas Hansson, Kees Goossens, Marco Bekooij, and Jos Huisken. CoMPSoC: A template for composable and predictable multi-processor system on chips. *Transactions on Design Automation of Electronic Systems (TODAES), ACM*, 2009.

[38] Andreas Hansson, Kees Goossens, and Andrei Rădulescu. A unified approach to constrained mapping and routing on network-on-chip architectures. In: *Proc. Int'l Conf. on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pp. 75–80, September 2005.

[39] Andreas Hansson, Maarten Wiggers, Arno Moonen, Kees Goossens, and Marco Bekooij. Applying dataflow analysis to dimension buffers for guaranteed performance in networks on chip. In: *Proc. Int'l Symposium on Networks-on-Chip (NOCS)*, 2008.

[40] Andreas Hansson, Maarten Wiggers, Arno Moonen, Kees Goossens, and Marco Bekooij. Enabling application-level performance guarantees in network-based systems on chip by applying dataflow analysis. *To appear in IET Computers & Digital Techniques*, 2009.

[41] D. Harel and A. Pnueli. On the development of reactive systems. pp. 477–498, 1985.

[42] J.L. Hennessy and D.A. Patterson. *Computer Architecture A quantitative Approach*. Morgan Kaufmann Publishers, 2003.

[43] S. Hosseine-Khayat and A.D. Bovopoulos. A simple and efficient bus management scheme that supports continuous streams. *ACM Transactions on Computer Systems*, 13 (2): pp. 122–140, May 1995.

[44] http://www.systemc.org. Systemc community. Website.

[45] M. Jersak, K. Richter, and R. Ernst. Performance analysis of complex embedded systems. *International Journal of Embedded Systems*, 1 (1–2): pp. 33–49, 2005.

[46] J.A. Kahle, M.N. Day, H.P. Hofstee, C.R. Johns, T.R. Maeurer, and D. Shippy. Introduction to the cell multiprocessor. *IBM Journal of Research and Development*, 49 (4): pp. 589–604, 2005.

[47] Kurt Keutzer, A. Richard Newton, Jan M. Rabaey, and Alberto Sangiovanni-Vincentelli. System-level design: orthogonalization of concerns and platform-based design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19 (12): pp. 1523–1543, 2000.

[48] Sanjeev Kohli. *Cache aware scheduling for synchronous dataflow programs*. Master's thesis, University of California, Berkeley, CA, 2004.

[49] H. Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, 1997.

[50] E.A. Lee and D.G. Messerschmitt. Synchronous data flow. In: *Proceedings of the IEEE*, 1987.

[51] E.A. Lee and T.M. Parks. Dataflow process networks. In: *Proc. of the Institute of Electrical and Electronics Engineers (IEEE)*, volume 83, pp. 773–799, 1995.

[52] P. Martin. *A comparison of Network-on-Chip and Busses*. Technical report, white paper downloadable from the Arteris website (www.arteris.com), 2005.

[53] A. Maxiaguine, Y. Zhu, S. Chakraborty, and W.-F. Wong. Tuning SoC platforms for multimedia processing: Identifying limits and tradeoffs. In: *Proc. Int'l Conf. on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2004.

[54] A.J.M. Moonen. *Modelling and simulation of guaranteed throughput channels of a hard real-time multiprocessor system*. Master's thesis, Eindhoven University of Technology, 2004.

[55] A.J.M. Moonen, C. Bartels, M.J.G. Bekooij, R. van den Berg, H. Bhullar, K. Goossens, P. Groeneveld, J. Huisken, and J. van Meerbergen. Comparison of an Aethereal network on chip and traditional interconnects - two case studies. In: Giovanni De Micheli, Salvador Mir, and Ricardo Reis (Eds.), *VLSI-SoC: Research Trends in VLSI and Systems on Chip*, number 249 in International Federation for Information Processing (IFIP). Springer, 2007. Fourteenth International Conference on Very Large Scale Integration of System on Chip (VLSI-SoC2006), October 16-18, 2006, Nice, France.

[56] A.J.M. Moonen, M.J.G. Bekooij, R. van den Berg, and J. van Meerbergen. Decoupling of computation and communication with a communication assist. In: *Proc. Euromicro Symposium on Digital System Design (DSD)*, 2007.

[57] A.J.M. Moonen, M.J.G. Bekooij, R. van den Berg, and J. van Meerbergen. Practical and accurate throughput analysis with the cyclo static dataflow model. In: *Proc. Int'l Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2007.

[58] A.J.M. Moonen, M.J.G. Bekooij, R. van den Berg, and J. van Meerbergen. Cache aware mapping of streaming applications on a multiprocessor system-on-chip. In: *Proc. Design, Automation and Test in Europe Conference and Exhibition (DATE)*, 2008.

[59] A.J.M. Moonen, M.J.G. Bekooij, and J. van Meerbergen. Timing analysis model for network based multiprocessor systems. In: *Proc. Workshop of Circuits, System and Signal Processing (ProRISC)*, pp. 91–99, Veldhoven, The Netherlands, November 2004.

[60] A.J.M. Moonen, R. van den Berg, M.J.G. Bekooij, H. Bhullar, and J. van Meerbergen. A multi-core architecture for in-car digital entertainment. In: *Proc. Int'l Conf. on Global Signal Processing (GSPx)*, Oct 2005.

[61] O.M. Moreira and M.J.G. Bekooij. Self-timed scheduling analysis for real-time applications. *EURASIP Journal on Advances in Signal Processing*, 2007: pp. 1–14, 2007.

[62] Steven S. Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann Publishers, 1997.

[63] T. Parks. *Bounded Scheduling of Process Networks*. PhD thesis, University of California at Berkeley, 1995.

[64] T.M. Parks, J.L. Pino, and E.A. Lee. A comparison of synchronous and cycle-static dataflow. In: *Proc. Conf. on Signals, Systems and Computers (Asilomar)*, 1995.

[65] Philips Semiconductors. *Device Transaction Level (DTL) Protocol Specification. Version 2.2*, July 2002.

[66] Joseph Pompei. Audio spotlight. Holosonic Research Labs, http://www.holosonics.com, 1998.

[67] Selliah Rathnam and Gert Slavenburg. An architectural overview of the programmable multimedia processor, TM-1. In: *Proc. Int'l Computer Conf. (COMPCON)*, 1996.

[68] E. Rijpkema, K. Goossens, A. Radulescu, J. Dielissen, J. van Meerbergen, P. Wielage, and E. Waterlander. Trade offs in the design of a router with both guaranteed and best-effort services for networks on chip. IEE Proceedings Computers and Digital Techniques, September 2003.

[69] Sebastian Ritz, Matthias Pankert, Vojin Zivojnovic, and Heinrich Meyr. Optimum vectorization of scalable synchronous dataflow graphs. In: *Proc. Int'l Conf. on Application-Specific Array Processors*, 1993.

[70] Andrei Rădulescu, John Dielissen, Santiago González Pestana, Om Prakash Gangwal, Edwin Rijpkema, Paul Wielage, and Kees Goossens. An efficient on-chip network interface offering guaranteed services, shared-memory abstraction, and flexible network programming. *IEEE Transactions on CAD of Integrated Circuits and Systems*, 24 (1): pp. 4–17, January 2005.

[71] Quino Sandifort, Lucien Breems, Carel Dijkmans, and Han Schuurmans. IF-to-digital converter for FM/AM/IBOC radio. In: *Proceedings European Conference on Solid-State Circuits (ESSCIRC)*, pp. 707–710, 2003.

[72] R. Schiffelers, R. van den Berg, J. van den Braak, H.S. Bhullar, S. de Feber, and M. Klaarwater. Epics7B - a learn and mean concept. In: *Proc. Int'l Conf. on Global Signal Processing (GSPx)*, 2003.

[73] Janis Sermulins, William Thies, Rodric Rabbah, and Saman Amarasinghe. Cache aware optimization of stream programs. In: *Proc. Int'l Conf. on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, 2005.

[74] M. Sgroi, M. Sheets, A. Mihal, K. Keutzer, S. Malik, J. Rabaey, and A. Sangiovanni-Vincentelli. Addressing the system-on-a-chip interconnect woes through communication-based design. In: *Proc. Design Automation Conference (DAC)*, pp. 667–672, June 2001.

[75] A Sriram and E.A. Lee. Determining the order of processor transactions in statically scheduled multiprocessors. *Journal of VLSI Signal Processing*, 1997.

[76] S. Sriram and S.S. Bhattacharyya. *Embedded Multiprocessors: Scheduling and Synchronization*. Marcel Dekker, Inc, 2000.

[77] D. Stiliadis and A. Varma. Latency-rate servers: a general model for analysis of traffic scheduling algorithms. *IEEE/ACM Transactions on Networking*, 6 (5): pp. 611–624, October 1998.

[78] J. Stott. Digital radio mondiale: key technical features. *Electronics & communication engineering journal*, pp. 4–14, February 2002.

[79] S. Stuijk, M.C.W. Geilen, and T. Basten. Exploring trade-offs in buffer requirements and throughput constraints for synchronous dataflow graphs. In: *Proc. Design Automation Conference (DAC)*, 2006.

[80] S. Stuijk, M.C.W. Geilen, and T. Basten. SDF^3: SDF For Free. In: *Proc. Int'l Conf. on Application of Concurrency to System Design (ACSD)*, 2006.

[81] S. Stuijk, M.C.W. Geilen, and T. Basten. Throughput-buffering trade-off exploration for cyclo-static and synchronous dataflow graphs. *IEEE Transactions on Computers*, 57 (10): pp. 1331–1345, 2008.

[82] Sander Stuijk. *Predictable Mapping of Streaming Applications on Multiprocessors*. PhD thesis, Eindhoven University of Technology, 2007.

[83] A.S. Tanenbaum. *Computer Networks*. Englewood Cliffs, NJ: Prentice-Hall, 1996.

[84] J. Teich and S.S. Bhattacharyya. Analysis of dataflow programs with interval-limited data-rates. *The Journal of VLSI Signal Processing*, 43 (2): pp. 247–258, 2006.

[85] 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 langauge. In: *Proc. Int'l Conf. on Formal Methods and Models for Codesign (MEMOCODE)*, 2007.

[86] 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: *Proc. Int'l Conf. on Formal Methods and Models for Codesign (MEMOCODE)*, 2006.

[87] L. Thiele, S. Chakraborty, and M. Naedele. Real-time calculus for scheduling hard real-time systems. In: *Proc. Int'l Symposium on Circuits and Systems (ISCAS)*, May 2000.

[88] R. van den Berg and H.S. Bhullar. Next generation philips digital car radios, based on a sea-of-DSP concept. In: *Proc. Int'l Conf. on Global Signal Processing (GSPx)*, 2004.

[89] L. Wehmeyer and P. Marwedel. Influence of memory hierarchies on predictability for time constrained embedded software. In: *Proc. Design, Automation and Test in Europe Conference and Exhibition (DATE)*, 2005.

[90] P Wielage, E J Marinissen, and C Wouters. Design and DFT of a high-speed area-efficient embedded asynchronous FIFO. In: *Proc. Design, Automation and Test in Europe Conference and Exhibition (DATE)*, 2007.

[91] M. H. Wiggers, M. Bekooij, and G. J. M. Smit. Efficient computation of buffer capacities for cyclo-static dataflow graphs. In: *Proc. Design Automation Conference (DAC)*, 2007.

[92] M. H. Wiggers, M. Bekooij, and G. J. M. Smit. Modelling runtime arbitration by latency-rate servers in dataflow graphs. In: *Proc. Int'l Workshop on Software and Compilers for Embedded Systems (SCOPES)*, 2007.

[93] M. H. Wiggers, M. Bekooij, and G. J. M. Smit. Computation of buffer capacities for throughput constrained and data dependent inter-task communication. In: *Proc. Design, Automation and Test in Europe Conference and Exhibition (DATE)*, 2008.

[94] M. H. Wiggers, M. J. G. Bekooij, P. G. Jansen, and G. J. M. Smit. Efficient computation of buffer capacities for cyclo-static real-time systems with back-pressure. In: *Proc. Symposium on Real-Time and Embedded Technology and Applications (RTAS)*, 2007.

[95] Reinhard Wilhelm et. al. The worst-case execution time problem overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems*, 7 (3), 2008.

[96] Selwyn Wright. anti-sound / silence machine. University of Huddersfield, UK.

Curriculum Vitae

Arno Moonen was born in Weert, The Netherlands, on September 2nd, 1978. After the primary school, he finished the study electrical engineering at the Lower Technical School (LTS) in Weert and at the Middle Technical School (MTS) in Roermond in 1994 and 1998, respectively. In 2001 he received the Bachelor Degree in electrical engineering at Fontys University of Applied Sciences in Eindhoven.

For his Master Degree, he studied at the Eindhoven University of Technology. The topic of his M.Sc. thesis was on modelling of communication channels in hard real-time multiprocessor systems. This research was conducted at Philips Research in Eindhoven.

After receiving his Master Degree in 2004, he started as a Ph.D. research assistant within the Electronic Systems group at Eindhoven University of Technology. His research was in cooperation with NXP Research in Eindhoven and NXP Semiconductors, Business-Line Car Infotainment, in Nijmegen. It has led among others to several publications and this thesis.

Since July 2008, Arno is working within the digital design group at Prodrive B.V. in Son, The Netherlands.

List of publications

First author

[58] A.J.M. Moonen, M.J.G. Bekooij, R. van den Berg, and J. van Meerbergen. Cache Aware Mapping of Streaming Applications on a Multiprocessor System-on-Chip. In *Proc. Design, Automation and Test in Europe Conference and Exhibition (DATE)*, 2008.

[55] A.J.M. Moonen, C. Bartels, M.J.G. Bekooij, R. van den Berg, H. Bhullar, K. Goossens, P. Groeneveld, J. Huisken, and J. van Meerbergen. Comparison of the Æthereal Network on Chip and Traditional Interconnects - Two Case Studies. In International Federation for Information Processing (IFIP), Volume 249, *VLSI-SoC: Research Trends in VLSI and Systems on Chip*, Boston: Springer, 2007.

[57] A.J.M. Moonen, M.J.G. Bekooij, R. van den Berg, and J. van Meerbergen. Practical and Accurate Throughput Analysis with the Cyclo Static Dataflow Model. In *Proc. Int'l Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2007.

[56] A.J.M. Moonen, M.J.G. Bekooij, R. van den Berg, and J. van Meerbergen. Decoupling of Computation and Communication with a Communication Assist. In *Proc. Euromicro Symposium on Digital System Design (DSD)*, 2007.

[60] A.J.M. Moonen, R. van den Berg, M.J.G. Bekooij, H. Bhullar, and J. van Meerbergen. A Multi-Core Architecture for In-Car Digital Entertainment. In *Proc. Int'l Conf. on Global Signal Processing (GSPx)*, 2005.

[59] A.J.M. Moonen, M.J.G. Bekooij, and J. van Meerbergen. Timing analysis model for network based multiprocessor systems. In *Proc. Workshop of Circuits, System and Signal Processing (ProRISC)*, 2004.

[54] A.J.M. Moonen. *Modelling and simulation of guaranteed throughput channels of a hard real-time multiprocessor system.* Master thesis, Eindhoven University of Technology, 2004.

Co-author

[40] A. Hansson, M. Wiggers, A.J.M. Moonen, K. Goossens, and M.J.G. Bekooij. Enabling Application-Level Performance Guarantees in Network-Based Systems on Chip by Applying Dataflow Analysis. *To appear in IET Computers & Digital Techniques*, 2009.

[39] A. Hansson, M. Wiggers, A.J.M. Moonen, K. Goossens, and M.J.G. Bekooij. Applying Dataflow Analysis to Dimension Buffers for Guaranteed Performance in Networks on Chip. In *Proc. Int'l Symposium on Networks-on-Chip (NOCS)*, 2008.

[5] M.J.G. Bekooij, A.J.M. Moonen, and J. van Meerbergen. Predictable and composable multiprocessor system design: a constructive approach. In *Proc. Bits & Chips Symposium on Embedded Systems and Software*, 2007.

[28] A.H. Ghamarian, M.C.W. Geilen, S. Stuijk, T. Basten, A.J.M. Moonen, M.J.G. Bekooij, B.D. Theelen, and M.R. Mousavi. Throughput Analysis of Synchronous Data Flow Graphs. In *Proc. Int'l Conf. on Application of Concurrency to System Design (ACSD)*, 2006.