Predictable Mapping of Streaming Applications on Multiprocessors

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 donderdag 25 oktober 2007 om 16.00 uur

door

Sander Stuijk

geboren te Breda

Dit proefschrift is goedgekeurd door de promotoren:

prof.dr. H. Corporaal
en
prof.dr.ir. J. van Meerbergen

Copromotor:
dr.ir. T. Basten

CIP-DATA LIBRARY TECHNISCHE UNIVERSITEIT EINDHOVEN

Stuijk, Sander

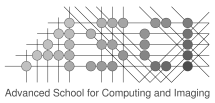
Predictable Mapping of Streaming Applications on Multiprocessors
/ by Sander Stuijk. - Eindhoven : Technische Universiteit Eindhoven, 2007.
Proefschrift. - ISBN 978-90-386-1624-7
NUR 959
Trefw.: multiprogrammeren / elektronica ; ontwerpen / multiprocessoren /
ingebedde systemen.
Subject headings: data flow graphs / electronic design automation /
multiprocessing systems / embedded systems.

Predictable Mapping of Streaming Applications on Multiprocessors

Committee:

prof.dr. H. Corporaal (promotor, TU Eindhoven)
prof.dr. J. van Meerbergen (promotor, TU Eindhoven)
dr.ir. T. Basten (copromotor, TU Eindhoven)
prof.dr. K.G.W. Goossens (TU Delft)
prof.dr. A. Jantsch (Royal Institute of Technology, Kista, Sweden)
dr. R. Marculescu (Carnegie Mellon University, Pittsburgh, USA)
prof.dr.ir. R.H.J.M. Otten (TU Eindhoven)

The work in this thesis is supported by the Dutch government in their NWO research program within the PROMES project 612.064.206.



This work was carried out in the ASCI graduate school.
ASCI dissertation series number 152.

Printing: Printservice Technische Universiteit Eindhoven

Abstract

Predictable Mapping of Streaming Applications on Multiprocessors

The design of new consumer electronics devices is getting more and more complex as more functionality is integrated into these devices. To manage the design complexity, a predictable design flow is needed. The result should be a system that guarantees that an application can perform its own tasks within strict timing deadlines, independent of other applications running on the system. This requires that the timing behavior of the hardware, the software, as well as their interaction can be predicted.

A heterogeneous multi-processor System-on-Chip (MP-SoC) is often mentioned as the hardware platform to be used in modern electronics systems. An MP-SoC offers good potential for computational efficiency (operations per Joule) for many applications. Networks-on-Chip (NoCs) are proposed as interconnect in these systems. A NoC offers scalability and guarantees on the timing behavior when communicating data between various processing and storage elements. Combining this with a predictable resource arbitration strategy for the processors and storage elements gives a predictable platform. To obtain a fully predictable system, also the timing behavior and resource usage of an application mapped to such an MP-SoC platform needs to be analyzable and predictable. The Synchronous Dataflow (SDF) model of computation fits well with the characteristics of streaming applications, it can capture many mapping decisions, and it allows design-time analysis of timing and resource usage. Therefore, this thesis aims to achieve predictable, streaming application behavior on NoC-based MP-SoC platforms for applications modeled as SDF graphs (SDFGs).

The most important timing requirement for streaming applications is usually related to the throughput that should be achieved. A major aspect influencing the achievable throughput, and in fact also the computational efficiency, is the storage space allocated to the data streams being processed in the system. In an SDFG model, the edges in the graph typically correspond to data streams. The storage allocation problem for an SDFG is the problem of assigning a fixed storage size to its edges. This size must be chosen such that the throughput requirement of the system is met, while minimizing the required storage space. The first major contribution of this thesis is an exact technique to explore the

throughput/storage-space trade-off space. Despite the theoretical complexity of the storage allocation problem, the technique performs well in practice. By considering the entire trade-off space, it is possible to cope with situations where the precise throughput constraint is not (yet) known or might dynamically change. In multi-processor mapping, for example, scheduling decisions can influence the achievable throughput. This introduces uncertainty on the relation between the throughput and the storage requirements of an SDFG in the early phases of the trajectory when these scheduling decisions are not yet made.

In any multi-processor mapping trajectory, an application must be bound to and scheduled onto the processors and storage elements in the MP-SoC. An important contribution of this thesis is such a technique to bind and schedule an SDFG onto an MP-SoC. Existing binding and scheduling techniques can only deal with single-rate execution dependencies between tasks. SDFGs can express multi-rate dependencies between tasks. Dependencies in an SDFG can be expressed in single-rate form, but then the problem size may increase exponentially making the binding and scheduling infeasible. The binding and scheduling technique presented in this thesis works directly on SDFGs, building on an efficient technique to calculate throughput of a bound and scheduled SDFG.

When the application tasks have been bound to and scheduled onto the platform components, it remains to schedule the communication onto the MP-SoC interconnect. This thesis presents three different scheduling strategies that schedule time-constrained communications on a NoC while minimizing resource usage by exploiting all scheduling freedom offered by NoCs. It is shown that these strategies outperform existing NoC scheduling techniques. Furthermore, a technique is presented to extract the timing constraints on the communication from a bound and scheduled SDFG, connecting the NoC scheduling strategies to the binding and scheduling strategy for SDFGs mentioned earlier.

Finally, the techniques presented in this thesis are embedded into a coherent design flow. The starting point is a streaming application that is modeled with an SDFG and a NoC-based MP-SoC that offers a predictable timing behavior. The objective of the flow is to minimize the resource usage while offering guarantees on the timing behavior, in practice the throughput, of the application when mapped to the system. A case study is performed that maps a set of multimedia applications (H.263 encoder/decoder and an MP3 decoder) onto a NoC-based MP-SoC. It shows that the design flow, SDFG mapping techniques, and SDFG analysis techniques presented in this thesis enable a mapping of a streaming application onto a NoC-based architecture that has a predictable timing behavior. This makes it the first complete design flow that maps a time-constrained SDFG to a NoC-based MP-SoC while providing throughput guarantees.

Contents

Abstract **i**

1 Introduction **1**
1.1 Trends in Embedded Systems: A Consumers Perspective 1
1.2 Trends in Embedded Systems: A Designers Perspective 3
1.3 Problem Statement 7
1.4 A Predictable Design Flow 10
1.5 Contributions 14
1.6 Thesis Overview 14

2 Streaming Multimedia Applications **17**
2.1 Overview 17
2.2 Application Domain 17
2.3 Application Modeling 20
2.4 Analyzing Actor Resource Requirements 23
2.5 Summary 25

3 Network-on-Chip-based Multi-Processor Platform **27**
3.1 Overview 27
3.2 Multi-Processor Platform Template 27
3.3 Resource Arbitration Strategies 29
3.4 Resource Arbitration in the Platform 32
3.5 Summary 34

4 Dataflow Preliminaries **35**
4.1 Overview 35
4.2 Synchronous Dataflow 35
4.3 Synchronous Dataflow and Time 37
4.4 Throughput Analysis 40
4.5 Scheduling 43
4.6 SDF^3: Synchronous Dataflow for Free 47
4.7 Comparison of Dataflow Models 49

4.8 Summary 52

5 Modeling Memory Mappings **55**
5.1 Overview 55
5.2 Related Work 56
5.3 SDF model for Memory Accesses 58
5.4 Memory Allocation 60
5.5 Experimental Evaluation 63
5.6 Summary 66

6 Resource Allocation **67**
6.1 Overview 67
6.2 Related Work 68
6.3 Platform Graph 69
6.4 Resource-Aware Application Graph 71
6.5 Resource Allocation Problem 72
6.6 Throughput Analysis 73
6.7 Resource Allocation Strategy 80
6.8 Experimental Evaluation 83
6.9 Summary 87

7 Throughput-Buffering Trade-Off Exploration **89**
7.1 Overview 89
7.2 Related Work 90
7.3 Storage Requirements 92
7.4 Storage Dependencies 94
7.5 Design-Space Exploration 98
7.6 Experimental Evaluation 102
7.7 Approximation of Buffer Sizes 105
7.8 Buffer Requirements for Binding-aware SDFGs 110
7.9 Summary 112

8 Network-on-Chip Routing and Scheduling **113**
8.1 Overview 113
8.2 Related Work 114
8.3 Communication Modeling 115
8.4 Interconnect Graph 117
8.5 Time-Constrained Scheduling Problem 119
8.6 Scheduling Strategies 125
8.7 Benchmark 129
8.8 Experimental Evaluation 131
8.9 Extracting Communication Scenarios from SDFGs 136
8.10 Summary 141

9 Design Flow **143**
9.1 Overview . . . 143
9.2 Related Work . . . 145
9.3 Starting Point . . . 148
9.4 MP-SoC Configuration . . . 150
9.5 Memory Dimensioning . . . 151
9.6 Constraint Refinement . . . 156
9.7 Tile Binding and Scheduling . . . 160
9.8 NoC Routing and Scheduling . . . 161
9.9 Implementation . . . 162
9.10 Summary . . . 163

10 Case Study **165**
10.1 Overview . . . 165
10.2 Applications . . . 166
10.3 Hardware Architecture . . . 170
10.4 Mapping . . . 172
10.5 Discussion on the Design Flow . . . 177
10.6 Summary . . . 179

11 Conclusions and Future Work **181**
11.1 Conclusions . . . 181
11.2 Open Problems and Future Research . . . 183

Bibliography **187**

Glossary **201**

Samenvatting **207**

Acknowledgments **211**

Curriculum Vitae **213**

List of Publications **215**

Chapter 1

Introduction

1.1 Trends in Embedded Systems: A Consumers Perspective

The number of consumer electronics devices sold worldwide is growing rapidly. A total of 2.1 billion consumer electronics devices with a total value of $1.3 trillion were sold worldwide in 2006. It is expected that by 2010 this has grown to over 3 billion devices with a total value of around $1.6 trillion [68]. Most of these devices contain one or more processors that are used to realize the functionality of the device. This type of devices are called **embedded systems**. Embedded systems range from portable devices such as digital cameras and MP3-players, to systems like a television or the systems controlling the flight of an airplane. These systems are everywhere around us in our daily live. Most of them are becoming intelligent micro-systems that interact with each other, and with people, through (wireless) sensors and actuators. Embedded systems form the basis of the so-called post-PC era [88], in which information processing is more and more moving away from just PCs to embedded systems. This trend is also signaled by ubiquitous computing [149], pervasive computing [57] and ambient intelligence [1]. These three visions describe all a world in which people are surrounded by networked embedded systems that are sensitive to their environment and that adapt to this environment. Their objective is to make information available *anytime, anywhere*. Embedded systems provide the necessary technology to realize these visions [16]. Realization of these visions implies that the number of embedded systems surrounding us in our daily lives will increase tremendously.

An important subclass of embedded systems are **embedded multimedia systems**. These systems combine multiple forms of information content and information processing (e.g. audio, video, animations, graphics) to inform or entertain the user. Examples of such systems are mobile phones, game consoles, smart cameras and set-top boxes. Many of the applications that perform the in-

Figure 1.1: Embedded multimedia systems: PlayStation 3 and iPhone.

formation processing in these systems process audio, video and animation. These types of data are inherently streaming. So, many embedded multimedia systems contain **streaming applications** [142]. These applications typically perform a regular sequence of transformations on a large (or virtually infinite) sequence of data items.

The functionality integrated into new embedded multimedia systems is ever increasing. The Sony PlayStation has, for example, transformed itself from a simple game console to a complete entertainment center. It not only allows users to play games, it can also be used to watch movies, listen to music and to browse the Internet or chat online with other PlayStation 3 users. Another example of a true multimedia platform is the Apple iPhone. It includes many different applications next to the mobile-phone functionality. It has, for example, a wide-screen LCD display that allows users to watch movies and browse through their collection of photos that are taken with the build-in camera. The phone contains also an MP3-player which allows users to listen for up-to 16 hours to their favorite music. While traveling, users can also use the phone to browse the Internet, send emails or use online navigation software such as Google-maps. It is expected that even more functions will be integrated into future embedded multimedia systems. This trend was already signaled by Vaandrager in 1998 who stated that "for many products in the area of consumer electronics the amount of code is doubling every two years" [145].

Current embedded multimedia systems have a robust behavior. Consider for example a modern high-end television system. Such a system splits the incom-

ing video stream from its accompanying audio stream. Many different picture enhancement algorithms are executed on the video stream to improve its quality when displayed on the screen. Despite the complex processing going on inside the television, the video and audio stream are output in sync on the screen and the speakers. Consumers expect that future embedded multimedia systems provide the same robust behavior as current systems have despite the fact that more and more media processing is performed in software [26].

In summary, the following trends in embedded (multimedia) systems are observed from the perspective of consumers.

- The number of embedded systems surrounding people in their daily lives is growing rapidly, and these systems are becoming connected more and more often.
- Increasingly more functionality is integrated into a single multimedia system.
- Users expect the same seamless behavior of all functions offered by novel multimedia systems as offered by existing systems.

1.2 Trends in Embedded Systems: A Designers Perspective

The previous section outlines the most important trends in the field of embedded systems from the perspective of consumers. It shows that embedded systems have to handle an increasing number of applications that are concurrently executed on the system. At the same time, guarantees must be provided on the behavior of each application running on the system. This section considers the same systems, but it looks at the trends visible in their design(-process).

The omnipresence of embedded systems in people's lives is leading to a tremendous increase in the amount of data that is being used. Today, people have gigabytes of photos, music and video on their systems. That data must be processed in real-time to be made useful. Embedded systems must provide the required computational power to do this. At the same time, their energy consumption should be kept at a minimum as many of these devices are battery powered (e.g., mobile-phone, MP3-player, digital-camera). To fulfill these requirements, the use of **multi-processor systems-on-chip** (MP-SoCs) is becoming increasingly popular [15, 73]. For example, Intel has shifted from increasing the clock frequency of every processor generation to a strategy in which multiple cores are integrated on a single chip. This paradigm shift is outlined in their platform 2015 vision [25]. It describes the expected evolution of Intel processor architectures from single core systems, via multi-core systems toward many-core systems (see Figure 1.2). The Cell architecture [74] that is used in the PlayStation 3 is another example

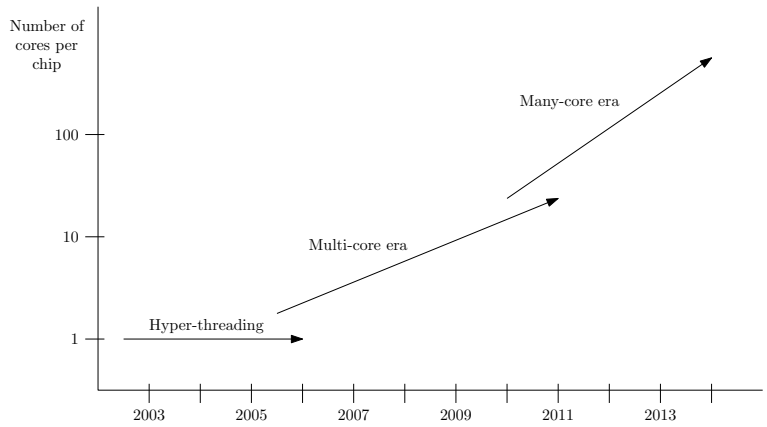


Figure 1.2: Current and expected eras for Intel processor architectures [25].

that shows the increasing popularity of MP-SoCs. It combines a PowerPC core with 8 synergetic processors that are used for data-intensive processing. A third example is the Nexperia digital video platform [36] from NXP. It supports digital television, home gateway and networking, and set-top box applications. An advanced member of the Nexperia family is the PNX8550 that combines two TriMedia processors, a MIPS processor and several hardware accelerators in a single chip.

An important design question that arises when developing an MP-SoC is whether a homogeneous or heterogeneous solution should be used. Amdahl observed that the less parallel portion of an application can limit the performance on a parallel system [6]. This observation is an important reason for using heterogeneous multi-processor systems. Different types of cores can be used in such a system to reduce the time needed to run the less parallel code. For example, consider an application in which 10% of the execution time is spent in code that cannot be executed in parallel on a 25-processor system. Suppose that in order to run this sequential code twice as fast, a processor would be needed that requires five times as much area as the cores used in the 25-processor system. A heterogenous 21-processor system with 20 cores similar to the cores used in the 25-processors and one core with the improved performance, would require the same area as the homogeneous 25-processor system. To compare the performance of both systems, the speed-up of the application on both systems can be computed using Amdahl's law [61]. This law computes the maximum expected improvement, i.e., reduction in run-time, to an overall system when only part of the system is improved. The speed-up of the application when executed on a homogeneous 25-processor system and a heterogenous 21-processor system relative to a sequential implementation of the application are:

$$Speedup_{homogeneous} = \frac{1}{0.1 + 0.9/25} = 7.4x$$

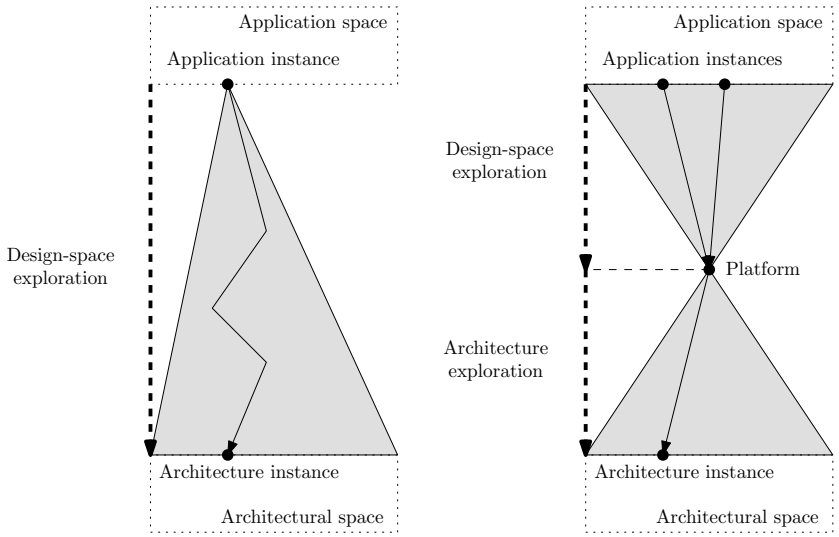
$$Speedup_{heterogeneous} = \frac{1}{0.1/2 + 0.9/20} = 10.5x$$

This example shows that using a **heterogeneous system** with some potentially large processors can be advantageous for the achieved speedup of the whole application. In addition, heterogeneous systems can show significant advantages in energy consumption as compared to homogeneous systems as the instruction set of the various processors can be optimized for their specific tasks.

The processing and storage elements that make up an MP-SoC must be interconnected. Traditionally, this has been done using on-chip buses or crossbar switches. These approaches do not scale very well when more processors are integrated in a system. To address these issues, **Networks-on-Chip** (NoCs) have been introduced [19, 32]. NoCs provide a scalable interconnect that can be shared between the processors and memories that are connected to it. Furthermore, it can provide guarantees on the time needed to send data through the NoC [93, 122]. This property makes NoCs suitable for use in MP-SoCs with a predictable timing behavior which is key for building reliable embedded multimedia systems.

The growing complexity of embedded multimedia systems leads to a large increase in their development effort. At the same time, the market dynamics for these systems push for shorter and shorter development times. It will soon be obligatory to keep to a strict design time budget that will be as small as six months to go from initial specification to a final and correct implementation [78]. Furthermore, the non-recurring engineering cost associated with the design and tooling of complex chips is growing rapidly. The International Technology Roadmap for Semiconductors (ITRS) predicts that while manufacturing complex Systems-on-Chip will be feasible, the production cost will grow rapidly as the costs of masks is raising drastically [69]. To address these issues, a **platform-based design** methodology is proposed in [39, 78]. The objective of this design methodology is to increase the re-use across different products that share certain functionality and the re-use between different product generations. The first form of re-use decreases the production cost as the same hardware can be used in more products. The second form of re-use lowers the development time as functionality implemented for a product does not have to be re-implemented for a successor product.

The traditional design methodology is a single monolithic flow that maps an application onto an architecture (see Figure 1.3(a)). It starts with a single application which is shown at the top of Figure 1.3(a). The bottom of the figure shows the set of architectures that could support this application. The design process (black arrow) selects the most attractive solution as defined by a cost function. Synthesis of this architecture is often an explicit objective of the de-



(a) Traditional. (b) Platform-based.

Figure 1.3: Design-space exploration strategies.

sign methodology [12, 38]. The platform-based design methodology [39, 78] no longer maps a single application to an architecture that is optimal for this single application. Instead, it maps an application onto a hardware/software platform that can also be used for different applications from the same application space (see Figure 1.3(b)). This platform consists of a set of interconnected hardware components (e.g., processors, memories, etc.), potentially software components with, for example, operating-system type of functionality and an application program interface (API) that abstracts from the underlying hardware and software. This API allows replacing one platform instance from the architecture space with another platform instance without the need to re-implement the application on the platform. The platform-based design methodology stimulates the use of a common "platform" denominator between multiple applications from the same application space. As a result, future design flows that map an application to a platform will focus on compiling an application onto an existing platform [78].

The trends signaled above show that the context in which applications are executed is becoming more dynamic. In future systems, multiple applications are running concurrently on an MP-SoC, and the set of active applications may change over time. At the same time, users expect a reliable behavior [10] of each individual application independent of the context in which it is operating. Virtualization of the resources in a system has been proposed as a concept to tackle this problem. The idea behind **virtualization** is that an application is given the

illusion that it is running on its own copy of the real hardware which however has only a fraction of the resources that are available in the real platform. For example, a processor which can do 100 million instructions per second could use a Time-Division Multiple-Access (TDMA) scheduler to present itself to an application A as a processor which can run 50 million instructions per second. This leaves room for another application B to use the remaining 50 million instructions per second without knowing that application A is also running on this processor. Virtualization has become popular in recent years in server and desktop computers [25, 61]. The concept is also employed in embedded systems. The Cell architecture [74] of IBM uses virtualization to avoid that programmers have to think about sharing processing resources and to guarantee the real-time response characteristics of applications. The Hijdra architecture [17] of NXP is another example of an embedded multi-processor system that uses virtualization. This architecture assumes that every hardware component has a scheduler that allows it to be shared between applications without them influencing each others timing behavior.

In summary, the following trends in the design of embedded systems are observed from a design perspective.

- Heterogeneous multi-processor systems are used to provide the required computational power for novel embedded multimedia systems.
- Networks-on-chip are used to provide a scalable interconnect with timing guarantees between the processors in the system.
- Platform-based design reduces production cost, design cost and design time of embedded systems.
- Virtualization of resources is used to guarantee a predictable behavior of applications in a dynamic environment.

1.3 Problem Statement

The trends outlined in the first two sections of this chapter show that the design complexity of future embedded multimedia systems is growing rapidly. Consumers expect that the number of applications integrated into these systems is increased as compared to existing systems. At the same time, they expect that this increased functionality does not affect the reliable behavior and quality of these systems, and that the price of these systems does not increase (too much). System-designers do not only have to meet these criteria, they also have to cope with a decreasing time-to-market and increasing design cost. To deal with these conflicting requirements, designers are starting to use multi-processor systems-on-chip, virtualization and a platform-based design methodology. The challenge is to combine these elements into a design method that enables designers to build

systems with a predictable timing behavior. In such a **predictable system**, applications are mapped to a platform while timing guarantees are provided for each application independent of the other applications executing simultaneously on the platform.

This thesis deals with the design of systems with a predictable timing behavior. Three components are needed to build these systems. First, a platform must be used that offers a predictable timing behavior to individual applications independent of other applications running on the same platform. Second, a model should be used that allows timing analysis of the application and its mapping to the platform. Third, a design flow should be used that allocates sufficient resources for an application in the platform to meet its timing requirements. The remainder of this section discusses all three aspects in some more detail and it explains the choices made in this thesis.

Multi-processor systems that use a NoC interconnect will be a dominant hardware platform in future embedded multimedia systems. The tile-based architecture presented in [30] enables the structured design of these NoC-based MP-SoCs. The template of this architecture allows any type of interconnect. Figure 1.4 shows the template instantiated with a NoC interconnect. Each tile contains a processor (P), a memory (M), a communication assist (CA) and a network interface (NI). The latter two resources are needed to decouple the computation from the communication and to connect the tile with the NoC. This NoC connects all tiles together via its routers (R) and links. To use this NoC-based MP-SoC in a predictable system, the platform should offer a resource sharing mechanism that allows multiple applications to use the platform resources simultaneously while guarantees can be provided on the amount of time an application has access to the resources and frequency of these accesses. A platform that offers these guarantees is called a **predictable platform**. The NoC-based MP-SoC shown in Figure 1.4 can be turned into a predictable platform through the use of virtualization on its resources. Due to the virtualization it is possible to consider a single application at a time when designing a system in which multiple applications are executed concurrently. This avoids that all combinations of applications have to be analyzed when verifying the timing constraints of an application that is mapped to the platform.

The design of a predictable system requires that the timing behavior of the application and its mapping to the platform can be analyzed. This can be done by modeling the application and mapping decisions in a **Model-of-Computation** (MoC) that allows timing analysis. A MoC captures, in an abstract manner, the relevant features of a computation [73]. Which features are relevant depends on the context in which a MoC is used. For the purpose of designing a predictable MP-SoC system, it is important that the MoC can express the concurrency that is present in an application. This concurrency should be exploited when mapping

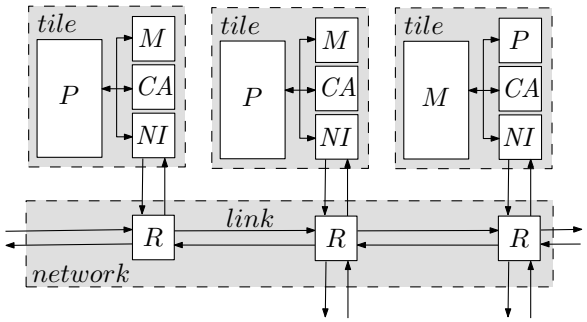


Figure 1.4: Predictable platform.

the application to the MP-SoC that is inherently concurrent. The MoC should also express the synchronization and communication between the concurrent tasks of an application. Concurrent tasks have to communicate with each other and they should be able to agree on the use of shared resources. Furthermore, the MoC must capture the timing behavior of the tasks and allow analysis of the timing behavior of the application. This makes it possible to verify whether the timing constraints imposed on the application are satisfied. Finally, the MoC should allow a natural description of the application in the model. Multimedia applications are typically streaming applications. The class of dataflow MoCs fits well with this behavior. A dataflow program is specified by a directed graph where the nodes (called **actors**) represent computations that communicate with each other by sending ordered streams of data-elements (called **tokens**) over their edges. The execution of an actor is called a **firing**. When an actor fires, it consumes tokens from its input edges, performs a computation on these tokens and outputs the result as tokens on its output edges. In this thesis, the **Synchronous Dataflow** (SDF) MoC is used to model streaming applications. Actors in an SDF graph (SDFG) consume and produce a fixed amount of tokens on each firing. This makes it possible to analyze the throughput [46] and latency [47, 128] of these graphs. An example of an SDFG modeling an H.263 decoder is shown in Figure 1.5. Every of the four actors performs part of the frame decoding. The frame decoding starts in the actor VLD and a complete frame is decoded when the data is processed by actor Motion Comp. (motion compensation). Data that must be preserved between subsequent firings of an actor is modeled with an initial token on the self-edges of the actors. The partially decoded data is communicated via the edges at the bottom (left-to-right). The edges at the top (right-to-left) model the storage-space constraints on the bottom edges. This shows another important property of SDFGs. They allow modeling of many mapping decisions in the graph [13]. This enables analysis of the timing behavior of the application under these design decisions.

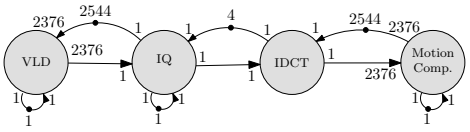


Figure 1.5: SDFG of an H.263 decoder.

When building a predictable system, a predictable platform should be used. Furthermore, a MoC should be used which allows analysis of the timing behavior of an application under certain mapping decisions. The third aspect that should be considered when designing a predictable system is the design flow that maps an application to a platform. This design flow should reserve sufficient resources in the platform such that the application can meet its timing-constraints when executed on the platform. A design flow that can determine this mapping is called a **predictable design flow**.

In this work, it is assumed that a platform with a predictable timing behavior is available. It is also assumed that applications are modeled as SDFGs. This allows timing analysis of an application and its mapping to the platform. The third element, a predictable design flow, is studied in this thesis. The next section sketches the steps needed in a predictable design flow. It also introduces the most important problems that must be solved in developing such a flow.

1.4 A Predictable Design Flow

This thesis presents techniques to map a time-constrained streaming application to a NoC-based MP-SoC. The objective is to minimize the resource usage (processing, memory, communication bandwidth) while offering guarantees on the throughput of the application when mapped to the system. A design flow that provides this guarantee is shown in Figure 1.6. The design flow consists of thirteen steps which are divided over four phases. This section introduces the various steps in the flow. The details of these steps can be found in the chapters mentioned in Figure 1.6. The motivation for the ordering of the steps in the flow can be found in Chapter 9.

The design flow assumes that the application that it has to map to the NoC-based MP-SoC is modeled with a **(streaming) application SDFG** with accompanying throughput constraint. The application SDFG specifies for every actor the required memory space and execution time for all processors on which this actor can be executed. It also gives the size of the tokens communicated over the edges. The NoC-based MP-SoC is described with a platform graph and an interconnect graph. The **platform graph** describes all resources in the SoC except that it abstracts from the NoC interconnect. The NoC resources are captured in

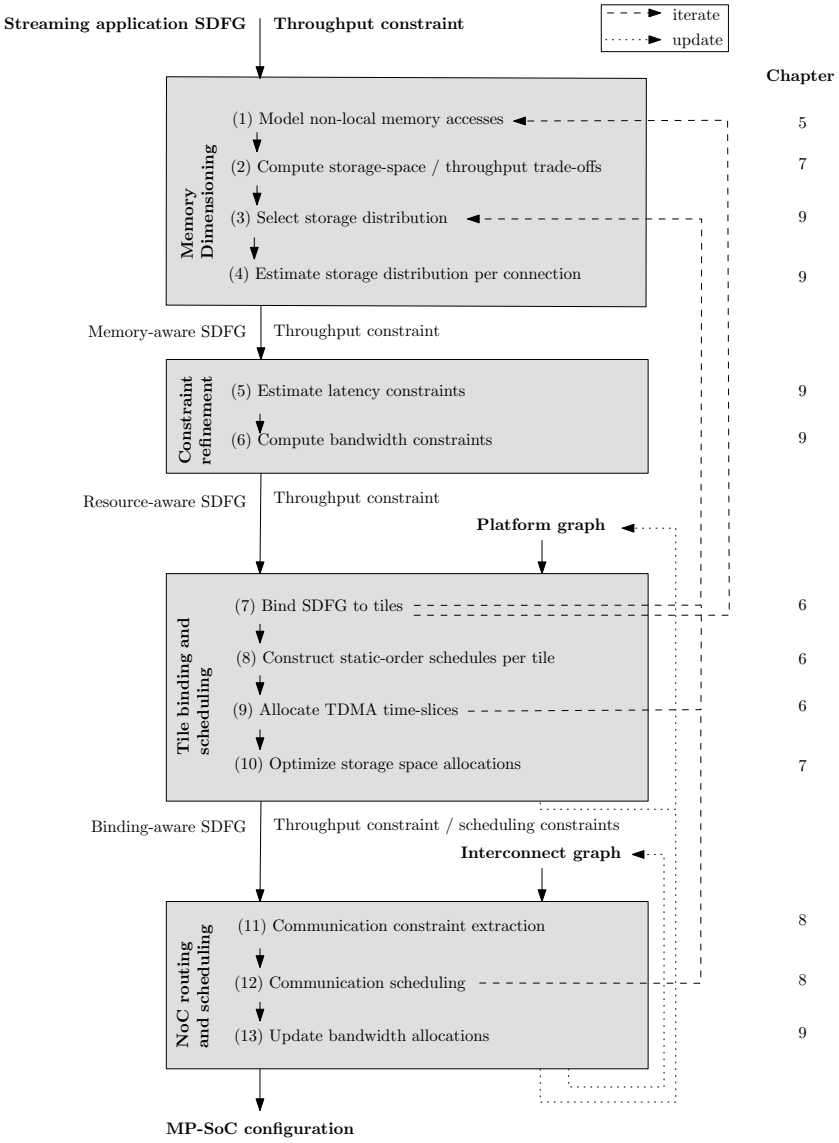


Figure 1.6: SDFG-based MP-SoC design flow.

the **interconnect graph**. The result of the design flow is an **MP-SoC configuration** which specifies a binding of the actors to the processors and memories, a schedule for the actors on the processors and a schedule for the token communication over the NoC.

Tokens that are communicated over the edges of an application SDFG must be stored in memory. The allocation of storage space for these tokens is dealt with in the **memory dimensioning** phase. Some tokens might be too large to fit into the memory that is available inside the tiles executing the actors that process the data in these tokens. Such tokens should be placed in some remote tile which has sufficient memory space. The actors that use these tokens must then access the tokens over the NoC. These accesses to the NoC must be modeled in the SDFG to allow timing analysis of this design decision. The first step of the design flow identifies the tokens which should be stored in a remote tile. It also transforms the streaming application SDFG to model the design decision that these tokens are stored in this remote tile.

The streaming application has a throughput constraint that must be satisfied by the MP-SoC configuration. A major aspect influencing the achieved throughput is the storage space allocated to the edges of the graph. Allocating space for more than one token to an edge might increase throughput because it may increase pipelining opportunities. The size of the storage space must be chosen such that the throughput requirement is met, while minimizing the required storage space. As the exact throughput constraint for parts of the system at various stages of the design is often unknown, a trade-off must be made between the realizable throughput and the storage requirements for an SDFG. The second step of the flow computes the throughput-storage space trade-off space. It finds all distributions of storage space that achieve the maximal throughput under a given total storage size constraint. One of these storage distributions is selected in the third step of the flow to limit the storage space of the edges in the application SDFG. The source and destination actor of an edge might in the end be bound to different tiles. In that case, the storage space allocated to an edge has to be split over both tiles. Step 4 estimates the storage space that should be allocated in both tiles to meet the throughput constraint. The SDFG that results from the first phase of the design flow is called a **memory-aware SDFG**. It models access to tokens stored in a remote memory and it constrains the storage space of the edges in the graph.

In the SDF MoC, it is assumed that edges have an infinite bandwidth and no latency. In other words, communication of tokens over an edge takes no time. Edges whose source and destination actors are bound to different tiles will be bound to a connection in the NoC. This connection has a latency and finite bandwidth. The **constraint refinement** phase of the design flow estimates the maximal latency (step 5) and minimal bandwidth (step 6) needed for edges when bound to a connection such that the throughput constraint is met. The resulting SDFG is called a **resource-aware SDFG**.

The constraints computed in steps 3 through 6 are used to steer the resource

allocation that takes place in the third phase of the flow. This **tile binding and scheduling** phase binds actors and edges from the resource-aware SDFG to the resources in the platform graph (step 7). When a resource is shared between different actors or applications, a schedule should be constructed that orders the accesses to the resource. The accesses from the actors in the resource-aware SDFG to a resource are ordered using a static-order schedule. This schedule is constructed in step 8 of the flow. Step 9 allocates TDMA time slices on all resources that are used by the resource-aware SDFG. These TDMA time slices provide virtualization of the resources to the application.

The storage space allocated to the edges of the resource-aware SDFG could be too large when the mapping decisions made in step 7 through 9 are considered. Step 10 computes the throughput-storage space trade-off space of the resource-aware SDFG considering the binding and scheduling decisions taken so far. The storage space allocations of the edges are then updated based on the smallest storage space allocation from this trade-off space that satisfies the throughput constraint.

The third phase of the design flow, binds and schedules the actors to the resources of the platform. This phase does not consider the scheduling of the communication on the NoC. This problem is considered in the **NoC routing and scheduling** phase of the design flow. The actor bindings and schedules impose timing-constraints on the communication that must be met when constructing a communication schedule. These timing constraints are extracted from the binding-aware SDFG in step 11. Next, the communication is scheduled onto the NoC. The objective of the NoC scheduling is to minimize the resource usage while satisfying all timing constraints. Based on the actual bandwidth usage of the communication schedule, the resource availability in the platform and interconnect graph can be updated. This is done in step 13 of the design flow. The updated graphs can then be used to map another application on the same platform while considering the resources used by the already mapped application(s).

The mapping of the streaming application to the NoC-based MP-SoC may fail at various steps of the design flow. This may occur due to lack of resources (step 7) or infeasible timing constraints (step 9 and 12). In those situations, the design flow iterates back to the first or third step of the flow and design decisions made in those steps are revised. When going back to step 1, more or different tokens should be placed in a memory that is accessed over the NoC. Reverting back to step 3 implies that the storage space allocated to the edges is too constrained for meeting the throughput constraint. So, a different storage distribution should be chosen.

The design process ends as soon as a mapping of the application to the NoC-based MP-SoC is found that satisfies the throughput constraint or till all storage distributions from the space found in step 2 are tried unsuccessfully. In the latter case, the design flow is not able to find an MP-SoC configuration that satisfies the throughput constraint. More resources should be added to the platform or the application and its constraint should be modified in order to find an MP-SoC

configuration that meets the throughput constraint.

1.5 Contributions

This thesis makes several contributions to develop a predictable design flow as sketched in the previous section.

- An SDF model is presented that allows reasoning about the timing behavior of an actor which uses data stored in a memory that is accessed over an interconnect (Chapter 5). An earlier version of this work was published in [135, 136].
- A cost-function driven heuristic algorithm is proposed for binding and scheduling a throughput-constrained SDFG on the tiles of a multi-processor system (Chapter 6). This work was published in [132].
- An efficient technique is presented to calculate the throughput of a bound and scheduled SDFG (Chapter 6). This work was published in [132].
- An algorithm is presented to compute the trade-off space between storage-space allocation for the edges of an SDFG and the maximal throughput that can be realized under these storage constraints (Chapter 7). This work was published in [137].
- Several routing and scheduling algorithms for mapping time-constrained communication on a NoC are presented. These algorithms minimize resource usage by exploiting all scheduling freedom offered by NoCs while guaranteeing that the timing constraints are met (Chapter 8). This work was published in [134] and an extended version is published in a special issue of the Journal of Systems Architecture on the best papers of the Digital System Design conference [133].
- A design-flow is proposed that maps a throughput-constrained application modeled with an SDFG onto a NoC-based MP-SoC (Chapter 9).
- The SDF^3 tool-kit implements all techniques presented in this thesis, the predictable design-flow and existing SDFG analysis and visualization techniques, as well as a graph generator (Chapter 4). An earlier version of this work has been published in [138].

1.6 Thesis Overview

This thesis is organized as follows. The next chapter discusses the characteristics of modern streaming multimedia applications. It considers both the modeling of these applications as dataflow graphs as well as the estimation of their resource

requirements. Chapter 3 presents the NoC-based MP-SoC platform template assumed in this thesis with its scheduling strategies. These scheduling strategies make sure that the platform can provide timing guarantees to individual applications when these applications reserve resources from the platform. The synchronous dataflow model is introduced in Chapter 4. This chapter discusses also existing techniques for analyzing the throughput of an SDFG and scheduling it on single and multi-processor systems. Chapter 5 presents an SDF model that allows reasoning about the timing behavior of an actor that uses data stored in a memory that is accessed over the NoC. A technique to bind and schedule an SDFG to the resources of an MP-SoC is presented in Chapter 6. The minimal storage-space for the edges of an SDFG that must be allocated by this binding and scheduling technique can be computed using the algorithm presented in Chapter 7. This algorithm can compute the complete trade-off space between the storage-space and throughput of an SDFG. The resource allocation technique presented in Chapter 6 does not construct a schedule for the communication on the NoC. Several scheduling techniques for this problem are presented in Chapter 8. All techniques presented in this thesis are embedded into a coherent and complete design flow in Chapter 9. A case study is performed in Chapter 10 that maps a set of multimedia applications (H.263 encoder/decoder and an MP3 decoder) onto a NoC-based MP-SoC. Finally, Chapter 11 concludes this thesis and gives recommendations for future work.

Chapter 2

Streaming Multimedia Applications

2.1 Overview

This chapter gives an overview of the main characteristics of streaming multimedia applications. Section 2.2 introduces the application domain and its most widely used applications. The properties that should be captured when modeling these applications are discussed in Section 2.3. To perform timing analysis and resource allocation, properties like the execution time and required memory space need to be extracted from an application. Section 2.4 discusses techniques to extract these requirements from the source code of applications.

2.2 Application Domain

Multimedia applications constitute a huge application space for embedded systems. They underlie many common entertainment devices, for example, cell phones, digital set-top boxes and digital cameras. Most of these devices deal with the processing of audio and video streams. This processing is done by applications that perform functions like object recognition, object detection and image enhancement on the streams. Typically, these streams are compressed before they are transmitted from the place where they are recorded (**sender**) to the place where the are played-back (**receiver**). Applications that compress and decompress audio and video streams are therefore among the most dominant streaming multimedia applications [152].

The compression of an audio or video stream is performed by an encoder. This encoder tries to pack the relevant data in the stream into as few bits as possible. The amount of bits that need to be transmitted per second between the sender and receiver is called the bit-rate. To reduce the bit-rate, audio and video encoders usually use a lossy encoding scheme. In such an encoding scheme, the encoder removes those details from the stream that have the smallest impact on

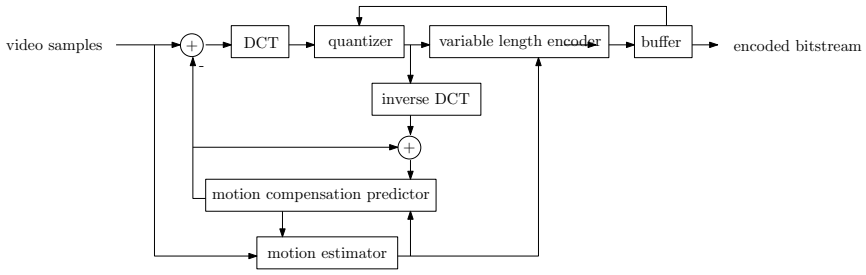


Figure 2.1: Structure of a video encoder [152].

the perceived quality of the stream by the user that has to listen to it or look at it. Typically, encoders allow a trade-off between the perceived quality and the bit-rate of a stream. The receiver of the stream must use a decoder to reconstruct the received audio or video stream. After decoding, the stream can be output to the user, or additional processing can be performed on it to improve its quality (e.g., noise filtering), or information can be extracted from it (e.g., object detection). There exist a number of different audio and video compression standards that are used in many embedded multimedia devices. The remainder of this section discusses the most commonly used standards.

Popular video coding standards are MPEG-2 [96] (e.g., used for DVD) and its successor MPEG-4 [97]. These video coding standards focus on high resolution video streams. The H.263 [70] and H.264 [71] coding standards focus on the mobile domain. These standards are meant for video compression of streams with low resolutions. The basic structure of the encoder used in all four video compression standards is shown in Figure 2.1. All compression standards use a discrete cosine transformation (DCT) to identify information in a frame that can be removed. The DCT separates the information into spatial frequencies. The higher spatial frequencies represent finer details that could be eliminated first. This elimination of details occurs in the quantizer. Depending on the required quality level, more or less details are removed from a frame. Motion estimation and compensation are also used in all coding standards to reduce the bit-rate of the compressed stream. Motion estimation compares part of one frame to a reference frame and determines the distance over which the selected part is shifted in the frame with respect to the reference frame. A motion compensator uses this motion vector to reconstruct the frame. The disadvantage of using a motion estimator is that the sender must hold a reference frame in its memory. However, the use of motion estimation and compensation greatly reduces the bit-rate. Motion compensation is performed both in the decoder and in the encoder. The encoder constructs, by performing motion compensation, the exact same frame as the decoder will construct. This frame is then used as a reference by the motion estimator to

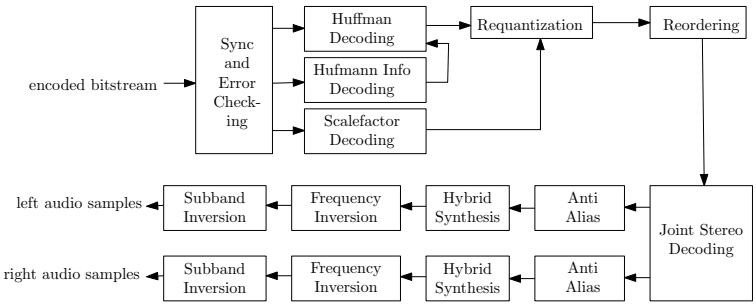


Figure 2.2: Structure of an MP3 decoder [126].

compute the motion vectors for the next frame. To further reduce the bit-rate, all standards use on top of the lossy encoding a loss-less encoding scheme. For this purpose, Huffman encoding or variable length encoding are typically used. These encoding techniques remove entropy from the final data stream that is sent to the decoder.

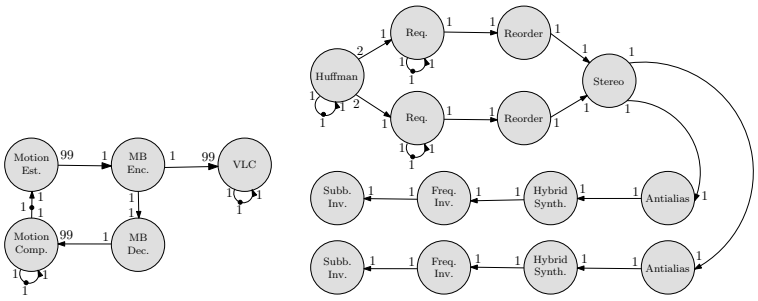
Audio streams are often compressed using the MPEG-1 Layer 3 (MP3) standard [83] or the Ogg Vorbis standard [147]. MP3 uses a combination of subband coding and a psychoacoustic model to compress the audio stream. The psychoacoustic model relies on the characteristics of the human hearing system. For example, when the human ear hears one tone, followed by another tone at a nearby frequency, the second tone cannot be heard for some interval. This effect is called masking and it is exploited in the psychoacoustic mechanism of the MP3 encoder. This allows the encoder to eliminate masked tones to reduce the amount of information that is sent to the decoder. The structure of an MP3 decoder is shown in Figure 2.2. It reverses the operations performed by the encoder in order to reconstruct the audio stream. The 'Sync and Error Checking' block finds a starting point in the bitstream and it checks the received stream for errors due to the data transmission. The 'Huffman' block uncompresses the bitstream using a Huffman decoder. The coefficients required for this operation are supplied by the 'Huffman Info Decoding' block. The output of the Huffman decoder is a stream of audio samples. The MP3 standard groups these samples into frames of 576 samples that are processed together in the remainder of the decoder. After the Huffman decoding, a frame is requantized. In this process, each audio sample inside the frame is multiplied with a coefficient that is supplied by the 'Scalefactor Decoding' block. Next, the audio samples inside a frame are reordered and an inverse discrete cosine transformation is applied. The latter operation is performed in the 'Joint Stereo Decoding' block. The remainder of the decoder consists of a series of audio filters that remove noise inserted into the audio stream by the operations performed in the encoder and decoder.

Ogg Vorbis is another audio compression standard that also uses a psychoacoustic model to compress the audio stream. Because it has been developed after the MP3 standard, the designers could use more sophisticated models. As a result, an Ogg Vorbis encoder can typically compress an audio stream with the same quality but with a lower bit-rate than an MP3 encoder.

2.3 Application Modeling

The design of a predictable system requires that the timing behavior of the application and its mapping to the platform can be analyzed. More precisely, it requires that the throughput and latency of an application when bound and scheduled on a platform can be predicted. Both the throughput and latency are influenced by the amount of buffering (memory space) that is used in the platform. Throughput is typically increased when larger buffers are used. Increasing buffer sizes may also reduce the variation in the time needed to process data items, which may lead to a reduced latency. However, an increase in buffer sizes means that more memory is needed in the platform. As a result, the energy consumption of the system will increase. From an energy perspective, it is important to minimize the memory usage and thus the buffering. When designing an embedded multimedia system, the designer should try to minimize the required buffering while meeting the throughput and latency constraints. Throughput is often the most important timing aspect when designing systems for streaming multimedia applications. Therefore, this thesis focuses on analyzing throughput and buffer requirements of an application and its mapping to a platform. This requires that the application and mapping decisions are modeled in a Model-of-Computation (MoC) that allows throughput and buffer analysis. The MoC that should be used depends on the features of the application that are relevant for building a predictable system. The Synchronous Dataflow (SDF) MoC has been selected for this purpose. A detailed motivation for this choice and a comparison between SDF and other MoCs can be found in Section 4.7. Essentially, the reason for this choice is that the SDF MoC provides a good compromise between expressiveness, modeling ease, analysis potential, and implementation efficiency. Modeling ease refers to the aspect that a MoC should allow modeling of applications from the targeted application domain in a natural and straightforward manner. This section discusses the application characteristics that are the most important when modeling streaming multimedia applications. It also explains how an application can be modeled with an SDF graph (SDFG) and it discusses the limitations of the SDF MoC.

The previous section shows that the dominant applications from the targeted application domain operate on streams of data. The SDF MoC can model streaming data in a natural way. In this MoC, an application is modeled with a directed graph where the nodes (called **actors**) represent computations (tasks) that com-



(a) H.263 decoder. (b) MP3 decoder.

Figure 2.3: SDFG model of applications.

municate with each other by sending ordered streams of data-elements (called **tokens**) over their edges. The graph makes the parallelism that is available in an application explicit. This is important when mapping the application onto a multi-processor system, which inherently offers parallelism. The execution of an actor is called a **firing**. When an actor fires, it consumes tokens from its input edges, performs a computation on these tokens and outputs the result as tokens on its output edges. The number of tokens that are consumed and produced on a firing are fixed in the SDF MoC and called the consumption and production **rates**. The rates attached to the source and destination of an edge can be different in the SDF MoC. This makes it possible to express multi-rate dependencies between the actors (tasks) in an application. This is important as different tasks in a multimedia application may operate at a different granularity, as illustrated with the examples discussed in the remainder of this section.

Figure 2.1 shows the structure of an H.263 encoder. An H.263 encoder divides a frame in a set of macro blocks (MBs). A macro block captures all image data for a region of 16 by 16 pixels. The image data inside a MB can be subdivided into 6 blocks of 8 by 8 data elements. Four blocks contain the luminance values of the pixels inside the MB. Two blocks contain the chrominance values of the pixels inside the MB. A frame with a resolution of 174 by 144 pixels (QCIF) contains 99 MBs. These 99 MBs consist, in total, of 594 blocks. An H.263 encoder that encodes frames with QCIF resolution can be modeled with the SDFG shown in Figure 2.3(a). The motion estimator block shown in Figure 2.1 is modeled in this SDFG with the Motion Est. actor. The motion compensation predictor block and the variable length encoder block are modeled with the Motion Comp. and VLC actors. The MB decoding (MB Dec.) actor models the inverse DCT block. The DCT and quantizer block are modeled together in the MB encoding (MB Enc.) actor. Buffering of the encoded bitstream is not taken into account in the SDFG. Therefore, the buffer block is not modeled in the graph. The motion estimation, motion compensation and variable length encoding blocks operate on a complete

video frame (i.e., 99 MBs). The other blocks in Figure 2.1 process a single MB at a time. These different processing granularities are modeled with the fixed rates in the SDFG of Figure 2.3(a). This application model shows that SDFGs are well-suited for modeling multi-rate behavior. The self-edges on the Motion Comp. and VLC actors model that part of the data that is used by these actors during a firing must be stored for a subsequent firing. In other words, these self-edges model the global data that is kept between executions of the code segments that are modeled with these actors.

Another example of an application modeled with an SDFG is shown in Figure 2.3(b). This SDFG models the MP3 decoder application shown in Figure 2.2. Most tasks shown in the block diagram of Figure 2.2 have a corresponding actor in the SDFG. The 'Sync and Error Checking', 'Huffman Info Decoding' and 'Scalefactor Decoding' from the block diagram have no corresponding actor in the SDFG. These blocks represent functionality of the decoder that is executed only at the beginning of the stream. The data produced by these blocks is used to initialize the 'Huffman decoding' and 'Requantization' blocks and is needed during the whole execution of the MP3 decoder (i.e., it is global data). The self-edges on the Huffman and Req. actors model the storage of this data between subsequent firings of these actors. This is similar to the self-edges in the H.263 encoder. All actors in the SDF MoC are executed periodically. This implies that an initialization phase, as occurs in the MP3 decoder, cannot be modeled. Therefore, these three blocks are not modeled in the SDFG of the MP3 decoder. However, these blocks do represent part of the application that does take time when executed. It might therefore be necessary to take this time into account when analyzing the throughput of the SDFG in order to get a conservative estimate of the throughput of the application when executed on a platform at any point in time, including the initialization. This can be done by adding the execution time of the code that corresponds to the 'Sync and Error Checking' and 'Huffman Info Decoding' blocks to the execution time of the code segment that is modeled with the Huffman actor. Similarly, the execution time of the 'Sync and Error Checking' and 'Scalefactor Decoding' blocks can be added to the execution time of the Req. actor. Alternatively, the execution time of these blocks can be ignored as these blocks are only executed during the start-up phase of the MP3 decoder. They do not influence the execution time of the application in the long-run.

The blocks in the block diagram of Figure 2.2 consume and produce data in blocks of 576 (audio) samples. One such a block of samples is called a frame. Each frame can be modeled with a token in the SDFG and the actors produce one token on each firing. These rates are annotated to the edges of the SDFG shown in Figure 2.3(b). The Huffman decoder blocks in the MP3 decoder of Figure 2.2 always operate on two frames at the same time, while all other blocks process one frame at a time. This is correctly modeled with the fixed rates in the SDFG of Figure 2.3(b). Many audio and video codecs can accurately be modeled with fixed rates as these codecs do not have a very dynamic, data-dependent behavior. Therefore, the rates (and also often the actor execution times) can be upper-bounded

without a large over-estimation. This over-estimation cannot be avoided when designing the system for its worst-case behavior. However, the dynamism in the application may lead to a resource allocation that is too conservative for some situations [49, 99]. The solution to prevent over-allocation of the resources is to split the application over multiple scenarios. Each scenario represents a behavior of the application for a constrained range of the rates and execution times. Identification of different scenarios can be done using the technique described in [49, 50]. An SDFG can then be constructed for each scenario and this SDFG can subsequently be mapped onto a platform using the techniques described in this thesis. However, there are two known issues with this approach. First, the design flow presented in this thesis deals with a single SDFG at a time. When mapping multiple SDFGs of different scenarios, the flow should be extended to take the relations between the actors in the different graphs into account. Second, run-time switching from one SDFG implementation to another has not yet been (extensively) studied. A switch may involve aspects such as task migration, memory reallocation, and/or NoC reconfiguration. It is currently not possible to analyze the timing behavior when switching between graphs. Extending the design flow to deal with multiple SDFGs that model different scenarios of the same application is an important direction of future work.

2.4 Analyzing Actor Resource Requirements

A design flow must allocate resources for the streaming applications that it maps onto a platform. To do this, the design flow needs information on the resource requirements of the application being mapped. The application is modeled with an SDFG. The actors in an SDFG communicate by sending tokens between them. Memory space is needed to store these tokens. The amount of tokens that must be stored simultaneously is determined by the design flow (see Chapter 9). However, the design flow must know how much memory space is needed for a single token. This is determined by the number of bytes of the data type that is communicated with the token. This information can easily be extracted through a static-code analysis. Some information is also needed on the resource requirements of the actors. These actors represent code segments of the application. To execute a code segment, processing time is needed as well as memory space to store its internal state. The internal state contains all variables that are used during the execution of the code segment but that are not preserved between subsequent executions of the code segement. Global data that is used inside a code segment is not considered part of the internal state. The SDF MoC requires that global data that is used by a code segment is modeled explicitly with a self-edge on the actor that models this code segment (see Figure 2.3(b) for an example). This self-edge contains one initial token whose size is equal to the size of the global data used in the code segment. The maximal size of the internal state is determined by the worst-case stack size and the maximal amount of memory allocated

Actor	Worst-case execution time [cycles]
Huffman	151977
Req.	72695
Reorder	34684
Stereo	53602
Antialias	409
Hybrid Synth.	7414
Freq. Inv.	4912
Subb. Inv.	1865001

Table 2.1: Worst-case execution times MP3 decoder on ARM7.

on the heap. In many embedded applications, no dynamic memory allocation is used. So, the memory space requirement for the internal state of an actor can be found by analyzing its worst-case stack size. The amount of processing resources that needs to be allocated to an actor depends on the throughput constraint of the application. To provide throughput guarantees for a mapped application, the design flow must analyze the throughput of the application using the worst-case execution times for the actors. Using these worst-case execution times, the design flow can compute a conservative bound on the actual throughput that is achieved when the application is executed on the platform. The worst-case execution times of the actors must be extracted from the code segments that are modeled with these actors before the design flow is started.

Tools are available to analyze the worst-case stack size and the execution time of applications [60, 63]. These tools take an application as input and analyze its memory and execution time requirements. An actor in an SDFG models a code segment from a full application. So, these tools cannot directly be used to analyze the requirements of the actors in an SDFG. It requires that each code segment that is modeled with an actor is isolated from the application code. The separate code segments can then be analyzed for their worst-case stack size and execution time. Most worst-case stack size and execution time analysis tools support only analysis of one or a few different processor types. The predictable design flow presented in this thesis targets a heterogeneous multi-processor platform which may contain many different processor types. Therefore, to complement existing tools, an analysis tool has been developed that can easily be reconfigured for a large number of different processor types [51]. The tool, called CTAP, supports ANSI-C as this is the main programming language for embedded systems. It uses the observation that the time-bound on a C statement that does not affect the control flow of an application is equal to the sum of the time-bounds on the assembly instructions it is translated to. The tool assumes that the number of cycles (time) needed to execute each assembly instruction can be upper bounded, an assumption valid for many processors used in the embedded domain. This im-

plies a very simple and portable architecture model. Only two things are needed: a mapping of each assembly instruction onto a constant delay and a translation of the source code to assembly instructions of the target processor. The former can be obtained from the processors datasheet, the latter using a compiler. Once CTAP has computed the bound on the C statements of an application, it uses static code analysis to compute the worst-case execution time of the whole application. Using CTAP, the worst-case execution times of the actor in the MP3 decoder SDFG (see Figure 2.3(b)) have been computed. Table 2.1 shows these execution times assuming that the actors are mapped to an ARM7 processor type [8]. The static code analysis technique that is used in CTAP to compute the worst-case execution time can also be used to find the worst-case stack size of actors. A first prototype tool exists to compute these worst-case stack sizes.

2.5 Summary

This chapter gives an overview of the most important characteristics of multimedia applications. These applications are composed of a set of communicating tasks that operate on streams of data. There are often multi-rate dependencies between these tasks. The most important constraints when designing a system that runs a multimedia application are the throughput realized by the application when mapped to a platform and the buffering required for the data communicated between the tasks of the application. The SDF MoC allows to analyze these constraints, to model the parallelism between the tasks of the application and to model the multi-rate behavior of the tasks. This chapter explains how an application can be modeled with an SDFG. It also briefly presents a tool to analyze the worst-case resource requirements of the tasks inside the application. A predictable design flow needs this information to allocate sufficient resources for the application such that it can provide throughput guarantees.

Chapter 3

Network-on-Chip-based Multi-Processor Platform

3.1 Overview

This chapter introduces a predictable platform template. It combines a Network-on-Chip-based Multi-Processor-System-on-Chip (NoC-based MP-SoC) platform with a resource scheduling strategy that can provide the required timing guarantees. Section 3.2 presents the template of this platform. Various scheduling strategies are compared in Section 3.3 with respect to several properties that are important for a predictable platform. Section 3.4 discusses the use of these scheduling strategies in the predictable platform.

3.2 Multi-Processor Platform Template

The multi-processor platform template that is used in this thesis is shown in Figure 3.1. Multi-processor systems like Daytona [3], Eclipse [125], Hijdra [18], and StepNP [111] fit nicely into this template. The template is based on the tile-based multi-processor platform described by Culler et al. in [30]. It consists of multiple tiles connected with each other by an interconnection network. Each **tile** contains a set of communication buffers, called the network interface (NI), that are accessed both by the local processing elements inside the tile and by the interconnect. The NI is also responsible for packatization of data send over the interconnect. A tile has also a small controller, called the communication assist (CA), that performs accesses to the local memory (M) on behalf of the NI. It decouples the communication and computation, allowing a better resource utilization. The CA acts also as the memory access arbiter, i.e., it decides when memory request from the local processor and NI are granted access to the local

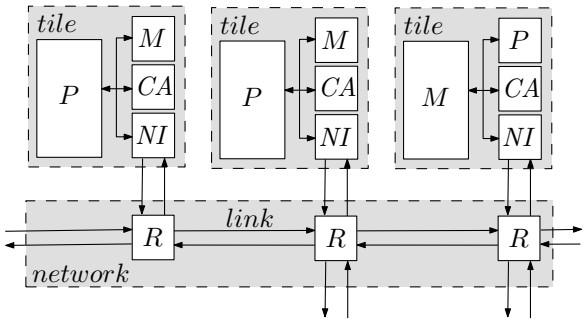


Figure 3.1: Multi-processor template.

memory.

Two different types of tiles are distinguished based on the functionality of the processing element (P) inside the tile and the size of the memory (M). The first type, called **processing tile**, contains a processor which executes the code of the applications running on the platform. The application code and some of the data structures needed when executing it are stored in the local memory of the tile. Multimedia applications may require access to large data structures. A video processing application may for example require access to a complete video-frame. A single frame at HDTV resolution consists of over two million pixels each using three bytes of memory. The memories in a processing tile will typically not be large enough to store this amount of data [116]. Therefore a second type of tiles that contains large memories is included in the architecture template. These tiles are called **memory tiles**. A memory tile contains besides a large memory also a small processing element (P) that schedules the request from multiple, different processing tiles to access its local memory. From the memory's perspective, only the local NI and P try to access this memory. Arbitration between these two elements is performed by the local CA.

The interconnect between the tiles in the platform template should offer unidirectional point-to-point connections between pairs of NIs. In a predictable platform, these connections must provide guaranteed bandwidth, and a tightly bounded propagation delay per connection - i.e. they must provide a **guaranteed throughput**. The connections must also preserve the ordering of the communicated data. A Network-on-Chip (NoC) can provide all these properties. The NoC consists of a set of routers which are connected to each other in an arbitrary topology. Each tile is connected through its NI with a router (R) in the NoC. The connections between routers and between routers and NIs are called links. Examples of NoCs providing the required properties are Æthereal [122] and Nostrum [93].

3.3 Resource Arbitration Strategies

Resources in the NoC-based MP-SoC platform may have to be shared between tasks (i.e. actors in case of an SDFG) of one or more applications that are executing on the platform. This may lead to resource conflicts when multiple tasks want to access a resource at the same time. A resource arbitration strategy should resolve these conflicts. This requires that the accesses of tasks to a shared resource are ordered (scheduled) in time.

To allow analysis of the timing behavior of applications, a predictable platform must provide a guarantee on the maximum amount of time between the moment that the task is ready to be executed and the completion of its execution. This amount of time is called the **worst-case response time** (WCRT) of a task. A predictable design flow has to consider the worst-case response times of tasks that form an application when it analyzes the timing behavior of this application when mapped to the platform. From a cost perspective (i.e. area, energy), the resource usage should be minimized. Therefore, a resource arbitration strategy should try to minimize the worst-case response time. This reduces the over-allocation of resources needed for an application to meet its timing constraints.

When building a predictable system, it is also important that the timing behavior of one application is not influenced by other applications running on the same platform. This enables analysis of the timing behavior of applications in isolation. A resource arbitration strategy that allows this is called **composable** [81]. Another important aspect that needs consideration when comparing resource arbitration strategies is the **flexibility** of a strategy to handle potentially changing dependencies between tasks of an application. A strategy is said to be flexible when it can deal with dynamically changing dependencies between tasks. In this section, three commonly used resource arbitration strategies (static-order, round-robin and time-division multiple-access) are compared on their worst-case response time, composability and flexibility.

Static-order. Following [100], a static-order schedule for a set T of tasks (i.e. actors of potentially different SDFGs) is defined as a finite or infinite sequence $t_1 t_2 t_3 ... \in T$. Practical infinite static-order schedules consist of a (possible empty) sub-sequence which is seen once followed by a finite sub-sequence which is infinitely often repeated. A resource arbitration strategy that uses a static-order schedule starts with waiting till the first task in the sequence is ready to execute. After executing this task, the scheduler executes the next task in the sequence if it is ready, or it waits till the task becomes ready to be executed. Once this task is executed, it continues with the next task. This process is repeated till the schedule is finished or indefinitely in the case of an infinite schedule.

A static-order schedule is not flexible nor composable as it can only be constructed when all dependencies between the tasks are fixed and known. Dependencies that are not taken into account can lead to a deadlock of the schedule. For example consider a task t_2 which is scheduled after another task t_1. If for

some reason at some point t_1 needs data of the execution of t_2 that is scheduled after t_1, the schedule will never make progress. So, the worst-case response times of the tasks in a deadlocked schedule are infinite. When deadlock is avoided, the worst-case response time depends on the moment a task becomes ready to execute and the time it takes till all other tasks which are scheduled before this task finish their execution.

In an SDFG, all dependencies between actors (tasks) are specified in the graph. A static-order schedule can exploit the fact that the dependencies between tasks are known in advance. Tasks can be ordered in such a way that the waiting time of tasks is minimized. Furthermore, a static-order schedule only waits when the task that should be executed next is not ready yet. Minimizing both the waiting time of actors and the waiting time in the schedule limits the amount of resource needed in the system to meet the timing constraints of an application. So, these aspects limit the over-allocation of resources from the system.

Round-robin. A round-robin schedule uses, similar to a static-order schedule, a list of ordered tasks. The difference between the two strategies is the order in which tasks from the list are executed. The round-robin scheduler repeatedly checks all tasks in the list in the specified order. When a task is ready to execute at the moment it is checked, its execution is started. After completion of the execution, or when the task is not ready to execute, the scheduler continues checking the next task in the list. This avoids that a round-robin scheduler deadlocks and it reduces waiting time due to the schedule. This gives the scheduling strategy the required flexibility to handle tasks (e.g., applications) for which the order of execution or data dependencies are not known when constructing the schedule.

The worst-case response time of a task t from a set T of tasks occurs when t becomes enabled directly after it is checked by the scheduler and all other tasks in the round-robin schedule are executed before t is checked again. The execution of t is then only completed after all tasks from the list are executed. The worst-case response time is thus equal to:

$$WCRT(t) = \sum_{t_i \in T} \tau(t_i), \tag{3.1}$$

where $\tau(t_i)$ is the execution time of task t_i.

The response time of a task depends on the execution time of all tasks in the round-robin schedule. When a task is added to the schedule, the response time will increase. This scheduling technique is therefore not composable.

TDMA. A time-division multiple-access (TDMA) scheduler uses the concept of a periodically rotating time wheel w. A time slice ω_i, a fraction of the wheel w, is allocated to each task t_i from the set T of tasks (see Figure 3.2). The scheduler executes t_i when this task is ready to be executed and the scheduler is within the time slice ω_i. When the end of the time slice is reached and a task has not

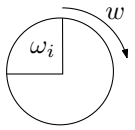


Figure 3.2: TDMA time wheel.

finished its execution, the execution is pre-empted and continued during the next period.

The worst-case response time of a task t_i to which a slice ω_i is allocated on a time wheel with size w occurs when the task becomes enabled directly after the end of the slice. The execution of t_i is then started after the non-reserved portion of the time wheel has passed, i.e. it is started after $w - \omega_i$ time units. The execution of the task ends when the total elapsed time that is reserved on the time wheel is equal to the task's execution time ($\tau(t_i)$). Note that this may require multiple iterations of the time wheel. This worst-case response time for the ready task t_i is given by:

$$WCRT(t_i) = \left\lceil \frac{\tau(t_i)}{\omega_i} \right\rceil \cdot w \tag{3.2}$$

The equation shows directly that there is no dependency between the response time of a task and the other tasks executed in the same TDMA schedule. This implies that new tasks can be added to the TDMA schedule, as long as there are unreserved slices available on the time wheel, without affecting the worst-case response time of the already scheduled tasks. The TDMA scheduling technique is therefore composable.

Table 3.1 compares the three resource arbitration strategies on the properties that are important when designing a predictable platform. The column 'over-allocation' gives for each strategy the over-allocation of resources relative to the other strategies. It assumes that in all cases a deadlock-free schedule is used that is optimized for the set of tasks that must be executed. The entries for the over-allocation of the round-robin and TDMA resource arbitration strategies require a small explanation. When TDMA scheduling is used on a resource, part of the time wheel may be unoccupied. This part can be used to schedule new tasks on the resource without affecting the worst-case response times of the already scheduled tasks. However, when part of a time wheel is unoccupied, the set of tasks that is scheduled on the resource using TDMA scheduling will have a larger worst-case then when round-robin scheduling is used to schedule the same set of tasks on the resource. To compensate for these larger worst-case response times, more resources may have to be allocated for TDMA scheduling as compared to round-robin scheduling. Therefore, TDMA resource arbitration leads to a (potentially) larger over-allocation then round-robin resource arbitration.

Table 3.1: Comparison of the resource arbitration strategies.

strategy	over-allocation	composable	flexible
static-order	small	no	no
round-robin	medium	no	yes
TDMA	large	yes	yes

3.4 Resource Arbitration in the Platform

Virtualization of the platform resources is important when designing a system with a predictable timing behavior. It enables analysis of the timing behavior of an application without the need to consider other applications that are simultaneously executing on the platform. The resource arbitration strategies that are used on the resources in the platform should provide this virtualization. At the same time, these strategies should avoid too much over-allocation of resources.

The resource arbitration strategies used for the different resources in the platform template are summarized in Table 3.2. Selection of the resource arbitration strategies is done considering the situation that the mapped applications are described with an SDFG. The dependencies between actors (tasks) of an SDFG are known in advance. This property can be exploited to reduce the allocation of resources needed for an application to meet its timing constraints. For this reason, some resources use a different strategies to schedule actors that belong to the same SDFG (application) and actors that belong to different applications. The choices for the different resource arbitration strategies on the resources of the platform are motivated in the remainder of this section.

The processors in the platform use a TDMA scheduler to order the execution of different applications. This guarantees that the timing behavior of one application is not influenced by other applications running on the same resources. A TDMA scheduler can also be used to schedule multiple actors of the same application on a single processor [17]. Time slices are then reserved for individual actors. This gives very conservative estimates on the worst-case response time of actors [128]. It assumes that two dependent actor executions, executed on the same processor, can only occur with a complete time wheel rotation in between. In practice, when a large enough time slice is reserved, it may never be needed to wait for this period of time. Throughput analysis performed using these worst-case response times may falsely suggest that an application cannot meet its throughput constraints. If throughput constraints are met, usually more resources than strictly needed are claimed. The timing behavior and resource usage is also negatively influenced by the fact that a time slice reserved for an individual actor cannot be used for other actors which belong to the same application. So, as an alternative to guarantee actor response times and at the same time obtain an efficient resource utilization, static-order scheduling is used between the actors of

Table 3.2: Resource arbitration in the NoC-based MP-SoC platform.

	within an application	between applications
Processor	static-order	TDMA
Memory	round-robin	round-robin
NoC	static-order	TDMA

an SDFG.

The accesses to the memory inside a tile are controlled by the CA. It uses a round-robin scheduler to arbitrate memory accesses that are requested by the processor and NI inside the tile. This resource arbitration strategy allows efficient usage of the memory resource. It avoids stalling when only the NI or processor needs access to the memory. This results in a good average case response time. It also has a worst-case response time which is smaller than the worst-case response time of a TDMA strategy. Using a static-order schedule is not possible as it is not known when the processor or NI need to access the memory.

Round-robin scheduling is not composable (see Table 3.1). To still obtain a platform that offers a composable behavior to the applications mapped on it, the worst-case access time to the memory should be taken into account in the execution times of the tasks (actors). When the resource requirements of a task are determined (e.g., using using the technique proposed in Section 2.4), the execution time of an instruction that accesses the memory is determined by the worst-case time needed on the processor to execute this instruction plus the worst-case time needed for the memory and its arbitration mechanism to complete the access.

The NoC uses a TDMA scheduling strategy to send streams of data from different applications between the tiles in the platform. This scheduling strategy virtualizes the communication resources. This makes it possible to provide guarantees on the bandwidth and latency of a connection through the NoC to an application. As such, it provides the required composability for mapping multiple applications to the NoC.

Dependencies between actors in an SDFG are known in advance. Furthermore, the actors of an SDFG have a predictable (worst-case) timing behavior when executed on the tiles of the platform. This makes it possible to derive worst-case time bounds on the availability of data for communication over the NoC and the maximal amount of time that a communication may take. Based on these time bounds, it is possible to derive a static-order schedule for all communication between the actors of an SDFG that takes place via the NoC. This static-order schedule has typically lower resource requirements than when a TDMA or round-robin schedule would be used to schedule the communication between actors of the same SDFG on the NoC resources.

3.5 Summary

This chapter introduces a NoC-based MP-SoC platform template that offers a predictable timing behavior. It consists of multiple tiles connected with each other by a NoC. Each tile contains both processing and storage resources. To provide timing guarantees, the resources inside the tiles and the NoC resources must use an arbitration mechanism that offers worst-case time bounds on its behavior. Several resource arbitration mechanisms are compared in this chapter on their worst-case behavior, composability and flexibility. Based on this comparison, resource arbitration mechanisms are selected for the various resources in the platform template.

Chapter 4

Dataflow Preliminaries

4.1 Overview

This chapter introduces terminology and definitions used in this thesis, and formalizes, in Section 4.2, the synchronous dataflow (SDF) model of computation that was introduced informally in the first chapter. A variant of the SDF model that takes time into account is presented in Section 4.3. Throughput analysis techniques for these timed SDF graphs (SDFGs) are discussed in Section 4.4. Existing scheduling techniques for SDFGs are presented in Section 4.5. A tool that implements most of the existing SDFG analysis and scheduling techniques is introduced in Section 4.6. A comparison between SDF and a number of related dataflow models of computation is made in Section 4.7. The section motivates also the choice made in this thesis to model applications with SDFGs.

4.2 Synchronous Dataflow

Let $\mathbb{N}$ denote the positive natural numbers, $\mathbb{N}_0$ the natural numbers including 0, and $\mathbb{N}_0^\infty$ the natural numbers including 0 and infinity (∞). Formally an SDFG is then defined as follows. Assume a set $Ports$ of ports; with each port $p \in Ports$ a finite rate $Rate(p) \in \mathbb{N}$ is associated.

Definition 1. (Actor) *An actor a is a tuple (I, O) consisting of a set $I \subseteq Ports$ of input ports (denoted by $I(a)$) and a set $O \subseteq Ports$ of output ports with $I \cap O = \emptyset$.*

Definition 2. (SDFG) *An SDFG is a tuple (A, D) consisting of a finite set A of actors and a finite set $D \subseteq Ports^2$ of dependency edges. The source of a dependency edge is an output port of some actor, the destination is an input port of some actor. The operator $SrcA$ ($DstA$) gives the source (destination) actor of a dependency edge and the operator $SrcP$ ($DstP$) gives the source (destination) port*

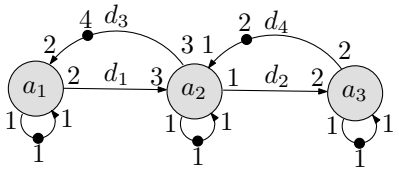


Figure 4.1: An example SDFG.

of a dependency edge. All ports of all actors are connected to precisely one edge, and all edges are connected to ports of some actor. For every actor $a = (I, O) \in A$, we denote the set of all dependency edges that are connected to the ports in I (O) by $InD(a)$ ($OutD(a)$).

Figure 4.1 shows an example of an SDFG with three actors a_1, a_2 and a_3. The actors a_1 and a_2 are connected to each other through the dependency edge d_1 and d_3. The edges d_2 and d_4 connect a_2 and a_3 to each other. The dependency sources and destinations are annotated with port rates. The execution of an actor is called a **firing**. When an actor a starts its firing, it removes $Rate(q)$ tokens from all $(p, q) \in InD(a)$. At the end of the firing, it produces $Rate(p)$ tokens on every $(p, q) \in OutD(a)$. The rates determine how often actors have to fire with respect to each other such that the distribution of tokens over all dependency edges is not changed. This property is captured in the repetition vector of an SDFG.

Definition 3. (Repetition vector and consistency) *A repetition vector of an SDFG (A, D) is a function $\gamma : A \to \mathbb{N}_0$ such that for every edge $(p, q) \in D$ from $a \in A$ to $b \in A$, $Rate(p) \cdot \gamma(a) = Rate(q) \cdot \gamma(b)$. A repetition vector γ is called non-trivial if and only if for all $a \in A$, $\gamma(a) > 0$. An SDFG is called consistent if it has a non-trivial repetition vector. The smallest non-trivial repetition vector of a consistent SDFG is called the repetition vector.*

The repetition vector of the SDFG shown in Figure 4.1 is equal to $(a_1, a_2, a_3) \to (3, 2, 1)$. This shows that the graph is consistent as the repetition vector is non-trivial. Consistency and absence of deadlock are two important properties for SDFGs which can be verified efficiently [23, 86]. Any SDFG which is not consistent requires unbounded memory to execute or deadlocks. When an SDFG deadlocks, no actor is able to fire, which is due to an insufficient number of tokens in a cycle of the graph. Any SDFG which is inconsistent or deadlocks is not useful in practice. Therefore, only consistent and deadlock free SDFGs are considered in this thesis.

There exists a special class of SDFGs in which all rates associated to ports equal 1. These graphs are called **Homogeneous Synchronous Data Flow Graphs** (HSDFGs) [86]. As all rates are 1, the repetition vector for an HSDFG associates 1 to all actors. Any consistent SDFG $G = (A, D)$ can be converted to an

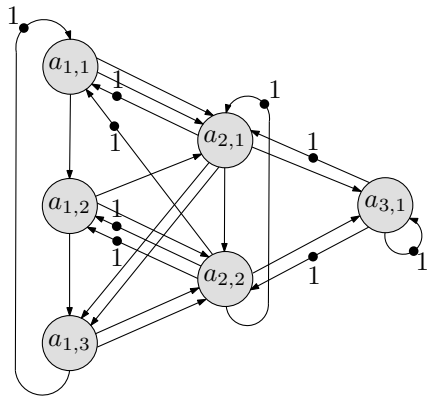


Figure 4.2: HSDFG of the example SDFG.

equivalent HSDFG $G_H = (A_H, D_H)$ [86, 128], by using the conversion algorithm in [128, Section 3.8]. Equivalence in this context means that there exists a one-to-one correspondence between the SDFG and HSDFG actor firings. Figure 4.2 shows the HSDFG that corresponds to the SDFG shown in Figure 4.1, with rates one omitted for readability. The number of copies of an actor a_i of the SDFG in the HSDFG is equal to its entry in the repetition vector. Therefore, three copies of actor a_1 are present in the HSDFG of Figure 4.2.

4.3 Synchronous Dataflow and Time

The firings of actors in an SDFG, as defined in the previous section, are atomic. There is no notion of time associated with it. Many system properties like throughput and latency require a notion of time to be associated with the firings of actors in an SDFG. A timed SDFG and its execution are defined as follows.

Definition 4. (Timed SDFG) *A timed SDFG is a triple* (A, D, Υ) *consisting of an SDFG* (A, D) *and a function* $\Upsilon : A \to \mathbb{N}$ *that assigns to every actor* $a \in A$ *the time it takes to execute the actor once.*

An example of a timed SDFG is shown in Figure 4.3. The execution time of each actor is denoted with a number in the actor. The execution of a timed SDFG is formalized through a labeled transition system. This requires appropriate notions of states and of transitions. To measure quantities related to dependency edges, such as the number of tokens present in, read from or written to edges, the following concept is defined.

Definition 5. (Edge quantity) *An edge quantity on the set* D *of dependency edges is a mapping* $\delta : D \to \mathbb{N}_0$*. If* δ_1 *is an edge quantity on* D_1 *and* δ_2 *is an edge*

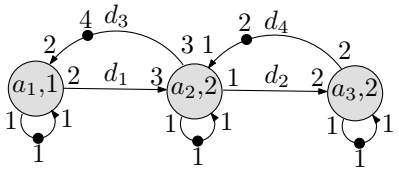


Figure 4.3: A timed SDFG example.

quantity on D_2 with $D_1 \subseteq D_2$, we write $\delta_1 \preceq \delta_2$ if and only if for every $d \in D_1$, $\delta_1(d) \leq \delta_2(d)$. Edge quantities $\delta_1+\delta_2$ and $\delta_1-\delta_2$ are defined by pointwise addition of δ_1 and δ_2 and subtraction of δ_2 from δ_1; $\delta_1 - \delta_2$ is only defined if $\delta_2 \preceq \delta_1$.

The amount of tokens read at the start of a firing of some actor a can be described by an edge quantity $Rd(a) = \{((q,p), Rate(p))|(q,p) \in InD(a)\}$ and the amount of tokens produced at the end of a firing by an edge quantity $Wr(a) = \{((p,q), Rate(p))|(p,q) \in OutD(a)\}$.

Definition 6. (State) *The state of a timed SDFG (A, D, Υ) is a pair (δ, υ). Edge quantity δ associates with each dependency edge $d \in D$ the amount of tokens in that edge in that state. To keep track of time progress, an actor status $\upsilon : A \to \mathbb{N}^{\mathbb{N}}$ associates with each actor $a \in A$ a multiset of numbers representing the remaining times of different ongoing firings of a. We assume that the initial state of an SDFG is given by some initial token distribution δ, which means the initial state equals $(\delta, \{(a, \{\}) \mid a \in A\})$ (with $\{\}$ denoting the empty multiset).*

The dynamic behavior of the SDFG is described by transitions. Three different types are distinguished: start of actor firings, end of firings, and time progress in the form of clock ticks. The following definition interprets time as **processing time**; that is, at any point in time, an arbitrary subset of the executing actors may progress depending e.g. on the allocation of sufficient resources. This is in contrast to the interpretation of time as *runtime* or *real time*, in which time always progresses for all active firings.

Definition 7. (Transition) *A transition of a timed SDFG (A, D, Υ) from state (δ_1, υ_1) to state (δ_2, υ_2) is denoted by $(\delta_1, \upsilon_1) \xrightarrow{\beta} (\delta_2, \upsilon_2)$ where label $\beta \in (A \times \{start, end\}) \cup \{clk\}$ denotes the type of transition.*

- *Label $\beta = (a, start)$ corresponds to the firing start of actor $a \in A$. This transition may occur if $Rd(a) \preceq \delta_1$ and results in $\delta_2 = \delta_1 - Rd(a)$, $\upsilon_2 = \upsilon_1[a \mapsto \upsilon_1(a) \uplus \{\Upsilon(a)\}]$, i.e., υ_1 with the value for a replaced by $\upsilon_1(a) \uplus \{\Upsilon(a)\}$ (where $\uplus$ denotes multiset union).*
- *Label $\beta = (a, end)$ corresponds to the firing end of $a \in A$. This transition can occur if $0 \in \upsilon_1(a)$ and results in $\upsilon_2 = \upsilon_1[a \mapsto \upsilon_1(a) \backslash \{0\}]$ (where $\backslash$ denotes multiset difference), and $\delta_2 = \delta_1 + Wr(a)$.*

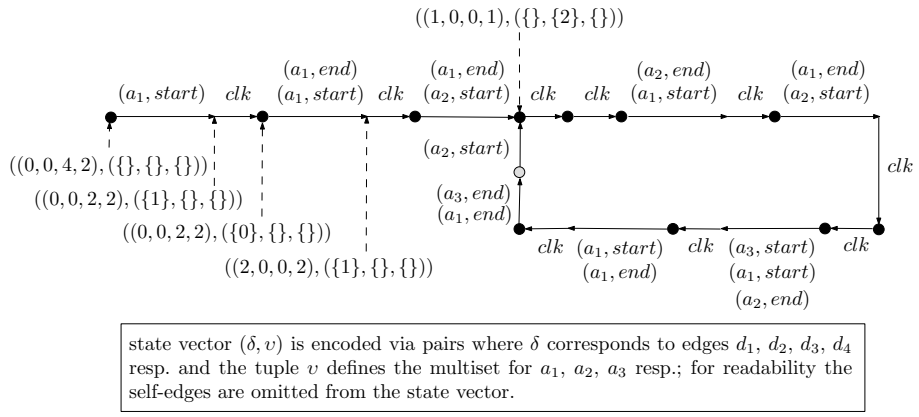


Figure 4.4: State-space of the example SDFG.

- *Label $\beta = clk$ denotes a clock transition. It is enabled if no end transition is enabled. Clock transitions can occur for any submultiset of executing actors and nondeterministically results in $\delta_2 = \delta_1$, $\upsilon_2 = \{(a, M) | a \in A\}$ with M equal to $\upsilon_1(a)$ with some arbitrary submultiset of its members decremented by one.*

A clock transition decreases the remaining execution time of an arbitrary submultiset of ongoing actor firings with one time-unit. The ability to select different submultisets in different clock transitions makes it possible to take resource constraints (e.g. static-order schedules or TDMA schedules) into account (see Section 6.6.2).

Definition 8. (Execution) *An execution of an SDFG is an infinite alternating sequence of states and transitions $s_0 \xrightarrow{\beta_0} s_1 \xrightarrow{\beta_1} \ldots$ from some designated initial state s_0.*

There exists one type of execution, namely self-timed execution, which gives maximal throughput [128]. It requires that each actor fires as soon as it is enabled and that always all active firings make progress.

Definition 9. (Self-timed execution) *In a self-timed execution, clock transitions only occur if no start transition is enabled and only clock transitions that reduce the remaining execution time of all active firings can occur.*

Figure 4.4 shows the transition system of the self-timed execution of the timed SDFG shown in Figure 4.3. All clock transitions are shown explicitly. Between clock transitions, there can be multiple start and/or end transitions enabled simultaneously. These start and end transitions are independent of each other. Independent of the order in which they are applied, the final state before each

clock transition, and the first state after each clock transition, are always the same. Therefore, all start and end transitions are shown as one annotated step. The only exception to this in Figure 4.4 is the gray state, which is made explicit to facilitate the illustration of throughput calculations below. The transition system consists of a finite sequence of states and transitions, called the **transient phase**, followed by a sequence of states and transitions which is repeated infinitely often and is called the **periodic phase**. In [46], it is proved that the state-space of the self-timed execution of any consistent and strongly connected timed SDFG consists of a finite sequence of states and transitions followed by a sequence that is periodically repeated. This is done by proving the following proposition (see [46, Proposition 11]).

Proposition 1. *For every consistent and strongly connected timed SDFG, the self-timed state-space consists of a transient phase, followed by a periodic phase.*

An important concept related to the execution of an SDFG is the notion of an iteration.

Definition 10. (ITERATION) *Given a timed SDFG* (A, D, Υ) *with a repetition vector* γ*. An iteration is a set of actor firings such that for each* $a \in A$*, the set contains* $\gamma(a)$ *firings of* a*.*

In the execution of Figure 4.4, the periodic phase consists of precisely one iteration. In general, it is possible that iterations in an execution of an SDFG overlap in time.

4.4 Throughput Analysis

Throughput is an important design constraint for embedded multimedia systems. Intuitively, the throughput of an SDFG refers to how often an actor produces an output token. The throughput of an SDFG is traditionally defined as 1 over the **maximal cycle mean** (MCM) of the corresponding HSDFG (see e.g., [128]). In the literature on cycle mean algorithms, the HSDFG is often seen as a weighted directed graph. This is a directed graph in which the weight, called cost, of each node is equal to the execution time of the actor in the HSDFG and the weight, called transit time, of each edge is equal to the number of initial tokens on the edge in the HSDFG. The cycle mean of some cycle of such a weighted directed graph is defined as the total cost of the nodes over the total transit time of the edges in that cycle. The maximum cycle mean over all cycles in the graph is called the MCM of the graph. To distinguish the case in which all edges in the directed graph have unit transit time from the general case in which arbitrary edge weights are allowed, in the latter context one sometimes uses the terms cycle ratio and **maximum cycle ratio** (MCR) for the general case and cycle mean and MCM for the restricted case.

To compute the throughput of an SDFG, the SDFG is traditionally converted to an equivalent HSDFG. The throughput can then be computed using a Maximum Cycle Ratio (MCR) algorithm or a Maximum Cycle Mean (MCM) algorithm that operates on a directed graph with unit transit time for the edges [34, 76]. In Section 2.5.3 and Section 2.5.4 of [11] an approach is suggested to convert an HSDFG to a weighted directed graph with unit transit time. Existing MCR and MCM algorithms are quite efficient. In [33], state of the art algorithms for MCR/MCM analysis are compared. Although MCR/MCM analysis can be done efficiently, it can only be applied to an HSDFG and the HSDFG can be exponentially larger in size (in terms of the number of actors) than the original SDFG, making the approach as a whole not particularly efficient for SDFG throughput analysis. For this reason, an alternative approach to compute the throughput of an SDFG is introduced in [46]. A self-timed execution of the SDFG is used for the throughput analysis. As already mentioned, this type of execution is known to give maximal throughput [128]. Throughput analysis is usually restricted to strongly connected SDFGs. In this thesis, throughput analysis is also only performed on strongly connected SDFGs. This is not a very limiting assumption as the storage space of edges in an SDFG is finite when the SDFG is implemented on a real system. Modeling storage space constraints on the edges of a connected SDFG transforms this SDFG into a strongly connected SDFG (see Section 7.3). A generalization to perform throughput analysis for arbitrary, connected SDFGs can be found in [45]. The throughput of a strongly connected SDFG is defined as follows.

Definition 11. (THROUGHPUT) *The throughput $Th(a)$ of an actor a for the self-timed execution σ of a consistent and strongly connected timed SDFG $G = (A, D, \Upsilon)$ is defined as the average number of firings of a per time unit in σ. The throughput of G is defined as*

$$Th(G) = \frac{Th(a)}{\gamma(a)},$$

where γ is the repetition vector of G and a an arbitrary actor. The throughput of G gives the average number of iterations of the graph per time unit in σ.

In [46], this notion of throughput for an SDFG is proved to be equivalent to one over the maximal cycle mean of the corresponding HSDFG. It also provides a technique to compute the throughput of an actor from the self-timed state-space. It uses Proposition 1 and the following proposition (see [46, Lemma 14]):

Proposition 2. *For every consistent and strongly connected timed SDFG (A, D, Υ), the throughput of an actor $a \in A$ is equal to the average number of firings per time-unit in the periodic part of the self-timed state-space.*

The throughput of actor a_3 of the example SDFG shown in Figure 4.3 is computed as follows. Consider again its self-timed state-space as shown in Figure

4.4. Actor a_3 ends its firing for the first time after 9 clock transitions in the gray state. At that moment, the actor is in the periodic phase of the schedule and fires each 7 time units. The periodic phase is repeated indefinitely. Hence, the average time between two firings over the whole schedule converges to the average time between two firings in the periodic phase. So, the throughput of a_3 is $1/7$. The throughput of the graph can then be computed using Definition 11. It uses the repetition vector of the graph which equal to $(a_1, a_2, a_3) \rightarrow (3, 2, 1)$. Using this repetition vector, it is found that the throughput of the graph is equal to the throughput of a_3. The throughput of the actors a_1 and a_2 can now be computed by multiplying the graph throughput with their entry in the repetition vector. The SDFG shown in Figure 4.3 models an application. Typically, a designer is interested in the throughput with which the application produces output data. Definition 11 gives a normalized throughput that can easily be converted to the rate at which output data is produced. Assume that actor a_3 produces this output data at a rate of 70 bytes per firing (on some output port of the application that is not shown in the figure). When a time-unit is equal to 1ms, the throughput with which output data is produced is equal to $1/7$ firings/time-unit $\cdot$ 70 bytes/firing $= 10$ bytes/ms.

States visited during the execution of a timed SDFG $G = (A, D, \Upsilon)$ need to be stored in order to detect the periodic phase. However, the lengths of the transient and periodic phases can be fairly long and a large number of states may have to be stored. Fortunately, due to the determinism in the state-space, only a selected set of states needs to be stored. Suppose that a state that is not stored is revisited. The execution continues then in the same way as the first time, revisiting the same states. If at least one of the states in the periodic part is actually stored, it will be encountered and a cycle in the state-space will be detected. Only a single state of every iteration has to be stored as the periodic behavior consists of a whole number of iterations ([46, Proposition 12]). In this way, the periodic behavior always includes at least one state that is stored. (To detect deadlock, it must also be checked whether a clock transition remains in the same state.) The throughput of G is now calculated as follows. First, an arbitrary actor $a \in A$ is picked. Every iteration of the SDFG includes $\gamma(a)$ start and end transitions of a. All the states reached immediately after every $\gamma(a)$-th execution of an end transition of a are stored. Using this method, the period can be detected and also the number of iterations of the period. The length in clock transitions can be easily calculated if additionally the number of clock transitions between each two stored states is kept. With this information and using Proposition 2 and Definition 11, the throughput of an SDFG can be calculated. Note that to compute the throughput for the example (see Figure 4.4), only the gray state must be stored as this is the only state in which a firing of a_3 ends.

4.5 Scheduling

A multi-processor platform will be used in future embedded multimedia systems. Multiple actors will often be mapped to a single processor from the platform. The firing of these actors must be ordered (scheduled) on the processor. Many multi-processor scheduling techniques are based on single-processor scheduling techniques. This section gives an overview of basic scheduling techniques for single- and multi-processor platforms. It also discusses the modeling of an ordered schedule in an SDFG.

Scheduling of SDFGs for single-processor systems was introduced in [86]. The objective of single-processor scheduling strategies is to minimize the code size and/or the required storage-space for the tokens communicated between the actors of the SDFG.

Bhattacharyya introduces in [22] the class of single appearance schedules. These schedules can be described with a regular expression in which each actor appears only once. This guarantees that the schedule has minimal code size. An acyclic SDFG has always a single appearance schedule. Initially there is always at least one actor which can fire as many times as indicated by the repetition vector. Firing this actor this number of times enables at least one other actor to fire as often as indicated by the repetition vector. Repeating this scheduling/firing strategy for all actors in the graphs yield a single appearance schedule. In [23, p. 85], it is shown that an arbitrary SDFG has a single appearance schedule if and only if each strongly connected component has a single appearance schedule. It is also shown that a strongly connected SDFG has a single appearance schedule only if the actors can be partitioned into two sets A_1 and A_2 such that any edge directed from A_1 to A_2 contains sufficient tokens to perform as many firings of every actor in A_2 as indicated by the repetition vector before firing any actor from A_1. Note that multiple, different single appearance schedules may exist. These schedules differ in the amount of storage-space required for the tokens communicated between the actors. A heuristic to minimize the required storage-space of a single appearance schedule is presented in [100]. A variant on the single appearance schedule using dynamic loop counts is presented in [108]. It is claimed that this allows the construction of schedules with minimal code and data size for arbitrary SDFGs. However, the overhead of storing all dynamic loop counts is neglected. The size of arrays used to store these loop counts can be exponentially large with respect to the number of actors in the graph.

SDFGs can also be scheduled onto multi-processor systems. When sufficient resources are available, schedules with a throughput equal to the maximal achievable throughput can be constructed. Existing scheduling techniques for multi-processor systems use HSDFGs [128, Section 3.8]. The conversion of an SDFG to an HSDFG can lead to an exponential increase of the number of actors in the graph. To limit the impact of the increase in the number of actors in the conver-

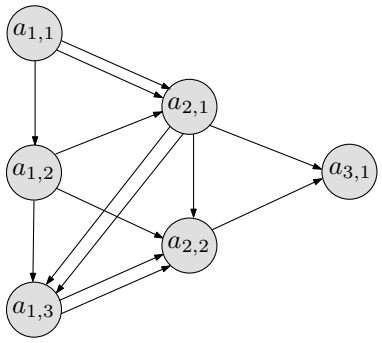


Figure 4.5: Acyclic precedence graph of the example SDFG.

1	2	3	4	5	6	7	8	9
a_1	a_1	a_2	$\_a_2$	a_1	a_2	$\_a_2$	a_3	$\_a_3$
							a_1	a_1

Figure 4.6: Self-timed schedule of the example SDFG of Figure 4.3.

sion, clustering algorithms are used [87, 114]. These algorithms cluster a set of actors into a single actor. All actors in the cluster are then scheduled onto a single processor using a single-processor scheduling technique. Only across clusters a multi-processor scheduling technique is used.

Two different forms of multi-processor schedules for HSDFGs exist: non-overlapping schedules and overlapping schedules. Both scheduling forms distinguish different iterations of an HSDFG from each other. The i-th *iteration* of an HSDFG is defined as the set of actor firings that contains the i-th firing of all actors in the HSDFG. In non-overlapping schedules, the HSDFG is converted to an Acyclic Precedence Graph (APG). This conversion removes all edges from the HSDFG that contain one or more initial tokens. In other words, it removes those edges which model a dependency between the i-th firing of some actor a_i and the (i+n)-th firing of some actor a_j where $n > 0$. The APG for the HSDFG of Figure 4.2 is shown in Figure 4.5. The APG shows only the intra-iteration precedences between the actor firings. It does not exploit the inter-iteration parallelism. This parallelism can be exploited by scheduling the SDFG over multiple iterations. A blocked schedule with **blocking factor** N is a non-overlapping schedule of the SDFG graph which is **unfolded** N times. Unfolding an SDFG N times means considering N successive iterations of the graph. The blocking factor which allows scheduling of the SDFG with the maximal achievable throughput is called the optimal blocking factor. A technique to compute the optimal blocking factor for blocked, non-overlapping schedules is presented in [102]. Not all SDFGs have a finite optimal blocking factor. For those SDFGs, it is not possible to construct a non-overlapping schedule which achieves maximal throughput. An example of

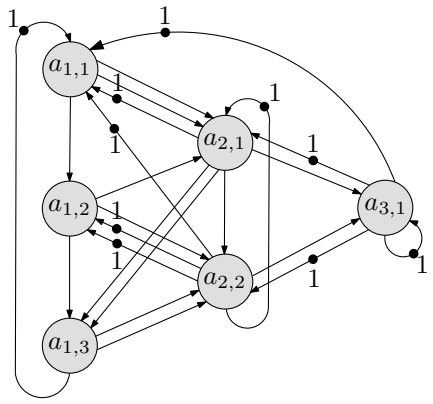


Figure 4.7: Static-order schedule $((a_1)^2 a_2 a_1 a_2 a_3)^*$ modeled in the HSDFG.

such a graph is the SDFG shown in Figure 4.3. The self-timed schedule, which achieves maximal throughput, is shown in Figure 4.6. The schedule starts with firing actor a_1 at time-step 1. This actor is fired a second time at time-step 2. The firing of a_2 is started in the 3th time-step and continued in the 4th time-step. The continuation of an actor firings is denoted with an underscore in front of the actor's name. After 9 time-steps, the first iteration of the graph is completed. The schedule continues with repeatedly firing the actors scheduled in the time-steps 3 through 9. To achieve maximal throughput, the two firings of actor a_1 which occur at time-steps 8 and 9 must always overlap with the firing of a_3. Any finite blocking factor will force the firings of a_1 to be non-overlapping with the firing of a_3 at some point in time. This results in a lower throughput than the maximal achievable throughput. In overlapped scheduling, the inter-iteration parallelism is fully taken into account. When resource constraints are absent, maximal throughput can always be achieved with this type of schedule [118]. This makes overlapped scheduling fundamentally more powerful than non-overlapped scheduling.

In [129], it is shown how a static-order schedule of an HSDFG $G = (A, D)$ can be modeled into an HSDFG $G_s(A, D_s)$ such that in the self-timed execution of G_s actors are fired in the order specified by the static-order schedule. The method that derives G_s from G does not change the set of actors. It only adds additional edges to G_s when compared to G (i.e. $D_s \cap D = D$). These edges form a cycle in G_s that enforces the ordering in which the actor firings should occur. Assume that the actors $a_1 a_2 a_3 \cdots a_n$ are scheduled in this order in a static-order schedule. To enforce this schedule, the method of [129] adds a cycle $((a_1, a_2), (a_2, a_3), \cdots, (a_{n-1}, a_n), (a_n, a_1)$ with one initial token on (a_n, a_1) to G_s. An example of an HSDFG that enforces the static-order schedule $((a_1)^2 a_2 a_1 a_2 a_3)^*$ on the HSDFG of Figure 4.2 is shown in Figure 4.7. The constructed HSDFG G_s

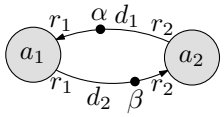


Figure 4.8: Counter example SDFG.

can be used to analyze the throughput that can be achieved with G under the given static-order schedule. The following theorem shows that it is not always possible to model a static-order schedule in an SDFG without using more actors than used in this schedule.

Theorem 1. (Modeling of static-order schedules) *Not every static-order schedule for N different actors can be modeled with a consistent SDFG (A, D) of which the ordering of the actor firings in its self-timed execution is equal to the static-order schedule and with $|A| = N$.*

Proof. The proof is given with a counter example. Consider the static-order schedule $(((a_1)^3 a_2)^2 a_1 a_2)^*$. This schedule contains two different actors; a_1 and a_2. The SDFG shown in Figure 4.8 contains also two actors. The edges d_1 and d_2 form a cycle which expresses the cyclic dependency of a_1 and a_2 that is required by the schedule. The rates on the ports of the actors fix the ratio of the number of actor firings between the different actors in the graph (i.e. repetition vector, Definition 3). Every edge going from an actor a_i to an actor a_j must have the same ratio. Otherwise the graph is inconsistent. This implies that in a consistent SDFG the rates on different edges going from a_i to a_j can only be a multiple of each other. So, without loss of generality it can be assumed that only the smallest possible rates are used to model the firing ratio of the actors. Note that it is not useful to have multiple edges from a_1 to a_2 or vice versa with different numbers of initial tokens on them. The reason for this is that the execution of the graph is only determined by the edge from a_i to a_j that has the smallest number of initial token on it amongst all edges from a_i to a_j. It remains to compute the rates of the actors and the number of initial tokens on the edges d_1 and d_2. It follows from the schedule that actor a_1 fires 7 times for each 3 firings of a_2. This implies $r_1 = 3$ and $r_2 = 7$. The number of initial tokens α and β on d_1 and d_2 can be derived from the static-order schedule. Consider the constraint that the first firing of a_2 takes place after three firings of a_1. Actor a_2 fires once before the 4th, 5th and 6th firing of actor a_1 occur. To allow these 6 firings of a_1 it must hold that $\alpha - 9 + 7 \geq 9$ which is equal to $\alpha \geq 11$. To avoid that more than 3 firings of a_1 could occur at the start of the schedule, it is required that $\alpha < 12$. So, it must hold that $\alpha = 11$. The number of initial tokens on d_2 can be derived in the same way. When $\beta > 0$, the actor a_2 can fire after two (or less) firings of a_1. So, it must hold that $\beta = 0$. The rates and initial token distribution of the SDFG have been fixed based on the requirements of the static-order schedule. Note that this

Table 4.1: Tokens in the edges d_1 and d_2 of the SDFG shown in Figure 4.8.

		a_1^3	a_2	a_1^3	a_2	a_1	a_2
d_1	11	2	9	0	**7**	4	11
d_2	0	9	2	11	**4**	7	0

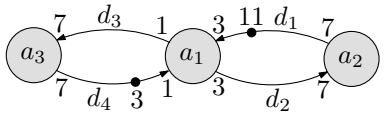


Figure 4.9: Static-order schedule $(((a_1)^2a_2)^2a_1a_2)^*$ modeled in an SDFG.

is the only SDFG that contains two actors and that satisfies the above mentioned criteria from the static-order schedule.

It remains to verify whether these values satisfy all constraints of the schedule. This is only true when the self-timed execution of the graph is equal to the static-order schedule. It is now shown that these requirements are not fulfilled by the constructed SDFG. Consider the number of tokens in the edges d_1 and d_2 when the SDFG is executed following the given static-order schedule and with $r_1 = 3$, $r_2 = 7$, $\alpha = 11$ and $\beta = 0$. The distribution of the tokens over d_1 and d_2 after every step of the schedule is shown in Table 4.1. It shows that there are seven tokens on d_1 after four steps of the schedule (shown bold). According to the static-order schedule, one firing of a_1 should occur after this step. However, two firings of a_1 are enabled. So, this SDFG does not realize the given static-order schedule. Hence, this example shows that not any arbitrary static-order schedule can be realized with an SDFG that contains a number of actors equal to the number of actors used in the schedule. □

This theorem leaves the possibility that the number of actors in an SDFG G_s that models a static-order schedule for an SDFG G may contain more actors than G. For example, the SDFG shown in Figure 4.9 realizes the static-order schedule that is used in the proof. One extra actor is used to enforce the static-order schedule $(((a_1)^3a_2)^2a_1a_2)^*$ in the self-timed execution of the graph. Currently, no method is known that tries to minimize the number of actors that are needed to model a given static-order schedule of a graph G. The only known technique is to convert the SDFG to its corresponding HSDFG and apply the method from [129]. This can lead to an exponential increase in the number of actors in the graph.

4.6 SDF^3: Synchronous Dataflow for Free

Development of new SDF-based design techniques is hampered by the availability of only a limited set of test graphs. Inspired by Task Graphs For Free [35] that

can only generate acyclic task graphs with non-pipelined point-to-point communication, a tool called **SDF For Free (SDF^3)** [138] was developed. It generates random sets of SDFGs, with support to analyze and visualize these graphs. The tool with C++ source code is freely available from `http://www.es.ele.tue.nl/sdf3`.

SDF^3 implements an SDFG generation algorithm that constructs graphs which are connected, consistent, and deadlock-free. Unconnected SDFGs can always be constructed by combining two or more SDFGs generated with the algorithm. A user specifies in a configuration file the most important parameters determining the characteristics of the graph. These parameters are the fixed number of actors in the graph and the average and variance of their degree (i.e., the number of input and output ports) and port rates. The actual degree of actors and the port rates are random values picked by a random number generator [92]. To get a better control over the characteristics of the generated graphs, a minimum and maximum value, bounding the range of possible values, can also be specified.
For many SDFG analysis algorithms, properties must be assigned to the actors, tokens, and edges, or the SDFG as a whole. For example, algorithms which deal with the throughput of a graph require timing annotations on actors. Buffer sizing algorithms typically require token sizes which can be annotated to the edges. SDF^3 contains functions to assign randomly selected values to actors and edges which represent these properties. Currently, it can assign an execution time and memory requirement to each actor, token sizes, latencies and buffer sizes to edges, and sets a throughput constraint on the graph. This annotation mechanism can easily be extended by users of the tool.
By default, the generated SDFGs are connected in an arbitrary way. If desired, the tool can restrict the connections between the actors making the graph a chain, a-cyclic, or strongly connected. DSP and multimedia applications, which are often of these forms, can easily be mimicked in this way.

Besides generation of random graphs, the tool offers a library that contains all SDFG analysis and transformation techniques discussed in this chapter. This includes implementations of Karp's, Howard's, and Young-Tarjan-Orlin's MCM algorithms. SDF^3 uses an XML-based format for SDFGs which enables simple exchange of graphs between different tools. It also offers a function to visualize SDFGs through the popular graph visualization tool dotty [41]. Furthermore, all techniques and algorithms presented in this thesis have been integrated into the tool. This makes it a versatile tool that can be used for DSP synthesis and designing multi-processor systems with a predictable timing behavior. The latter aspect is studied in this thesis.

4.7 Comparison of Dataflow Models

Various dataflow Models of Computation (MoCs) exist. These MoCs differ in their expressiveness and succinctness, analyzability and implementation efficiency. The **expressiveness** and **succinctness** of a model indicate which systems can be modeled and how compact these models are. For example, the behavior of an H.263 decoder can be modeled with an SDFG consisting of 4 actors. Alternatively, the same behavior can be modeled with an HSDFG containing 4754 actors. Clearly, constructing the SDF model is easier than constructing the large HSDF model. This example illustrates that the SDF model is more succinct than the HSDF model. Furthermore, some properties (e.g., data-dependent behavior) can be modeled in some dataflow MoCs but not in others, which is a difference in expressiveness. The second aspect that differentiates MoCs is their analyzability. The **analyzability** of a MoC is determined by the availability of analysis algorithms and the run-time needed for an algorithm on a graph with a given number of nodes, independent of the MoC considered. The third aspect that is relevant when comparing MoCs is their **implementation efficiency**. This is influenced by the complexity of the scheduling problem and the (code) size of the resulting schedules. Also for this aspect it is important that different MoCs are compared assuming a graph with an equal number of nodes. This decouples the succinctness of a MoC from the other aspect that is considered. In this section, the most important dataflow MoCs are compared on the three aforementioned aspects. The result of this comparison is visualized in Figure 4.10. The objective of the comparison is to motivate the choice made in this thesis to model an application with an SDFG.

The first two dataflow MoCs considered are HSDFGs and SDFGs. As a note aside, it is interesting to remark that HSDFGs and SDFGs correspond to subclasses of **Petri nets**, which is a well known general purpose MoC, not limited to dataflow, with a rich literature [112, 117]. These subclasses are **marked graphs** [29] and **weighted marked graphs** [139] respectively. The Petri net literature may provide theoretical results and analysis algorithms that are applicable to (H)SDFGs.

The important difference between SDFGs/weighted marked graphs and HSDFGs/marked graphs is that the former MoCs can deal with multi-rate dependencies. This makes these MoCs more succinct. However, existing analysis algorithms for HSDFGs/marked graphs for properties like throughput or latency have a polynomial time complexity. Similar algorithms for SDFGs/weighted marked graphs have a non-polynomial time complexity. This implies that the analyzability of SDF/weighted marked graphs is less. The fact that these MoCs support multi-rate dependencies makes their schedules and the scheduling problem also more complex. So, their implementation efficiency is less than that of a graph modeled in the HSDF or marked graph MoC.

Karp and Miller introduced in 1966 the **computation graph** model [77].

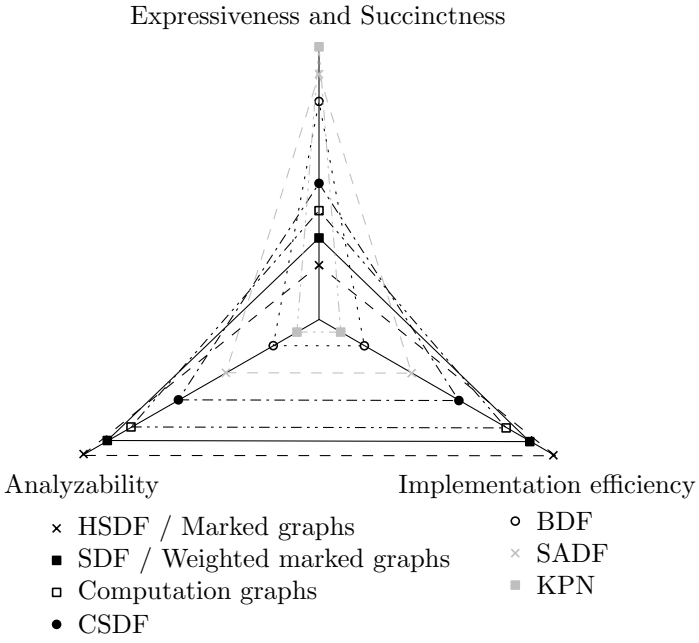


Figure 4.10: Comparison of dataflow models of computation.

Similar to SDFGs, actors in a computation graph consume and produce a fixed amount of tokens on each firing and edges may contain initial tokens. To each input port of an actor, a threshold parameter is attached. This threshold indicates that the connected actor is only allowed to fire if the number of tokens on the edge is at least equal to the threshold. An SDFG can be seen as a computation graph in which this threshold is equal to the number of tokens consumed by the input port during a firing. Karp and Miller provide necessary and sufficient conditions for liveness of a computation graph. Liveness indicates that all parts of the graph can execute infinitely often. The article proves also conditions under which buffer sizes on the edges remain bounded when executing a computation graph. Similar conditions exist for SDFGs. In [45], all necessary and sufficient conditions for liveness and boundedness of an SDFG are given.

The distinction made in the computation graph model between the number of tokens needed for an actor to fire and the actual number of tokens consumed by a firing does not make it more expressive than SDFGs but it does make it more succinct. However, this makes the conditions under which an actor can be scheduled (fired) more complex. For the same reason, analysis algorithms become more time consuming.

An important property of the MoCs discussed so far is that the rate at which actors consume tokens from their inputs and produce tokens on their outputs is

constant. The **Cyclo-Static Dataflow** (CSDF) MoC [84, 24] relaxes this constraint. In this model, the rate of a port may change between subsequent firings. The sequence of the rates of the port must be finite and periodically repeated. Using the algorithm from [84], any CSDFG can be converted into an equivalent HSDFG. This guarantees that a static-order, periodic schedule can be constructed for a CSDF graph (CSDFG). It also shows that CSDF is not more expressive then SDF, but it is more compact for certain aspects. The complexity of CSDF schedules will typically be larger than the schedule that results from an SDFG with an equal number of actors. Existing state-space based analysis techniques for SDFGs can be extended to CSDFGs by considering the rate sequences as part of the state. As CSDFGs have more information in their state when compared to SDFGs, the analysis algorithms will be slower.

It is not possible to model data-dependent behavior in any of the MoCs discussed so far. This makes it impossible to express the property that an actor chooses between two inputs depending on the value of a token on a third input. The **Boolean Dataflow** (BDF) MoC allows the modeling of this type of data-dependent behavior. This model was introduced by Lee in [85] and studied extensively by Buck in [27]. BDF extends SDF with a *switch* and *select* construct. The *switch* copies the data it reads from its input to one of its outputs based on the value of a boolean *control token*. The *select* reads, based on the value of a control token, data from one of its inputs and copies this data to the output. The switch and select introduce data-dependent behavior, which makes a BDF graph (BDFG) in general non-analyzable at design time. The data-dependent behavior makes it also impossible to construct a static-order schedule for a BDFG. Therefore, a run-time scheduling mechanism must be used when implementing an application modeled with a BDFG. This makes the implementation of a BDFG less efficient than any of the MoCs mentioned before.

The **Scenario-Aware Dataflow** (SADF) model, introduced in [140], is another dataflow MoC which enables modeling and analysis of data-dependent behavior. In an SADF graph (SADFG) the rates of the actors are data-dependent and controlled with a control actor. Both the BDF model and the SADF model are Turing complete. This implies that they can both model the same behavior. However, an SADFG will often be more compact than a BDFG that models the same functional behavior, as control actors in SADF allow modeling of more complex control structures than the simple switch and select from the BDF model.

Recently a number of analysis algorithms have been developed for SADFGs. These algorithms have a non-polynomial time complexity. The execution time of these algorithms is larger as the time needed to execute a similar analysis algorithm on an equally sized CSDFG or SDFG. This is because typically more different behaviors are possible in an SADFG and all these behaviors need to be analyzed. SADFGs can be partially scheduled at design-time and must partially be scheduled at run-time.

The **Kahn Process Network** (KPN) MoC is proposed by Kahn in [75]. In this model, processes communicate with each other by sending data to each other

over edges. A process may write to an edge whenever it wants. When it tries to read from an edge which is empty, it blocks and must wait till the data is available. The amount of data read from an edge may be data-dependent. This allows modeling of any continuous function from the inputs of the KPN to the outputs of the KPN with an arbitrarily small number of processes. Continuity corresponds informally to the expected dataflow property that extra input can only lead to extra output (not to less). KPN in fact is sufficiently expressive to capture precisely all data dependent dataflow transformations. This implies that it is not possible to analyze properties like the throughput or buffer requirements of a KPN without considering all possible inputs. Every input may require a different schedule. So, run-time scheduling must be used when a KPN is implemented.

A MoC that is used in the predictable design flow described in this thesis must have sufficiently fast analysis techniques. Furthermore, the model should allow an efficient implementation as the application modeled in the MoC should be realized on a platform while minimizing resource usage. Modeling applications in the used MoC should also be as simple as possible to allow application designers to describe their application in the most natural way.

The first requirement is fulfilled by the HSDF, SDF, computation graphs, and CSDF MoC. The HSDF MoC was not chosen because models of realistic applications can be very large. Consider for example the H.263 decoder discussed at the beginning of this section. This makes the analysis algorithms too time-consuming to be useful in practice despite their polynomial time complexity.

When this research began, analysis algorithms for SDFGs, computation graphs and CSDFGs did not exist (e.g. for exact buffer sizing). Development of a design flow that uses the SDF MoC required the development of novel, sufficiently fast analysis techniques. Efficient analysis techniques for more general MoCs like CSDF are inherently more difficult to develop. For this reason, SDF was selected as the MoC that is used in the predictable design flow.

The analysis techniques that have been developed over the last few years for the SDF MoC are fast enough to be used in a predictable design flow, as this thesis shows. These techniques can most likely be extended to the CSDF MoC with limited execution time overhead. Therefore, it seems logical to consider in future work an extension of the design flow and techniques proposed in this thesis to CSDFGs.

4.8 Summary

This chapter formalizes the SDF model and it extends the model to take time into account. It presents analysis techniques to compute the throughput of timed SDFGs. Throughput is an important design constraint when building embedded multimedia systems. The chapter discusses also different single and multi-

processor scheduling techniques for SDFGs. It is shown that modeling static-order schedules in an SDFG requires, in general, a conversion of the graph to its equivalent HSDFG. The chapter briefly presents the SDF^3 tool for SDFG generation, scheduling, analysis, and visualization. It ends with a comparison between some of the most widely used dataflow models for their use in a predictable design flow. This comparison shows that the SDF model is suitable for use in such a design flow.

Chapter 5

Modeling Memory Mappings

5.1 Overview

The first step of the design flow sketched in Chapter 1 concerns the memory mapping and dimensioning. This chapter explains the details of this step.

Until now, embedded systems designers lived comfortably with dedicated memory close to the computational logic, thereby allowing predictable and short access times. For cost reasons, it is no longer affordable for different sub-systems to have separate, large memories. This suggests the need for a high level of re-use of these memories. In future platforms, memories will be distant and shared among potentially many computational resources. The platform, introduced in Section 3.2, follows this trend as it contains large, shared memories in so-called memory tiles. These memory tiles have limited processing capabilities. Therefore, actors that perform a computation cannot be executed on these tiles.

An SDFG that models a multimedia application may communicate large tokens. An actor in a video encoder may, for example, require access to a complete video frame which can be modeled with one token. A frame in HDTV-resolution has a size of several megabytes. Memories in processing tiles are typically not large enough to contain a complete frame. The frame (token) must therefore be stored in a memory tile. The actor that uses this token must be mapped to a processing tile as insufficient processing capabilities are available in the memory tile. The memory tile might be shared by a number of actors potentially of different applications. To build a system with a predictable timing behavior, memory accesses to a distant and shared memory should be modeled in the application SDFG. This chapter presents an SDF model that can be used to model the memory access pattern of actors. The model allows analysis of the timing behavior of an SDFG when an actor uses tokens stored in a memory tile.

Figure 5.1 shows an SDFG with an actor a that has a self-edge d_s with one token and that consumes c tokens during its firing from the dependency edge

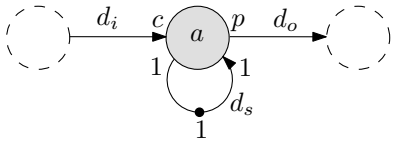


Figure 5.1: Example SDFG with large tokens on d_i and d_s.

d_i. Actor a produces during a firing also p tokens on the dependency edge d_o. Assume that the tokens on d_o can be stored in the memory inside the processing tile to which a is mapped, but the tokens on d_i and d_s cannot be stored in this memory. These tokens must be stored in a memory tile. The SDFG of Figure 5.1 is used in this chapter to show how the remote memory accesses of a to the tokens on the edges d_i and d_s are modeled in an SDFG. The resulting model can be used to refine an application model; the refined model then serves as input to the remainder of the design flow, which makes remote memory accesses analyzable and predictable. Models for other distributions of the tokens over the memories in the processing and memory tiles can be constructed in a similar way.

The next section discusses related BDF and HSDF models that capture memory accesses of an actor in a memory tile. The SDF memory access model is presented in Section 5.3. A technique to extract the SDF memory access model from an actor is outlined in Section 5.4. The SDF memory access model is used in Section 5.5 to model the remote memory accesses of a real application.

5.2 Related Work

An actor represents a code segment in an application that can be executed without blocking when all its input tokens (data) are available. This chapter addresses the problem that the data needed to execute a code segment (actor) does not fit into the memory of the tile on which the code is executed (i.e. the data does not fit in the memory of a processing tile). The solution to this problem is to split the code segment in several parts (also actors) such that the data needed for each individual part fits into the memory of the processing tile. All other data is stored in a memory tile. Between the execution of the various code segments, data must be transfered between the processing and memory tile. To allow timing analysis, both the splitting of a code segment into a set of smaller code segments and the introduced memory transfers must be modeled into the application SDFG.

The problem of modeling memory accesses in a memory tile was first identified in [136]. A BDF and HSDF model for this problem were presented in [135]. The BDF model describes a generic memory access pattern in which repeatedly data is read from and written to the memory tile. After an unknown number of read/write iterations, a result (token) is produced on the output edge(s) of the model. The timing behavior of this model cannot be analyzed due to the unknown number of

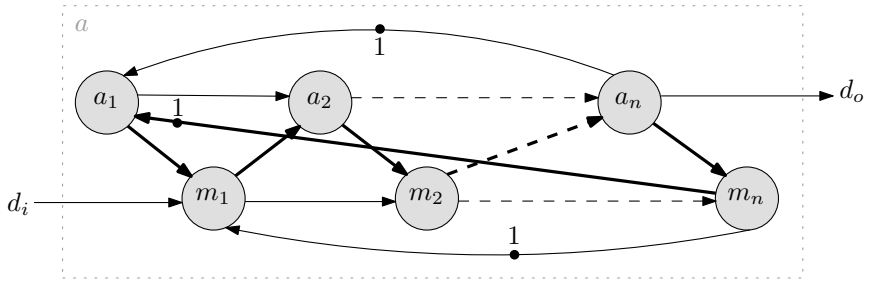


Figure 5.2: HSDF model for remote memory access.

read/write iterations. In multimedia applications, the number of iterations is often bounded or even fixed. In those situations, the BDF model can be transformed into an HSDF model. The HSDF memory access model that results from this transformation, when applied on the simple example of Figure 5.1, is shown in Figure 5.2. Actor a is split in this model into a sequence of n actors a_i that perform all part of the computation done by a. The model therefore in fact is a parameterized model. It assumes that a data element needed by actor a_{i+1} (computation) must explicitly be fetched from actor m_i (memory). When actor a_i requests some data elements from actor m_i, the firing of a_{i+1} is postponed till the memory returns the requested data. In case the access pattern to the remote data token is known in advance, the data can be prefetched from the memory while actor a_i is still performing its transformation on the previously fetched data elements. In [135], it is shown how this prefetching can be taken into account in the HSDF memory access model. For this purpose, both the a_i and m_i actors are subdivided into two separate parts. The resulting HSDF model is shown in Figure 5.3. The actors $a_{i,1}$ and $m_{i,1}$ deals with the prefetching of data from the memory tile, actor $a_{i,2}$ performs the computation on the data and actor $m_{i,2}$ deals with fetching data from the memory tile. It is important to note that not all edges in the graph shown in Figure 5.2 and Figure 5.3 represent an actual transfer of data in the system. Only tokens that are sent over the bold edges must physically be transfered from one memory location to another. Tokens sent over the non-bold edges require no actual transfer of data (i.e. transformations can be done in place). This convention is used in the remainder of this chapter for all edges.

The HSDF model contains a pair of actors a_i, m_i for each access that is performed on the memory inside the memory tile. This way of modeling memory access patterns can result in very large HSDFGs. In [135], a case study is presented on a motion vector computation algorithm. Such an algorithm is typically used in video encoders. The original SDFG that models the algorithm is similar to the SDFG shown in Figure 5.1. Both the token on the edges d_i and d_o model a frame with HDTV resolution and must be stored in a memory tile. The HSDFG that models all memory accesses contains 28800 actors. When prefetching from the

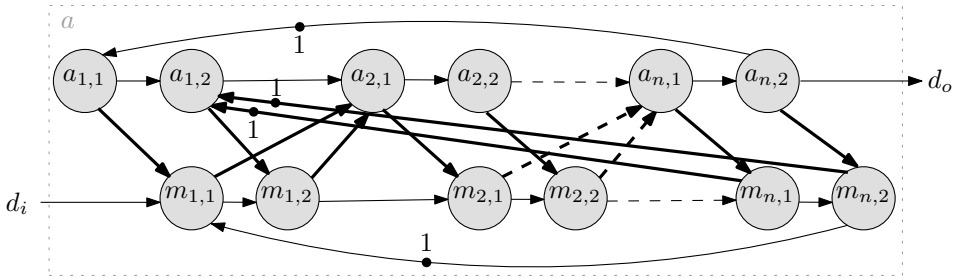


Figure 5.3: HSDF model for remote memory access with prefetching.

memory tile is taken into account, the number of actors increases to 57600.

5.3 SDF model for Memory Accesses

The number of actors in the HSDF memory access model is proportional to the number of accesses to the memory tile. For realistic applications, this can result in very large HSDFGs. These large graphs can result in very time consuming throughput computations. This section abstracts the HSDF memory access model of [135] into an SDF model that takes into account the prefetching of data from a memory tile. The abstraction is based on the observation that the execution times of the actors a_i and m_i in the HSDF model of Figure 5.2 are often independent of the index i. Typically, the actors have a very similar (or even equal) execution time for different indices. To analyze the timing behavior of the graph, it is not necessary to distinguish the different actions from each other. Using this observation, it is possible to construct an SDF memory access model in which the number of actors is independent of the number of accesses to the memory tile.

The HSDF memory access model split the original actor a (see Figure 5.1) in n pairs of actors a_i and m_i. The functionality of a is split over the a_i actors. The m_i actors model the operations that need to be performed on the memory tile. To deal with prefetching in the HSDF model, both the a_i and m_i actors are subdivided into two separate parts (see Figure 5.3). One part deals with the prefetching of data from the memory tile (actors $a_{i,1}$ and $m_{i,1}$). The other part deals with performing the computation on the data and fetching data from the memory tile (actors $a_{i,2}$ and $m_{i,2}$). The SDF memory access model shown in Figure 5.4 makes the same division. The parameter n that is used in this SDF memory acccess model is in line with the parameter used in the model of Figure 5.3. Actor a_2 models the time needed to execute the functionality of $a_{i,2}$. It receives the data needed for its firing from the memory (i.e. from m_1 and m_2) and it sends a request to the memory tile for the data needed in its next firing. Before a_2 fires for the i-th time, actor a_1 sends a request to prefetch data from

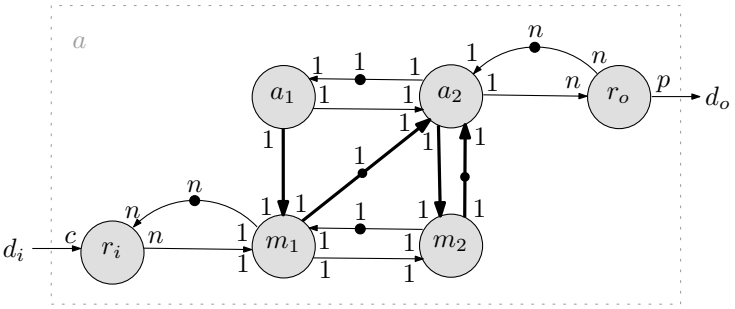


Figure 5.4: SDF model for remote memory access of tokens on an input and self-edge.

the memory needed for the $(i+1)$-th firing of a_2 to m_1. The firing of m_1 can then occur in parallel with the firing of a_2. Note that the initial tokens on the edge from m_1 to a_2 and on the edge from m_2 to a_2 make the first firing of a_2 independent of any firing of m_1 and m_2. This models the initial (possibly empty) code segment that is executed in the source code, modeled by a_2, before the first remote memory access occurs. The parallel firing of m_1 and a_2 makes it possible that part of the data needed for the $(i+1)$-th firing of a_2 is fetched in parallel with the i-th firing of a_2. This reduces the amount of data that needs to be fetched for the $(i+1)$-th firing of a_2 after its i-th firing has finished. The amount of data that can be prefetched depends obviously on whether the data access pattern of the application is known in advance. It also depends on the available storage space in the local tile memory. This should be sufficiently large to hold all data requested to be prefetched by a_1 and the data needed for the firing of a_2. It is interesting to observe that the (pre)fetching model is independent of the prefetching strategy. A designer can choose which strategy to use and use the SDF model of Figure 5.4 to analyze its timing behavior. The actor r_i and the cycle through r_i and a_1 enforce the input behavior of the actor a in Figure 5.1. When c tokens are available on the edge d_i, the actor r_i fires and consumes these tokens. This firing produces also n tokens on the edge from r_i to m_1. These tokens enable n successive firings of the actors m_1, m_2, a_1 and a_2. After the n firings of m_1, m_2, a_1 and a_2 have ended, the actor r_o produces p tokens on the edge d_o. It also produces n tokens on the edge from r_o to a_2. This enables the next set of firings of the actors a_1, a_2, m_1 and m_2.

The SDF memory access model can be optimized in certain situations. It is possible that due to storage-space constraints or data-dependencies in the code segments no data can be prefetched from the memory. In that situation, the actors a_1 and m_1 can be removed from the memory access model shown in Figure 5.4. The edges between r_i and a_1 should then be connected to a_2. Furthermore, self-edges with one initial token must be added to a_2 and m_2. These tokens

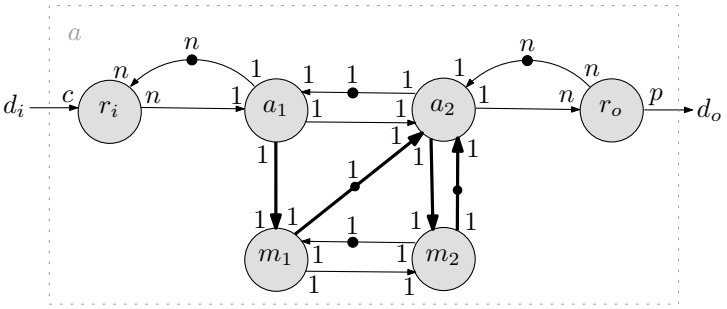


Figure 5.5: SDF model for remote memory access of tokens on a self-edge.

model the data stored in the memory of the processing and memory tile. Another optimization is possible when all data can be prefetched. In this situation, actor a_2 never requests data from the memory tile that it needs for its next firing (i.e. the size of the tokens send from m_2 to a_2 is zero). To avoid that resources are allocated for the empty memory access, the actor m_2 and all edges connected to it should be removed from the SDF memory access model. To model the memory inside the memory tile, a self-edge with one initial token should be added to actor m_1. This optimization is applied in the experiment presented in Section 5.5.

The SDF memory access model shown in Figure 5.4 deals with the situation sketched in Figure 5.1. In this SDFG, the tokens on the edges d_i and d_s, which are needed for a firing of a, are stored in a memory tile. Similar SDF models can be constructed for any combination of input, output and self-edges whose tokens are stored in a memory tile. Figure 5.5 shows, for example, the SDF model for an actor of which only the token(s) on the self-edge are stored in the memory tile. The tokens on the actor's input and output edges are stored into the memory of the processing tile that executes the actors a_1 and a_2.

5.4 Memory Allocation

Section 5.1 introduced the problem that tokens needed to fire an actor may not fit in the processing tile on which the actor is executed. To solve this problem, the actor must be split into a set of actors that use smaller tokens which fit in the memory of the processing tile. An actor represents a code segment of an application that can be executed without blocking when all its input tokens (data) are available. Splitting an actor implies dividing the code segment into smaller code segments. The requirement for each individual code segment is that the data needed to execute this segment fits into the memory of the processing tile. This behavior is often referred to as the **non-blocking actor semantics**. A technique is needed to partition a code segment into smaller segments that respect the non-blocking actor semantics. In other words, the data used by each of these smaller

code segments should fit in the memory of the processing tile that executes these code segments. This technique must also decide when data is moved between the processing tile that executes the code segments and the memory tile that is used as an additional storage resource. In other words, the technique should take a code segment that can be modeled with a single SDF actor (e.g., Figure 5.1) as input and it must output a set of code segments that can be modeled with the SDF memory access model presented in Section 5.3. This section outlines a technique that can perform this transformation. The implementation of this technique in the design flow that is presented in this thesis is left as future work.

Several techniques have been developed in the field of scratch-pad memories for similar problems. A scratch-pad is a fast compiler-managed memory that replaces the hardware managed cache. Scratch-pads are becoming more popular in embedded systems [14] as they offer, compared to caches, considerable energy saving and tighter worst-case execution time bounds [2, 148]. Various algorithms exist to distribute variables used in a code segment over a scratch-pad and background memory. These algorithms can be split into two different classes: static allocation algorithms [9, 131] and dynamic allocation algorithms [143, 144, 146]. Static allocation algorithms decide for each variable that is used in a code segment whether it should be placed in the scratch-pad or background memory. This placement is static during the whole execution of the code segment. Dynamic allocation algorithms move variables in and out of the scratch-pad during the execution of a code segment. These algorithms enable a compiler-controlled swapping of the data stored in a scratch-pad memory. The objective of these algorithms is to swap data in and out of the scratch-pad such that the total time needed to execute a code segment is minimized.

The problem addressed in this chapter is closely related to the allocation problem of scratch-pad memories. The memory inside a processing tile can be seen as a scratch-pad memory and the memory inside a memory tile as the background memory. The main difference between both problems is that both the static and dynamic allocation problem of scratch-pads do not require that all variables, which are used in a code segment, are stored in the scratch-pad. In fact, this need not even be an optimal solution to the scratch-pad allocation problem. However, all data used in the executed code segment must be available in the scratch-pad to realize the non-blocking actor semantics of the SDF model. A static scratch-pad allocation algorithm does not move data at run-time between the scratch-pad and background memory. When part of the data needed for a code segment does not fit in the scratch-pad it is placed in the background memory and accessed from this memory. This violates the non-blocking actor semantics. In other words, the code segments constructed by a static scratch-pad allocation algorithm provides only non-blocking actor semantics when all data used by the original code segment fits already in the scratch-pad memory. In this situation, it is not needed to partition the original code segment. Segmentation of the original code is required when not all data needed for a code segment (actor) fits at the same time in the scratch-pad (memory inside a processing tile). Dynamic memory allocation algorithms can

decide to move data between the scratch-pad and background memory between the execution of code segments. This makes it possible to guarantee the non-blocking actor semantics on the code segments that are constructed by such an allocation algorithm. The remainder of this section present the changes that should be made to an existing dynamic memory allocation algorithm to guarantee the non-blocking actor semantics for the code segments it constructs.

In [143], a dynamic memory allocation algorithm is presented that partitions a code segment into smaller code segments. This algorithm can be adapted such that all variables that are used in a code segment are stored in the scratch-pad before the code segment is started. In other words, the adapted version of the algorithm can partition a code segment (actor) in smaller code segments (set of actors) for which it can guarantee an execution that behaves according to the non-blocking actor semantics. Between the execution of the actors, data is moved between the memory in the processing tile and the memory in a memory tile. When sufficient storage space is available in the processing tile, (part of) these data movements can actually be done in parallel with the actor executions (i.e. data can be prefetched). The dynamic allocation algorithm from [143] starts with a heuristic that partitions a code segment into smaller segments that are called regions. For each region, it identifies which variables are used. Next, it allocates space in the scratch-pad memory for the variables that are used in the first region. It then continues with the second region etcetera. At some point, not all variables that are used in a region may fit into the scratch-pad. Some variables that are not used in this region should then be moved to the background memory to create space in the scratch-pad. The decision which variables to swap out of the scratch-pad memory into the background memory is made by a function called FindSwapoutSet. This function computes for every variable V that is used in the region a set of variables that should be moved from the scratch-pad to the background memory in order for V to fit into the scratch-pad. It uses a function called FindBenefit to compute for every variable that it considers as candidate to be moved out of the scratch-pad the benefit of this move. This benefit depends on the execution time that is saved by having V in the fast scratch-pad instead of the slow background memory. It also takes into account the time lost when it has to access variables from the background memory during the execution of the region. To achieve actor semantics, variables used in the region should be placed inside the scratch-pad. To enforce this, the execution time needed to access data from the background memory should be set to ∞ in the algorithm from [143]. As a result, the execution time of the region is infinite when it uses at least one variable that is not placed in the scratch-pad memory. The algorithm from [143] attempts to avoid this situation by moving as many variables out of the scratch-pad till all variables needed for the considered region fit into the scratch-pad. When it is impossible to put all variables needed for this region into the scratch-pad, the algorithm will abort. In this case, the actor should be mapped to a different tile with a larger memory, or the memory inside the processing tile must be enlarged to get sufficient space to store all variables that are used in the region

in its memory, or the region should be split into a set of regions that each have smaller memory requirements.

The result of the adapted dynamic memory allocation algorithm that is proposed above is a code that is partitioned in regions. Each region has a non-blocking execution semantics as all its data (tokens) are stored in the memory of the processing tile. Between the execution of actors, transfers between the memory of the processing tile and memory tile are scheduled. This execution behavior is captured in the SDF memory access model presented in the previous section. The execution time of the actors in the model can be obtained by using an analysis technique on the regions as discussed in Section 2.4.

5.5 Experimental Evaluation

This section shows how the SDF memory access model can be used to analyze the timing properties of a motion estimation application that uses data stored in a memory tile. It shows also how the model can be refined to take other architectural aspect (e.g., the interconnect) into account. As demonstrated, this enables analysis of resource requirements like storage-space allocations and interconnect bandwidth usage.

A motion estimator computes a set of motion vectors for a video frame. These motion vectors describe the motion of a block of pixel data from one frame to another. They are typically used to perform spatial up-conversion or to achieve a higher data compression (e.g., in an H.263 encoder). The motion estimation algorithm divides the current frame into a sequence of blocks of typically 8 by 8 pixels. It then takes for each block in the current frame a window of typically 32 by 32 pixels centered around the block's position from the previous frame. The motion vectors are determined by the best match, using the sum-of-absolute differences, between the pixels in the block from the current frame and the pixels in the window. The computation of all motion vectors for a frame can be modeled with the SDF actor shown in Figure 5.1. The actor reads in the current frame from the input (d_i) and the previous frame from the self-edge (d_s). Both frames are modeled with a single token. Next, it computes and outputs all motion vectors, and it outputs the current frame on the self-edge. The current frame becomes in this way the previous frame in the next firing.

Consider the situation in which the tokens that model the current and previous frame do not fit into the memory of the tile to which the motion estimation actor is mapped. These tokens must be stored in a memory tile and the motion estimation actor has to access them via the interconnect. This situation can be modeled with the SDF model shown in Figure 5.4. On each firing, actor a_2 computes a motion vector and it also requests data from the memory it needs for its next firing. The latter is done by sending a token to actor m_2. The actor a_1 sends requests to the remote memory to prefetch data. The handling of these requests in the remote memory is modeled by actor m_1.

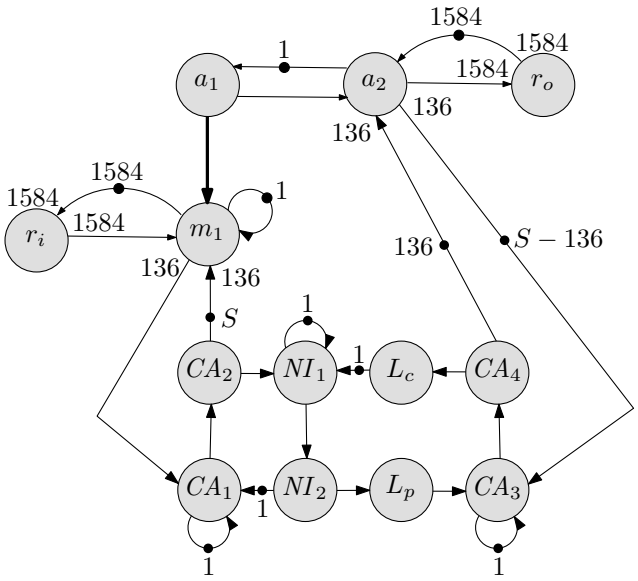


Figure 5.6: SDFG for motion estimation with remote memory access.

Assume now that the video frames that are used have a resolution of 352 by 288 pixels (CIF resolution) and that a window of 32 by 32 pixels is used. Actor a_2 computes one motion vector at a time. So, it must fire 1584 times to compute all motion vectors of one frame (i.e. $n = 1584$ in Figure 5.4). To compute one motion vector, the actor a_2 needs storage space for one window from the previous frame and a block from the current frame in the tile to which it is mapped. Assume that this tile offers sufficient storage space to store at least two windows and two blocks. In that case, the window (10124 pixels) and the block (64 pixels) needed for the next firing a_2 can be prefetched completely while a_2 is firing. Prefetching this data is modeled in the SDFG of Figure 5.4 with the transfer of one token from m_1 to a_2. This token has a size of 1088 bytes, assuming that each pixel has a size of one byte.

The bold edges in Figure 5.4 represent data that is transfered over the interconnect. In the SDF Model-of-Computation it is assumed that edges introduce no delay (see Section 4.3). In an actual implementation, the transfer of data over the interconnect has a delay. To take this delay into account, the bold edges in Figure 5.4 should be replaced with an SDF model of the interconnect. In [94], an SDF model is presented that models a guaranteed-throughput connection in the Æthereal NoC [123]. Assume that the architecture onto which the motion estimation actor is mapped uses Æthereal guaranteed throughput connections to implement the bold edges in Figure 5.4. The edge from m_1 to a_2 is the only bold edge in Figure 5.4 that involves the transfer of a large token over the interconnect.

The tokens on the other bold edges model only data transfers of a few bytes that introduce a negligible delay. To take the delay on tokens send over the edge from m_1 to a_2 into account, this edge must be refined with the connection model of [94]. Figure 5.6 shows the SDFG that models the remote memory accesses of the motion compensator to the current and previous frame. (Rates 1 have been omitted for clarity.) The SDFG contains no actor m_2 since all data needed for the firings of a_2 can be prefetched. Instead, a self-edge with one token is added to m_1. This token models the data stored in the memory tile. This optimization was already explained in Section 5.4. The edge from m_1 to a_2, present in the SDF memory access model shown in Figure 5.4, has been refined in the SDFG of Figure 5.6 with the connection model of [94]. This connection model takes into account the delay introduced by the network (actors L_p and L_c), the network interfaces (actors NI_1 and NI_2) and the communication assists at the sending and receiving side (actors CA_1, CA_2, CA_3 and CA_4). The tokens sent between the CA_i, NI_i, L_c and L_p actors model a flit in the NoC. A flit contains 64 bits of data (assuming a flit-size of 96 bits of which 32 bits are needed for the header). The tokens on the edge between m_1 and a_2 in Figure 5.4 represent 1088 bytes of data. To send this data through the NoC, a total of 136 flits is needed. The edge from CA_2 to m_1 and the edge from a_2 to CA_3 model the amount of storage space, S, that is allocated in the tiles onto which a_2 and m_2 are mapped. The number of initial tokens on these edges constrain the number of tokens that can be present in the edge from m_1 to CA_1 and the edge from CA_4 to a_2. Initially, there are already 136 tokens in the edge from CA_4 to a_2. These tokens correspond to the initial token in the edge from m_1 to a_2 in the SDFG shown in Figure 5.4. The execution time $\Upsilon(NI_2)$ of actor NI_2 models the bandwidth assigned to the guaranteed throughput connection that is used between the tile to which m_1 is mapped and the tile to which a_2 is mapped.

The number of frames that can be processed by the SDFG shown in Figure 5.6 depends on the allocated storage space S and allocated bandwidth (i.e. execution time of NI_2). Figure 5.7 shows the trade-off space of the storage space allocations, bandwidth allocation and the throughput of the motion estimation application. This trade-off space has been computed by repeatedly performing a throughput computation on the SDFG shown in Figure 5.6 for different values of S and $\Upsilon(NI_2)$. The execution times of the other actors in the graph are kept constant. These actors have the following execution times: $\Upsilon(CA_1) = \Upsilon(CA_2) = \Upsilon(CA_3) = \Upsilon(CA_4) = \Upsilon(NI_1) = \Upsilon(L_p) = \Upsilon(L_c) = 0.1\mu$s, $\Upsilon(m_1) = 20\mu$s, $\Upsilon(a_1) = 10\mu$s, and $\Upsilon(a_2) = 40\mu$s. The results show that allocating storage space for more than 170 flits has no impact on the throughput. They also show that the throughput decreases drastically when a bandwidth below 100Mbit/s is allocated.

This experiment shows that the SDF memory access model allows reasoning about the timing aspects of using memory tiles in the system. It also demonstrates how an SDF model can be refined to take other architectural aspects into account. This makes it possible to reason about storage space and interconnect bandwidth requirements. Exploration of this storage-space/bandwidth/throughput trade-

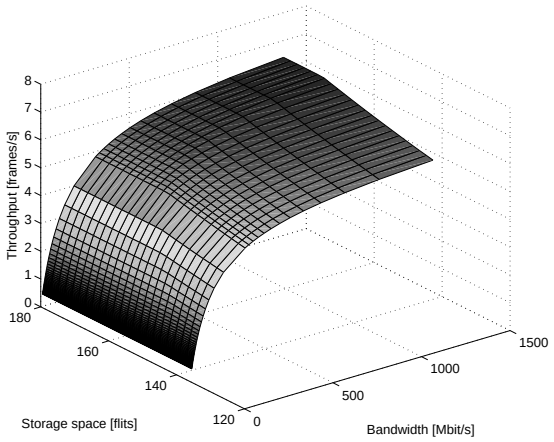


Figure 5.7: Trade-offs for motion estimation application.

off space and selecting an appropriate point from this space for a throughput-constrained application are studied in the remainder of this thesis.

5.6 Summary

This chapter presents an approach to deal with large data structures in on-chip multi-processors while guaranteeing performance. It assumes a tile-based multi-processor architecture with small memories close to the processing resources and large memories that are shared by different subsystems and applications. An SDF model is presented that allows reasoning about the timing aspects of using large, shared memories in the system. A technique is also outlined to extract the SDF memory access model from a code segment. The SDF model can take prefetching of data from shared memories into account. This prefetching is important as it allows hiding of the delay introduced by using shared memories that are distant from the processing resources.

Chapter 6

Resource Allocation

6.1 Overview

The multi-processor systems-on-chip platform presented in Chapter 3 meets with the growing computational demands of modern embedded multimedia applications. It allows multiple applications to execute concurrently. To guarantee the performance of these applications, it is required that every application running on the platform has a predictable timing behavior which is independent of other applications running on the same platform. A resource allocation strategy, which binds tasks from an application to the resources and schedules the tasks and the inter-task communication on the assigned resources, should offer this predictability.

Existing resource allocation strategies for time-constrained applications are based on HSDFGs or acyclic dependency graphs (acyclic HSDFGs). Techniques for acyclic graphs, as opposed to SDFGs, often do not take streaming (iterative, overlapping execution of the graph) into account, which makes them not very suitable for throughput-constrained multimedia applications. Moreover, an SDFG model of an application must be converted to an HSDFG to apply any of these techniques. This conversion can drastically increase the problem size, rendering this approach often infeasible. For example, the HSDFG corresponding to the SDFG shown in Figure 6.1 contains 4754 actors. The biggest problem with working on an HSDFG instead of on its corresponding SDFG is the time needed to compute the throughput. A resource allocation strategy must compute the throughput of an application bound to the system at least once in order to verify whether the throughput constraint is met. Throughput is determined by the cycles in a graph. The fastest method to compute the throughput of an HSDFG is the use of a maximum cycle ratio algorithm [128]. The fastest known variant has a run-time of 21 minutes on a P4 at 3.4GHz for the HSDFG of the H.263 decoder shown in Figure 6.1. So any HSDFG-based resource allocation strategy runs for

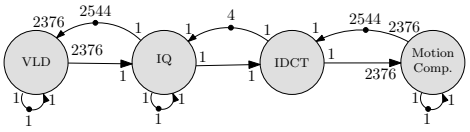


Figure 6.1: SDFG of an H.263 decoder.

at least 21 minutes on the H.263 decoder. Typically, a resource allocation strategy performs a throughput computation more than once in order to get a notion of the critical cycles in the application and tune the resource allocation. This chapter presents a novel technique for task binding and scheduling directly on SDFGs. This technique is used in step 7, 8 and 9 of the design flow introduced in Section 1.4. Because it works directly on SDFGs, it keeps the problem size typically much smaller and allows resource allocation for a larger class of applications within a limited run-time. For example, the proposed strategy has a run-time of less than 2 seconds on the H.263 decoder. It performs 21 throughput checks directly on the provisionally mapped SDFG during the trajectory to find the binding and scheduling.

The remainder of this chapter is organized as follows. The next section discusses related work in the field of resource allocation for dataflow graphs. The platform, introduced in Chapter 3, is formalized in Section 6.3 and the application model, a refinement of the SDFG MoC that annotates an SDFG with resource requirements, is formalized in Section 6.4. The resource allocation problem is defined in Section 6.5. Throughput computation for an SDFG bound to an MP-SoC is explained in Section 6.6. An SDFG-based resource allocation strategy is described in Section 6.7. An experimental evaluation is presented in Section 6.8.

6.2 Related Work

An overview of traditional scheduling and binding techniques for dataflow graphs can be found in [128]. These techniques range from fully static to fully dynamic scheduling and consider both single-processor and multiprocessor systems. All mentioned techniques for MP-SoCs use acyclic graphs in which every task must be executed once. SDFGs contain tasks (actors) that are repeatedly executed and different tasks may be executed with different rates. SDFGs also allow cyclic dependencies between different (pipelined) executions of the same task.

Resource allocation for acyclic graphs with timing guarantees is studied in [65, 82, 124]. Hu et. al assume that every task can only be bound to a single processor type [65]. The strategy decides on which processor (i.e. location) to use. It also constructs a schedule for communication on the NoC interconnect. The strategy presented in this chapter has to decide on both the processor type and its location and works, as explained, for a larger class of models. However, it abstracts from the NoC scheduling problem. It is assumed that this problem can

be solved in a later phase of the design flow (see Chapter 9). In [82], the resource allocation problem is formulated as a constraint satisfaction problem. Cyclic dependencies which determine, for example, the throughput of an application cannot be expressed in this framework. Another resource allocation strategy for acyclic graphs is presented in [124]. The strategy maps a throughput constrained graph onto a homogeneous platform. This chapter presents a strategy that targets a heterogeneous platform.

A multi-objective evolutionary algorithm to bind an application described as a Kahn Process Network to a heterogeneous MP-SoC is presented in [37]. The approach can deal with cyclic task graphs, but no timing guarantees are provided on the resulting binding.

In [95], an approach is presented to perform resource allocation for a time-constrained HSDFG on a homogeneous MP-SoC. Binding is done using a multi-dimensional bin-packing algorithm that considers the same resources as the strategy presented in this chapter does. However, the latter can handle arbitrary SDFGs while targeting a heterogeneous MP-SoC.

A method to bind an application described as a Cyclo-Static Dataflow graph onto a heterogeneous MP-SoC is given in [24]. It tries to maximize the throughput which can be realized with the available resources. Only a single application can be mapped to the system. Resource allocation to multiple applications with throughput guarantees for each of them is not considered. The strategy presented in this chapter tries to minimize resource usage under given throughput constraints, thus maximizing the number of applications that can run concurrently on the system while providing throughput guarantees.

6.3 Platform Graph

The multi-processor platform template described in Chapter 3 consists of a collection of tiles that are connected by an interconnection network. Each tile contains one processor (P) and a local memory (M). A tile contains also a set of communication buffers, called the network interface (NI), that are accessed both by the local processor and the interconnect. The resources in a tile can be described as follows. Let PT be the set of all processor types.

Definition 12. (Tile) *A tile is a 6-tuple* (pt, w, m, c, i, o) *with* $pt \in PT$ *the processor type,* $w \in \mathbb{N}_0$ *the size of the processor's TDMA time wheel (in time units),* $m \in \mathbb{N}_0$ *the memory size (in bits),* $c \in \mathbb{N}_0$ *the maximum number of connections supported by the NI, and* $i, o \in \mathbb{R}$ *the maximum incoming and outgoing bandwidth (in bits/time-unit) of the tile.*

In practice, a time wheel may already be partially occupied when binding an application to a tile. The function $\Omega : T \rightarrow \mathbb{N}_0$, with T the set of tiles, gives for a tile the size of the time wheel which is already occupied. Other resources in a tile may also be (partially) occupied. For simplicity, it is assumed that all memory

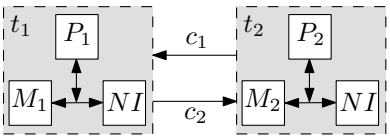


Figure 6.2: Example platform.

Table 6.1: Properties of the example platform.

	pt	w	m	c	i	o		$\mathcal{L}(c)$
t_1	p_1	10	700	5	100	100	c_1	1
t_2	p_2	10	500	7	100	100	c_2	1

(m), connections (c), and incoming (i) and outgoing (o) bandwidth specified by a tile are available for an application. Resources that are not available (i.e., used by other applications) should not be specified.

The tiles in the multi-processor platform are connected through a NoC. The NoC provides timing guarantees on the maximal latency encountered when sending data between the tiles. In this chapter, the links and routers in the NoC are ignored. It is assumed that the NoC provides point-to-point connections with a fixed latency between the tiles. When scheduling the communication of the data-elements sent through the connection on the NoC, the NoC scheduling strategy has to guarantee that the latency of the data-elements is within the latency bound of the connection. Using this assumption, the internal structure of the NoC can be abstracted away into these connections. Given this abstraction, the multi-processor platform can be described with the following graph structure.

Definition 13. (Platform Graph) *A platform graph $(T, C, \mathcal{L})$ consists of a set T of tiles, a set $C \subseteq T^2$ of connections and a latency function $\mathcal{L} : C \to \mathbb{N}$. A connection is a tuple $c = (u, v)$ through which data can be sent from a tile u to a tile v with a latency $\mathcal{L}(c)$ (in time units).*

The connections between tiles introduce a latency when data is sent between them. Each connection can have a different latency. In this way, the latency of different connections through a NoC can be taken into account. The amount of data which can be sent per time-unit (i.e. bandwidth) is limited by the incoming, i, and outgoing bandwidth, o, of the tiles. Table 6.1 gives the values of all elements in the platform of Figure 6.2.

The platform template described in Chapter 3 contains a communication assist (CA) that is ignored in the platform graph. The CA acts as a memory arbiter between the processor and NI inside a tile. In this thesis, it is assumed that the worst-case timing behavior of the CA is taken into account in the execution time of the actors (see Section 3.4). Hence, the CA can be abstracted away in the platform graph.

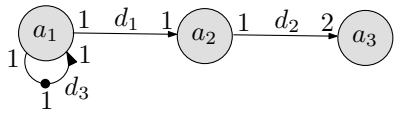


Figure 6.3: Example resource-aware graph.

6.4 Resource-Aware Application Graph

The structure of an application can be described with an SDFG. A resource allocation strategy needs also information on the resource requirements of the actors and edges in the graph. It must, for example, know to which processor types an actor can be bound and how many CPU cycles it requires on these processors. Furthermore, the model must also provide a throughput constraint which must be satisfied when the application is bound to the platform graph. An application with its resource requirements and throughput constraint is described by a resource-aware (application) graph.

Definition 14. (Resource-aware application graph) *A resource-aware application graph* $(A, D, \Gamma, \Theta, \lambda)$ *is a 5-tuple of an SDFG* (A, D)*, the functions* $\Gamma : A \times PT \rightarrow \mathbb{N}^{\infty} \times \mathbb{N}_0^{\infty}$ *and* $\Theta : D \rightarrow \mathbb{N}_0^5 \times \mathbb{R}$*, and the throughput constraint* $\lambda \in \mathbb{R}$*. Function* Γ *gives for each actor* $a \in A$ *and each processor type* $pt \in PT$ *a tuple* (τ, μ) *with* τ *and* μ *respectively the execution time (in time units) and memory requirement (in bits) of* a *when assigned to a processor of type* pt *or* ∞ *if* a *cannot be assigned to a processor of type* pt*. Function* Θ *gives for each dependency edge* $d \in D$ *from an actor* $a_i = SrcA(d)$ *to an actor* $a_j = DstA(d)$ *a 6-tuple* $(sz, \alpha_{tile}, \alpha_{src}, \alpha_{dst}, \rho, \beta)$ *with* sz *the size of a token (in bits),* α_{tile} *the memory (in tokens) required when* a_i *and* a_j *are assigned to a single tile,* α_{src} *and* α_{dst} *the memory (in tokens) required in the source and destination tile when* a_i *and* a_j *are assigned to different tiles,* ρ *the minimal latency (in time units) between the production and consumption of a token on* d *when* a_i *and* a_j *are assigned to different tiles and* β *the bandwidth (in bits/time-unit) required when* a_i *and* a_j *are assigned to different tiles.*

The minimal latency can capture timing constraints on the allowed interconnect (NoC) schedules, which will be added later in the design flow (see Chapter 8). The higher the minimal latency, the higher the interconnect scheduling freedom.

Table 6.2 shows the values of the functions Γ and Θ for the actors and edges of the resource-aware graph shown in Figure 6.3. The resource requirements can be obtained for an application using the techniques described in Section 2.4. The throughput constraint is usually a user requirement.

Table 6.2: Properties of the example resource-aware graph.

	$p_1(\tau,\mu)$	$p_2(\tau,\mu)$		sz	α_{tile}	α_{src}	α_{dst}	ρ	β
a_1	$(1,10)$	$(4,15)$	d_1	7	1	2	2	3	100
a_2	$(1,7)$	$(7,19)$	d_2	100	2	2	2	1	10
a_3	$(3,13)$	$(2,10)$	d_3	1	1	0	0	0	0

6.5 Resource Allocation Problem

A resource allocation strategy must bind each actor from the resource-aware graph $(A, D, \Gamma, \Theta, \lambda)$ to a tile in the platform graph $(T, C, \mathcal{L})$. As a consequence, also each dependency edge in the resource-aware graph is assigned to a connection between two tiles or to the memory inside a tile. The binding of actors to tiles is given by the *binding function*.

Definition 15. (BINDING FUNCTION) *A binding function is a function $\mathcal{B} : A \to T$ which gives for every actor $a \in A$ the tile $t \in T$ to which it is bound.*

Multiple applications are scheduled on a tile using a TDMA scheduler, as explained and motivated in Section 3.4. For each resource-aware graph, a time slice should be reserved on each tile which executes actors from the graph. A static-order schedule, ordering the actor execution of a resource-aware graph on a processor, must also be constructed for each tile. Both the size of the time slice and the static order schedule are given for each tile by the *scheduling function*.

Definition 16. (SCHEDULING FUNCTION) *A scheduling function is a function $\mathcal{S} : T \to \mathbb{N}_0 \times SO$, where SO is the set of all static order schedules. It gives for a tile $t \in T$ from the platform graph a tuple (ω, S), where ω is the size of the TDMA time slice reserved for the resource-aware graph and S is a static order schedule for the actors from the resource-aware graph which are bound to t.*

The following notations are used in the remainder. For each tile $t \in T$, $t = (pt_t, w_t, m_t, c_t, i_t, o_t)$ and $\mathcal{S}(t) = (\omega_t, S_t)$; for each actor $a \in A$ and processor type $pt \in PT$, $\Gamma(a, pt) = (\tau_{a,pt}, \mu_{a,pt})$, and, for each dependency edge $d \in D$, $\Theta(d) = (sz_d, \alpha_{tile,d}, \alpha_{src,d}, \alpha_{dst,d}, \rho_d, \beta_d)$. The set of all actors $a \in A$ bound to $t \in T$ is denoted with A_t. Using the set A_t, three sets of dependency edges are defined. The first set $D_{t,tile}$ contains all dependency edges of which both the source and destination actor are bound to t. Set $D_{t,src}$ contains all dependency edges of which the source actor is bound to t and the destination actor is bound to a different tile; $D_{t,dst}$ contains all dependency edges of which the destination actor is bound to t and the source actor is bound to a different tile.

Binding and scheduling functions give a resource allocation for a resource-aware graph on a platform graph. This allocation is called valid if and only if the throughput constraint is met and not more resources are allocated than available. The next section explains how throughput is computed. To guarantee that not

more resources are allocated than available, the following must hold for each tile $t \in T$:

1. the allocated time slice is available:
   $\omega_t \leq w_t - \Omega(t)$,
2. not more memory is allocated than available:
   $\sum_{d \in D_{t,tile}} \alpha_{tile,d} \cdot sz_d + \sum_{d \in D_{t,src}} \alpha_{src,d} \cdot sz_d + \sum_{d \in D_{t,dst}} \alpha_{dst,d} \cdot sz_d + \max_{a \in A_t} \mu_{a,pt} \leq m_t$,
3. not more connections are allocated than available:
   $|D_{t,src}| + |D_{t,dst}| \leq c_t$,
4. not more input and output bandwidth is allocated than available:
   $\sum_{d \in D_{t,dst}} \beta_d \leq i_t \wedge \sum_{d \in D_{t,src}} \beta_d \leq o_t$.

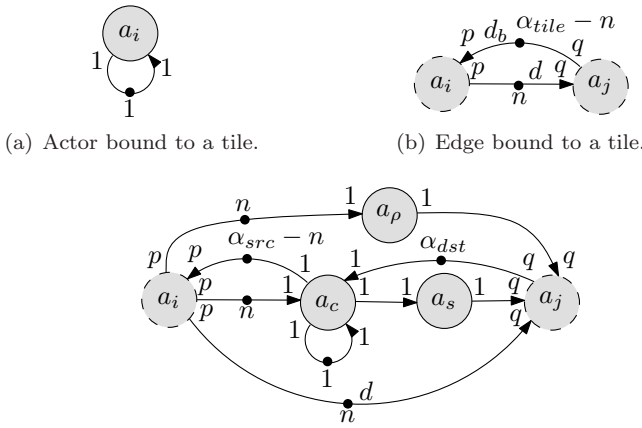
The second constraint guarantees that the amount of memory allocated on a tile does not exceed the total amount of memory available on the tile. The constraint uses the assumption that edges cannot share memory space. However, actors from the same application are assumed to share their memory space. The static-order schedule on the actors guarantees that only one actor is firing at the same tile. So, only this actor needs memory space to store its state. When the actor firing ends, the state can be discarded. Any data that is needed for a next firing of this actor should be outputted on a self-edge of the actor. Doing so, makes the memory requirements explicit.

6.6 Throughput Analysis

In Section 4.3, the timed SDFG model with its operational semantics is presented along with a technique to compute the throughput of the graph. It assumes a platform with infinite resources (i.e. actors do not share processors and communication does not introduce a delay). Various aspects of resource sharing can be modeled into a timed SDFG. Other aspects like the timewheel and static order schedules of tiles, must be handled in the operational semantics. This reduces the number of possible ways to fire an actor in an SDFG to those who respect the constraints that come from the resource allocation.

6.6.1 Modeling Resource Allocations in SDFGs

A resource allocation for a resource-aware graph $(A, D, \Gamma, \Theta, \lambda)$ on the resources in platform graph $(T, C, \mathcal{L})$ is given by a binding function $\mathcal{B}$ and a scheduling function $\mathcal{S}$. The resource allocation decisions are modeled as follows into a timed SDFG $G_b = (A_b, D_b, \Upsilon)$. For every actor $a \in A$ there is a corresponding actor $a_b \in A_b$. Its execution time, $\Upsilon(a_b)$, is equal to the execution time of a on the



(a) Actor bound to a tile.

(b) Edge bound to a tile.

(c) Edge bound to a connection.

Figure 6.4: Modeling of resource constraints in a timed SDFG.

processor type contained in the tile $t = (pt, w, m, c, i, o) \in T$ to which a is bound (i.e. $\Upsilon(a_b) = \tau(\Gamma(a, pt))$. Only one instance of a_b can be executing at the same moment in time on a tile (i.e. no auto-concurrency). This is modeled by adding a self-edge on a_b with rate one to the set D_b (see Figure 6.4(a)) and having one token on it in the initial state of the SDFG. Every edge $d \in D$ with $\Theta(d) = (sz, \alpha_{tile}, \alpha_{src}, \alpha_{dst}, \rho, \beta)$ of which both the source and destination actor are bound to the same tile ($\mathcal{B}(SrcA(d)) = \mathcal{B}(DstA(d))$), is modeled in G_b with two edges as shown in Figure 6.4(b). The edge d_b with initially $\alpha_{tile} - n$ tokens, with n the number of tokens in the initial state of the resource-aware graph, limits the storage space of the edge d to the memory size allocated for the edge d by the resource allocation strategy. Any edge $d \in D$, with n initial tokens, whose source and destination actor are bound to different tiles ($\mathcal{B}(SrcA(d)) \neq \mathcal{B}(DstA(d))$) uses a connection $c \in C$ in the platform graph. The delay and memory constraint introduced by this resource binding are modeled with the graph shown in Figure 6.4(c). The memory constraint at the source (destination) tile is modeled with the dependency edge from a_c to a_i (a_j to a_c). The self-edge on a_c enforces that tokens are sent sequentially over the connection. Actor a_c is used to model the delay for sending a token over the connection. Its execution time, $\Upsilon(a_c)$, is equal to $\mathcal{L}(c) + \lceil sz/\beta \rceil$. Actor a_c is a very simple connection model. It can be replaced with a more detailed model, such as the network-on-chip connection model of [94]. Actor a_ρ models the minimal latency ρ between the production and consumption of a token on the dependency edge d when it is bound to a connection in the platform graph. The execution time of a_ρ is equal to $\Upsilon(a_\rho) = \rho$. TDMA scheduling guarantees that the throughput of an application is not influenced by other applications running on the same platform. No assumptions are made on

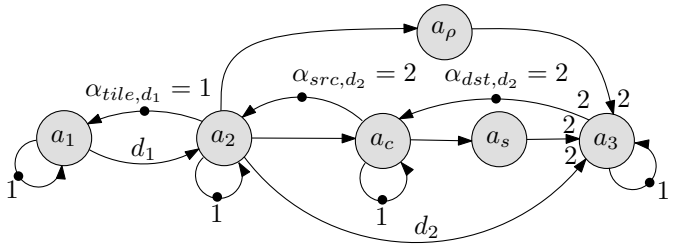


Figure 6.5: Binding-aware SDFG for the example SDFG of Figure 6.3.

the position of a time slice on the time wheel. This guarantees that a time wheel can always be defragmented in one rotation and it leaves flexibility for the run-time scheduling of applications on a platform. Consider the worst-case situation in which a token needed to fire actor a_j running at a tile $t = (pt, w, m, c, i, o) \in T$ arrives exactly at the end of the time slice with size ω allocated to the application by scheduling function $\mathcal{S}$. In that situation, a_j has to wait at most $w - \omega$ time steps before it can fire. To guarantee that the throughput analysis of the model is conservative with respect to an implementation, it is assumed that a token sent between actors bound to different tiles is always delayed by this amount of time. This is modeled by a_s, which has an execution time $\Upsilon(a_s) = w - \omega$. The model shown in Figure 6.4(c) contains an edge d that expresses the dependency of actor a_j on actor a_i, which was also present in the resource-aware SDFG. The same dependency between the two actors is also captured with the two edges going from a_i via a_ρ to a_j. Therefore, an efficient implementation can ignore the edge d.

Definition 17. (Binding-aware SDFG) *A binding-aware SDFG is a 6-tuple* $(A_b, D_b, \Upsilon, T, \mathcal{B}, \mathcal{S})$ *consisting of a timed SDFG* (A_b, D_b, Υ) *that models the binding of a given application graph to a set of tiles* T*, by binding function* $\mathcal{B}$ *and the time-slice allocation given by scheduling function* $\mathcal{S}$*, as explained above.*

Consider as an example the resource-aware graph shown in Figure 6.3 and the platform of Figure 6.2. Assume that the actors a_1 and a_2 are bound to tile t_1 and actor a_3 to t_2. The dependency edge d_1 is then bound to t_1 and the edge d_2 is bound to the connection c_1. The binding-aware SDFG that models these binding decisions is shown in Figure 6.5 (omitting rates 1 for clarity). The execution time of a_1 and a_2 is then equal to 1 and the execution time of a_3 is equal to 2 (see Table 6.2). The execution time, $\Upsilon(a_c)$, of the actor a_c that models the delay for sending a token over the connection, is equal to $\mathcal{L}(c_1) + \lceil sz_{d_2}/\beta_{d_2} \rceil$. Actor a_s, modeling the worst-case delay due to time slice allocations, has an execution time $\Upsilon(a_s) = w_{t_2} - \omega_{t_2}$, which depends on the schedule function $\mathcal{S}$. Note that the actors a_c, a_s and a_ρ are not bound to a processor, as these actors only model the interconnect between two tiles.

6.6.2 Operational Semantics

One option to model static order schedules in dataflow graphs is proposed in [13]. As shown in Section 4.5, this requires a conversion of the SDFG to an HSDFG. This conversion leads to an increase in the time needed for the throughput computation. To avoid this issue, the scheduling function, i.e., the time wheel allocations and static-order schedules, is not modeled into the binding-aware SDFG. Instead, these aspects are taken as constraints on the execution of the binding-aware SDFG. This requires an extension of the state of a timed SDFG (see Definition 6) to keep track of the state of the static-order schedules and time wheels. Recall the definition of a static-order schedules introduced in Section 3.3; a static-order schedule for a set A of actors is defined as a finite or infinite sequence $a_1 a_2 a_3 ... \in A$. Practical infinite static-order schedules consist of a (possible empty) sub-sequence which is seen once followed by a finite sub-sequence which is infinitely often repeated. The initial position of a static-order schedule is denoted with s_0, and the function ϱ gives the next position of a static-order schedule.

Definition 18. (State) *The state of a binding-aware SDFG* $(A_b, D_b, \Upsilon, T, \mathcal{B}, \mathcal{S})$ *is a 4-tuple* $(\delta, \upsilon, \iota, \kappa)$*. The pair* (δ, υ) *represents the state of the timed SDFG* (A_b, D_b, Υ)*. For each tile* $t \in T$*, the position of the time wheel in that state is given by* $\iota : T \rightarrow \mathbb{N}_0$*. The function* κ *associates to every tile* $t \in T$ *the state of the schedule (i.e. position in the static-order schedule). The initial state of a binding-aware SDFG is given by some initial token distribution* δ *and the initial position of the static-order schedules, which means that the initial state equals* $(\delta, \{(a, \{\}) \mid a \in A\}, \{0 \mid t \in T\}, \{s_0(S(\mathcal{S}(t))) \mid t \in T\})$*.*

The dynamic behavior of a timed SDFG is described by transitions. Three different types are distinguished: start of actor firings, end of firings, and time progress in the form of clock ticks. The binding and scheduling functions $\mathcal{B}$ and $\mathcal{S}$ impose additional constraints when these transitions are allowed to occur in a binding-aware SDFG as compared to a timed SDFG.

Definition 19. (Transition) *A transition of a binding-aware SDFG* $(A_b, D_b, \Upsilon, T, \mathcal{B}, \mathcal{S})$ *from state* $(\delta_1, \upsilon_1, \iota_1, \kappa_1)$ *to state* $(\delta_2, \upsilon_2, \iota_2, \kappa_2)$ *is denoted by* $(\delta_1, \upsilon_1, \iota_1, \kappa_1) \xrightarrow{\beta} (\delta_2, \upsilon_2, \iota_2, \kappa_2)$ *where label* $\beta \in (A \times \{start, end\}) \cup \{clk\}$ *denotes the type of transition. The change from* (δ_1, υ_1) *to* (δ_2, υ_2) *in the various transitions of a binding-aware SDFG is identical to the state change in the timed SDFG and therefore omitted in the following.*

- *A start of firing transition of* $a \in A$ *is enabled in a binding-aware SDFG if it is enabled in the corresponding timed SDFG and when* $t = \mathcal{B}(a) \neq NIL$*; it must further hold that* $\varrho(\kappa_1(t)) = a$*. It results in* $\iota_2 = \iota_1$*, and* $\kappa_2 = \kappa_1$*.*
- *An end of firing transition of* $a \in A$ *is enabled in a binding-aware SDFG if it is enabled in the corresponding timed SDFG. It results in* $\iota_2 = \iota_1$*, and*

$\kappa_2 = \kappa_1$ *if* $\mathcal{B}(a) = NIL$ *or if* $\mathcal{B}(a) \neq NIL$, $\kappa_2(t) = \varrho(\kappa_1(t))$ *for* $t = \mathcal{B}(a)$ *and* $\kappa_2(t) = \kappa_1(t)$ *for* $t \neq \mathcal{B}(a)$.

- *A clock transition is enabled in a binding-aware SDFG if it is enabled in the corresponding timed SDFG. It results in* $\iota_2 = \{(\iota_1(t) + 1) \bmod w(t) | t \in T\}$, $\kappa_2 = \kappa_1$ *and the set* M *of actors whose remaining execution time is reduced contains all actors* $a \in A$ *for which* $t = \mathcal{B}(a) = NIL$ *or* $\iota_1(a) \geq w(t) - \omega(\mathcal{S}(t))$.

In a clock transition, the position of all time wheels is advanced with one position and after a complete rotation it is set to zero. The transition reduces the remaining execution time of all active firings of actors which are not bound to a tile and of those which are bound to a tile on which the current TDMA time slot is reserved. The reason to advance the time for actors not bound to a tile is to ensure proper progress of delay actors like a_s in Figure 6.5. The transition system guarantees that when an actor $a \in A$ is bound to a tile $t \in T$, its remaining execution time is lowered if and only if the position of the time wheel of tile t is larger or equal to the size of the wheel minus the time slice reserved for the application. In practice, an actor firing on a tile will never occur later than this moment. This guarantees that the throughput analysis, which is explained below, is conservative. Whenever an actor is firing when the time slice of the application ends, the firing is pre-empted and continued in the next rotation of the time wheel.

6.6.3 Throughput Computation

Similar to the self-timed execution of a timed SDFG (see Definition 9), the constrained self-timed execution of a binding-aware SDFG is defined as follows. Note that Definition 19 already guarantees that the proper set of actor firings makes progress, so it is only necessary to prohibit clock transitions when actor firings are enabled.

Definition 20. (Constrained self-timed execution) *A constrained execution is self-timed if and only if clock transitions only occur if no start transition is enabled.*

Theorem 2. (Periodic behavior) *The state space of a constrained self-timed execution of a binding-aware SDFG* $(A_b, D_b, \Upsilon, T, \mathcal{B}, \mathcal{S})$ *contains always exactly one cycle (in terms of macro steps from one clock transition to another clock transition).*

Proof. The binding-aware SDFG $(A_b, D_b, \Upsilon, T, \mathcal{B}, \mathcal{S})$ is by definition strongly connected as every edge has a bounded storage space which is modeled in the graph using the constructs of Figure 6.4. This means that every actor depends on tokens from every other actor, which limits the difference between the number of firings of actors with respect to each other. This implies that there exists a bound on the number of simultaneous actor firings and the number of tokens in

any edge. Furthermore, every tile has a static-order schedule of finite length and a time wheel with only a finite number of slices. Hence, there is only a finite number of different reachable states. Further, in the transition system, there is always at least one transition enabled (even in a deadlock state, there is still a clock transition enabled), which implies that the number of transitions that will occur is infinite. By the pigeon hole principle, at least one of the finite number of reachable states is visited infinitely often. Since the self-timed execution is deterministic (when considering the execution in 'macro steps' from one clock transition to another), there is only one transition to leave any (recurrent) state. Hence, there is exactly one cycle in the state space (in terms of macro steps). □

The throughput of a graph refers to how often an actor produces an output token. In a constrained self-timed execution, actors fire as soon as possible while respecting the constraints from the binding and scheduling functions $\mathcal{B}$ and $\mathcal{S}$. This guarantees that the execution gives maximal throughput under the resource constraints.

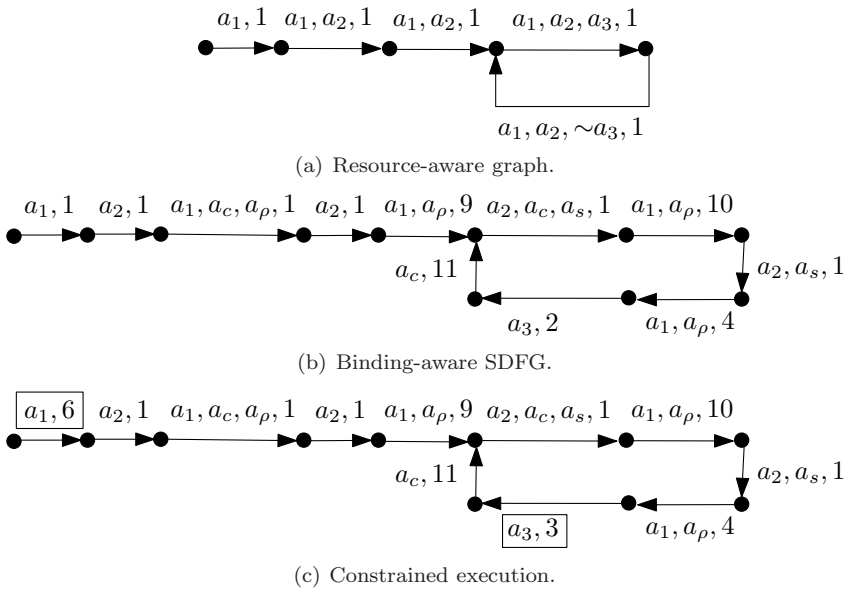
Definition 21. (THROUGHPUT) *The throughput $Th(a)$ of an actor a for the constrained self-timed execution σ of a binding-aware SDFG $G = (A_b, D_b, \Upsilon, T, \mathcal{B}, \mathcal{S})$ is defined as the average number of firings of a per time unit in σ. If SDFG (A_b, D_b) is consistent, then the throughput of G is defined as*

$$Th(G) = \frac{Th(a)}{\gamma(a)},$$

for some arbitrary actor $a \in A_b$, where γ is the repetition vector of (A_b, D_b).

As explained in Section 4.4, the throughput of an actor in an SDFG can be computed by executing the SDFG in a self-timed manner while remembering all visited states until a state is revisited. At that point, the periodic phase is reached and the throughput of an actor a is equal to the number of firings of a in one period of the periodic phase divided by the number of clock transitions in the period. The throughput of a binding-aware SDFG can be computed in the same way.

As an example, a comparison is made between the throughput found when analyzing the self-timed execution of the resource-aware graph shown in Figure 6.3 or the self-timed execution of the corresponding binding-aware SDFG shown in Figure 6.5 (assuming execution times 1, 1, and 2 for the actors respectively) or the constrained self-timed execution of this binding-aware SDFG. Figure 6.6(a) shows the state-space of the resource-aware graph. As before, states are represented by black dots and state transitions are indicated by edges. The label with a transition indicates which actors start their firing in this transition and the elapsed time till the next state is reached. The continuation of an actor firing is indicated by a tilde in front of the actor label. Actor a_3 executes once every 2 time-units in the self-timed execution of the resource-aware SDFG (i.e. its throughput is 1/2).



(a) Resource-aware graph.

(b) Binding-aware SDFG.

(c) Constrained execution.

Figure 6.6: State-space of the resource-aware graph, binding-aware SDFG and the constrained self-timed execution.

Since a_3 occurs only once in the repetition vector, also the graph throughput is $1/2$. This is the maximal achievable throughput taking into account only the dependencies inherent in the SDFG. The self-timed state-space of the binding-aware SDFG is shown in Figure 6.6(b). The limited storage space of d_1 causes a_1 and a_2 to fire in sequence and communication and synchronization is taken into account via firings of actors c and s. Actor a_3 executes once every 29 time-units, which is the maximal achievable throughput of the actor and the graph that can be guaranteed under the worst-case synchronization conditions while taking the binding into account. The explored state-space which considers the other resource constraints on the execution of the SDFG is shown in Figure 6.6(c). The chosen static-order schedules $(a_1 a_2)^*$ and a_3^* are in line with the self-timed schedule, so they do not affect the result. 50% of the TDMA time wheels are allocated to the application. These time slot allocations cause actor a_1 and a_3 to post-pone their firings for respectively 5 and 1 time-unit in the worst case (see boxes in Figure 6.6(c)). As a result, actor a_3 fires only once every 30 time-units. The throughput is therefore equal to $1/30$.

In [17], TDMA time slice allocations are modeled by increasing the execution time of every actor firing with the fraction of the TDMA time wheel which is not reserved by the application. This increases the execution time of actor a_3 in the example with 5 time units. This is the maximum time the constrained self-timed execution will post-pone the firing of an actor. In many situations, the time with

which a firing is post-poned is less. Hence, the constrained self-timed execution gives a more accurate throughput result. This reduces the resource requirements of the application while guaranteeing its timing behavior.

6.7 Resource Allocation Strategy

The resource allocation strategy consists of three main steps which are each executed once. First, an actor binding is constructed, then a static order schedule for each tile containing actors of the resource-aware graph, and finally time slices are allocated.

6.7.1 Resource binding

The resource binding step must bind every actor from the resource-aware graph to a tile in the platform graph. An important objective in the resource allocation strategy is to meet the throughput constraint specified by the resource-aware graph. For this reason, it is important that actors whose execution time have a large impact on the throughput of the application are considered first. The throughput of an SDFG is known to be limited by its critical cycle [128]. This is a cycle in the corresponding HSDFG with the maximal ratio between the sum of the execution time of the actors on the cycle and the sum of the number of tokens on the edges of the cycle. The conversion of an SDFG to an HSDFG can lead to an exponential increase in the number of actors in the graph [46]. This may make it infeasible to analyze the HSDFG of a resource-aware graph to identify the actors on its critical cycle. Therefore, the binding step tries to estimate the criticality of all cycles in the graph (and the actors on them) directly on the SDFG. This is done with the cost function given by Equation 6.1, with $a \in A$ an actor, C the set of cycles through a, γ the repetition vector of the resource-aware SDFG and n the number of tokens on edge d in the initial state of the SDFG.

$$cost(a) = \max_{c \in C} \frac{\sum_{actors\ b \in c} \gamma(b) \cdot \operatorname*{avg}_{\{pt \in PT \mid \tau_{b,pt} \neq \infty\}} \tau_{b,pt}}{\sum_{edges\ d=(u,v) \in c} n/Rate(v)}, \tag{6.1}$$

After sorting the actors in decreasing order, the resource allocation strategy tries to bind the actors in the given order to the tiles. For each actor $a \in A$, it may have to choose from a number of different tiles $T' \subseteq T$. The objective of the resource allocation strategy is to balance the load of the application equally over all tiles and to minimize the latency of the connections that are used. The load of a tile is estimated by the amount of processing performed on its processor relative to the total amount of processing needed for the application, the fraction of memory used and the average fraction of occupied connections and bandwidth. Given a (partial) binding, and the corresponding sets A_t, $D_{t,tile}$, $D_{t,src}$, $D_{t,dst}$ (see Section

6.5), these aspects are captured in the following definitions, with t a tile and γ the repetition vector of the resource-aware graph.

$$l_p(t) = \frac{\sum_{a \in A_t} \gamma(a) \cdot \tau_{a,pt}}{\sum_{a \in A} \gamma(a) \cdot \max_{\{pt \in PT | \tau_{a,pt} \neq \infty\}} \tau_{a,pt}}$$

$$l_m(t) = (\max_{a \in A_t} \mu_{a,pt} + \sum_{d \in D_{t,tile}} \alpha_{tile,d} \cdot sz_d + \sum_{d \in D_{t,src}} \alpha_{src,d} \cdot sz_d + \sum_{d \in D_{t,dst}} \alpha_{dst,d} \cdot sz_d)/m_t$$

$$l_c(t) = avg\left(\sum_{d \in D_{t,src}} \frac{\beta_d}{o_t}, \sum_{d \in D_{t,dst}} \frac{\beta_d}{i_t}, \frac{|D_{t,src}| + |D_{t,dst}|}{c_t}\right)$$

Besides the load of the resources in a tile, the resource allocation strategy must also consider the latency of the connections between the used tiles. It should try to minimize the sum of the latencies of the connections that are used. This minimizes the time spent on communication. Furthermore, the latency of a connection is typically related to the locality of the two tiles it connects. Minimizing latency results then in binding actors to tiles which are physically close to each other. The latency load of binding an actor a to a tile t is estimated by the relative latencies of the connections used with this binding compared to binding the actor to alternative tiles. Given a (partial) binding and the corresponding sets A_t, $D_{t,tile}$, $D_{t,src}$, $D_{t,dst}$ (see Section 6.5). The latency load when binding an actor a to a tile t is given by the following definition. It uses a set $D_{t,a} = \{D_{t,src} \bigcup D_{t,dst} | SrcA(d) = a \vee DstA(d) = a\}$ that contains all dependency edges connected to a that are bound to a connection in the platform graph.

$$l_l(t,a) = \frac{\sum_{d \in D_{t,a}} \mathcal{L}(\mathcal{B}(SrcA(d)), \mathcal{B}(DstA(d)))}{\max_{t \in T'} \sum_{d \in D_{t,a}} \mathcal{L}(\mathcal{B}(SrcA(d)), \mathcal{B}(DstA(d)))}$$

Equation 6.2 combines the processing load, memory load, communication load and latency load into in a single cost function for binding an actor a to a tile t.

$$cost(t,a) = c_1 \cdot l_p(t) + c_2 \cdot l_m(t) + c_3 \cdot l_c(t) + c_4 \cdot l_l(t,a) \tag{6.2}$$

The constants in the function are specified by the user of the binding step. This enables the user to trade-off how the various loads of the tile are weighted with respect to each other. The algorithm tries to bind actor a to a tile $t \in T'$ in the increasing order given by the tile cost function based on the current partial

Table 6.3: Binding of actors to tiles.

c_1, c_2, c_3, c_4	a_1	a_2	a_3
$1, 0, 0, 0$	t_1	t_1	t_2
$0, 1, 0, 0$	t_2	t_1	t_1
$0, 0, 1, 0$	t_1	t_1	t_1
$0, 0, 0, 1$	t_1	t_1	t_1
$1, 1, 1, 1$	t_1	t_1	t_2

binding with a bound to t. When a tile $t \in T'$ is found for which it holds that the binding of a to t does not conflict with the constraints given in Section 6.5 it binds a to t and the algorithm continues with the next actor. When all tiles are tried and no valid binding is found, the problem is considered infeasible. Table 6.3 shows the resulting binding of actors of the running example for various settings of the constants.

After binding all actors to a tile, an optimization is performed to improve the load balance of the tiles. This is done by considering all the actors in reverse order. When reconsidering the binding of an actor $a \in A$ which is bound to a tile $t \in T$, its binding is first removed. Next, all tiles T' to which a can be bound are sorted using Equation 6.2, considering the load of all tiles when the whole application graph except actor a is bound. The algorithm then tries to bind a to a tile $t \in T'$ in the increasing order given by the cost function. Note that it will always be possible to find a valid binding as the original binding is one of the bindings which is tried.

6.7.2 Constructing static-order schedules

For each tile, a static-order schedule must be constructed that orders the firings of all actors bound to it. A list-scheduler is used to construct these static-order schedules for all tiles at once. The schedules are constructed via an execution of the binding-aware SDFG, assuming that for each tile 50% of the available time wheel is allocated to the resource-aware graph. Experiments have shown that the percentage of the time wheels that is assumed to be allocated to the resource-aware graph has little influence on the constructed static-order schedules because typically only a few actors, which are mapped to the same processor, are ready to fire at the same moment in time. Through actors like a_s in Figure 6.5 the delay for tokens sent between tiles is taken into account in the schedule construction. When an actor becomes enabled in the execution of the binding-aware SDFG, it does not start its firing immediately. Instead the actor is added to the ready list of the tile it is bound to. When no actor is firing on the tile, the first actor is removed from the list and its firing is started. At this moment, the actor is added to the schedule of the tile. The execution ends as soon as a recurrent state is found. At this point, a finite-length schedule has been constructed for

each tile. For the example, the scheduler constructs for tile t_1 a schedule with 17 states - $a_1a_2a_1a_2a_1a_2a_1a_2a_1(a_2a_1a_2a_1a_2a_1a_2a_1)^*$. After constructing the schedule, an optimization is performed to remove all recurrent occurrences of the same scheduling sequence. In this way, the schedule on t_1 is reduced to $(a_1a_2)^*$.

After the static-order scheduling, it is possible to perform life-time analysis on the tokens in the dependency edges that are bound to a tile. This allows tighter memory allocation for these edges. This optimization is left as future work.

6.7.3 Time slice allocation

The final step of the resource allocation strategy involves the allocation of time slices for all tiles. A binary search algorithm is used, which guarantees that a time slice allocation satisfying the throughput constraint is found if it exists. The search between the initial bounds of 1 time slice and the entire (remaining) time wheel continues until the throughput of the binding-aware SDFG constrained by the current slice allocation is at most 10% larger than the throughput constraint. The stop criterion makes a trade-off between the run-time of the resource allocation strategy and the resource usage of the mapped application. The time slice allocation ends unsuccessfully if the allocation of the entire remaining time wheels is insufficient to meet the throughput constraint.

If successful, the slice allocation step so far allocates equal fractions of the remaining time wheel for each tile to which at least one actor is bound. This is based on the assumption that the processing load is perfectly balanced over the tiles. However, in case of an imperfect load balance it may be possible to reduce the allocated time slices using another binary search. The upper bound for every tile $t \in T$ is equal to the slice found in the previous step (i.e. ω_t) and the lower bound for every tile t is $\left\lfloor \frac{l_p(t) \cdot \omega_t}{\max_{t \in T} l_p(t)} \right\rfloor$ which takes into account the relative load of each tile. The binary search is continued on all tiles simultaneously using the above mentioned bounds. The binary search is ended when the slices can no longer be reduced without violating the throughput constraint.

6.8 Experimental Evaluation

6.8.1 Experimental setup

A benchmark is needed to evaluate the run-time and quality of the resource allocation strategy and explore the impact of different parameter values in the tile-cost function. A benchmark of four sets of resource-aware graphs was generated using SDF^3 (see Chapter 4.6). The first set contains processing intensive graphs that have large execution times, do not communicate too often and have small token sizes and states. The second and third set are memory and communication intensive. The fourth set contains both SDFGs which are balanced with respect to their processing, memory and communication requirements and graphs which

Table 6.4: Average number of resource-aware graphs bound.

cost function	c_1, c_2, c_3, c_4	set 1	set 2	set 3	set 4
1:	$1, 0, 0, 0$	16.89	5.22	9.78	9.33
2:	$0, 1, 0, 0$	18.67	8.56	12.67	11.33
3:	$0, 0, 1, 0$	25.56	8.44	14.00	13.22
4:	$0, 0, 0, 1$	16.44	3.33	7.11	6.33
5:	$1, 1, 1, 1$	17.00	5.78	11.00	11.56
6:	$0, 1, 4, 0$	22.67	8.56	13.76	14.11

are dominated by one or two of these aspects. The sets were ordered to mimic the order in which applications are considered for mapping. For each set, three different sequences of graphs were generated to eliminate effects from the random generator.

Three different platform graphs are used in the experiments. Each platform graph is a 3x3 mesh-based platform with 3 different types of processors. The graphs differ in the memory size and number of supported connections. All processors have an equally sized time wheel. The connections between the tiles are assigned a latency which is small compared to the execution time of the actors. This is realistic as the latency of an interconnect in an MP-SoC is typically much smaller than the execution time of the executed tasks.

Each set of graphs from our benchmark has been tested with six different settings for the tile-cost function (see first column Table 6.4). For a given tile-cost function, platform graph, and sequence of resource-aware graphs, resources are allocated to resource-aware graphs till no valid resource allocation is found for a graph. This gives a conservative estimate on the number of applications for which resources can be allocated on the platform. A design-time pre-processing step that orders the applications to optimize the order in which they are handled, a (run-time) mechanism that rejects an application and continues with the next one or another implementation version of the rejected application, and/or a platform dimensioning step may improve the results.

6.8.2 Experiments on the benchmark

Table 6.4 shows the number of resource-aware graphs which could be bound for each tile-cost function and set of graphs from the benchmark. It averages over the three sequences of graphs contained in each set and the three platform graphs used in the experiments. The average run-time of the strategy for a single resource-aware graph on a P4 at 3.4GHz is 5 seconds. On average, each run of the algorithm invokes 16.1 times the throughput computation technique described in Section 6.6 which would make a trajectory based on a conversion to HSDFG very expensive. The result of the set with computation intensive tasks (set 1) shows that it is important to consider not only the processing (1st tile-cost function), but also

the communication (3rd tile-cost function). The reason for this is that when the processing load is balanced, many dependency edges are bound to a connection, requiring synchronization between tiles. As a result, larger time slices need to be allocated on the tiles to meet the throughput constraint than when more actors of a single resource-aware graph are bound to the same tile. The latter effect is achieved by the 3rd tile-cost function. In-line with its objective, the 2nd tile-cost function, which considers the memory resources, performs well on the memory constrained graphs (set 2). As expected, the 3rd tile-cost function, which considers the connection and bandwidth resources, performs best on the communication intensive graphs (set 3). The 4th tile-cost function, which tries to minimize the interconnect latency, performs worst on all sets. It tries to bind as many actors as possible to the same tile, given the memory, connection and bandwidth constraints. However, the time slice allocation step quickly fails to find large enough time slices to meet the throughput constraint. As no iteration is performed between the steps of the resource allocation, the strategy fails to find a valid resource allocation. The results show further that the 5th tile-cost function, which considers all resources, gives an average result for all sets. This is to be expected as it balances all resources and as such does not give priority to the most constrained resource in any of the sets. The results show that it is important to minimize the number of connections in order to limit the synchronization overhead. They also show that balancing the memory usage is an important secondary objective as the 2nd tile-cost function gives good results for most sets. Based on these observations, a 6th tile-cost function $(0, 1, 4, 0)$ was devised. This tile-cost function emphasizes minimization of the number of connections while balancing memory usage. Using this cost function, the largest number of application graphs is allocated onto the architecture for the set with mixed resource requirements (set 4). This shows that it is possible to guide the resource allocation through the tile-cost function. The number of mapped applications is however close to the number of applications mapped with the 3th tile-cost function. This confirms that the most important objective is the minimization of the number of connections. Connections have a large impact on the memory requirements (storage space must be reserved in two tiles) and on the time slice allocation (large enough slices are needed to compensate for the worst-case synchronization time when communicating between tiles).

The objective of the resource allocation strategy is to perform resource allocation for as many graphs as possible while keeping the total amount of resources used as low as possible. Table 6.5 shows the resource usage after resource allocation for the graphs from the 4th set. For comparison, the resource usage of each resource is normalized with respect to the largest usage of this resource when using any of the 6 tile-cost functions. The results show that the 3rd tile-cost function achieves a good result by allocating the largest number of resource-aware graphs (see Table 6.4) to the smallest amount of resources. It shows also that the 6th tile-cost function effectively uses the available resources. These results confirm that communication has a major impact on resource usage (or reservations).

Table 6.5: Resource efficiency for set 4.

cost function	timewheel	memory	connections	input bw	output bw
1:	0.46	0.78	0.82	0.81	1.00
2:	0.93	0.96	1.00	1.00	1.00
3:	0.91	0.88	0.67	0.67	0.63
4:	0.08	0.50	0.30	0.30	0.29
5:	0.97	0.89	0.94	0.94	0.93
6:	1.00	1.00	0.96	0.96	0.94

This is as expected because no synchronization is assumed between time wheels, meaning that communication has a large impact on the guaranteed throughput that can be obtained under worst-case synchronization conditions. Communication needs to be balanced by allocation of large time slices on the communicating processors. The table also shows that for all tile-cost functions, except the 4th, the resource occupancy of the various resources is similar, indicating that all these tile-cost functions give a balanced resource utilization. It also shows that focusing only on minimization of the interconnect latency results in a very poor resource utilization.

When doing resource allocation using the 6th tile-cost function on the graphs from the 4th set of the benchmark, on average 67% of the resources in the platform graphs are used. This result is reasonable because it is achieved without any optimizations. Resource utilization can be increased when doing system dimensioning, re-ordering applications before allocation and/or applying mechanisms to transform applications or to continue allocating applications after one application fails to be bound. If, for example, the mapped applications have a mugh higher processing/memory requirement ratio then the ratio present in the platform, a high resource utilization is not possible. Note that the resource utilization can, for example be increased by redimensioning the platform.

6.8.3 Experiments on a multimedia system

Besides the synthetic graphs, a multimedia system consisting of three H.263 decoders (each 4 actors) and an MP3 decoder (13 actors) is used to evaluate the resource allocation strategy. The four resource-aware graphs are bound and scheduled on a platform with 2 generic processors and 3 accelerators. The used tile-cost function $(2, 0, 1, 0)$ focuses on balancing the processing load and it tries to limit the communication. The memory usage is ignored as the total amount of memory needed in every potential bindings is similar and sufficient memory resources are available. The strategy finds a resource allocation with a balanced resource utilization. The run-time of the strategy is 8 seconds of which approximately 87% is spent on the time slice allocation. The time slice allocation step performs 85 times a throughput computation in order to minimize the slices used by the appli-

cations. Resource allocation techniques that convert the SDFG to an HSDFG and compute throughput on the HSDFG would take several hours when performing a similar amount of throughput checks. (Recall that one throughput computation for the H.263 decoder takes in that case 21 minutes.) This experiment shows that the resource allocation strategy can handle SDFGs whose corresponding HSDFGs are large (14275 actors) within a limited run-time. It also shows that through a combination of modeling resource allocation decisions in the SDFG (as proposed e.g. in [13, 24]) and by constraining the execution of the graph it becomes feasible to analyze the throughput of realistic applications when bound to a heterogeneous MP-SoC.

6.9 Summary

This chapter presents the first resource allocation strategy that can bind multiple SDFGs to a heterogeneous multi-processor system while giving throughput guarantees to each individual application, also in a context of resource sharing. The technique can deal with multi-rate and cyclic dependencies between actors without converting it to a homogeneous SDFG. The strategy uses generic cost-functions to steer the binding of the application to the architecture and incorporates an efficient technique to compute the throughput of a bound and scheduled SDFG. The experiments show that this enables a balanced resource allocation of time-constrained applications bound to a multi-processor system-on-chip, which makes the resource allocation technique a versatile tool in any SDFG mapping flow aiming to provide throughput guarantees.

The technique presented in this chapter assumes that buffering requirements between actors are given, and that the interconnect is abstracted into point-to-point connections with given latency guarantees. Buffer sizing under throughput constraints and a NoC scheduling technique that provides latency guarantees are considered in the next two chapters.

Chapter 7

Throughput-Buffering Trade-Off Exploration

7.1 Overview

The actors in an SDFG communicate tokens with each other. Storage space, buffers, must be allocated for these tokens. At design time, the allocation (size) of this storage space must be determined. The predictable design flow, introduced in Section 1.4, computes the storage space allocations for the edges of an SDFG in its first phase, the memory dimensioning phase. The objective is to minimize the storage space allocated for the graph as the available storage space in an embedded system is usually very limited. Minimizing storage has the additional advantage that it saves energy.

Consider the timed SDFG shown in Figure 7.1. This figure shows the same graph as earlier introduced in Chapter 4. Storage space must be allocated for the dependency edges d_1 and d_2. The self-loops model absence of auto-concurrency and will not require storage space in a real implementation and can thus be ignored. The amount of storage space that is allocated to the edges influences the throughput that the graph can achieve. Figure 7.2 shows the complete trade-off space between allocated storage space and the achieved throughput for the SDFG. The points in Figure 7.2 represent the smallest distributions of storage space over the dependency edges that achieve a certain throughput. These points provide optimal trade-offs (Pareto points) between the throughput and buffer size of the SDFG.

Traditionally, research has been done on finding the smallest amount of storage space needed to execute an SDFG on a single processor. The left-most point of the trade-off space gives the minimal storage requirements for this type of execution. Also techniques have been studied to find the smallest amount of storage space needed to execute an SDFG while it realizes its maximal throughput. With this

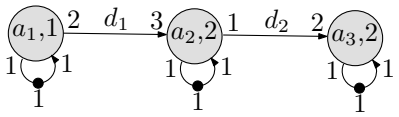


Figure 7.1: Example SDFG.

storage space allocation, the SDFG realizes the same throughput as if the graph is executed without any bounds on its storage space. The right-most point of the trade-off space gives the minimal storage space requirements to execute the SDFG with its maximal throughput.

For multimedia applications, it is interesting to find the complete trade-off space between the buffer requirements and the throughput of an SDFG. This allows a designer or run-time QoS manager to trade-off resource requirements (storage space) with quality (throughput).

This chapter presents a technique to find all trade-off points between the throughput and buffer size of an SDFG. The experimental evaluation shows that the technique completes on real applications within milliseconds or seconds, except for one case. For an H.263 decoder with 3255 throughput-buffering Pareto points, the technique takes 53 minutes. To improve scalability, an approximation technique is presented that can be used to explore the design space while trading off run-time of the algorithm with quality of the end result, in terms of buffer size overestimation. An analytical bound on the overestimation of the heuristic is also provided. The results show that the approximation heuristic scales well. When applied to the H.263 decoder, it approximates the throughput-buffering trade-off space within a few milliseconds. The minimal buffer size needed for maximal throughput is then approximated with less than 0.2% overestimation.

In line with earlier work on buffer sizing for SDFGs, the technique considers only constraints which come from the dependencies between the actor firings. Other constraints, such as resource sharing between actors, are not taken into account. In Section 7.8, it is explained how the technique can be extended to take resource constraints into account.

7.2 Related Work

NoC-based MP-SoCs use buffers to decouple communication and computation elements in the architecture (see Section 3.2). Minimization of the buffer requirements is important as buffers have a large influence on the area and power consumption of a NoC [28]. Techniques to minimize buffer sizes in a NoC have been studied before. In [28], a buffer sizing algorithm is presented that is based on network calculus. An analytical method for sizing buffers is presented in [40]. This method assumes that an application has a periodic behavior with some jitter. An approach to minimize buffers using queuing theory is presented in [66]. In

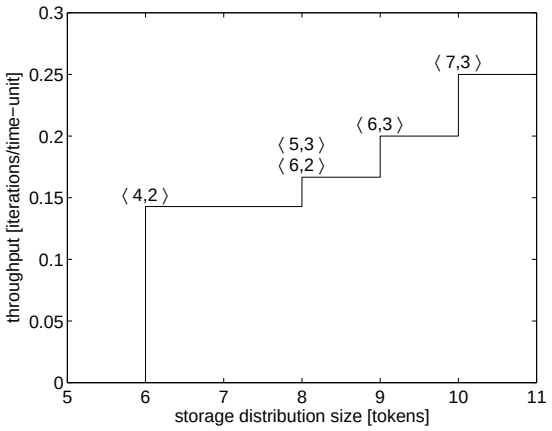


Figure 7.2: Pareto space for SDFG shown in Figure 7.1.

this thesis it is assumed that applications are modeled with an SDFG. The SDF MoC is different from the before mentioned MoCs. Therefore, the solution to the buffer minimization problem is also different. This chapter studies the buffer sizing problem for the SDF MoC.

Minimization of buffer requirements in SDFGs has been studied before, see for example [4, 23, 21, 44, 54, 67, 101, 104, 109]. The proposed solutions target mainly single-processor systems. Modern media applications, however, often target multi-processor systems. Furthermore, they have timing constraints expressed as latency or throughput constraints. Only looking for the minimal buffer size which gives a deadlock-free schedule as done in [4, 23, 21, 44, 101, 109] may result in an implementation that cannot be executed within these timing constraints. It is necessary to take the timing constraints into account while minimizing the buffers. Several approaches have been proposed for minimizing buffer requirements under a throughput constraint. In [54], a technique based on linear programming is proposed to calculate a schedule that realizes the maximal throughput while it tries to minimize buffer sizes. Hwang et al. propose a heuristic that can take resource constraints into account [67]. This method is targeted towards a-cyclic graphs and it always maximizes throughput rather than using a throughput constraint. Thus, it could lead to additional resource requirements. In [104], buffer minimization for maximal throughput of a subclass of SDFGs (homogeneous SDFGs) is studied. The proposed algorithm is based on integer linear programming. In general, the buffer sizes obtained with this approach cannot be translated to exact minimal buffer sizes for arbitrary SDFGs. A technique to schedule an SDFG under a given throughput constraint is presented in [150, 151]. The scheduling algorithm tries to minimize the required storage space. The technique does not always find the minimal storage space for the given throughput

constraint. Also no results are presented on the over-dimensioning of the storage requirements for realistic applications. This chapter presents, in contrast to existing work, an exact technique to determine all trade-offs (Pareto points) between the throughput and buffer size for an SDFG, as well as an approximation technique to approximate this space, while providing guarantees on throughput and worst-case buffer size overestimation. An interesting observation is that both the exact and the approximation technique can also be applied after the search space has been pruned by a heuristic, which may in general lead to reduced buffering requirements for the given throughput. For the mentioned heuristic of [151], one of the experiments shows that the technique presented in this chapter can compute the exact result within four second when starting from the result of the heuristic. This makes the work presented in this chapter complementary to fast heuristics.

In [23], it is shown that the buffer minimization problem of SDFGs is NP-complete. Both the exact and the approximation technique are based on state-space exploration. Explicit state-space exploration techniques are frequently applied successfully to solve NP-complete (and sometimes worse) scheduling problems [5, 7, 127]. For buffer minimization, [44] proposed a state-space exploration technique to find minimal buffer requirements to execute an SDFG with a deadlock-free schedule. The results presented in [44] indicate that it might be feasible to apply a state-space exploration based technique to the problem of storage-throughput trade-off analysis. However, the technique presented in this chapter is in general not exhaustive. It prunes the search space in an efficient way, as confirmed by the experiments, without loosing any Pareto points.

7.3 Storage Requirements

Dependency edges in an SDFG are assumed to have unbounded storage space. However, in practice storage space must be bounded. Bounded storage space for edges can be realized in different ways. One option is to use a memory that is shared between all edges. The required storage space for the execution of an SDFG is then determined by the maximum number of tokens stored at the same time during the execution of the graph. Murthy et al. use this assumption to schedule SDFGs with minimal storage space [101]. This is a logical choice for single-processor systems in which actors can always share the memory space. A second option is to use a separate memory for each edge, so empty space in one cannot be used for another. This assumption is logical in the context of multi-processor systems, as memories are not always shared between all processors. The edge capacity must be determined per edge over the entire schedule, and the total amount of memory required is obtained by adding the edge capacities up. Minimization of the memory space with this variant is considered in [4, 21, 44]. Hybrid forms of both options can be used [44]. In this thesis, it is assumed that edges cannot share memory space. This gives a conservative bound on the required memory space when the SDFG is implemented using shared memory. In that

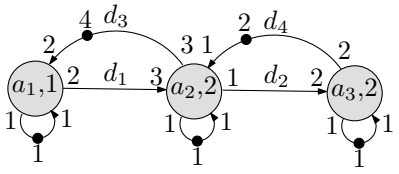


Figure 7.3: SDFG with storage distribution $\langle 4, 2 \rangle$.

case, the SDFG may require less memory, but it will never require more memory than determined by the method presented in this chapter.

The maximum number of tokens which can be stored in an edge (**edge capacity**) is given by a storage distribution.

Definition 22. (Storage distribution) *A storage distribution of an SDFG (A, D) is an edge quantity δ, as defined in Definition 5, that associates with every $d \in D$, the capacity of the edge.*

The storage space required for a storage distribution is called the distribution size. Note that if tokens on different edges represent different amounts of data, this can easily be accounted for in the definition of distribution size. In the remainder, it is assumed that all tokens are of equal size.

Definition 23. (Distribution size) *The size of a storage distribution δ of an SDFG (A, D) is given by: $|\delta| = \sum_{d \in D} \delta(d)$.*

A possible storage distribution for the SDFG shown in Figure 7.1 would be $\delta(d_1) = 4$ and $\delta(d_2) = 2$, denoted as $\langle d_1, d_2 \rangle \mapsto \langle 4, 2 \rangle$. It has a distribution size of 6 tokens.

In an SDFG state, an edge (p, q) from actor a_1 to actor a_2 does not contain an arbitrary number of tokens. Assume that the edge contains in the initial state of the execution n tokens. The number of tokens in the edge after x firings of a_1 and y firings of a_2 is given by the following equation:

$$n + x \cdot Rate(p) - y \cdot Rate(q)$$

This can be re-written to:

$$\left\lfloor \frac{n + x \cdot Rate(p) - y \cdot Rate(q)}{k} \right\rfloor \cdot k + n \bmod k,$$

with $k = \gcd(Rate(p), Rate(q))$. The equation shows that the number of tokens in an edge, and hence the storage space which can be used usefully, depends on the gcd of the rate at which the actors a_1 and a_2 produce and consume tokens.

This gcd is called the **step size** of the edge.

The bound on the storage space of each edge can be modeled in an SDFG (A, D) by adding for edge $(p, q) \in D$ from an actor $a_1 \in A$ to an actor $a_2 \in A$ an edge (q_δ, p_δ) from a_2 to a_1 with $Rate(p) = Rate(p_\delta)$ and $Rate(q) = Rate(q_\delta)$. This modeling construct was already used in Section 6.6.1. The number of initial tokens on the edge (q_δ, p_δ) determines the storage space of the edge (p, q). Subscript 'δ' denotes elements used to model storage space. The SDFG which models the storage distribution δ in an SDFG (A, D) is denoted (A_δ, D_δ). Figure 7.3 shows the timed SDFG which encodes the storage distribution $\langle 4, 2 \rangle$ for the timed SDFG shown in Figure 7.1. Note that no storage space is allocated for the self-loops on the actors. These self-loops are introduced to model absence of auto-concurrency and will not require storage space in a real implementation and can thus be ignored. In fact, the technique presented in this chapter allows in general to specify which edges should be considered buffers, and which edges model other dependencies. The self-loop dependencies added to limit auto-concurrency are just one example of this flexibility.

At the start of a firing, an actor consumes its input tokens. This includes the tokens it consumes from the edges which model the storage space of edges to which the actor will write. The consumption of these tokens can be seen as allocation of storage space for writing the results of the computation. At the end of the firing, the actor produces its output tokens. This includes the production of tokens on edges which model the storage space of edges from which the actor has read tokens at the beginning of the firing. The production of these tokens can be seen as the release of the space of the input tokens. In other words, the model assumes that space to produce output tokens is available when an actor starts firing and the space used for input tokens is released at the end of the firing. The chosen abstraction is conservative with respect to storage and throughput if in a real implementation space is claimed later, or released earlier or data tokens are written earlier.

7.4 Storage Dependencies

The maximal throughput of an SDFG is limited by the availability of tokens on edges. In the self-timed execution of the SDFG an actor may, for example, be waiting for tokens on an edge d_δ (modeling the storage space of edge d). Adding tokens to d_δ (i.e. increasing the storage space of d) might enable the actor to fire earlier and possibly increase the maximal throughput of the SDFG. The dependency of an actor firing on tokens produced by the end of another firing is called a causal dependency.

Definition 24. (Causal Dependency) *A firing of an actor a_i causally depends on the firing of an actor a_j via an edge d if and only if the firing of a_i consumes a*

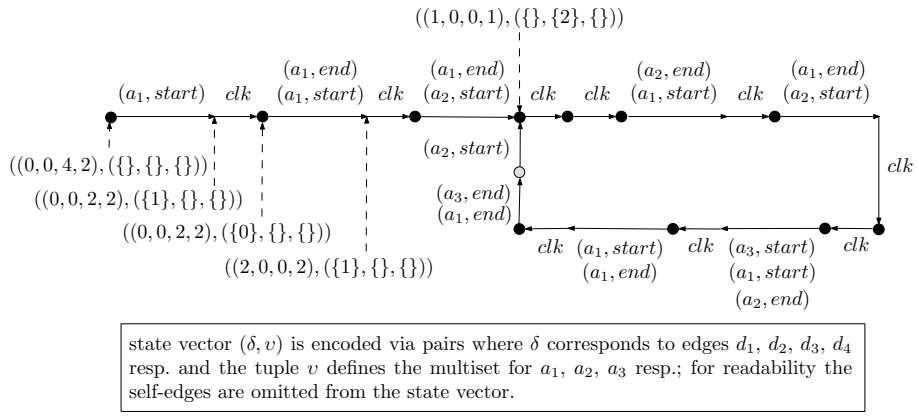


Figure 7.4: State-space of the example SDFG.

token from d produced by the firing of a_j on d without a clock transition between the start of the firing of a_i and the end of the firing of a_j.

Recall from Section 4.3 that the self-timed execution of an SDFG is periodic (after some initial phase). If a causal dependency appears in the periodic phase of the execution, the actor will repeatedly (infinitely often) not be able to fire earlier which on its turn may influence the throughput. Throughput may increase if these dependencies are resolved. All causal dependencies between the actor firings of the periodic phase can be captured in a causal dependency graph. It is sufficient if only the dependencies between actor firings in one period of the periodic phase are considered as the dependencies are equal in all periods.

Definition 25. (Dependency Graph) *Given a timed SDFG $(A_\delta, D_\delta, \Upsilon)$ incorporating a storage distribution δ and a sequence of states and transitions p corresponding to one period of the self-timed execution of $(A_\delta, D_\delta, \Upsilon)$ (starting at some arbitrary state in the period). The causal dependency graph (N, E) contains a node $n_{i,k}$ for the k-th firing in p of actor $a_i \in A_\delta$. The set of dependency edges E contains an edge if and only if there exists a causal dependency between the corresponding firings.*

The state-space of the self-timed execution of the running example is shown in Figure 7.4. The corresponding dependency graph is shown in Figure 7.5, assuming the gray state as the start state.

The throughput of an SDFG is limited by an infinite sequence of causal dependencies between the actor firings, captured by a causal dependency cycle in the dependency graph.

Definition 26. (Causal Dependency Cycle) *A causal dependency cycle is a simple cycle in the causal dependency graph.*

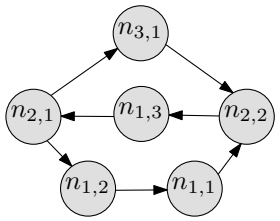


Figure 7.5: Dependency graph of the example SDFG.

A causal dependency cycle is a sequence of actor firings that causally depend on each other, starting and ending with the same actor firing. Causal dependencies caused by edges which model storage space are of interest when looking at the trade-off between storage space and throughput. Adding more tokens to these edges (i.e. increasing the storage space of the corresponding edge) may resolve causal dependency cycles and increase throughput.

Definition 27. (Storage Dependency) *Given a timed SDFG* $(A_\delta, D_\delta, \Upsilon)$ *incorporating a storage distribution* δ *and its dependency graph* Δ*. An edge* $d \in D_\delta$ *has a storage dependency in* Δ *if and only if there exists a causal dependency via edge* d_δ *in some dependency cycle of* Δ*.*

Consider again the dependency graph shown in Figure 7.5. Every edge in the graph belongs to a causal dependency cycle. A firing of actor a_1 causally depends on a firing of a_2. This causal dependency goes via the dependency edge d_3 in the SDFG (see Figure 7.3). This dependency edge models the storage space of dependency edge d_1. So, edge d_1 has a storage dependency in the dependency graph shown in Figure 7.5. Also dependency edge d_2 has a storage dependency as a firing of actor a_2 causally depends on a firing of a_3.

Storage dependencies can be used to determine which storage capacities can be enlarged to increase the throughput of the graph. However, two issues remain to be solved. First, if the graph deadlocks (because of lack of storage space or an inherent deadlock due to the dependencies), the dependency graph is empty and provides no information about which edge capacities to enlarge. Second, having a node for every firing of every actor, the dependency graph may become prohibitively large (a multiple of the sum of entries in the repetition vector). The latter issue is solved first and subsequently the deadlocking case is considered.

Cycle detection in the dependency graph can become very time consuming. To solve this, an abstract version of the dependency graph can be constructed in which the number of nodes is equal to the number of actors in the SDFG. Therefore, the abstract dependency graph allows faster cycle detection as compared to the dependency graph.

Definition 28. (Abstract dependency graph) *Given a timed SDFG* $(A_\delta, D_\delta, \Upsilon)$ *incorporating a storage distribution* δ *and its dependency graph* (N, E)*. The*

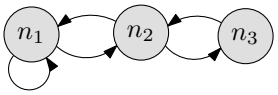


Figure 7.6: Abstract dependency graph of the example SDFG.

abstract dependency graph (N_a, E_a) *contains an abstract dependency node* $n_i \in N_a$ *for each actor* $a_i \in A_\delta$*. For each dependency edge* $(a_{i,k}, a_{j,l}) \in E$*, there is an edge* (n_i, n_j) *in* E_a*.*

The abstract dependency graph of the running example is shown in Figure 7.6. It contains fewer nodes than the dependency graph (see Figure 7.5). Compared to the SDFG of the example (see Figure 7.3), the abstract dependency graph contains an equal number of nodes but fewer edges. In practice, the abstract dependency graph can be constructed by traversing through the cycle in the state-space of the self-timed execution once. Initially, an abstract dependency graph must be constructed which contains a node for every actor in the SDFG and no edges. When during the traversal of the cycle in the state-space a causal dependency is found, a corresponding edge is added to the abstract dependency graph. An important property of the abstract dependency graph is that it includes at least all storage dependencies present in the full dependency graph. (The definition of a storage dependency carries over to the abstract dependency graph.)

Theorem 3. (Preservation of storage dependencies) *The set of storage dependencies of an abstract dependency graph contains all storage dependencies of the corresponding dependency graph.*

Proof. Given a dependency graph $\Delta = (N, E)$ and its corresponding abstract dependency graph $\Delta_a = (N_a, E_a)$. Any dependency edge $(a_{i,k}, a_{j,l}) \in E$ from a node which corresponds to the k-th firing of actor a_i to the l-th firing of actor a_j maps in Δ_a to an edge from the abstract dependency node of actor a_i to the abstract dependency node of actor a_j. Any cycle of dependencies in Δ is a sequence of dependency edges which in the abstract dependency graph (using the mapping of dependency nodes in Δ to nodes in Δ_a) is also a sequence starting and ending in one node, i.e. it is also a cycle in Δ_a. Hence, the edge is also on some cycle in Δ_a and thus also on some simple cycle. □

In the case that the SDFG deadlocks, the regular dependency graph is empty by definition. To find out which edge capacities need to be increased, if any, the following alternative definitions are used in that case.

Definition 29. (Causal Dependency in Deadlock) *In a deadlocked state, an actor* a_i *causally depends on an actor* a_j *via an edge* d *from* a_j *to* a_i *if and only if a firing of* a_i *is prohibited by lack of tokens on edge* d*.*

Based on this definition, a causal dependency graph for the deadlock case can be defined. Via Definition 26 and Definition 27, this defines the storage dependencies for the deadlock case.

Definition 30. (Causal Dependency Graph in Deadlock) *Given a timed SDFG* $(A_\delta, D_\delta, \Upsilon)$ *incorporating a storage distribution* δ*, with throughput zero (i.e., self-timed execution (eventually) deadlocks). The causal dependency graph* (D, E) *contains a node* a *for every actor* $a \in A_\delta$*. The set of dependency edges* E *contains an edge if and only if there exists a causal dependency between the corresponding actors in the deadlock state.*

In the remainder, (causal) dependency graph refers to either the abstract causal dependency graph in case of SDFGs with positive throughput and the causal dependency graph in deadlock for the case when the graph deadlocks.

7.5 Design-Space Exploration

Section 7.3 explained how a storage distribution δ can be modeled into an SDFG G. The throughput of G under the storage distribution δ can be computed with the technique presented in Section 4.4. In this way, the throughput for a given storage distribution can be found. Using this approach, it is possible to find the trade-offs between the distribution size and the throughput, i.e., the Pareto space. Figure 7.2 shows this Pareto space for the example SDFG. It shows that storage distribution $\langle 4, 2\rangle$ is the smallest distribution with a throughput for actor a_3 larger than zero. The throughput can never go above 0.25. This can be seen as follows. The repetition vector for the graph is $(3, 2, 1)$. Thus, actor a_2 always has to fire twice per iteration of the SDFG, which requires four time steps. Therefore, the throughput of a_2 is limited to 0.5, and the normalized graph throughput to 0.25. It is not difficult to check that other actors are constraining the throughput less than a_2. With a distribution size of 10 tokens (or more), the maximal throughput can be achieved.

Definition 31. (Minimal Storage Distribution) *A storage distribution* δ *with throughput* Th *is minimal if and only if for any other storage distribution* δ' *with throughput* Th'*,* $|\delta'| < |\delta|$ *implies* $Th' < Th$ *and* $|\delta'| = |\delta|$ *implies* $Th' \leq Th$*.*

Distributions $\langle 0, 0\rangle$, $\langle 4, 2\rangle$, $\langle 5, 3\rangle$, $\langle 6, 2\rangle$, $\langle 6, 3\rangle$ and $\langle 7, 3\rangle$ in the example are minimal, but distribution $\langle 5, 2\rangle$ is not.

Algorithm 1 is used to find all minimal storage distributions for an SDFG G with maximal throughput Th_{max}. It uses a set U which contains all storage distributions which it may explore. Initially, the set U contains only the storage distribution $\langle 0, \ldots, 0\rangle$. The algorithm explores all storage distributions in U with a size sz before it explores a storage distribution with size $sz + 1$. When a storage distribution δ with size sz exists in the set U, it is removed from U. Next, the algorithm computes the storage dependency graph Δ and throughput

Algorithm 1 Find all minimal storage distributions

Input: A timed SDFG G with maximal throughput Th_{max}
Result: A set P of pairs (storage distribution, throughput) containing precisely all minimal storage distributions

```
1:  procedure findMinStorageDist(G, Th_max)
2:      Let U be the set of unexplored storage distributions
3:      U ← [⟨0, ..., 0⟩]
4:      P ← ∅
5:      sz ← 0
6:      while no (δ, Th) ∈ P with Th = Th_max do
7:          while a storage distribution δ with size sz in U do
8:              U ← U\δ
9:              Create SDFG G_δ which models δ in G
10:             Compute throughput Th and dependency graph Δ of G_δ
11:             P ← P ∪ {(δ, Th)}
12:             Let S be the set of storage dependencies in Δ
13:             for each edge d in S do
14:                 δ_n ← δ
15:                 δ_n(d) ← δ(d) + step(d)
16:                 U ← U ∪ {δ_n}
17:             end for
18:         end while
19:         sz ← sz + 1
20:     end while
21:     Remove non-minimal storage distributions from P
22: end procedure
```

Th for this distribution. The distribution-throughput pairs are kept in a set P. The algorithm continues by constructing a new storage distribution δ_n for each edge d which has a storage dependency in Δ. In storage distributions δ_n, the storage space of d is increased by the step size of the edge, as explained in Section 7.3. All other dependency edges have the same storage space in δ_n as in δ. The storage distribution δ_n is then added to the set U. Because the algorithm explores storage distributions with increasing size and because the step size is always a positive number, it is guaranteed that all storage distributions with size sz which will be explored are added to U before the algorithm starts exploring distributions of this size. The outermost while loop of the algorithm terminates when some storage distribution realizing the maximal throughput has been found. Due to the loop nesting, no other storage distributions of equal size remain to be explored. Finally, all non-minimal storage distributions are removed from P. Observe that this can be done via a single traversal through all the explored storage distributions stored in P by keeping this set sorted according to throughput. In an efficient implementation, also set U is kept sorted according to distribution size.

A few lemmas are needed to prove that Algorithm 1 finds all minimal distributions for an SDFG.

Lemma 1. *Given a storage distribution δ_i with throughput Th_i. For any storage distribution $\delta_j \preceq \delta_i$ with throughput $Th_j < Th_i$ and dependency graph Δ, there is a dependency edge d with a storage dependency in Δ such that $\delta_i(d) > \delta_j(d)$.*

Proof. If δ_j is such that the graph deadlocks, the result is straightforward to prove using the dependency graph for the deadlock case. The case that Th_j is positive is discussed in more detail below.

In a self-timed execution, each actor firing has a causal dependency with at least one earlier actor firing (unless it is one of the first firings consuming the initial tokens). This gives chains of causal dependencies between all actor firings that occur during the execution. These chains of causal dependencies start with the initial firings of the graph, and either end at some point, or they are of infinite length. In a self-timed execution however, there must be at least one such chain of infinite length. (Otherwise, the graph would have delayed some firing unnecessarily.) Such infinite chains determine the throughput.

There is a finite number of states in the periodic phase and states in this phase are revisited each period. Hence, also the same causal dependencies are encountered again and again. In other words, each infinite chain of causal dependencies implies a cycle in the dependency graph and conversely, every cycle in the dependency graph corresponds to an infinite chain of causal dependencies in the execution.

To increase the throughput, each of the causal dependency cycles must be broken. Since δ_i has a higher throughput than δ_j, it is known that every causal dependency cycle of δ_j includes a storage dependency in Δ_j (because otherwise Th_j would be maximal, contradicting $Th_j < Th_i$). Since decreasing the capacity

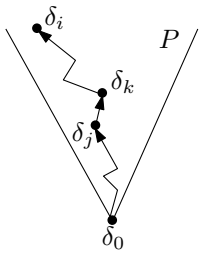


Figure 7.7: Distributions reached by Algorithm 1.

of edges can never increase throughput, to achieve throughput $Th_i > Th_j$, at least one storage dependency of each causal dependency cycle in Δ_j has been resolved by increasing the capacity of the corresponding edge. Hence, there must be some edge d with a storage dependency in Δ_j such that $\delta_i(d) > \delta_j(d)$. □

Lemma 2. *Given a storage distribution δ_i with throughput Th_i and a storage distribution $\delta_j \in P$ such that $\delta_j \preceq \delta_i$ with throughput $Th_j < Th_i$. Then Algorithm 1, from distribution δ_j, explores a storage distribution δ_k for which $\delta_j \preceq \delta_k \preceq \delta_i$, $|\delta_j| < |\delta_k|$ and $Th_j \leq Th_k \leq Th_i$.*

Proof. Let S_j be the set of storage dependencies of δ_j. From Lemma 1, it follows that there exists a dependency edge $d \in S_j$ for which $\delta_j(d) < \delta_i(d)$. So, d can be enlarged with at least one step before the storage space becomes equal to the storage space assigned to it in δ_i. Because $d \in S_j$, Algorithm 1 does increase the storage space of d, which results in a new storage distribution δ_k. As (only) the storage space of d is increased, but not beyond its capacity in δ_i, it must hold that $\delta_j \preceq \delta_k \preceq \delta_i$ and $|\delta_j| < |\delta_k|$. From $\delta_j \preceq \delta_k \preceq \delta_i$, it also follows that $Th_j \leq Th_k \leq Th_i$. □

Theorem 4. (Correctness of Algorithm 1) *The set of all storage distributions contained in P which is constructed using Algorithm 1 contains precisely all minimal storage distributions.*

The proof of Theorem 4, the correctness of Algorithm 1, is illustrated by Figure 7.7.

Proof. For throughput 0, $\delta_0 = \langle 0, \ldots, 0 \rangle$ is the (only) minimal storage distribution. Therefore, let δ_i be some minimal storage distribution with positive throughput Th_i. It must be shown that $\delta_i \in P$, i.e., that the algorithm will explore distribution δ_i. Initially, Algorithm 1 starts from the storage distribution $\delta_0 = \langle 0, \ldots, 0 \rangle$ which satisfies the conditions of Lemma 2. Repeatedly using Lemma 2, it can be shown that the algorithm explores a series δ_k of storage distributions with $\delta_0 \preceq \delta_1 \preceq \delta_2 \preceq \ldots \preceq \delta_m \preceq \delta_i$ and for each k, $|\delta_k| < |\delta_{k+1}|$. From

$\delta_k \preceq \delta_i$, it follows that $|\delta_k| \leq |\delta_i|$ and hence, after a finite number of m steps it must be that Lemma 2 no longer applies. Thus, it follows that $Th_m = Th_i$ and, because δ_i is minimal, that $|\delta_m| = |\delta_i|$. For a distribution $\delta_m \preceq \delta_i$ such that $|\delta_m| = |\delta_i|$, it must be that $\delta_m = \delta_i$, which shows that δ_i is explored by the algorithm.

So far, it is shown that set P contains all minimal storage distributions. However, it may also contain non-minimal distributions. The last line of the algorithm removes non-minimal storage distributions, completing the proof. □

From the literature on SDFGs, lower bounds on the storage space required for each edge to avoid deadlock (i.e., throughput equal to zero) are known [4, 101]. These bounds can be used to speed up the initial phase of Algorithm 1. Distribution $\langle 0, \ldots, 0 \rangle$ with all zero entries is by definition the only minimal storage distribution realizing zero throughput. Thus, to find all Pareto points with non-zero throughput, it is sufficient to start from the mentioned lower bounds.

An important and relevant question that remains is whether Algorithm 1 terminates. It can be shown that when at least one actor has a bounded throughput, then Algorithm 1 terminates. If all actors can increase their firing rate indefinitely, then there are infinitely many minimal storage distributions, and the algorithm cannot terminate.

Theorem 5. (Termination) *For any connected SDFG G that contains an actor with bounded throughput, Algorithm 1 terminates.*

Proof. Given a connected SDFG G, Th_{max} is equal to the minimum of the throughput of all strongly connected components in G. As there is at least one actor with bounded throughput, it must hold that $Th_{max} < \infty$. The throughput Th_{max} is achievable within finite memory. This implies that there exists some storage distribution δ_{max} of size $N < \infty$ which reaches throughput Th_{max}.

Algorithm 1 explores the storage distributions with increasing size. There are only a finite number of storage distributions of some size n and only a finite number of different sizes are explored before the algorithm explores the storage distributions with size N. This implies that only a finite number of different storage distributions exist that have a size at most $|\delta_{max}|$. Within a finite number of steps, all distributions with size up to N are explored and a storage distribution with throughput Th_{max} is found, causing the algorithm to terminate. □

7.6 Experimental Evaluation

Experiments have been performed to see how the approach performs in practice on a number of real DSP and multimedia applications modeled as SDFGs. From the DSP domain, the set contains a modem [21], a satellite receiver [121] and a sample-rate converter [21]. From the multimedia domain, an H.263 decoder (see

Figure 1.5), an H.263 encoder [110], an MP3 decoder (see Figure 2.3(b)) and an MP3 playback application [151]. The MP3 decoder graph models the parallelism inside an MP3 decoder, while the MP3 playback application models the MP3 decoder as a single actor and adds to this actors which model a sample-rate converter, an audio post processing filter and a digital-to-analog converter. The test bench includes also the example SDFG shown in Figure 7.1 and the often used bipartite SDFG from [21]. For each of the SDFGs, the complete design-space was explored. This resulted in a Pareto space showing the trade-offs between the throughput and distribution size for each graph.

The results of the experiments are summarized in Table 7.1. It shows the number of actors in each graph and the number of edges for which the buffers are being sized, the minimal distribution size for the smallest positive throughput, the maximum throughput that can be achieved and the distribution size needed to realize this throughput. They also show the number of Pareto points and the number of minimal storage distributions that were found during the design-space exploration. The results show that one Pareto point of the example graph contains two different minimal storage distributions. In all other situations, each Pareto point contains a single storage distribution.

An upper bound on the storage space required for each dependency edge to achieve maximal throughput with finite edge capacities can be found [86]. This upper bound and the lower bound found with the technique from [4], can be used to compute the number of different storage distributions in the design-space (see row '#Distr. in space' of Table 7.1). The next row shows the number of storage distributions explored by the algorithm (excluding the zero-size minimal distribution, which is not actively explored, as explained in the previous section). The results show that the algorithm explores only very few distributions from the space. This shows that the algorithm successfully prunes the design-space.

The algorithm computes the throughput for each storage distribution it tries. This is done via a self-timed execution of the graph. The row 'Max. #states visited' shows the maximal number of different states that is visited during a throughput computation. Only a selected number of states must be stored (see Section 4.4) to compute the throughput of the graph. The number of states that is stored is shown in the row 'Max. #states stored'.

The experiments show that it is feasible to perform a design-space exploration for reasonable application kernels. As all SDFGs, except the H.263 decoder, show a run time in the order of seconds or even milliseconds to explore the complete design space when the exploration is performed on a P4 at 3.4GHz. The run time for the H.263 decoder is large due to the large number of Pareto points contained in the space.

It is interesting to consider the MP3 playback model of [151] in a bit more detail. In [151], a heuristic is presented that computes a storage-space distribution under a throughput constraint. The objective is to minimize the size of the storage distribution. The reported results state that the heuristic can compute within the order of 10^{-2}s a storage distribution which is 10% larger than the

Table 7.1: Experimental results.

	Example	Bipartite	Sample Rate	Modem	Satellite
#actors / #sized edges	3/2	4/4	6/5	16/19	22/26
Min. pos. throughput (s^{-1})	1/7	$4{\cdot}10^4$	$15{\cdot}10^4$	$3{\cdot}10^4$	$18{\cdot}10^4$
Distr. size	6	28	32	38	1542
Max. throughput (s^{-1})	1/4	$6{\cdot}10^4$	$17{\cdot}10^4$	$6{\cdot}10^4$	$23{\cdot}10^4$
Distr. size	10	35	34	40	1544
#Pareto points	4	9	4	4	3
#Min. distr.	5	9	4	4	3
#Distr. in space	27	$1{\cdot}10^8$	$9{\cdot}10^{12}$	$1{\cdot}10^{10}$	$2{\cdot}10^{65}$
#Distr. checked	7	51	3	4	4
Max. #states visited	21	652	$6{\cdot}10^6$	134	10377
Max. #states stored	2	20	5328	2	241
Exec. time	$1ms$	$1ms$	$1ms$	$2ms$	$7ms$

	MP3 decoder	MP3 playback	H.263 decoder	H.263 encoder
#actors / #sized edges	13/12	4/2	4/3	6/6
Min. pos. throughput (s^{-1})	$7{\cdot}10^3$	4.2	50	54
Distr. size	12	1977	4753	299
Max. throughput (s^{-1})	$8{\cdot}10^3$	8.3	100	244
Distr. size	16	2898	8006	397
#Pareto points	4	620	3255	20
#Min. distr.	4	620	3255	20
#Distr. in space	4096	$91{\cdot}10^6$	$3{\cdot}10^{10}$	$9.7{\cdot}10^5$
#Distr. checked	7	2152	$292{\cdot}10^3$	99
Max. #states visited	33579	19855	$8{\cdot}10^6$	605
Max. #states stored	212	2	1124	2
Exec. time	$2ms$	$21s$	$53min$	$9ms$

smallest storage distribution allowing maximal throughput. Using the algorithm presented in this chapter, the optimal solution can be found in 21s (see Table 7.1). Depending on the context in which buffer sizing is applied, this may or may not be acceptable. In general, however, the exponential worst-case complexity of Algorithm 1 could potentially lead to prohibitively large run-times. In those cases, the algorithm can be combined with any heuristic for buffer sizing. For example, the heuristic from [151] can be used to compute a storage distribution close to the optimum. A fraction of the storage space computed for each edge by the heuristic can then be used as a starting distribution in (line 3 of) the algorithm. In this way, the algorithm explores the trade-off space just below the storage distribution computed by the heuristic for a smaller distribution that still satisfies the throughput constraint. To test this approach, Algorithm 1 has been ran on the MP3 playback SDFG with the initial storage distribution equal to a fraction of the storage distribution requirements computed by the heuristic from [151] for this SDFG. When 90% of the storage distribution sizes computed by the heruristic are used as a starting point, the algorithm terminates immediately without increasing the size of one of the edges. This indicates that 90% of the storage space computed by the heuristic is already sufficient to achieve maximal throughput. Potentially, the storage space can even be reduced further. Setting the initial storage distribution equal to 80% of the storage distribution requirements computed by the heuristic, Algorithm 1 was able to find the optimal storage distribution within 3.4s. This illustrates how the algorithm can be combined with a heuristic, which will in general improve the results of the heuristic with little effort.

7.7 Approximation of Buffer Sizes

7.7.1 A Generic Approximation Technique

The experimental results of the previous section show that the search space of distributions is pruned efficiently by looking for storage dependencies. Nevertheless, the number of distributions that need to be explored may still be large, potentially leading to long run-times of the algorithm. An approximation of the exact result can be obtained by reducing the number of distributions that need to be explored, for instance by changing the step size for increasing the edge capacities. In Section 7.3, it is shown that for an edge d considering as sizes all multiples of $step(d)$ guarantees that all minimal distributions are found. In this section, exploration of only a set K_d of capacities for edge d is considered. For any capacity k, $\lceil k \rceil^{K_d}$ is used to denote the smallest capacity in K_d which is at least k. It is required that sets K_d be such that such the capacity always exists (i.e., that edge capacities can always be increased). Concrete examples of such sets, that are used for the experimental evaluation later in this section, are sets K_d^n, for any number $n \in \mathbb{N}$, defined as $\{k \in \mathbb{N} \mid k = n \cdot step(d)\}$, i.e., only multiples of the $step(d)$ are

considered for the given multiplication factor n.

Algorithm 1 is adapted as follows. Line 15 becomes:

$$\delta_n(d) \leftarrow \lceil \delta(d) + 1 \rceil^{K_d}.$$

That is, the next smallest capacity in the given set K_d is chosen. It can be proved that the adapted algorithm finds all minimal storage distributions δ among all distributions with edge capacities $\delta(d) \in K_d$ for all $d \in D$. This property is proved using Lemmas 1 and 2, while restricting attention to distributions within the limited set.

From the fact that the adapted algorithm finds all storage distributions that are minimal in the reduced set, the following bound on the discrepancy of the result from the optimal result can be derived.

Theorem 6. (Overestimation bound) *For every minimal storage distribution δ with throughput Th, found in the full search, there is a storage distribution δ' with throughput Th' that is minimal in the reduced search space such that $Th' \geq Th$ and $|\delta'| \leq \sum_{d \in D} \lceil \delta(d) \rceil^{K_d}$.*

Proof. An increase in edge capacity cannot decrease the throughput. If all edge capacities in δ are rounded up to $\lceil \delta(d) \rceil^{K_d}$, a distribution with throughput Th' is obtained that has at least throughput Th. If this distribution is minimal in the reduced search space, the theorem is proved. If it is not minimal, there exists a minimal one with the same throughput and smaller distribution size, or with a higher throughput and the same size. In both cases, this distribution satisfies the theorem. □

Note that the adapted algorithm still has an exponential worst-case complexity. However, it allows a trade-off between run-time and quality of the end result, by appropriately choosing the K_d. One can choose, for example, the sets K_d^n already mentioned above, possibly with different multiplication factors per edge. Theorem 6 then gives a bound on the worst-case loss in quality (buffer-size overestimation). Consider a single edge d. Theorem 6 and the definition of K_d^n imply that

$$\delta'(d) \leq \lceil \delta(d) \rceil^{K_d^n} \leq \delta(d) + (n-1)step(d).$$

It follows that the relative overestimation for d is bounded as follows:

$$\frac{\delta'(d) - \delta(d)}{\delta(d)} \leq \frac{(n-1)step(d)}{\delta(d)}, \tag{7.1}$$

which is the expected result that the overestimation per edge can be at most $n-1$ times the step size.

Figure 7.8 shows the worst-case overestimation for a single edge, for various edge capacities, step sizes, and multiplication factors. The edge capacities and step sizes are in line with those observed in the models of the benchmark. The

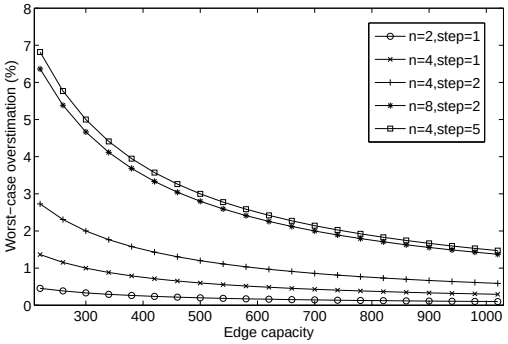


Figure 7.8: Worst-case overestimation for various edge capacities, step sizes, and multiplication factors.

relative worst-case overestimation is small for increasing edge size, which is when the approximation algorithm is most useful, because only for large distribution spaces the run-time of the exact technique may become problematic. Note that the relative worst-case overestimation does not change when considering multiple edges (assuming the same capacities, step sizes, and multiplication factors). The likelihood that the worst case will occur in fact decreases with increasing numbers of edges in the SDFG. Another way to limit the over-estimation is to search the trade-off space with several multiplication factors per edge, as illustrated below.

It is also possible to try to bound the relative overestimation via an appropriate choice of K_d, by choosing for K_d the set $\{\lceil(1+q)^n\rceil step(d) \mid n \in \mathbb{N}\}$, for some appropriately small q. Except for rounding effects, this choice for the K_d limits the overestimation to $q\%$.

7.7.2 Experimental Evaluation

The experimental results presented in Section 7.6 show that design-space exploration of the H.263 decoder and the MP3 playback application take the most time from all the tested models. For both applications, this is due to the large number of Pareto points in the trade-off space. However, the throughput of most of the Pareto points is close to each other. In practice, it is not interesting to find all these points. The approximation technique presented in the previous subsection can be used to reduce the number of different storage distributions that is explored.

Experiments have been performed with this approximation algorithm on the H.263 decoder and MP3 playback application. For both applications, a uniform multiplication factor has been used for the different edges. The step size in the H.263 decoder was multiplied with a factor of 3, 9, and 27. For the MP3 playback application, a multiplication factor of 3, 5, and 15 has been used. The results

Table 7.2: Results for the approximation technique.

	H.263 decoder			
	exact	$n = 3$	$n = 9$	$n = 27$
Min. pos. throughput (s^{-1})	50	50	50	50
Distr. size	4753	4753	4753	4753
Max. throughput (s^{-1})	100	100	100	100
Distr. size	8006	8006	8012	8021
#Pareto points	3255	1087	365	124
#Min. distr.	3255	1087	365	124
#Distr. checked	$292{\cdot}10^3$	28720	3613	558
Max. overest.	-	0.07%	0.24%	0.69%
Avg. overest.	-	0.03%	0.10%	0.33%
Min.buf. / max.thr. overest.	-	0%	0.07%	0.19%
Exec. time	$53min$	$5min$	$36s$	$7ms$

	MP3 playback			
	exact	$n = 3$	$n = 5$	$n = 15$
Min. pos. throughput (s^{-1})	4.2	4.2	4.2	4.2
Distr. size	1977	1977	1977	1977
Max. throughput (s^{-1})	8.3	8.3	8.3	8.3
Distr. size	2898	2994	2902	3867
#Pareto points	620	203	121	43
#Min. distr.	620	203	121	43
#Distr. checked	2152	376	160	55
Max. overest.	-	6.11%	11.66%	50.93%
Avg. overest.	-	1.02%	1.77%	12.32%
Min.buf. / max.thr. overest.	-	3.31%	0.14%	33%
Exec. time	$21s$	$3.7s$	$1.6s$	$0.49s$

Table 7.3: MP3 playback alternative approximations.

	exact	$n = (5, 15)$	$n = 3, 5$ (combined)
Min. pos. throughput (s^{-1})	4.2	4.2	4.2
Distr. size	1977	1977	1977
Max. throughput (s^{-1})	8.3	8.3	8.3
Distr. size	2898	2907	2902
#Pareto points	620	42	255
#Min. distr.	620	42	255
#Distr. checked	2152	55	536
Max. overest.	-	12.06%	5.47%
Avg. overest.	-	2.00%	0.71%
Min.buf. / max.thr. overest.	-	0.31%	0.14%
Exec. time	$21s$	$0.61s$	$5.3s$

for these experiments are shown in Table 7.2. The results show that the approximation technique drastically improves the run-time of the exploration at the cost of a reduced number of Pareto points found (but as already said, it is hard to imagine that hundreds or thousands of Pareto points are practically meaningful).

The approximation may lead to an overestimation of the required storage space for a given throughput. The table shows for each experiment the maximum overestimation observed for an arbitrary Pareto point in the complete trade-off space, the average overestimation over the entire space, and the overestimation of the minimal buffer requirements allowing maximal throughput. The results show that the overestimation is very small in general, which shows that dropping Pareto points for these models does not have much impact in terms of storage requirements computed for a given throughput constraint. The only exception is the overestimation for the MP3 playback application with multiplication factor 15, suggesting that this factor is too large. The peaks in maximum overestimation in the MP3 experiment are to be expected, because large overestimation may occur for large multiplication factors and/or large step sizes in combination with small buffer sizes. However, small buffers are part of the trade-off space that the algorithm explores first. In those cases, the exact technique, an approximation with a smaller multiplication factor, or an approximation aiming to bound the relative overestimation (as explained at the end of the previous subsection) can be used to determine the appropriate buffer sizes satisfying the throughput constraint.

The experiments reported in Table 7.2 use only one multiplication factor, which is the same for all the sized edges. There are several ways to improve the obtained results. One way is to carefully select different multiplication factors for the different edges. Based on the results for the MP3 playback application, it seems that a relatively small multiplication factor ($n = 5$) should be used for the first edge while the second edge should have a larger multiplication factor ($n = 15$). The column labeled $n = (5, 15)$ in Table 7.3 shows that, using these multiplication factors, the trade-off space can be approximated with very limited overestimation. Another way is to apply the exact technique (or another approximation) on a designated part of the approximated space, in the same way as the combination of the technique presented in this chapter with heuristics that was explained earlier. Given a throughput constraint, one can first make a coarse approximation of the trade-off space with a large multiplication factor. Then, one can choose a distribution that comes close to satisfying the constraint as the starting point for a finer grain approximation or an exact exploration of the trade-off space up to the point that the throughput constraint is satisfied. This approach was used to find the minimal storage distribution for maximal throughput of the MP3 playback application. First the trade-off space was approximated in 3.7s with multiplication factor $n = 5$ (see Table 7.2). The first storage distribution below the largest one found by this approximation was used as starting point for the exact algorithm. This algorithm found the minimal storage distribution for maximal throughput within 27ms. A third way to improve approximation

results is to simply combine the results of two or more approximations of the trade-off space. This may lead to a reduced overestimation, as illustrated by the last column of Table 7.3. Combining the approximations of the trade-off space obtained via multiplication factors 3 and 5 leads to an increased number of Pareto points found, resulting in a reduced average overestimation when compared to the two approximations in isolation. Table 7.3 shows the results when this approach is applied to the MP3 playback model. To obtain this result, the two approximations were computed explicitly and then combined. A more efficient implementation would first compute the approximation with the largest multiplication factor, and then use the information about distributions already explored while computing the approximation with the smaller multiplication factor. This would lead to an amount of checked distributions and a run-time which are less than the sums of those values for the individual approximations. The experiment shows the versatility of the approximation technique. Note that it does not make sense to combine approximations when one multiplication factor is a divisor of (one of) the other multiplication factor(s) (which is why Table 7.3 does not report any other combinations).

7.8 Buffer Requirements for Binding-aware SDFGs

The technique presented in the previous sections computes the trade-off space between the buffer requirements and throughput of an SDFG assuming unlimited processing resources. In other words, it assumes that an actor can be executed as soon as all its input tokens are available. Actor firings never have to be postponed till a processing resource becomes available. In practice, multiple actors are often mapped to the same resource and actor firings are delayed till the firing is scheduled on the resource. This section discusses how this resource sharing, as captured by a binding-aware SDFG (see Chapter 6), can be taken into account in the storage-space/throughput trade-off space exploration.

In Section 6.6, it is explained how the storage space constraints of edges are modeled in a binding-aware SDFG. The storage space of an edge whose source and destination actor are bound to the same tile is modeled with the SDF model shown in Figure 6.4(b). The initial tokens α_{tile} on the edge d_b model the amount of storage space allocated to the edge d. The SDF model shown in Figure 6.4(c) is used to model the situation in which the source and destination actor are bound to different tiles. The storage space constraint on the source tile is modeled with the initial tokens α_{src} on the edge from actor a_c to actor a_i. The storage space allocated on the destination tile is modeled with the initial tokens α_{dst} on the edge from a_j to a_c. Note that this puts a constraint on the combined storage space used by the edge from a_c to a_s and the edge from a_s to a_j.

The edges in Figure 6.4 that contain the initial tokens α_{tile}, α_{src} and α_{dst} model storage space allocations of a binding-aware SDFG. Changing the number of initial tokens α_{tile}, α_{src} and α_{dst} may influence the throughput of the binding-

aware SDFG in the same way as this happens when the storage-space allocations of edges in an SDFG are changed. Similar to an SDFG, a binding-aware SDFG has a storage-space/throughput trade-off space. This space makes a trade-off between the number of initial tokens α_{tile}, α_{src} and α_{dst} on the various edges of a binding-aware SDFG and the throughput of the graph. This section explains how the notion of a causal dependency (see Definition 24) must be changed in order to find all minimal storage distributions of a binding-aware SDFG.

A causal dependency indicates that an end of firing transition of an actor a_i, as defined by the operational semantics of an SDFG, enables a start of firing transition of an actor a_j. When ignoring resource sharing, the production of a token is the only reason for the existence of this relation. This follows immediately from the operational semantics of an SDFG as given by Definition 7. When resource sharing is considered, the start of an actor firing depends not only on the availability of tokens, but also on the schedule that is used on the shared resource. An actor is only fired when it has sufficient tokens and when the actor firing is scheduled on a shared resource (see Definition 19). This property is taken into account in the definition of a binding-aware causal dependency.

Definition 32. (Binding-Aware Causal Dependency) *An actor a_i has a binding-aware causal dependency on an actor a_j via an edge d when there is a causal dependency from a_i to a_j via d, or when a_i and a_j are bound to the same resource and a firing of a_i is scheduled in the static-order schedule directly after a firing of a_j.*

According to Definition 32, any causal dependency in the execution of a binding-aware SDFG is also a binding-aware causal dependency. The start of an actor firing also has a binding-aware causal dependency with the end of an actor firing when these actor firings occur on the same resource and one actor firing starts immediately after the end of the other actor firing. Using the definition of a binding-aware causal dependency, the abstract dependency graph is defined similar to Definition 28 (deadlock-free case) and Definition 29 (deadlocked case).

Algorithm 1 can be used to explore the storage-space/throughput trade-off space of a binding-aware SDFG when the notion of a binding-aware causal dependency is used. The static-order schedules and the time-slice constraints might hide causal dependencies between actor firings. By including successive firings as specified by the static-order schedules as dependencies in the dependency graph, as is done in the definition of a binding-aware causal dependency, it is guaranteed that no storage dependencies are missed. The optimality of the result of Algorithm 1 can now be proven in the same way as Theorem 4. Remember that Algorithm 1 will only change the number of initial tokens α_{tile}, α_{src} and α_{dst} in the binding-aware SDFG which is inputted to the algorithm as these initial tokens model the storage space allocated to the edges of the graph.

Algorithm 1 requires a (binding-aware) SDFG G with maximal throughput Th_{max}. This maximal throughput is needed to guarantee termination of Al-

gorithm 1 when all minimal storage distributions are found. It is possible to compute the maximal throughput of a binding-aware SDFG. A platform instance constrains the maximum amount of memory available for the storage space of the dependency edges. Every edge that models storage space in the binding-aware SDFG could be assigned a storage space equal to the total amount of memory available in the platform. After modeling this storage space allocation in the graph, the throughput Th_{max} can be computed through a constrained self-timed execution of the binding-aware SDFG (see Section 6.6.1).

7.9 Summary

This chapter presents a method to explore the trade-offs between the throughput and memory usage for SDFGs. It differs from existing buffer sizing methods as it can determine exact minimum memory bounds for any achievable throughput of the graph. Other methods can only determine an upper bound on the minimal memory requirement for the lowest non-zero or highest throughput of the graph.

In addition to the exact exploration algorithm, a generic and very versatile approximation technique is presented that is based on the exact algorithm. The approximation provides throughput guarantees, and it has a proven analytical upper bound on the overestimation in buffer sizes. Approximation of the trade-off space can be used when the run-times of the exact technique would become problematic. The results for the approximation technique show that it can drastically improve the run-time needed for the exploration of the trade-off space with only very limited overestimation of the storage space.

This chapter also shows how resource constraints due to binding and scheduling that are not modeled in the SDFG can still be taken into account in the exploration. This enables a design-flow step in which the memory requirements of an SDFG can be minimized under a throughput constraint once the SDFG is bound and scheduled onto a multi-processor system-on-chip.

Chapter 8

Network-on-Chip Routing and Scheduling

8.1 Overview

Current NoCs like Æthereal [123] and Nostrum [93] use circuit-switching to create connections through the NoC which offer timing guarantees. Today's routing and scheduling solutions however (a) often do not use all routing flexibility of NoCs and (b) make bandwidth reservations for connections with throughput/latency guarantees that are unnecessarily conservative. To illustrate the first point, for example, the scheduling strategies presented in [53, 64] restrict themselves to minimal length routes. Modern NoCs allow the use of other, more flexible, routing schemes. As an illustration of the second point, consider a simple NoC with three links l_1, l_2 and l_3. The data streams sent over l_1 and l_2, shown in Figure 8.1, are both sent over l_3. Traditional NoC scheduling strategies [53, 59] reserve two guaranteed throughput connections on the link ($l_{3,trad}$). However, given the timing of the data streams on l_1 and l_2, it is possible to combine both streams and preserve bandwidth ($l_{3,new}$). The essential idea is not to reserve bandwidth for guaranteed throughput connections permanently during the entire life time of an active data stream but only during certain intervals, typically per communicated message (a token in an SDFG). The use of non-minimal routes and the intelligent reservation of NoC bandwidth leads to a better resource utilization in the NoC.

Modern multimedia applications more and more exhibit dynamism that causes the application to have a number of different communication patterns, called **communication scenarios**. An extension to the techniques of [53, 59] to handle this type of dynamism is presented in [99], which introduces a technique to allocate resources for each scenario while guaranteeing that sufficient resources are available when switching between scenarios. It ignores that often information is available on the time needed to switch between the scenarios. As a result, it suffers from

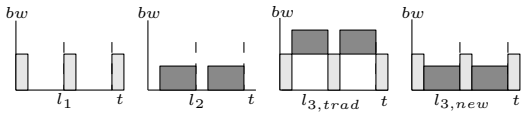


Figure 8.1: Motivating example.

the resource over-allocation problem as illustrated in Figure 8.1. Using this information, it is possible to minimize the resource usage when switching between scenarios.

This chapter explores and compares several new routing and scheduling strategies for data streams that exploit all scheduling freedom offered by modern NoCs and minimize resource usage. It is furthermore shown how communication scenario transitions can be taken into account. Scheduling strategies which minimize resource usage will be able to schedule problems with tighter latency constraints and/or larger bandwidth requirements. The proposed scheduling techniques can serve as the basis for step 12 of the design flow introduced in Section 1.4.

The remainder of this chapter is organized as follows. The next section discusses related work. Section 8.3 presents the communication task graph model. This model captures all timing constraints from an application that have to be satisfied by the NoC routing and scheduling strategies. The architecture of the NoC is discussed in Section 8.4. The time-constrained scheduling problem is formalized in Section 8.5. Several different scheduling strategies are presented in Section 8.6. The benchmark used to evaluate these strategies is presented in Section 8.7. The experimental evaluation of the proposed scheduling strategies on this benchmark is discussed in Section 8.8. A technique to extract a communication task graph model from an SDFG that is bound and scheduled on an MP-SoC is presented in Section 8.9. Section 8.10 concludes this chapter.

8.2 Related Work

This chapter considers scheduling streaming communication on a NoC within given timing constraints while minimizing resource usage. Only communication with timing constraints is considered in this chapter. In practice, some communication streams in an application may have no timing requirements. Scheduling techniques for these streams are studied in e.g. [120]. Those techniques can be used together with the approach presented in this chapter to schedule both the communication without and with timing constraints.

In [64], a state-of-the-art technique is presented to schedule time-constrained communications on a NoC when assuming acyclic, non-streaming communication. That is, tasks communicate at most once with each other (i.e., there is no notion of a pipelined execution of different iterations of the task graph). The communication model, presented in Section 8.3, allows modeling of communication streams

in which tasks periodically, i.e., repeatedly, communicate with each other. Acyclic communication can be dealt with by assuming a period with infinite length.

Scheduling streaming communication with timing guarantees is also studied in [53, 59, 123]. They apply a greedy heuristic and reserve bandwidth for streams, whereas this thesis proposes to reserve bandwidth per message and present several different heuristics. The results presented in this chapter show a clear improvement in resource usage. In [99], an extension to [53, 59] is presented to schedule multiple communication patterns onto a single NoC. It assumes that the streams of different communication patterns are independent of each other and no timing relation between them is known. As a result, streams from different patterns cannot share bandwidth. In this chapter, a technique is presented to share bandwidth between multiple communication scenarios when a timing relation between the scenarios is known. The scenario technique is inspired by scenario-based design approaches as proposed in [48, 50, 89].

Many NoCs like Nostrum [93], SPIN [56], and the NoC proposed by Dally and Towles [32] use regular NoC architectures like a mesh, torus or fat-tree. The regular structure of these NoCs fits well with simple routing schemes like XY-routing. These architectures assume that the computational elements connected to the NoC are all of similar size. In practice, existing IP-blocks do not always meet this requirement. Furthermore, applications with irregular communication requirements do not fit well with the regular NoCs. For these reasons, irregular NoC topologies and their accompanying routing and scheduling techniques are studied [62, 103, 106]. In [62], a NoC architecture is studied in which some links from a mesh are removed. A technique to design application-specific NoCs is studied in [103]. [106] studies a mesh-topology with added links that reduce the long latencies between vertexes in the mesh. The routing and scheduling technique presented in this chapter can be used in combination with any arbitrary (regular or irregular) NoC topology.

8.3 Communication Modeling

Multimedia applications, for instance an MP3 decoder, operate on streams of data. Many NoC scheduling and routing strategies use a communication centric model to describe these applications. They assume that an application consists of a set of periodically executed tasks that exchange messages with each other via (data) streams. These tasks are bound to the various processors in the system. Whenever multiple tasks are bound to one processor, the execution order of these tasks is fixed through a schedule. Dynamism in the application can cause differences in the time at which tasks consume and produce messages. When the timing difference is small or occurs infrequently, it can be considered as jitter on the communication pattern. To provide timing guarantees, the worst-case communication pattern which includes this jitter must be considered when allocating resources. It is also possible that the dynamism in the application causes changes

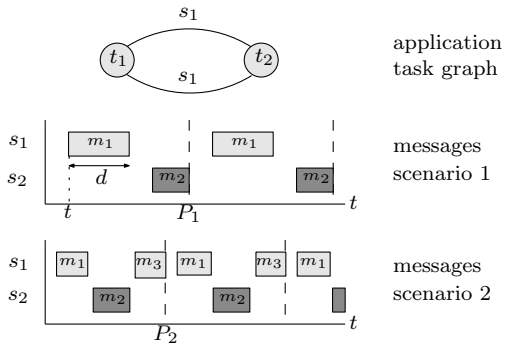


Figure 8.2: Example communication task graph.

in the communication pattern which are effective over a longer period of time. An MP3 decoder for example could switch from decoding a stereo stream to a mono stream. This situation could be taken into account by allocating resources for the worst-case communication pattern that can occur. However, this will result in a resource allocation which is too conservative for most situations [48, 99]. The solution to prevent over-allocation of the resources is to consider communication patterns which differ considerably from each other as separate scenarios. In an MP3 decoder, for example, the decoding of a stereo and mono stream could be seen as two distinct scenarios. It is possible that a switch from one scenario to another is time-constrained and that the two scenarios **overlap** for some time. These aspects should be taken into account when allocating resources.

Figure 8.2 shows an example of a simple communication task graph consisting of two tasks t_1 and t_2. The communication between the two tasks can follow two scenarios. In the first scenario, task t_1 sends a message m_1 through stream s_1 to task t_2 and t_2 sends a message m_2 through s_2 to t_1. This pattern is repeated with a period P_1. The second scenario has a different communication pattern and period. In the second scenario, task t_1 sends every period P_2 two messages m_1 and m_3 to task t_2 and t_2 sends a message m_2 to t_1. Note that also the time at which messages are sent and the size of them is different in the two scenarios. When running, switches from one scenario to another may occur. During this switching period, due to the streaming nature of applications, multiple scenarios may be active simultaneously. Table 8.1 gives the overlap when switching between the two scenarios of the example communication task graph. It shows, for example, that when switching from scenario 1 to scenario 2, both communication patterns overlap for 5 time-units. It is also possible that a switch from one scenario to another scenario cannot occur or happens without overlap.

The communication task graph model focuses on modeling the communication within an application that is already bound and scheduled on an MP-SoC. The schedule and timing constraints imposed on the application determine time

Table 8.1: Scenario overlap during switching

	scenario 1	scenario 2
scenario 1	-	5
scenario 2	3	-

bounds within which each task must be executed. Similarly, they determine time bounds within which messages must be communicated between the tasks. This chapter presents techniques to schedule messages from multiple scenarios, which are specified with time bounds, on the NoC. Furthermore, Section 8.9 shows how the execution of an SDFG that is bound and scheduled on a MP-SoC can be captured in a set of communication scenarios. It also shows how time bounds for the messages communicated within each scenario are derived.

8.4 Interconnect Graph

The multi-processor platform template introduced in Chapter 3 uses a NoC to connect the different tiles in the platform to each other. Each tile contains a **network interface** (NI) through which it is connected with a single router in the NoC. The **routers** can be connected to each other in an arbitrary topology. The connections between routers and between routers and NIs are called **links**. In this chapter, the connections between the processing elements and the NI inside a tile are ignored. It is assumed that these connections introduce no delay, or that the delay is already taken into account in the timing constraints imposed on communications, and that there is sufficient bandwidth available. Hence, the NI can be abstracted away into the tile. Given this abstraction, the interconnect can be described with the following graph structure.

Definition 33. (Interconnect graph) *An interconnect graph (V, L) is a directed graph where each vertex $u, v \in V$ represents either a tile or a router, and each edge $l = (u, v) \in L$ represents a link from vertex u to vertex v.*

Communication between tiles involves sending data over a sequence of links from the source to the destination tile. Such a sequence of links through the interconnect graph is called a route and is defined as follows.

Definition 34. (Route) *A route r between vertex u and vertex v with $u \neq v$ is a path in the interconnect graph of consecutive links from u to v without cycles. The operators src and dst give respectively the source and destination vertex of a route or a link. The length of a route r is equal to the number of links in the path, and denoted $|r|$. $l \in r$ is used to denote that the link l appears in the route r.*

Links can be shared between different communications by using a TDMA-based scheduler in the routers and NIs. All links have the same number of TDMA

slots, N, and each slot has the same bandwidth. At any moment in time, at most one communication can use a slot in a link. This guarantees that the NoC schedule is contention-free. Hence, no deadlock will occur. The data transfered over a link in a single slot is called a **flit** and it has size sz_{flit} (in bits). To minimize buffering in routers, a flit entering a router at time-slot t must leave the router at slot $t + 1$. Not all slots in a link may be available for use by a single application. Part of the slots may already be used by other applications mapped to the system.

Wormhole routing [31] is used to send the flits through the network. This technique requires limited buffering resources and offers strict latency bounds. A message is divided by the sending NI into flits. The flits are then routed through the network in a pipelined fashion. This reduces the communication latency considerably. All flits which belong to the same message and are sent in consecutive slots form a **packet**. The first flit in a packet (header flit) contains all routing information and leads the packet through the network. The header has a fixed size of sz_{ph} bits ($sz_{ph} \leq sz_{flit}$). The remaining $sz_{flit} - sz_{ph}$ bits in the header flit can be used to send (a part of) the actual message. The size of the header must be taken into account when allocating resources in the NoC. Two messages, possibly sent between different source and destination tiles but over one link at non-overlapping moments in time can use the same slot. For messages that use the same slot in the link between the source tile and the first router but a different route, the routing information stored in the NI for this slot must be changed. This **reconfiguration** of the NI can be implemented efficiently by sending a message from a processor or communication assist [30] inside a tile to its NI to change the routing information. Alternatively, the approach outlined in [58] can be used to implement the reconfiguration. In [58], the routing information of multiple different streams is associated with a TDMA slot. When a message is send within this TDMA slot, the NI uses the routing information associated with the stream that this message belongs to. The time-constrained scheduling problem, presented in Section 8.5, assumes that time required to reconfigure the NI is T_{reconf}. During this reconfiguration time, the slot may not be used to send messages.

Tasks in an application communicate with each other through streams of messages. The ordering of the messages in a stream must be preserved. To realize this, the NIs send messages onto the network in the same order as they receive them from the processors. The scheduling of communications on the NoC must also guarantee that the messages are received in the same order. No reordering buffers are thus needed in the NIs, which simplifies their hardware design. The NoC further requires that when the communication of a message is started, slots are claimed in the links it is using. These slots are only freed after the communication has ended. Preemption of a communication is not supported.

8.5 Time-Constrained Scheduling Problem

8.5.1 Overview

Informally, the time-constrained scheduling problem consists of finding a schedule for a set of scenarios CS, such that the schedule of each scenario has no conflicting resource requirements with the other (overlapping) scenarios and applications, and that the set of messages which make up a scenario are sent between different tiles in a system within given timing constraints. First, the problem of scheduling a single scenario $s \in CS$ is formalized in Section 8.5.2. The formalization is such that only a single period of the scenario needs to be scheduled. This schedule can then be repeatedly executed as often as necessary. It uses a function $\mathcal{U} : L \times \mathbb{N} \rightarrow \{used, not\text{-}used\}$ which indicates for every link at every moment in the time-span of one scenario period of the schedule whether a slot is occupied. The function $\mathcal{U}$ captures the resource constraints due to other applications using the same platform and from the schedules of other scenarios on the messages being scheduled from scenario s. Section 8.5.3 explains how the function $\mathcal{U}$ is constructed and used when scheduling multiple scenarios on the NoC. The complexity of the single-scenario scheduling problem is studied in Section 8.5.4.

8.5.2 Scheduling a Single Scenario

A communication scenario consists of a set of messages which must be scheduled on the NoC within their timing constraints. A message is formally defined as follows.

Definition 35. (MESSAGE) *Given an interconnect graph (V, L), a set of streams S and a period P. A message m is a 7-tuple $(u, v, s, n, \tau, \delta, sz)$, where $u, v \in V$ are respectively the source and the destination tile of the n-th message sent through the stream $s \in S$ during the period P. The earliest time at which the communication can start, relative to the start of the period, is given by $\tau \in \mathbb{N}_0$ $(0 \leq \tau < P)$. The maximum duration of the communication after the earliest start time is $\delta \in \mathbb{N}$ $(\delta \leq P)$. The size (in bits) of the message that must be communicated is $sz \in \mathbb{N}$.*

The communication task graph, shown in Figure 8.2, sends each period P_1 a message $m_1 = (u, v, s_1, n, \tau, \delta, sz)$ through the stream s_1 of scenario 1. This communication can start at time τ and must finish before $\tau + \delta$. Note that a communication may start in some period and finish in the next period. This occurs when $\tau + \delta > P_1$.

In practice, messages may not always have a fixed earliest start time, duration, or size. Conservative estimates on these figures should be used to construct the set of messages in order to guarantee that all communications fall within the timing and size constraints. Resources that are claimed but not used, due to for example a smaller message size, can be used to send data without timing requirements between tiles without providing guarantees, i.e. best-effort traffic.

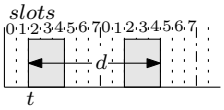


Figure 8.3: Scheduling entity on a link.

A message specifies timing constraints on the communication of data between a given source and destination tile. It does not specify the actual start time, duration, route and slot allocation. This information is provided by the scheduling entity.

Definition 36. (SCHEDULING ENTITY) *A scheduling entity is a 4-tuple* (t, d, r, st)*, where* $t \in \mathbb{N}_0$ *is the start time of the scheduled message relative to the start of the period and* $d \in \mathbb{N}$ *is the duration of the communication on a single link. The scheduled message uses the route* r *in the network and the slots it uses from the slot table of the first link* $l \in r$ *are given by the set* $st \subseteq \{0, \ldots, N-1\}$ *with* N *the slot-table size.*

The slots given in st are claimed on the first link of the route r at time t for the duration d. On the next link, the slot reservations are cyclically shifted over one position. So, these slots are claimed one time-unit later, i.e., at $t + 1$, but for the same duration d. The complete message is received by the destination at time $t + d + |r| - 1$. Figure 8.3 shows a scheduling entity which sends a message over a link with a slot-table of 8 slots. Starting at time $t = 2$, the slots 2, 3, and 4 are used to send the message. The communication ends after $d = 11$ time units. In total, two packets consisting both of three flits are used to send the message.

The relation between a message and a scheduling entity is given by the **schedule function**, formally defined below in Definition 37. Among all schedule functions, those respecting the constraints in Definition 37 are called **feasible**. One of the conditions that a feasible schedule needs to satisfy is that it is contention free, i.e., slot-tables should not be simultaneously reserved by different messages, different scenarios or different applications. An important aspect in this context is the relation between the slot-table size N and the period of the scenario P. Figure 8.4 shows an example of a link l with slot table size $N = 8$. The second slot from the slot table is occupied by another application. The message(s) from a scenario with period $P = 7$ are also scheduled on the link l. In the first period, the scenario uses the third slot from the slot table. In the next period, the scenario uses the second slot from the slot table. However, this slot is already occupied by another application. Hence, there is contention on the link at this moment in time as both schedules want to use the same slot at the same moment in time. This example shows that it is in general not sufficient to guarantee that the first period of some scenario is contention-free; also following periods must be free of contention. In fact, the number of periods of a schedule with period P which must be checked for contention is equal to the least common multiple of P and N, $\text{lcm}(P, N)$, divided

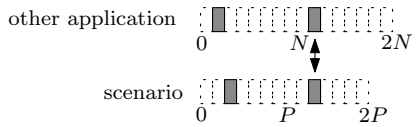


Figure 8.4: Contention in second period.

by P. After this number of periods, the first time-unit of the scenario coincides again with the first slot of the slot table. For simplicity, it is assumed that the period of a scenario is a multiple of the size of the slot table. It is then sufficient to check only a single period for contention, which simplifies the formulas in the remainder. This is not a restriction as any scenario whose period P which does not adhere to this requirement can be concatenated for $\text{lcm}(P, N)/P$ times before scheduling it. The period of the concatenated scenario is then a multiple of the slot table with size N.

Consider now Definition 37. The first two constraints make sure that the communication takes place between the correct source and destination tile. The third and fourth constraint guarantee that the communication falls within the timing constraints given by the message. The fifth constraint ensures that enough slots are reserved to send the message and packet headers over the network. It uses a function $\pi(e)$ which gives for a scheduling entity $e = (t, d, r, st)$ the number of packets which are sent between t and $t + d$ on the first link of the route r considering the slot reservations st and assuming that at time 0 the first slot of the slot table is active. The function $\varphi(e)$ gives the number of slots reserved by e between t and $t + d$. The sixth constraint makes sure that a scheduling entity does not use slots in links which are at the same moment in time used by other applications or scenarios. It uses a function $\sigma(e, l_k, x)$ which indicates for a scheduling entity e and the k-th link l_k on the route r of e whether it uses a slot from the slot-table with size N of l_k at time x; $\sigma(e, l_k, x) = \textit{used}$ when $(x + k) \bmod N \in \{(s + k) \bmod N | s \in st\}$ and $\sigma(e, l_k, x) = \textit{not-used}$ otherwise. The seventh constraint requires that the schedule is contention-free. The next constraint makes sure that there is enough time to reconfigure the NI between two messages which originate at the same NI and use the same slot but different routes. The last constraint enforces that the ordering of messages in a stream is preserved.

Definition 37. (Schedule function) *A schedule function is a function $\mathcal{S} : M \rightarrow E$ where M and E are respectively the set of messages and scheduling entities. The function $\mathcal{S}$ is called feasible if and only if, for all messages $m = (u, v, s, n, \tau, \delta, sz) \in M$ associated to scheduling entity $\mathcal{S}(m) = e = (t, d, r, st)$,*

1. *the route starts from the source tile: $u = src(r)$,*

2. *the route ends at the destination tile: $v = dst(r)$,*

3. *the communication does not start before the earliest moment in time at which the data is available:* $t \geq \tau$,

4. *the communication finishes not later than the deadline:* $t+d+|r|-1 \leq \tau+\delta$,

5. *the number of allocated slots is sufficient to send the data:* $sz + sz_{ph} \cdot \pi(e) \leq sz_{flit} \cdot \varphi(e)$,

6. *the communication uses no slots occupied by other applications or scenarios: for all links* $l \in r$ *and time instances* x *with* $0 \leq x < P$, $\sigma(e, l, x) =$ *not-used when* $\mathcal{U}(l, x) =$ *used*,

and for each pair of messages $m_1, m_2 \in M$ *with* $m_1 \neq m_2$, $m_1 = (u_1, v_1, s_1, n_1, \tau_1, \delta_1, sz_1)$, $\mathcal{S}(m_1) = e_1 = (t_1, d_1, r_1, st_1)$, $m_2 = (u_2, v_2, s_2, n_2, \tau_2, \delta_2, sz_2)$, *and* $\mathcal{S}(m_2) = e_2 = (t_2, d_2, r_2, st_2)$,

7. *for all* $l \in r_1 \cap r_2$ *and for all* x *with* $0 \leq x < P$, $\sigma(e_1, l, x) =$ *not-used or* $\sigma(e_2, l, x) =$ *not-used*,

8. *if* $u_1 = u_2$, $r_1 \neq r_2$, *and* $st_1 \cap st_2 \neq \emptyset$, *then there is enough time to reconfigure the NIs:* $(t_2 - t_1 - d_1) \bmod P \geq T_{reconf}$,

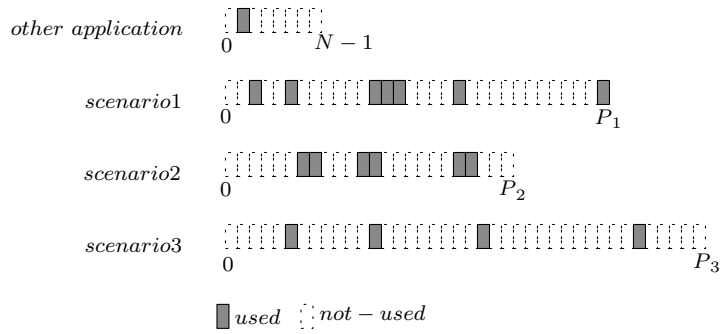
9. *if* $s_1 = s_2$ *and* $n_1 < n_2$, *then the ordering of these messages is preserved:* $t_1 + d_1 < t_2 \wedge t_1 + d_1 + |r_1| - 1 < t_2 + |r_2|$.

If a schedule function is not feasible, it means that one or more of the above rules are violated in at least one associated scheduling entity. Such a schedule is called **infeasible**. By construction, any feasible schedule is contention-free and hence free of deadlock and livelock [52].

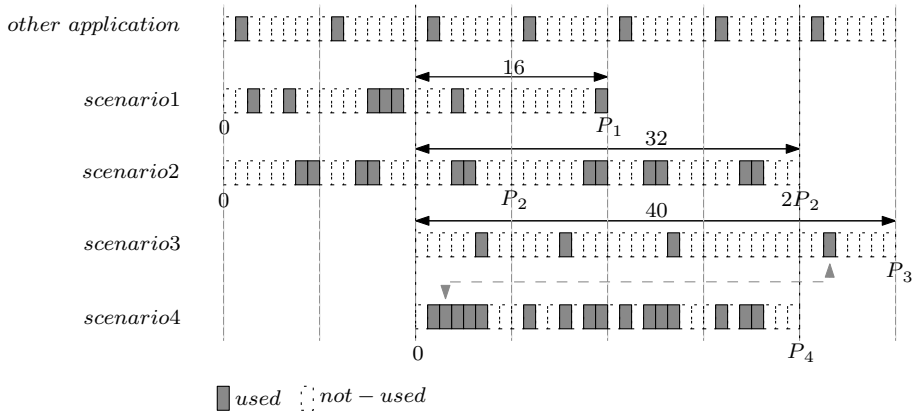
8.5.3 Scheduling Multiple Scenarios

The problem of scheduling a set of scenarios CS consists of finding a feasible scheduling function $\mathcal{S}$ for the streams in all scenarios $s \in CS$. Feasibility requires that resource constraints from a scenario $s_i \in S$ are considered when scheduling another scenario $s_j \in S$ which overlaps with s_i. Of course, slots occupied by other applications must also be taken into account. When scheduling a single scenario, all these constraints are taken into account through the function $\mathcal{U}$. This subsection explains with an example, illustrated in Figure 8.5, how the function $\mathcal{U}$ is constructed.

Consider the situation in which messages of four scenarios must be scheduled on a link l. Assume that the link has a slot table of size 8 of which the second slot in the slot table is already occupied by another application. This slot cannot be used for any scenario. Assume now that the first three scenarios are already scheduled on l. To schedule the messages of the fourth scenario, a function $\mathcal{U}$ must be constructed which takes into account the slots used by the three already scheduled scenarios and the slots occupied by other applications. Let the scenarios 1, 2 and



(a) Schedule of scenarios 1,2 and 3 and some other application.



(b) Constraints on scenario 4.

Figure 8.5: Constraints for link l on scenario 4.

3 have respectively a period of 32, 24 and 40 time-units. The slots occupied by the scheduled scenarios 1, 2 and 3 are shown in Figure 8.5(a). The fourth scenario has a period of 32 time-units and it has an overlap with the scenarios 1, 2 and 3 of respectively 16, 32 and 40 time-units. The latter implies that when switching, for example, from scenario 1 to scenario 4, the last 16 time-units of scenario 1 overlap with the first 16 time-units of scenario 4. Note that a consequence of the earlier assumptions that the start of a scenario aligns with a slot-table rotation and that the period of a scenario is a multiple of the slot-table size is that the overlap between two scenarios is a multiple of the slot-table size. This means that a new scenario can only be started at the beginning of a slot-table rotation. If desirable, this restriction can be relaxed to allow arbitrary overlap, but this makes the construction of $\mathcal{U}$ more tedious and it seems of little practical value.

The overlap of 16 time-units between scenarios 1 and 4 implies that the last 16 time-units of the last period of scenario 1 overlap with the first 16 time-units of the first period of scenario 4. After these 16 time-units, scenario 1 is no longer repeated. So, the slots occupied in the last 16 time-units of scenario 1 cannot be used for scenario 4. Due to the overlap between scenarios 2 and 4, more than one period of scenario 2 overlaps with scenario 4. The slots occupied by scenario 2 cannot be used for scenario 4. The overlap of scenario 3 on scenario 4 is even larger than a complete period of scenario 4. Slots occupied after a complete period of scenario 4 do constrain the next period of the scenario and should therefore also be considered as a constraint on the available slots. The dotted arrow between scenario 3 and 4 gives an example of such a constraint. Only slots which are not occupied by other applications or any of the overlapping scenarios can be used to schedule the messages of scenario 4. These slots are colored white in Figure 8.5(b). Note that scenario 4 does not necessarily overlap with the other three scenarios simultaneously. In fact, that would be impossible with the given schedules for scenario 1 and 2, because those would then cause contention, as can be seen from Figure 8.5(b).

8.5.4 Complexity

In this section, it is proved that the single-scenario scheduling problem is NP-complete.

Theorem 7. (Scheduling complexity) *Given an interconnect graph $G = (V, L)$ and a scheduling problem consisting of a set of messages M with a period P. The problem of finding a feasible schedule function is NP-complete.*

Proof. First, it is shown that the problem belongs to NP. The verification algorithm must check whether the scheduling entities satisfy the constraints of Definition 37. The first five constraints and the last two can be checked in $O(|M|)$ time. The other constraints can be easily checked in polynomial time for a single link $l \in L$, but need to be repeated for all $l \in L$. However, this is still polynomial in the problem size.

To prove that the problem is NP-complete, it is shown that the disjoint-path problem [42] can be reduced in polynomial time to the scheduling problem discussed in this section. The disjoint-path problem was proved to be NP-complete (even for planar graphs) [42]. In the disjoint-path problem, a set of edge-disjoint paths in a given graph must be found between a set of pairs of vertexes. The reduction of the disjoint-path problem to the time-constrained scheduling problem works as follows. Let G be the graph and the set $\{(u_1, v_1) \dots (u_k, v_k)\}$ the pairs of vertexes that form an instance of the disjoint-path problem. An instance of the scheduling problem is constructed with interconnect graph G and the number of slots in each link equal to 1. The ordering of the messages in the streams is ignored, i.e. it is assumed that each message belongs to a different stream. The length of the period is equal to 1. The set of messages $M = \{(u'_1, v'_1, s_1, 1, 0, 1, 1), \dots, (u'_k, v'_k, s_k, 1, 0, 1, 1)\}$, i.e., all messages have sequence number 1, starting time 0, deadline 1, and size 1. This construction can be done in polynomial time. Suppose that there are edge-disjoint paths $p_1, \cdots, p_k$ between $(u_1, v_1) \dots (u_k, v_k)$. For each i, the path p_i, which is in fact a route from vertex u'_i to v'_i in the scheduling problem, is exclusively dedicated to the message $(u'_i, v'_i, s_i, 1, 0, 1, 1)$. The message is scheduled to be sent at time 0 using all bandwidth in the links on its route. It is easy to see that this schedule function is feasible. Conversely, suppose that there is a feasible schedule function for the set M; then, the set of scheduling entities cannot share bandwidth, as sharing bandwidth leads to missing deadlines. Since bandwidths are all set to be 1, it trivially follows that the routes are all disjoint. Hence, any feasible schedule is a solution to the disjoint-path problem. □

8.6 Scheduling Strategies

8.6.1 Overview

Given a set M of messages, a scheduling strategy for a single scenario must find a schedule entity e for each message $m \in M$ and the set E of scheduling entities must form a feasible schedule function (i.e., all constraints from Definition 37 must be met). Given that an exhaustive approach is not tractable, several heuristic approaches are presented. The heuristics allow the user to trade off quality of solutions and effort spent on solving problems. First, a greedy strategy is presented in Section 8.6.2. Typically, the greedy approach gives a solution quickly. However, it also excludes a large part of the solution-space. The second strategy, ripup, adds backtracking to the greedy approach. This improves the quality (number of feasible solutions found for a set of problems), but it also increases the run-time. The backtracking tries to resolve scheduling conflicts when they occur. The third strategy, presented in Section 8.6.4, tries to avoid conflicts by estimating a priori the occupation of all links. This should steer the routing process

to avoid scheduling conflicts and as such minimizes the use of the backtracking mechanism. A feasible schedule for the messages of a single scenario takes into account the constraints that originate from other applications and scenarios. These constraints are captured in the function $\mathcal{U}$. A scheduling strategy for multiple scenario based on the scheduling strategies for a single scenario is presented in Section 8.6.5.

8.6.2 Greedy

The greedy strategy explores a small part of the solution-space. As a result, it has a small run-time. However, it may miss solutions or find non-optimal ones in terms of resource usage. The greedy strategy essentially tries to schedule the largest, most time-constrained messages first, via the shortest, least congested route that is available. It works as follows. First, all messages $m = (u, v, s, n, \tau, \delta, sz) \in M$ are assigned a cost using Equation 8.1 and sorted from high to low based on their cost. The cost function guarantees that messages are ordered according to their (integer) size (larger size first) and that two messages with the same size are ordered with respect to their duration (tighter constraint first).

$$\text{cost}_M(m) = sz(m) + \frac{1}{\delta(m)} \tag{8.1}$$

Next, a schedule entity $e = (t, d, r, st)$ must be constructed for the first message $m = (u, v, s, n, \tau, \delta, sz) \in M$. To minimize the resource usage, the scheduling strategy must try to minimize the length of the routes. For this reason, the greedy strategy determines a list R of all routes from u to v with the shortest length and assigns a cost to each route r using the following cost function that determines the minimum number of available slots in any link in a route during the time-span that the link might potentially be used by the message.

$$\text{cost}_R(r, m) = \min_{l_k \in r} \sum_{\tau(m)+k \leq x \leq \tau(m)+\delta(m)+k-|r|} \mathcal{F}(l_k, x) \tag{8.2}$$

with $\mathcal{F}(l_k, x) = 1$ when $\mathcal{U}(l_k, x) = \textit{not-used}$ and $\mathcal{F}(l_k, x) = 0$ otherwise. The routes are sorted from low to high cost giving preference to the least congested routes. Next, a schedule entity e is constructed using the first route r in R. The scheduling strategy should avoid sending data in bursts as this increases the chance of congestion. Therefore, the start time, t, of e is set equal to the earliest possible time respecting the third and last constraint from Definition 37. Given t and the fourth and last constraint from Definition 37, the maximal duration d of e can be computed. All slots available between t and the maximal duration on the first link of the route, respecting the sixth, seventh and eighth constraint from Definition 37, are located. From these slots, a set of slots, st, is selected which offer sufficient room to send the message and the packet headers. The scheduler tries to minimize the number of packets that are used by allocating consecutive

slots in the slot table. This minimizes the overhead of the packet headers, which in turn minimizes the number of slots needed to send the message and its headers. This leaves as many slots as possible free for other messages. It is possible that no set of slots can be found which offer enough room to send the message within the timing constraints. If this is the case, the next route in R must be tried. In the situation that all routes are unsuccessfully tried, a new set of routes with a length equal to the minimum length plus one is created and tried. This avoids using routes longer than needed and it never considers a route twice. A route which uses more links than the minimum required is said to make a **detour**. The length of the detour is equal to the length of the route minus the minimum length. If no set of slots is found when a user-specified maximum detour of X is reached, then the problem is considered infeasible. If a set st of slots is found, the minimal duration d needed to send the message via the route r, starting at time t using the slots st is computed using the fifth constraint from Definition 37. The scheduling entity $e = (t, d, r, st)$ is added to the set of schedule entities E. The new set of schedule entities $E \cup \{e\}$ is guaranteed to respect all constraints from Definition 37. The next message can then be handled. The process is repeated till a schedule entity is found for all messages in M, or until the problem is considered infeasible (i.e. a message cannot be scheduled).

8.6.3 Ripup

The ripup strategy uses the greedy strategy described in the previous section to schedule all messages. This guarantees that all problems that are feasible for the greedy strategy are also solved in this strategy. Moreover, the same schedule function is found. As soon as a conflict occurs (i.e. no schedule entity e_i can be found for a message m_i which meets the constraints given in Definition 37), an existing schedule entity e_j is removed from the set of schedule entities E. To choose a suitable e_j, the heuristic calculates for each schedule entity $e_j \in E$ the number of slots it uses in the links that can also be used by e_i. The higher this number, the larger the chance that e_j forms a hard conflict with e_i. A schedule entity e_j with the largest conflict is therefore removed from E. This process is continued until a schedule entity e_i for the message m_i can be created that respects the constraints given in Definition 37. After that, the messages of which the corresponding schedule entities were removed are re-scheduled in last-out first-in order. On a new conflict, the ripup mechanism is activated again. The user specifies the maximum number of times a ripup may be performed. This allows a trade-off between quality and run-time of the strategy.

8.6.4 Global knowledge

The ripup scheduler does not know a priori which unscheduled messages need to use links in the route it assigns to the message it is scheduling. It can only use local information to avoid congestion. The global knowledge strategy tries

to estimate, before scheduling messages, the number of slots that are needed in each of the links. This gives the scheduling strategy global knowledge on the congestion of links. This knowledge is used to guide the route selection process when scheduling the messages.

Communication of a message m can take place at any moment in time within the time interval specified by m. Within this interval, the scheduling entity requires at least $\left\lceil \frac{\lceil sz(m)/sz_{flit} \rceil}{\max(\lfloor \delta(m)/N \rfloor, 1)} \right\rceil$ slots in each link of the route it uses. In the optimal situation, all scheduled messages use a route with the shortest length. To estimate the congestion on all links in the NoC, the strategy assumes that only shortest length routes are used. For each link $l \in L$, the strategy computes the minimal number of slots required at each moment in time when all messages which can use l, as it is part of at least one of their shortest routes, would use the link l. The function $\mathcal{C} : L \times \mathbb{N}_0 \rightarrow \mathbb{N}_0$ gives the estimated number of slots used in a link $l \in L$ at a given time x.

The global knowledge strategy uses the same algorithm as the ripup strategy. However, a different cost function is used to sort the routes it is considering when scheduling a message. The cost function used by the greedy and ripup strategy (Equation 8.2) is replaced by the following cost function.

$$\text{cost}_R(r, m) = \sum_{l_k \in r} \max_{\tau(m)+k \leq x \leq \tau(m)+\delta(m)+k-|r|} \mathcal{C}(l, x) \tag{8.3}$$

This cost function ensures that the routes are sorted based on the estimated congestion of the links contained in the routes. Routes containing only links with a low estimated congestion are preferred over routes with links that have a high estimated congestion. This minimizes the number of congestion problems which occur during scheduling. As such, it makes more effective use of the allowed ripups.

8.6.5 Multiple Scenarios

Given a set CS of scenarios, a scheduling strategy for multiple scenarios must find a feasible scheduling function for each scenario while taking into account the resource constraints of the scenarios on each other. These constraints are captured in the function $\mathcal{U}$ (See Section 8.5.3).

The multi-scenario scheduling strategy must first decide on the order in which it schedules the scenarios. A scenario which overlaps with many other scenarios has potentially tight resource constraints as many other scenarios share resources with it. When this scenario is scheduled first, the scheduling functions can minimize the number of packets which it needs to send its messages. This minimizes the resource usage of the scenario. The overlap in time-units between two scenarios is given by the function $\mathcal{O} : CS \times CS \rightarrow \mathbb{N}_0$. Note that this function is not commutative as the overlap of a switch from a scenario s_i to a scenario s_j may

differ from the overlap when switching from s_j to s_i. To sort the scenarios according to the overlap which they have with each other, the multi-scenario scheduling strategy assigns a cost to each scenario using Equation 8.4 and sorts them from high to low based on their cost.

$$\text{cost}_{CS}(s_i) = \sum_{s_j \in CS} \mathcal{O}(s_i, s_j) + \mathcal{O}(s_j, s_i) \tag{8.4}$$

Next, a set of scheduling entities E_i must be constructed for the set of messages M_i which make up the first scenario s_i in the ordered set of scenarios. This is done using a single-scenario scheduling function. The greedy, ripup or knowledge strategy can all be used for this purpose. The function $\mathcal{U}$ contains at that moment only the constraints which originate from other applications. When the used single-scenario scheduling strategy finds a feasible scheduling function $\mathcal{S}$, the multi-scenario strategy updates the function $\mathcal{U}$ to include the constraints which originate from $\mathcal{S}(M_i)$. This is done using the procedure described in Section 8.5.3. The multi-scenario strategy continues by scheduling the next scenario in the ordered list of scenarios. This process of scheduling a scenario and updating the constraint function $\mathcal{U}$ continues till either no feasible scheduling function is found for a scenario or all scenarios are scheduled. In the latter case, the problem is called feasible; otherwise, it is called infeasible.

8.7 Benchmark

A benchmark is needed to test the quality of the scheduling strategies. It must contain a set of problems that covers a large part of the problem space typical of realistic applications. It should also be large enough to avoid optimization towards a small set of problems. It is not possible to construct a benchmark containing only real existing applications. Profiling these is too time-consuming and they are not representative for more demanding future applications at which NoCs are targeted. Therefore, a benchmark which consists of a set of randomly generated problems is used. A method to generate synthetic workloads for NoC performance evaluation is introduced in [141]. It assumes a communication model in which tasks exchange data-elements with each other through streams. The communication of data-elements has a periodic time behavior with some jitter on it. A benchmark generator is used to evaluate the scheduling strategies which is based on a similar idea. It is used to construct the benchmark that is used in the experimental evaluation of the scheduling strategies (Section 8.8). Experimental results on a set of realistic applications are reported in Chapter 10.

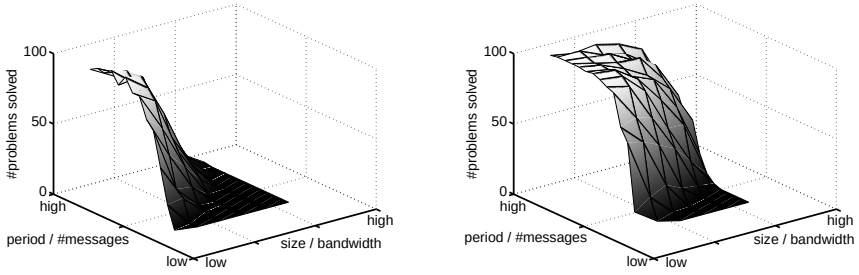
Many NoCs use a regular topology like a mesh [53, 59, 64, 93] or torus [32]. Tiles located at the edge of a mesh are restricted in the links that can be used as at least one direction is not available because of the topology. In a 3x3 mesh, this holds for all tiles except for one. In a 5x5 mesh, there are 16 edge tiles and 9 non-edge tiles and a 7x7 mesh has 24 edge tiles and 25 non-edge tiles. The ratio of

edge to non-edge tiles can possibly influence the scheduling strategies. To study this effect, problem sets are generated for a 3x3, 5x5 and a 7x7 mesh. All tiles in a bidirectional torus have the same number of links. So, a torus has compared to a mesh more scheduling freedom as it has more links. To study, the effect of the additional scheduling freedom on the scheduling strategy, torus topologies with the same dimensions as the mesh topology are included in the benchmark. The NoC topology can be optimized when the applications running on it are known at design-time. This may result in the use of irregular NoC topologies [62, 103, 106]. To study how the NoC routing and scheduling strategies behave on irregular topologies, two topologies with respectively 9 and 25 tiles are added to the benchmark. Following [62], these topologies are based on a regular mesh topology in which 10% of the links are removed. These irregular topologies are constructed such that communication between any pair of tiles remains possible.

A traffic generator is developed which creates a user-specified number of streams of messages between randomly selected source and destination tiles. The streams can model uniform and hotspot traffic. All messages in a stream are assigned a start time, size, and duration which consists of a randomly selected base value which is equal for all messages in the stream plus a random value selected for each individual message in the stream. The first part can be used to steer the variation in message properties between streams. The second part can be used to create variation between messages in a single stream (i.e. jitter).

The problem space can be characterized in a 2-dimensional space. The first dimension is determined by the number of messages which must be communicated within a period. The second dimension is determined by the ratio of the size of the messages communicated and the available bandwidth. When constructing the problem sets, it turned out that there is an area in the problem-space where problems change from being easy to solve to unsolvable. A selection of 78 equally distributed points around this area in the problem-space was made. For each point 100 problems were generated. This gives a benchmark with a set of 7800 different problems per topology-size and traffic model (uniform or hotspot). The mesh, torus and irregular topologies have similar topology sizes and can thus share the problem sets. Figure 8.6 shows for each point in the problem-space of the 5x5 mesh with uniform traffic how many problems are solved with the greedy and global knowledge strategies. The results for the greedy strategy show that most problems do not have a trivial solution. A solution is found for only 30% of the problems. The results of the global knowledge strategy show that 57% of the problems can be solved (and already suggest that global knowledge performs better than greedy). So, the benchmark contains problems which are not trivial to solve (i.e. greedy does not find a solution), but a solution does exist (i.e. global knowledge finds a solution). Note that when the demands on the resources are increased, the problems get harder to solve. More latency sensitive messages and/or larger messages (more bandwidth) need to be scheduled. Scheduling strategies which are more resource efficient will be able to solve more problems.

The benchmark must also contain multi-scenario scheduling problems in or-



(a) 5x5 mesh with greedy.

(b) 5x5 mesh with global knowledge.

Figure 8.6: Feasible problems in the problem-space.

der to benchmark the multi-scenario scheduling strategy. These problems can be constructed by combining multiple single-scenario scheduling problems into one multi-scenario scheduling problem and selecting a random overlap between them. Scheduling problems from the 5x5 topologies are used for this purpose. Only scheduling problems are used for which all scheduling strategies, including the reference strategy [59] introduced in Section 8.8.1, find a feasible scheduling function. This guarantees that when the multi-scenario scheduling strategy cannot find a feasible scheduling function, this must be due to the resource constraints which result from scenario overlaps. In total, 500 random combinations of two problems are selected with a random overlap value between 5% and 50% of the period of the problems.

8.8 Experimental Evaluation

8.8.1 Single-Scenario Reference Strategies

A state-of-the-art scheduling strategy is presented in [59]. The strategy allows the use of non-shortest routes but it assumes that slots cannot be shared between different streams. Reconfiguration of the NIs is not possible. As in the greedy strategy, this strategy does not reconsider scheduling decisions when a conflict occurs. This strategy is used in the experiments as the **reference strategy**. It is implemented using the greedy strategy with an adapted cost function to sort the routes. In addition, three restrictions are imposed on it. One, messages in one stream must use the same route. Two, streams are not allowed to share slots. Three, the reconfiguration time is equal to a period. This makes it impossible to reconfigure the NIs, i.e. slots cannot be shared between different streams. The cost function that is used to sort the routes computes the ratio between the number of slots that are currently not-used in the links of a route and the total number of slots in the links of the same route. The experimental results suggest that using a

backtracking mechanism is very effective. For this reason, the reference strategy is extended with the ripup mechanism. This strategy is used in the experiments as the **improved reference strategy**.

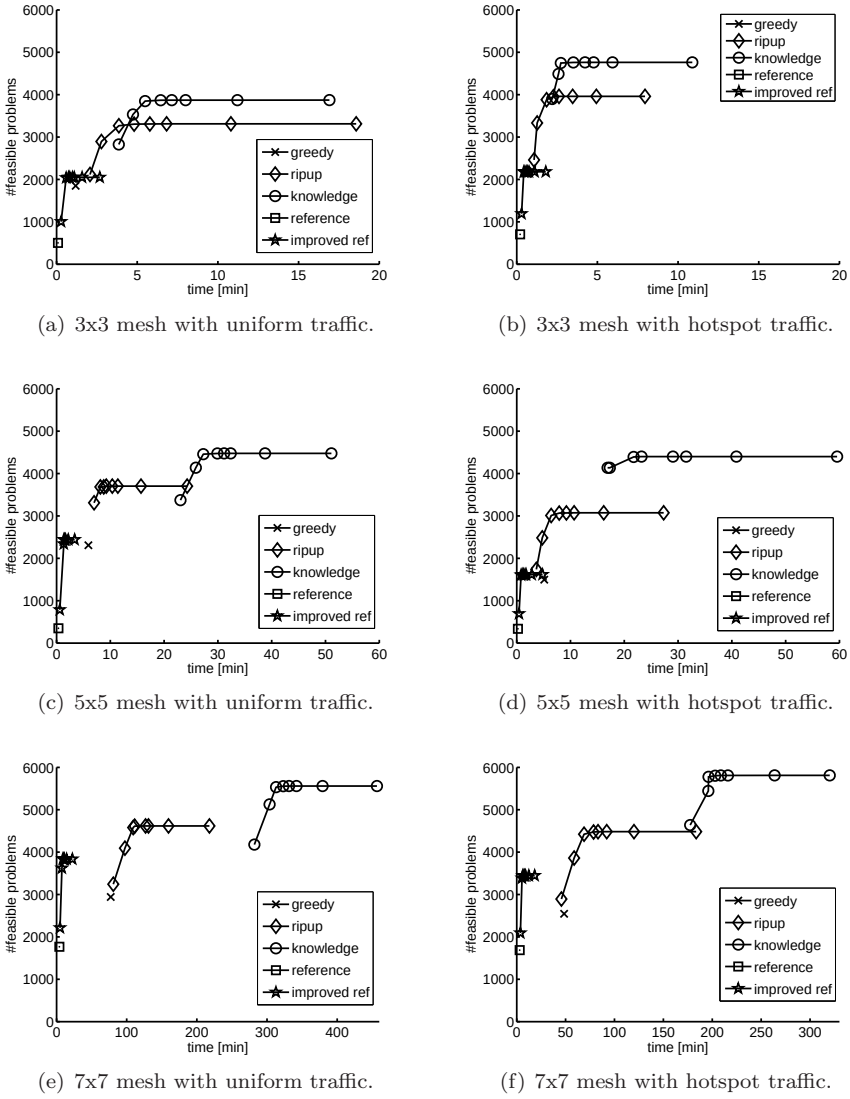
8.8.2 Single-Scenario Scheduling

All single-scenario scheduling strategies have been tested on the benchmark problems. The ripup, global knowledge and improved reference strategies have been tested with a number of different values for the maximum number of ripups (1, 10, 50, 100, 150, 200, 400, 800) to study the trade-off between the number of problems for which a solution is found and the run-time. A slot-table size of 8 slots is used in all experiments and the maximum detour (X) is initially set to 0. Note that $X = 0$ guarantees that any solution uses only shortest routes. This makes it possible to study for how many problems each strategy is able to find a solution with minimal resource requirements. The reconfiguration time of the NI, T_{reconf}, is set to 32 time units. This gives tiles 4 complete rotations of the slot table to reconfigure the NI. A processor or communication assist must send a message to update the routing information in the NI. The size of this message is less than the size of a single flit (i.e. it needs one time unit to be sent), so the value for T_{reconf} is conservative.

The trade-off between the run-time and the number of problems that is solved with the various strategies on the mesh topologies is shown in Figure 8.7. The trade-off curves for the other topologies are similar and therefore omitted. Table 8.2 summarizes the results for all strategies and all topologies assuming that 800 ripups are allowed. The column 'Improvement' shows the percentage of additional problems that is solved by all strategies compared to the reference strategy. The column 'Avg time' gives for each strategy the average run-time on a problem.

Looking at the number of problems solved, the results show that the reference strategy is outperformed by the improved reference strategy. This shows that adding backtracking to the state-of-the-art scheduling algorithm presented in [59] improves the results considerably. The results show further that the reference strategy solves less problems than greedy and the improved reference strategy solves less problems than the other two strategies using ripups. This result leads to the conclusion that not using the ability of NoCs to reconfigure their connections is a limiting factor. As modern NoCs do not have this limitation, problems scheduled using the reference strategies may unnecessarily be considered infeasible or use unnecessarily many resources. Slot sharing is especially advantageous for hotspot traffic (see Table 8.2). For this type of traffic, the strategies presented in Section 8.6 are able to solve up-to 81% more problems than the improved reference strategy. This shows that slot sharing reduces the problem of contention on links connected to a hotspot.

The results in Table 8.2 show that the global knowledge strategy always outperforms the other strategies. However, the average run-time on a problem is larger for this strategy than for the other strategies. This is caused by the con-



(a) 3x3 mesh with uniform traffic.

(b) 3x3 mesh with hotspot traffic.

(c) 5x5 mesh with uniform traffic.

(d) 5x5 mesh with hotspot traffic.

(e) 7x7 mesh with uniform traffic.

(f) 7x7 mesh with hotspot traffic.

Figure 8.7: Trade-off between feasible problems and run-time.

Table 8.2: Results single-scenario problem.

Mesh-topology			
	Improvement	Avg time [ms]	Detour ($X = 2$)
Greedy	118%	178	304%
Ripup	277%	615	365%
Knowledge	371%	1201	420%
Reference	0%	9	5%
Improved ref.	213%	80	263%
Torus-topology			
	Improvement	Avg time [ms]	Detour ($X = 2$)
Greedy	284%	54	434%
Ripup	417%	111	473%
Knowledge	458%	196	487%
Reference	0%	5	1%
Improved ref.	242%	14	255%
Arbitrary topology			
	Improvement	Avg time [ms]	Detour ($X = 2$)
Greedy	58%	66	295%
Ripup	260%	602	454%
Knowledge	449%	905	595%
Reference	0%	6	68%
Improved ref.	148%	167	339%

gestion estimation made at the start of the strategy. Simpler estimates might be used to reduce its run-time. The reference and improved reference strategy have always the lowest run-time. This is logical as route selection is done only once for all messages in a stream and the slot allocation does not have to consider reconfiguration of slots.

Modern NoCs allow the use of flexible routing schemes (i.e. routes may use a detour). More problems may be solved when this flexibility is used. To quantify this gain, all strategies were tested with a maximum detour of 2 on all problems in the benchmark. The results of this experiment are shown in column 'Detour' of Table 8.2, which shows the improvement in the number of problems solved when compared to the reference strategy with detour zero. It shows that using non-shortest routes helps in solving additional problems.

8.8.3 Multi-Scenario Scheduling

The multi-scenario strategy can use all available single-scenario scheduling strategies. For the experiments, the knowledge strategy with 800 ripups is used as this strategy solves the largest number of single-scenario scheduling problems. This indicates that this strategy is the most resource efficient strategy. The state-of-the-art strategy presented in [99] is used as a comparison for the multi-scenario scheduling strategy. This strategy assumes that two scheduling problems cannot share slots when the scheduling problems overlap. In [99], the strategy is used in combination with the reference strategy. The experiments on the single-scenario

scheduling strategies showed that all the strategies presented in Section 8.6 are more resource efficient than this strategy. To exclude the influence of the reference strategy on the results obtained with the strategy from [99], the strategy from [99] is used in combination with the best strategy from Section 8.6. This is the knowledge strategy with 800 ripups.

All multi-scenario problems from the benchmark are scheduled on a 5x5 mesh topology. The multi-scenario scheduling strategy presented in this chapter is able to find a feasible scheduling function for 74% of the problems. The strategy from [99] fails to find a feasible scheduling function for any of the 500 problems. This is due to the fact that the number of slots occupied by each individual scenario is too large to allow a combination of both scenarios in each of the problems on the NoC. To schedule the multi-scenario scheduling problems using the strategy from [99] a NoC with more resources (e.g. larger bandwidth, larger slot tables) are needed. Using the multi-scenario strategy from this chapter, many problems can be scheduled within the available resources. This shows that slot sharing between scenarios can have a large impact on the required resources.

8.8.4 Cost functions

Cost functions are used in the scheduling strategies to sort the messages M and routes R. The cost functions should minimize the chance of having a conflict when scheduling messages. They are constructed in such a way that the most resource constrained messages are handled first and that the resource usage is balanced over all links in the NoC. However, by doing so, they up-front exclude points from the solution-space. To circumvent this problem, randomly ordered sets M and R can be used as an alternative for the cost functions.

To test the impact of the cost functions on the quality of the strategies, the cost functions in the ripup strategy are replaced with a mechanism which assigns random costs to messages and routes. Experiments showed that the number of times this randomized strategy is executed with a fixed number of ripups on a given problem set did not have an influence on the number of problems for which a feasible schedule function is found. However, the number of problems solved within a limited run-time and randomly ordered messages and routes is far lower than the number of problems solved by any of the heuristics in the same time. This shows that the cost functions in the heuristics are effective in ordering the messages and routes.

8.8.5 Scalability

The experiments showed that the run-time of the various strategies increases when the size of the topology increases (see Figure 8.7). This is caused by the fact that the number of links in a route increases and that the number of routes which is considered when a scheduling strategy tries to find a schedule entity for a message increases. Therefore also some experiments are done for a 9x9 mesh.

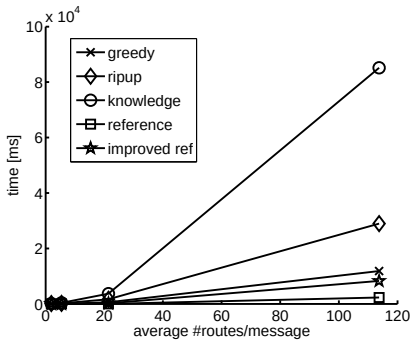
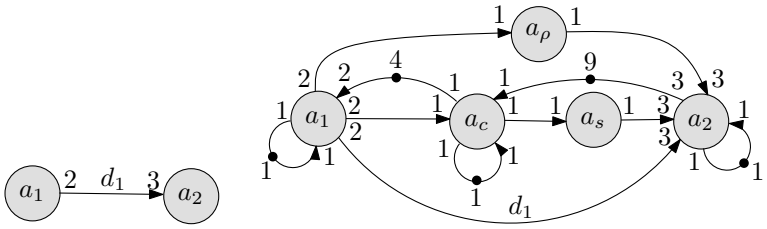


Figure 8.8: Relation between run-time and mesh-size.

When going from a 5x5 mesh to a 9x9 mesh and assuming that each tile has an equal chance of being the source/destination of a stream, the average number of links in a route goes from 3.33 links to 6.00 links and the average number of routes which is considered for a single scheduling entity increases from 5.41 routes to 113.73 routes. So, with increasing mesh-size, a scheduling strategy has to consider potentially more links and more routes when it tries to find a scheduling entity for a message. Figure 8.8 shows the relation between the run-time of the scheduling strategies and the average number of routes which is considered when scheduling a message on a mesh-topology. It shows that the run-time of the strategies increases more than linearly with increasing mesh-size. This could potentially lead to large run-times for the scheduling strategies when the size of the NoC increases. However, in many practical situations, an algorithm that maps tasks to tiles will try to keep the source and destination of a stream close to each other. In other words, the algorithm will try to map an application to a region of the platform and not utilizing resources all across the platform. As a result, the average length of a route and the number of routes which is considered when scheduling a message is in many practical cases not proportional with the topology-size. It grows (much) less rapidly. Furthermore, the experimental results presented in Table 8.2 show that the run-time of all strategies is still within seconds when an application is mapped to a region of 7x7 tiles. So, all scheduling strategies can definitely be used when an application is mapped to a region of this size.

8.9 Extracting Communication Scenarios from SDFGs

The communication task graph model that is used in the NoC routing and scheduling problem (see Section 8.3) has a communication centric view on an application. It assumes that an application can be described by a set of communication scenarios in which every scenario contains a set of messages that is communicated within time bounds between the tasks of the application. The predictable design-



(a) SDFG. (b) Binding-aware SDFG.

Figure 8.9: Example of an SDFG and its binding-aware SDFG.

flow, presented in Section 1.4 and in more detail in Chapter 9, is based on the synchronous dataflow (SDF) model. This section explains the relation between this SDF model and the communication task graph model.

The execution of a (binding-aware) SDFG can be captured in a state-space traversal (see Section 4.3 and Section 6.6). For example, Figure 8.10 shows the state-space of the self-timed execution of the binding-aware SDFG shown in Figure 8.9(b) (of which the details are explained below). An important observation is that the constrained execution consists of a transient phase that is executed once followed by a periodic phase that is executed infinitely often. The main idea is to construct a communication scenario for each of the two phases in the execution of the SDFG. The first scenario captures the communication of all token produced in the transient phase. The second scenario captures the communication of all tokens produced in one period of the periodic phase.

The procedure for extracting the two communication scenarios that describe all token communications taking place during the execution of an SDFG when bound and scheduled to a multi-processor system is illustrated with the example SDFG shown in Figure 8.9(a). As discussed in Chapter 6, the binding and scheduling of an SDFG to a multi-processor system can be described with a binding-aware SDFG. Assume now that the actors a_1 and a_2 of the example SDFG are bound to different tiles in the system. Figure 8.9(b) shows the binding-aware SDFG for the example. The actors a_1 and a_2 in the binding-aware SDFG have respectively an execution time of 4 and 2 time-units. Assume that the latency of the connection from the tile onto which a_1 is bound to the tile onto which a_2 is bound is 1 time unit and that the dependency edge d_1 also has a minimal latency of 1 time unit. The actors a_c and a_ρ, which model respectively the interconnect latency and the dependency edge latency, have therefore both an execution time of 1 time-unit. Assume now that the processors in both tiles have a timewheel with a size of 2 time-units. The complete timewheel has been reserved for the example SDFG on the processor to which a_1 is bound. On the processor in the other tile, only 50% of the timewheel (1 time-unit) has been reserved for the SDFG. Therefore, the execution time of the actor a_s that models the synchronization of the timewheels

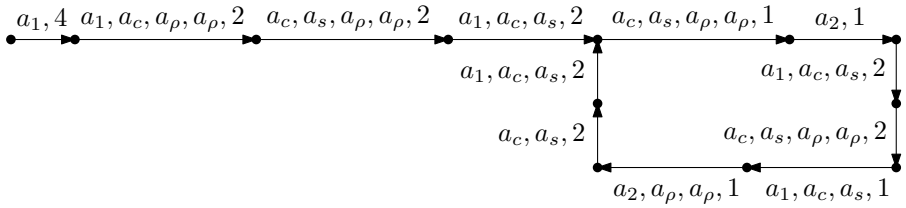


Figure 8.10: Execution of the binding-aware SDFG.

on the two tiles is equal to 1 time-unit. (See Section 6.6.1 for a detailed discussion on the actors a_ρ, a_c and a_s.) Static-order schedules are trivial as both a_1 and a_2 are bound to different tiles.

The actor bindings imply also that the dependency edge d_1 is bound to the interconnect between the tiles. The communication of tokens sent via this edge from a_1 to a_2 must be scheduled on the interconnect. Each token corresponds to a message. In general, a sequence of $\gcd(Rate(SrcP(d_1), DstP(d_1)))$ tokens can be combined into a message without introducing additional timing constraints. This minimizes the packetization overhead without constraining the scheduling problem. For the scheduling problem, time-bounds need to be derived on when the communications can be started and how long these communications may take. These timing constraints can be deduced from the state-space of the constrained self-timed execution of the binding-aware SDFG. The states visited during the constrained self-timed execution of the example binding-aware SDFG are shown in Figure 8.10. As before, states are represented by black dots and state transitions are indicated by edges. The label with a transition indicates which actors start their firing in this transition and the elapsed time till the next state is reached. The execution enters the periodic phase after 10 time-units. Hence, token communications that are started during the first 9 time-units of the SDFG execution belong to the scenario of the transient phase. All other token communications belong to the scenario of the periodic phase. The state-space shows that the periodic phase has a length of 12 time-units. This implies that all token communications that belong to the periodic phase are repeated every 12 time-units.

The state-space also captures the relation between the times at which tokens are produced onto d_1 by actor a_1 and the times at which these tokens are consumed from d_1 by a_2. This relation is shown in Figure 8.11. At time-unit 4, the first firing of a_1 ends with the production of 2 tokens on d_1. These tokens are consumed by the first firing of a_2, which starts at time-unit 11. This firing of a_2 consumes also one token that is produced by the second firing of a_1. The other token produced by the second firing of a_1 is consumed by the next firing of a_2 which starts at time-unit 17. The third firing of a_1 ends at time-unit 12 with the production of two tokens on d_1. The end of this firing occurs after the start of the periodic phase. So, the token communication related to the tokens produced in the first two firings of a_1 belong to the transient scenario. The subsequent firings produce tokens

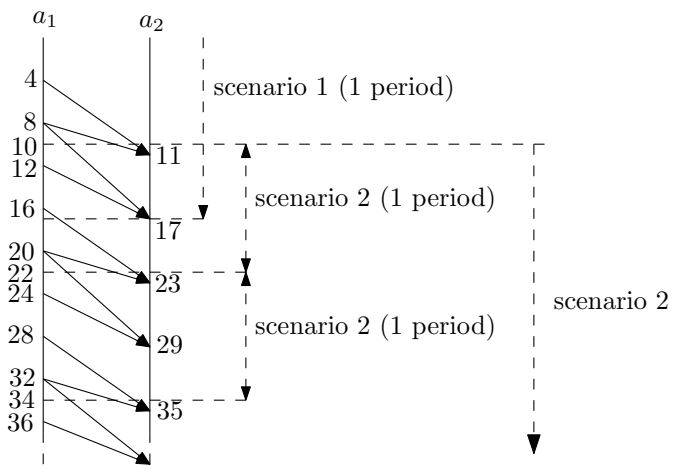
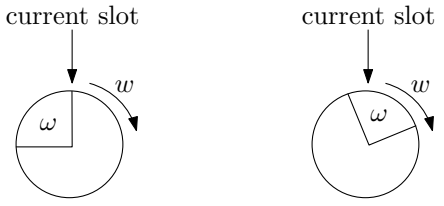


Figure 8.11: Communication on dependency edge d_1.

that result in token communications that take place in the periodic scenario. The tokens produced by the second firing of a_1 are consumed by the second firing of a_2 that starts at time-unit 17. At that moment, all tokens produced by firings of a_1 that belong to the transient scenario are related to firings of a_2. This marks the end of the transient scenario. In the example, it has therefore a length of 17 time-units.

Figure 8.11 shows the time at which tokens are produced on and consumed from d_1. The time of production is the latest possible time in which the firing of actor a_1 will finish on the processor. It can therefore be used as the earliest time at which the communication of the token on the interconnect can start as before this moment in time it cannot be guaranteed that the token is available. The consumption time of a token is the latest possible time at which the involved firing of actor a_2 starts. In an implementation, the firing of a_2 may start earlier due to the relative position of timewheels or the position of the allocated time slices on a timewheel. The worst-case situation, as illustrated in Figure 8.12(a), is that the token which enables the firing of a_2 arrives exactly after the reserved time-slice (ω) has passed. In this situation, the start of the firing is postponed for $w - \omega$ time-units. On the other hand, the token which enables the firing of a_2 can also arrive while the timewheel is within the reserved time-slice. The actor firing can then be started immediately. Recall that in order to guarantee that the throughput analysis is conservative, the self-timed constrained execution exhibits the worst-case allocation of the TDMA time wheel allocations from the perspective of the processors. A token should therefore be available in the receiving tile at least $w - \omega$ time-units before the start of the firing of a_2 as observed in the constrained self-timed execution to guarantee that the firing of a_2 is not delayed when the position of the timewheel is different in an implementation as compared to the



(a) Worst-case. (b) Best-case.

Figure 8.12: TDMA timewheel position.

constrained self-timed execution. Note that the time between the production and consumption of a token sent over the edge d_1 as seen in the constrained self-timed execution is always at least equal to the sum of the execution times of the actors a_c and a_s. The execution time of a_c is larger than zero and the execution time of a_s is equal to $w - \omega$. So, the duration in which a token must be sent over the interconnect is guaranteed to be at least equal to the execution time of a_c.

It is now straightforward to construct the sets of messages that capture the communication behavior of the example SDFG in both the transient and periodic scenario. Every token sent over the edge d_1 forms a message $m = (u, v, s, n, \tau, \delta, sz)$ in the communication task graph. The start-time τ of m is equal to the time at which the corresponding token is produced on the edge. The maximal duration δ of the message is equal to the consumption time of the corresponding token minus τ and minus $w - \omega$. The source vertex u and destination vertex v, stream s and token size sz follow directly from the binding-aware SDFG. The sequence number n of the message follows directly from the state-space. The sets of messages found for the example SDFG are shown in Figure 8.13. The overlap between the two scenarios also follows from the state-space. The state-space shows that the periodic phase is entered at time-unit 10. This marks the start of the periodic scenario. As mentioned before, the transient scenario ends at time-unit 17. So, the two scenarios have an overlap of 8 time-units.

The method described above can be used to extract the communication scenarios and their overlap from the execution of an SDFG. It cannot be guaranteed that the length of the periods and their overlap is a multiple of the slot table size. However, as explained in Section 8.5, the scheduling problem assumes that both the period and overlap are a multiple of the slot table size. In the example that is discussed in this section, a slot-table size of 8 slots is assumed. To guarantee that the overlap between the scenarios and the period of scenarios are a multiple of the slot table size, the length of the transient scenario must be extended such that this requirement is met. First, the end of the transient scenario is simply extended to guarantee that the overlap becomes a multiple of the slot table size. This is not needed for the example as the overlap between the two scenarios is already a multiple of the slot table size. Next, the start of transient scenario is

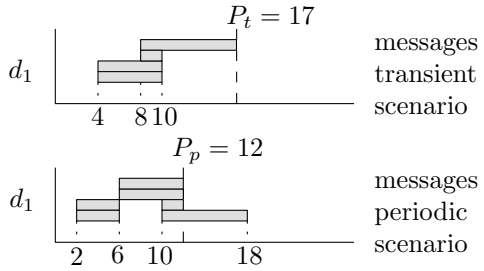


Figure 8.13: Scenarios in the SDFG execution.

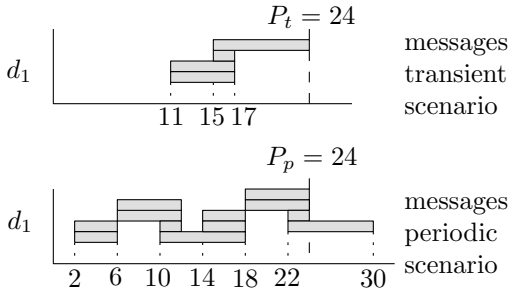


Figure 8.14: Scenarios with extended periods.

shifted such that the total period becomes a multiple of the slot table size. For the example SDFG, this means a shift over 7 time-units. To solve the potential mismatch between the period of the periodic scenario (P_p) and the slot table size (N), the scenario should be concatenated a number of times ($\text{lcm}(P_p, N)/P_p$) till the period of the resulting scenario is equal to $\text{lcm}(P_p, N)$. The resulting scenarios with their periods for the example are shown in Figure 8.14. The overlap between the two scenarios is 8 time-units.

8.10 Summary

This chapter studies the problem of scheduling time-constrained communication of streaming applications on a NoC. Several new strategies are presented to route and schedule streaming communication. The scheduling strategies use all routing and scheduling flexibility offered by modern NoCs while limiting resource usage. Short routes and the reservation of consecutive slots in slot tables minimize resource usage and packetization overhead. However, they also create potential bottlenecks in the NoC, which may render some resources unusable for scheduling other streams. The use of non-minimal routes and non-consecutive slot reservations might increase scheduling freedom for remaining streams. The strategies

presented in this chapter try to find a good compromise in the allocation of routes and slot-table slots. The experiments show that these strategies perform better than the state-of-the-art strategy of [59]. The reason is that the strategies exploit freedom offered by modern NoCs not used in the existing strategy. Additionally, it is shown that adding backtracking to this state-of-the-art strategy improves its results considerably with only a small overhead on its run-time.

This chapter also shows that the dynamism in communication patterns of an application can be captured in a set of time-constrained scheduling problems. Sharing slots from a slot-table between these problems is possible when the timing relations between the problems are taken into account. The experimental results show that this reduces the amount of resources needed to schedule an application onto a NoC outperforming the technique of [99].

Furthermore, a technique is presented to extract a time-constrained scheduling problem for a NoC from an SDFG that is bound and scheduled onto a multi-processor system. This technique can be used in combination with the scheduling strategies presented in this chapter to schedule the communication between actors of an SDFG on a NoC while providing timing guarantees.

Chapter 9

Design Flow

9.1 Overview

The number of applications integrated into new consumer electronics devices is increasing rapidly. At the same time, user expectations on the quality of these devices is ever increasing. To manage the design complexity, a predictable system and design flow are needed that guarantee that an application can perform its own tasks within strict timing deadlines independent of other applications running on the system. This requires that the timing behavior of the hardware, the software, as well as their interaction can be predicted.

This chapter combines the modeling, resource allocation, analysis and scheduling techniques presented in this thesis into a coherent design flow that maps a streaming application onto a NoC-based MP-SoC architecture while offering a predictable timing behavior. The objective is to minimize the resource usage while offering guarantees on the throughput of an application when mapped to the system. The design flow is shown in Figure 9.1 (which is a replication of Figure 1.6, for easy of readability). It takes as an input a throughput-constrained, streaming application that is modeled as an SDFG. This streaming application SDFG is mapped in 4 phases onto a NoC-based MP-SoC architecture. This architecture is a predictable platform as described in Chapter 3. The first phase of the design flow, called **memory dimensioning**, deals with the storage-space that is needed for the tokens communicated over the edges of the SDFG. When a token does not fit into the local memory of a processing tile, it transforms the streaming application SDFG to model accesses to this token when it is stored in a memory tile. The memory dimensioning phase computes also the trade-off space between the storage-space allocated to the edges of the graph and the maximal throughput that can be realized under those allocations. It uses this trade-off space to constrain the storage space of the edges in the application SDFG. The next phase,

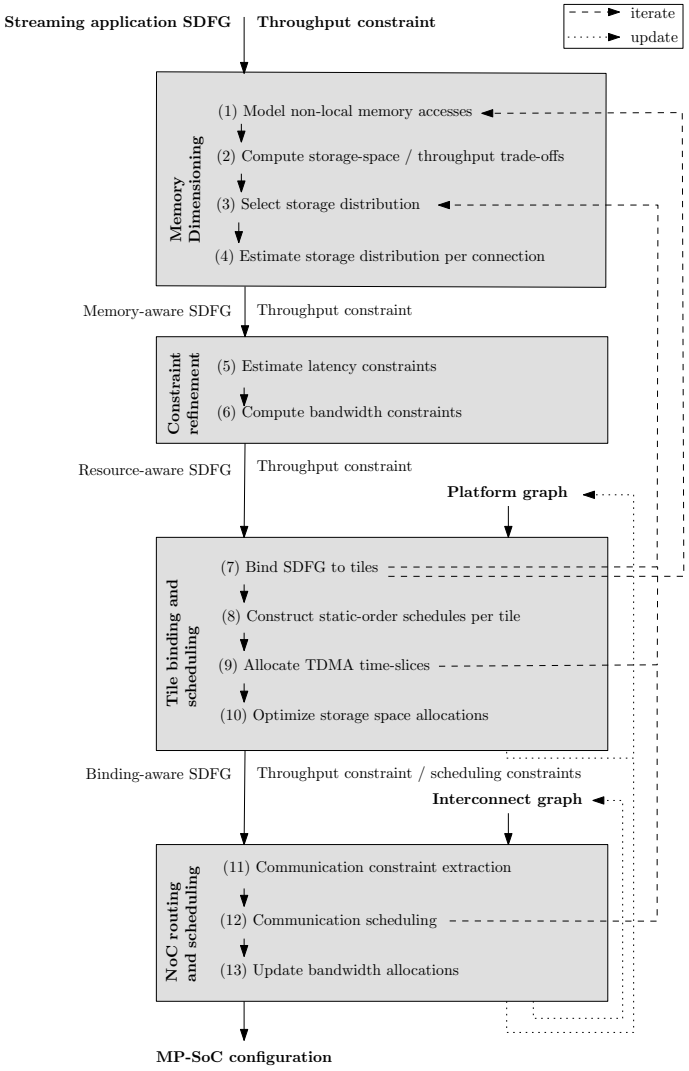


Figure 9.1: SDFG-based MP-SoC design flow.

called **constraint refinement**, uses these storage constraints to compute latency and bandwidth constraints on the edges of the graph. All these constraints are used to steer the binding of actors to the tiles of the MP-SoC architecture. This binding process is performed in the third phase of the design flow. This phase, called **tile binding and scheduling**, constructs also a static-order schedule for the actors of an application that are bound to the same tile. Furthermore, it allocates TDMA time slices on all tiles and it tries to minimize the storage-space allocations. The final phase of the design flow, the **NoC routing and scheduling** phase, deals with the allocation of TDMA slots on the links of the NoC. This starts with the extraction of the NoC scheduling problem from the binding and scheduling decisions made in the previous phase of the flow, after which a solution is searched for the NoC scheduling problem. Finally, the actual bandwidth usage of the NoC schedule is computed. This information is used to update the amount of resources that is available for the next application that should be mapped to the same NoC-based MP-SoC architecture. At various steps of the design flow, the used algorithms may not be able to find a solution which satisfies the resource and/or timing constraints. In those situations, an iteration over (part of) the design flow must be made. The possible iterations are shown with dashed lines in Figure 9.1. Iterations occur due to the lack of resources (step 7) or infeasible timing constraints (step 9 and 12). In those situations, the design flow iterates back to the first or third step of the flow and design decisions made in those steps are revised. When going back to step 1, more or different tokens should be placed in a memory that is accessed over the NoC. Reverting back to step 3 implies that the storage space allocated to the edges is too constrained for meeting the throughput constraint. So, a different storage distribution should be chosen.

The remainder of this chapter is organized as follows. The next section discusses related design flows. The input of the design flow is discussed in Section 9.3. The end result of the flow, a configured MP-SoC, is presented in Section 9.4. The four phases of the flow are discussed in detail in Section 9.5 till Section 9.8. The implementation of the design flow is presented in Section 9.9. The experimental evaluation of the design flow via a non-trivial case study is presented in the next chapter.

9.2 Related Work

Martin [91] and Jerraya [72] have recently signaled the need for the development of an appropriate programming model for MP-SoCs. This programming model must satisfy a number of requirements. First, it must make the concurrency in an application explicit. Second, the model should be easy to implement on an MP-SoC. Third, the model should allow simulation and analysis of its performance during the design process. The flow presented in this chapter satisfies all these requirements.

Benini and De Micheli introduce a system-level design methodology in [20]. They state that a programming model based on message passing is most suitable for programming a NoC-based MP-SoC architecture. This programming model is scalable, simple to implement on an MP-SoC and it fits well with the dataflow languages that are used by application designers. The SDFG model that is used in the design flow presented in this chapter can be implemented with message passing. The methodology presented in [20] focuses on implementation aspects and is as such not a design flow that constructs a mapping and it does not provide performance guarantees.

Besides a programming model, Jerraya signals in [72] also the need for a methodology that explores the design space in a structured way and that maps an application onto an MP-SoC architecture. An overview of existing design space exploration techniques can be found in [55]. Most of the exploration techniques discussed in this paper focus on one (or a few) aspects of the design trajectory (e.g., task binding or memory optimization). These techniques do not offer a complete design flow. The flow presented in this chapter covers a much larger part of the design trajectory as it provides a complete MP-SoC configuration.

Two different types of design flows can be distinguished based on the potential for customization of the MP-SoC architecture. The first type is a **synthesis flow** in which the architecture is constructed during the flow using a library of architecture components. The second type, called **compilation flow**, uses a fixed MP-SoC architecture. The design flow presented in this chapter is an example of a compilation flow. It does not change or optimize the MP-SoC architecture. Instead, it compiles a given application onto a given architecture.

The most important problems in the design of NoC-based systems are summarized in [105]. The paper focuses on the synthesis of a NoC-based MP-SoC. Therefore, not all problems discussed in the paper are related to the compilation flow that is presented in this chapter. Furthermore, the application model used in [105], acyclic task graphs, is more limited than the SDFG model used in the flow presented in this chapter. Nevertheless, in the context of a compilation flow for SDFGs, the flow in this chapter and the techniques presented in this thesis address and solve some of the problems mentioned in [105]. It addresses the problem of determining buffer sizes for NoC communication buffers (Chapter 7), a solution to the NoC communication scheduling problem is proposed (Chapter 8), and the problem of scheduling tasks on a processor is addressed (Chapter 6).

In [98], a flow is presented to synthesize a crossbar interconnect that meets the performance constraints of a set of applications while minimizing the resource usage. This flow does not consider the allocation of resources (e.g., storage space, processors) to tasks nor the scheduling of these tasks on the resources. The flow presented in this chapter does addresses these issues.

An approach to synthesize a NoC-based MP-SoC is presented in [59]. It assumes that a single task is running on a processor and edges between tasks have a latency and bandwidth constraint. The result of the synthesis flow is a system in which the latency and bandwidth constraints for all edges going through the

NoC are satisfied. The flow does not consider resource sharing between multiple applications as opposed to the flow presented in this chapter.

A compilation flow that maps an abstract model of an application onto an abstract model of the MP-SoC architecture is outlined in [90]. The methodology follows the Y-chart approach [79], which is a general approach to map an application to an architecture. After construction of a mapping, performance analysis is performed. If needed, the application, architecture or mapping are changed and the performance analysis is performed again. The methodology presented in [90] suggests to use stochastic automata networks [115] to model the application, architecture and their mapping. The paper shows how communication refinement can be performed on an application which is modeled with a stochastic automata network and that is mapped onto a NoC-based MP-SoC. It also gives a comprehensive overview of existing algorithms and techniques to map and schedule an application onto the architecture. Furthermore, the paper observes the need for system-level design methodologies that allow analysis of the design decisions. The design flow presented in this chapter is an example of such a design methodology.

Hu et al. present in [64] a compilation flow which considers the binding of tasks to processors and the scheduling of their communication on a NoC. The flow assumes that the application is modeled with an acyclic HSDFG. After binding the actors to the processors, a start time is chosen at which actor firings start. From these start times, the NoC scheduling problem is derived and subsequently solved. No additional constraints (e.g., storage-space dimensioning and allocation) are considered in this flow. The flow assumes that sufficient storage space is available to store all data between its production and consumption time. Due to the limitations of the application model, it cannot handle multi-rate behavior, iterative streaming behavior and cyclic dependencies.

The Artemis/Sesame flow [113, 37] is closely related to the flow presented in this chapter. Artemis uses a multi-objective optimization strategy based on an evolutionary algorithm to map an application onto a MP-SoC. Using Compaan [130], a design flow is available that can map an application modeled with a Kahn process network onto a MP-SoC architecture. In contrast with the flow presented in this chapter, their mapping strategy does not consider the dimensioning of storage space (buffer sizes), the scheduling of NoC communication and it does not provide any timing guarantees on the application when executed on the MP-SoC.

Another compilation flow is presented in [95]. This flow is in several aspects similar to the design flow presented in this chapter. The flow maps a throughput-constrained HSDFG to an homogeneous MP-SoC in which multiple actors can share processing resources. The design flow determines latency, bandwidth and storage-space constraints for the edges of the graph. It also allocates TDMA time slices for the actors that share a processing resource. However, the flow does not consider the scheduling of communication on the interconnect. Instead, it uses an abstract model for the interconnect with some ideal properties (e.g., no contention between data sent between different tiles), similar to the abstraction

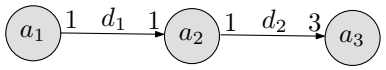


Figure 9.2: Example streaming application SDFG.

Table 9.1: Properties of the streaming application SDFG of Figure 9.2.

	$p_1(\tau,\mu)$	$p_2(\tau,\mu)$		sz	λ
a_1	$(5, 200)$	$(20, 200)$	d_1	128	0.0085
a_2	$(5, 350)$	(∞, ∞)	d_2	64	
a_3	(∞, ∞)	$(30, 100)$			

made in Chapter 6 of this thesis. The design flow presented in this chapter can deal with arbitrary SDFGs and heterogeneous MP-SoCs and it considers the scheduling of communication onto a NoC interconnect.

9.3 Starting Point

The design flow shown in Figure 9.1 starts with a throughput-constrained, streaming application and a NoC-based MP-SoC onto which the application must be mapped. This section describes the information that must be available about the application and the MP-SoC architecture at the start of the design flow.

A streaming application must be modeled with an SDFG. Annotations are needed to the actors and edges of the graph that specify properties like execution time or token size. Formally, a throughput-constrained streaming application SDFG serving as input to the design flow is defined as follows. Let PT be the set of processor types considered in the flow.

Definition 38. (Streaming application SDFG) *A streaming application SDFG $(A, D, \Gamma, \Theta, \lambda)$ is a 5-tuple consisting of an SDFG (A, D), the functions $\Gamma : A \times PT \rightarrow \mathbb{N}^{\infty} \times \mathbb{N}_0^{\infty}$ and $\Theta : D \rightarrow \mathbb{N}_0$, and the throughput constraint $\lambda \in \mathbb{R}$. Function Γ gives for each actor $a \in A$ and each processor type $pt \in PT$ a tuple (τ, μ) with τ and μ respectively the execution time (in time units) and memory requirement (in bits) of actor a when assigned to a processor of type pt or ∞ if actor a cannot be assigned to a processor of type pt. Function Θ gives for each dependency edge $d \in D$ the size sz of a token (in bits).*

Figure 9.2 shows an example of a streaming application SDFG. The requirements for the actors and edges are shown in Table 9.1. This table shows, for example, that actor a_1 can be bound to a processor of type p_1 and to a processor of type p_2. Actors a_2 and a_3 on the other hand can only be bound to one processor type. These properties must be provided by the application designer. They can

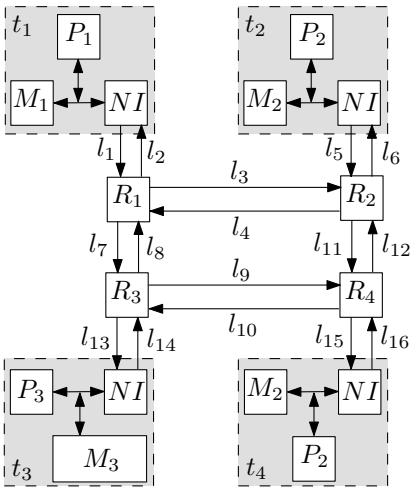


Figure 9.3: Example NoC-based MP-SoC architecture.

be derived using the code analysis techniques discussed in Chapter 2.

The design flow needs as input, besides the streaming application SDFG, a NoC-based MP-SoC architecture. An example of such an architecture is shown in Figure 9.3. The properties of the resources inside the tiles are shown in Table 9.2. The architecture consists of three processing tiles (t_1, t_2 and t_4) and one memory tile (t_3) that are interconnected via a NoC with four routers. For each tile, Table 9.2 specifies the processor type (pt), the TDMA time wheel size (ω, in time-units), the amount of memory (m, in bits), the number of supported connections (c), the input bandwidth (i, in bits/time-unit) and the output bandwidth (o, in bits/time-unit). The properties of the NoC are shown in Table 9.3. It shows that each link has a slot-table with 8 slots (N) and that 96 bits (sz_{flit}) can be transferred over a link in one time-unit (i.e. the flit clock of the NoC is equal to one time-unit), the size of a packet header (sz_{ph}) is 32 bits, and that the NI reconfiguration time is 32 time-units. This reconfiguration time specifies the time that is needed to update the routing information for a slot in the slot table of a NI (see Section 8.4).

The NoC-based MP-SoC architecture is used in the tile binding and scheduling phase of the design flow and in the NoC routing and scheduling phase. Both phases use a different abstraction of the architecture. The tile binding and scheduling phase abstracts from the links and routers in the interconnect. It uses a platform graph (see Definition 13) that assumes point-to-point connections with a fixed latency between the tiles in the platform. The NoC routing and scheduling phase on the other hand abstracts from the resources inside a tile. It only considers the links and routers in the NoC and their connections to the tiles. The structure

Table 9.2: Properties of the tiles in the architecture of Figure 9.3.

	pt	w	m	c	i	o
t_1	p_1	100	1000	10	96	96
t_2	p_2	100	500	10	96	96
t_3	p_3	100	1000	10	96	96
t_4	p_2	100	500	10	96	96

Table 9.3: Properties of the NoC in the architecture of Figure 9.3.

N	sz_{flit}	sz_{ph}	T_{reconf}
8	96	32	32

and properties of the NoC resources are described with an interconnect graph (see Definition 33).

As mentioned, the platform graph abstracts from the NoC by using the notion of point-to-point connections between tiles. A system designer must specify the available connections and their latency. A connection from a tile t_i to a tile t_j can only exist when there exists a route in the NoC from t_i to t_j. As a first estimate, the latency can be set equal to the number of links in the shortest route from t_i to t_j. Other properties of the links on the route (e.g. congestion) can also be taken into account. A designer can also decide not to specify a connection. This avoids that the source and destination actor of some dependency edge in the streaming application SDFG are bound to respectively t_i and t_j. In this way, a designer can partition the architecture into smaller sub-parts of which only one is used in the tile binding and scheduling phase.

9.4 MP-SoC Configuration

The end result of the NoC-based MP-SoC design flow is a streaming application SDFG that is bound and scheduled onto the resources of the NoC-based MP-SoC architecture. Formally, the MP-SoC configuration consists of a feasible resource allocation as discussed in Section 6.5 and a feasible NoC schedule as discussed in Section 8.5.

Figure 9.4 shows the binding and scheduling of the example application shown in Figure 9.2 onto the architecture of Figure 9.3 as determined by the design flow. In this case, the actors a_1 and a_2 are bound to tile t_1. Therefore, also edge d_1 is bound to t_1. Actor a_3 has been bound to tile t_2. This implies that the edge d_2 goes from tile t_1 through the NoC to tile t_2. Storage-space for 1 token of d_1 and 2 tokens of d_2 has been allocated on t_1. The actors a_1 and a_2 are executed on t_1 using the static-order schedule $(a_1 a_2)^*$. To separate the execution of this application from other applications, 75% of the TDMA time-wheel has been reserved on tile t_1. Similar allocations are made for edge d_2 and actor a_3

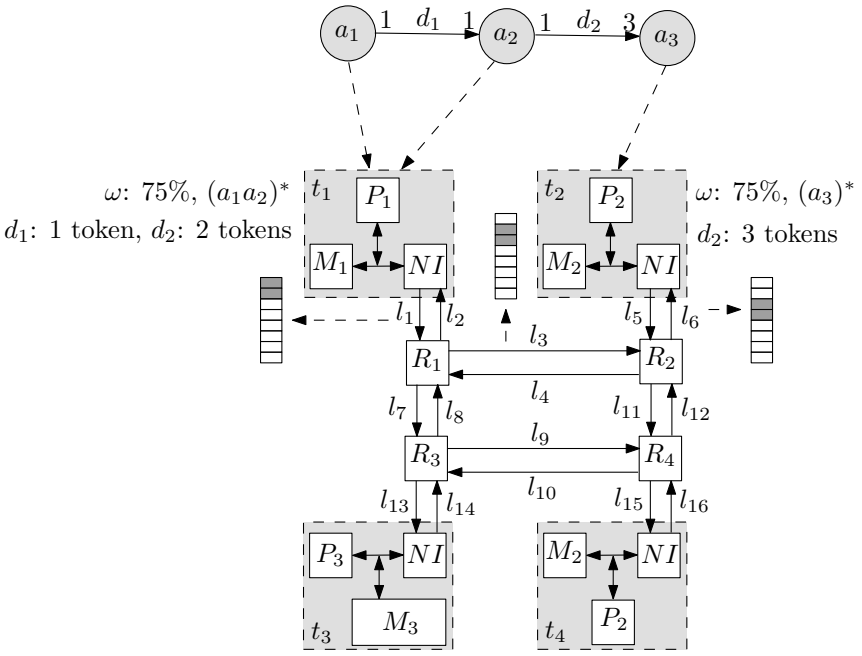


Figure 9.4: MP-SoC configuration.

on tile t_2. The MP-SoC configuration shown in Figure 9.4 specifies further the allocated slots on the links l_1, l_3 and l_6 that are used to send the tokens from actor a_2 through d_2 to a_3. Using this resource allocation, the throughput constraint of the application, λ (see Table 9.1), is met. The streaming application SDFG can realize under the given resource allocation and schedule a throughput of 0.0088 iterations/time-unit, which is above the throughput constraint of the application.

9.5 Memory Dimensioning

This section discusses the details of the memory dimensioning phase of the design flow. This is the first phase of the flow shown in Figure 9.1. The result of this phase is a memory-aware SDFG which is formally defined as follows. Compared to the streaming application SDFG (see Definition 38), the memory-aware SDFG contains additional storage constraints for the edges of the graph, redefining function Θ. These constraints are computed in the memory dimensioning phase.

Definition 39. (MEMORY-AWARE SDFG) *A memory-aware SDFG* $(A, D, \Gamma, \Theta, \lambda)$ *is a 5-tuple consisting of an SDFG* (A, D)*, the functions* $\Gamma : A \times PT \rightarrow \mathbb{N}^{\infty} \times \mathbb{N}_0^{\infty}$ *and* $\Theta : D \rightarrow \mathbb{N}_0^4$*, and the throughput constraint* $\lambda \in \mathbb{R}$*. Function* Γ *is defined as in Definition 38. Function* Θ *gives for each dependency edge* $d \in D$ *from an*

actor $a_i = SrcA(d)$ to an actor $a_j = DstA(d)$ a 4-tuple $(sz, \alpha_{tile}, \alpha_{src}, \alpha_{dst})$ with sz the size of a token (in bits), α_{tile} the memory (in tokens) required when a_i and a_j are assigned to a single tile, α_{src} and α_{dst} the memory (in tokens) required in the source and destination tile when a_i and a_j are assigned to different tiles.

The sets of actors and dependencies of the memory-aware SDFG may in fact differ from those sets in the application SDFG. This occurs if some of the data tokens need to be stored in non-local memories.

9.5.1 Modeling Non-Local Memory Accesses

The first step in the memory dimensioning phase, the model non-local memory accesses step, is responsible for dealing with tokens that are too large to fit into the memory of the processing tile to which the producing or consuming actor is bound. These tokens must be stored in a memory tile. The decision to store tokens in a memory tile must, for two reasons, be modeled into the streaming application SDFG. First, the modeling of this design decision makes the resource requirements explicit. Second, the transformed SDFG allows reasoning about the timing behavior when tokens in a memory tile are accessed by an actor which executes on a processing tile. An SDF model to make non-local memory accesses explicit is presented in Chapter 5. This model allows timing analysis and it also captures the resource requirements for storing tokens in a memory tile.

A system-designer must decide which tokens are stored in a memory tile. Often, multimedia applications use only a limited number of large data objects. These applications focus on streaming data and try to avoid producing large amounts of data in bursts. So, the number of edges that communicate tokens which are too large to be stored in the memory of a processing tile is limited. The streaming application SDFG specifies the size of the token communicated over the dependency edges. The designer can use this information to decide which tokens should be stored in a memory tile. Alternatively, a designer could initially decide to assume that all tokens fit into the memories of the processing tiles and continue with the next step of the design flow. This decision can always be revised when during a later stage of the design flow no binding of the graph to the architecture can be found due to insufficient memory resources in the processing tiles. At that moment, it will be clear which tokens cannot be stored in a processing tile. The designer must then return to the memory modeling step and model the design decision to store these tokens in a memory tile into the streaming application SDFG. If the system designer decides to map a token to a remote memory tile, he must also provide appropriate execution times, state sizes and token sizes for the actors and edges of the memory access model that is used to refine the streaming application SDFG. Currently, the decision to map tokens to non-local memories and the analysis of resource requirements resulting from such a mapping is not yet supported by automated tools in the flow. Providing tool support for this step is an interesting topic for future research.

Consider now the streaming application SDFG shown in Figure 9.2 and the targeted NoC-based architecture shown in Figure 9.3. The memories inside the processing tiles are large compared to the token sizes. Therefore it does not seem logical to store any of the tokens in a memory tile. So, it is not needed to model any remote memory accesses in the streaming application SDFG that is the input to the flow.

9.5.2 Compute Trade-Offs between Storage-Space and Throughput

After modeling the non-local memory accesses, the design flow continues with the second step of the flow. In this step, a trade-off is computed between the storage space allocated to the dependency edges of the SDFG and the maximal throughput that can be realized within these storage bounds. This trade-off space can be used as an indication of the storage-space that must be allocated to the dependency edges to realize a certain throughput.

Chapter 7 presents techniques to find the trade-offs between the storage-space distributions of an SDFG and the maximal throughput realized within this storage space. This trade-off space can be computed using the storage-space/throughput trade-off algorithm presented in Section 7.5 or the approximation technique of Section 7.7. When using the exact technique, these storage-space distributions are proved to be the minimal storage distributions needed to realize the associated throughputs.

The trade-off analysis of Chapter 7 allows auto-concurrency. This enables multiple copies of an actor to fire simultaneously. In an actual system, only one instance of an actor can be executed at the same time on a processor. To take this into account, the second step of the design flow adds a self-edge with one token to every actor in the memory-aware SDFG before computing the storage-space/throughput trade-off space. This step of the design flow assigns to every actor its minimal execution time on any processor type before executing the storage-space/throughput algorithm. The trade-off space contains in this case the trade-offs between the maximal throughput that can be realized within a given storage-space for any possible mapping. In other words, the trade-off space gives the trade-off between the storage space and the highest throughput that can be realized in the design flow. These trade-offs can be used in the remainder of the flow to minimize the amount of memory allocated to the dependency edges of the SDFG.

The trade-off space for the storage distributions and their maximal throughput of the example application is shown in Figure 9.5. The trade-off space shows that the maximal throughput of the application is 0.033 iterations/time-unit. This is an upper-bound on the throughput that can be realized when actors have to share processors and communication introduces a delay. The throughput constraint of the example application is below this maximal throughput. This indicates that it might be possible to find a mapping that satisfies the throughput constraint. When the throughput constraint exceeds the maximal throughput, it is certain

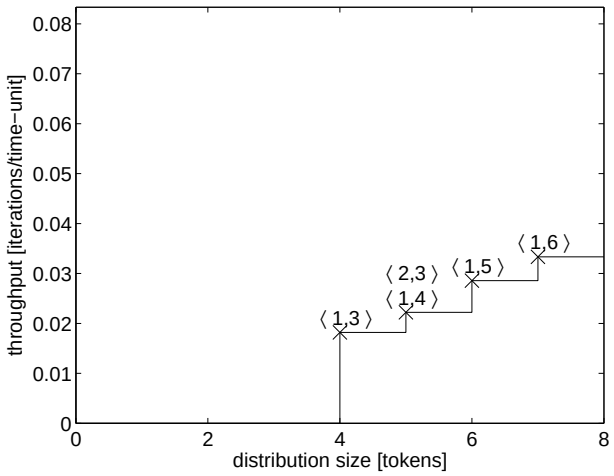


Figure 9.5: Pareto space for the example streaming application SDFG.

that no mapping exists that satisfies the timing constraints.

9.5.3 Select Storage Space Requirements Edges

The previous step of the design flow computed the minimal storage distributions of the streaming application SDFG. In this step of the design flow, one of these minimal storage distribution is selected. This storage distribution is used to constrain the storage space allocated to the dependency edges. This decision will also impact the latency and bandwidth constraints of these edges as computed in the next phase of the flow.

The smallest storage distribution with a non-zero throughput gives the storage-space that must be allocated to the dependency edges to allow a deadlock-free execution. So, it gives for every dependency edge the smallest storage-space needed to allow sequential, deadlock-free execution of its source and destination actor. A sequential execution occurs when the source and destination actor of a dependency edge $d \in D$ are bound to the same tile. To minimize the memory usage, the storage-space constraint α_{tile} for d, specified in the memory-aware SDFG, is chosen equal to the storage-space allocated to d in the smallest minimal storage distribution allowing a positive throughput.

The source and destination actor of an edge $d \in D$ can fire concurrently when they are bound to different tiles. When more storage-space is available for d than the minimum needed to guarantee a deadlock-free execution, the actors may be able to exploit this concurrency. This can lead to an increase in throughput. The optimal trade-offs between allocated storage space and throughput have been

computed in the previous step of the design flow. This step of the design flow selects one of these storage distributions, to constrain the storage space of the edges in the graph for the remainder of the flow. I.e., a storage distribution must be chosen to determine the α_{src} and α_{dst} constraints for each edge in the memory-aware SDFG. Initially, the smallest storage distribution δ_s is selected which has a throughput $\lambda_s \geq \lambda$. When the throughput constraint, λ, of the streaming application SDFG cannot be met in a subsequent step of the flow, the storage space requirements must be increased. In that case, the next minimal storage distribution from the trade-off space should be used to constrain the storage space of the edges. This procedure of enlarging the storage space of the dependency edges should be repeated till either a storage space distribution is used with which the complete flow can be completed or till all storage space distributions are tried without success. In the latter case, the design flow is not able to map the application to the platform while meeting the throughput constraint.

Consider the trade-off space shown in Figure 9.5 for the streaming application SDFG of Figure 9.2. The smallest storage distribution $\langle 1, 3 \rangle$ satisfies the throughput constraint. Therefore, this storage distribution is initially selected by the design flow to be used in the subsequent steps of the flow. This storage distribution is also used to constrain the storage space of the dependency edges d_1 and d_2 when their source and destination actors are bound to the same tile.

9.5.4 Estimate Storage Distribution per Connection

A storage distribution δ_s was selected in the previous step of the design flow. This step of the design flow uses δ_s to constrain the storage space allocated for a dependency edge $d \in D$ when its source and destination actor are bound to different tiles. The storage space of d is used to store data produced by an actor firing before it is consumed by another actor firing. Sufficient storage space should be allocated to d in both of the tiles to store all tokens produced or consumed during one firing in the memory of the tile. So, the storage space α_{src} allocated on the source tile should offer at least sufficient room for $Rate(SrcP(d))$ tokens. Similarly, the allocated storage space α_{dst} in the destination tile should be at least $Rate(DstP(d))$ tokens. Availability of at least these amounts of storage space guarantees that during the firing of an actor, none of the tokens produced or consumed by the firing have to be communicated through the interconnect. This enables analysis of the worst-case execution time of an actor by only considering the processor and the arbitration mechanism of the memory in the tile to which the actor is bound. The storage distribution δ_s may offer more storage space to d than the number of tokens needed for one firing of the source and destination actor. In the heuristic used in the flow, the choice is made to distribute the storage space evenly over the source and destination tile. This gives both the source and destination actor some flexibility to fire without having to wait before tokens are communicated over the interconnect. Dependency edge d may contain n initial tokens. It is assumed that these tokens are initially stored in the memory of the

Table 9.4: Properties of the example application for storage distribution $\langle 1, 3 \rangle$.

	$p_1(\tau, \mu)$	$p_2(\tau, \mu)$		sz	α_{tile}	α_{src}	α_{dst}	ρ	β
a_1	$(5, 200)$	$(20, 200)$	d_1	128	1	1	1	13	6.98
a_2	$(5, 350)$	(∞, ∞)	d_2	64	3	2	3	61	3.49
a_3	(∞, ∞)	$(30, 100)$							

tile to which the source actor of d is bound. This assumption models the worst-case situation in which the data needed to fire the destination actor is initially available, but must still be communicated over the interconnect. The storage space in the source tile should thus be large enough to keep all initial tokens. Based on the assumptions mentioned above, the storage space constraints for an edge d in the source and destination tile are given by:

$$\alpha_{src}(d) = \max(\left\lceil \frac{\delta_s(d)}{2} \right\rceil, Rate(SrcP(d)), n) \tag{9.1}$$

$$\alpha_{dst}(d) = \max(\left\lceil \frac{\delta_s(d)}{2} \right\rceil, Rate(DstP(d))) \tag{9.2}$$

In the previous step of the design flow, the storage distribution $\langle 1, 3 \rangle$ was selected to constrain the storage space of the dependency edges d_1 and d_2 when the source and destination actors of the example application are bound to different tiles. The storage space constraints for d_1 and d_2, as computed using Equation 9.1 and Equation 9.2, are shown in Table 9.4. This table shows also the α_{tile} constraint derived in the previous step and the latency (ρ) and bandwidth (β) constraints of the edges as computed in the next phase of the design flow.

9.6 Constraint Refinement

The tile binding and scheduling phase uses a resource-aware SDFG (see Definition 14). Compared to the memory-aware SDFG, it contains additional latency and bandwidth constraints for the dependency edges of the graph. The second phase of the design flow computes these constraints.

9.6.1 Estimate Latency Constraints

When analyzing the throughput of an SDFG, it is assumed that the transfer of a token over a dependency edge takes no time. In a NoC-based MP-SoC, a dependency edge may be bound to the NoC. In this case, some time is needed to transfer a token from the sending to the receiving tile. This is because the NoC over which the token is sent has a latency and a finite bandwidth. The token may further be delayed because the actor which consumes it has to wait till the TDMA time wheel reaches the time slice of the application. The resource-aware

SDFG specifies a minimal latency ρ for every edge d in the SDFG. This minimal latency specifies the minimal time between the production and consumption of a token on d when the source and destination actor are bound to different tiles. In other words, it sets the minimal amount of time that can be used to sent a token between two actors. This provides freedom to the scheduling problems solved in the design flow.

The minimal latency ρ is used in the construction of a binding-aware SDFG (see Section 6.6.1). Whenever an edge d from an actor a_i to an actor a_j is bound to the interconnect, this edge is replaced with the SDF model shown in Figure 6.4(c). The execution time of the actor a_ρ is equal to the minimal latency ρ of the edge. It is important to note that this latency is in general not equal to the latency of a connection in the interconnect. The latter is included in the execution time of the actor a_c. Actor a_c includes further the delay of tokens due to the finite bandwidth allocation for the edge d on the interconnect.

The latency assigned to edges of a resource-aware SDFG have an impact on the throughput of the application when mapped to the NoC-based MP-SoC. The reason for this is that all actors in the graph, due to the storage space allocations, are part of the same strongly connected component. The latency ρ on an edge may influence future firings of the edge's source actor. So, the throughput of the graph is related to the latency assigned to the edges. The latency assigned to an edge should therefore never be so large that it decreases the throughput of the application below its throughput constraint λ. In fact, to compute the latency constraint for the edges in the resource-aware SDFG, the more strict requirement is used that the assigned latencies should not reduce the throughput of the graph below the throughput λ_s of the minimal storage distribution selected in the third step of the flow. This couples the allocated storage space to the scheduling problems solved in the flow; increasing memory usage, which allows higher throughput, leads to tighter latency constraints. This makes it simpler for the tile binding and scheduling problem (3rd phase of the flow) to realize the throughput requirement. More storage space is available for buffering tokens and the communication latency, which affects the throughput, is smaller. However, it makes the NoC routing and scheduling (4th phase of the flow) more complex as the same amount of tokens has to be scheduled in less time.

To avoid an extensive design-space exploration involving many throughput calculations on the state-space of the SDFG (which may change with every latency assignment), an estimator for the throughput of an SDFG is used in the computation of the latency constraints. The throughput of an SDFG is known to be limited by its critical cycle [128]. This is a cycle in the corresponding HSDFG with the maximal cycle mean. The **cycle mean** is defined as the ratio between the execution time of the actors on the cycle and the number of tokens on the edges of the cycle. The throughput of an SDFG is equal to the inverse of the maximal cycle mean. The conversion of an SDFG to an HSDFG can lead to an exponential increase in the number of actors in the graph [46], which makes it infeasible to analyze the cycle mean of the cycles in the HSDFG. Therefore, as a

basis for the throughput estimator, an estimate of the cycle mean of every cycle in the SDFG is used. Following the definition of the cycle mean for an HSDFG, the cycle mean of a cycle c through a sequence of actors in a memory-aware SDFG is estimated by the following equation:

$$\mathrm{CM}(c) = \frac{\sum_{a \in c} \min_{pt \in PT} \tau(\Gamma(a, pt)) \cdot \gamma(a)}{\sum_{edges\ d=(u,v) \in c} n/Rate(v)}, \tag{9.3}$$

with γ the repetition vector of the SDFG and n the number of tokens on edge d in the initial state of the SDFG. The equation assumes the best-case binding of the actors on the cycle to the processor types. This gives a lower-bound on the time needed to execute all actors on the cycle when bound to the tiles. The tokens on the edges are normalized with respect to the rate at which they are consumed from these edges. The sum of these normalized tokens gives a notion of the number of iterations of the cycle that can fire concurrently. Equation 9.3 considers only simple cycles in the SDFG. It neglects that combinations of simple cycles in the SDFG can form a simple cycle in the corresponding HSDFG. Such a simple cycle in the HSDFG could be the critical cycle of the graph. Therefore, the throughput of the SDFG can be lower than suggested by the maximum cycle mean found when using Equation 9.3.

Before computing the latency constraints for the dependency edges in the memory-aware SDFG, the storage constraints that are computed in the previous step of the flow are also modeled into the SDFG. For every edge d in the memory-aware SDFG, an edge d_δ is added in the opposite direction to model the storage space of d (see also Section 7.3). The number of tokens on d_δ is equal to the sum of the storage space in the source and destination tile when the source and destination actor d would be bound to different tiles (i.e., it is equal to $\alpha_{src}+\alpha_{dst}$). Using this graph with storage constraints, the latency constraints for the edges in the memory-aware SDFG are determined.

The latencies assigned to the edges of a cycle c increase the cycle mean of c. The larger the latency assigned to the edges, the larger the freedom is for the scheduling problems solved in the design flow. However, when the latencies are too large, the throughput constraint can be violated. The latencies are assigned to all dependency edges such that the cycle mean of every cycle as estimated by Equation 9.3 becomes as large as possible, but never goes above $1/\lambda_s$. The latency ρ of an edge $d \in D$ that is part of all the cycles in set C is then given by Equation 9.4.

$$\rho = \min_{c \in C} \left\lfloor \frac{\left(\frac{1}{\mathrm{CM}(c) \cdot \lambda_s} - 1\right) \cdot \sum_{a \in c} \min_{pt \in PT} \tau(\Gamma(a, pt)) \cdot \gamma(a)}{\#actors\ in\ c} \right\rfloor \tag{9.4}$$

The equation computes for every cycle $c \in C$ of which d is a part the latency that can be added to the cycle mean of c such that it becomes equal to $1/\lambda_s$. This latency is divided evenly over the edges of the cycle. The latency ρ of d is equal to the minimum of the latencies computed for all cycles C to which it belongs.

Consider again the example application shown in Figure 9.2. The latency of the edges in this application, as computed using Equation 9.4, are listed in Table 9.4. To compute these constraints, the storage constraints α_{src} and α_{dst} that were computed in the previous step of the flow are used.

9.6.2 Compute Bandwidth Constraints

The source and destination actor of a dependency edge d can be bound to different tiles. Tokens communicated via d must then be transferred over the interconnect. The design flow must schedule the communication of these tokens on the interconnect. This is done in the NoC routing and scheduling phase of the design flow. Before doing so, actors are bound to tiles in the tile binding and scheduling phase of the flow. The tile binding determines whether a dependency edge uses the interconnect. To steer the binding decisions, the tile binding phase uses a notion on the amount of data that must be communicated per time-unit through a dependency edge d. The required bandwidth β for an edge d with token size sz is estimated by the design flow using the following equation:

$$\beta = Rate(SrcP(d)) \cdot \gamma(SrcA(d)) \cdot sz \cdot \lambda_s \tag{9.5}$$

The first three terms in the equation give the total amount of data (in bits) that must be sent in one iteration of the graph through the network. The throughput λ_s specifies the average number of iterations of the graph that are completed per time-unit, if the throughput of the storage distribution selected in step 3 of the flow is realized. The product of these two parts gives the average number of bits that must be sent per time-unit through the interconnect.

The estimated bandwidth requirement β is based on the assumption that the actors produce data at a constant rate. This is valid for streaming applications in which no bursts occur. Bursts may occur in practice; these should be dealt with when the communication is actually scheduled on the interconnect (step 12 of the flow). In step 13 of the flow, the actual bandwidth usage is observed and used to constraint the bandwidth that is available for other application that use the MP-SoC architecture simultaneously.

Table 9.4 shows the bandwidth constraints for the dependency edges of the example application shown in Figure 9.2. These bandwidth constraints are computed for the storage distribution $\langle 1, 3 \rangle$ which has a throughput of 0.018 iterations per time-unit. At the end of this step, all constraints required to transform the memory-aware SDFG (Figure 9.2 and Table 9.1) to a resource-aware SDFG are computed. The resulting resource-aware SDFG is shown in Figure 9.2 and Table 9.4. For this example, the structure of the resource-aware SDFG is the same as

the structure of the application SDFG. Only if tokens are mapped to non-local memories, the structure is different.

9.7 Tile Binding and Scheduling

The previous phase of the design flow derived constraints on the edges of the memory-aware SDFG. These constraints are used in the tile binding and scheduling phase of the design flow to steer the binding of the resource-aware SDFG to the NoC-based MP-SoC architecture. The first three steps of this phase are identical to the resource allocation strategy presented in Chapter 6. This strategy starts with binding actors to the tiles in the architecture (step 7 of the flow). The resources from the MP-SoC architecture that can be used in this binding process are described by the platform graph (see Definition 13). After the tile binding, a static-order schedule is constructed to order the execution of actor from the same application on a processor (step 8). Step 9 allocates TDMA time slices on the processors to allow multiple applications to share these processors. The result of the resource allocation strategy is a binding-aware SDFG. It models all binding and scheduling decisions made till this point of the design flow.

It is not guaranteed that the storage space allocations for the edges bound to the interconnect are optimal after the binding and scheduling decisions made by the resource allocation strategy. It might be possible to assign a smaller storage space to one or more edges and still realize the same throughput. This is due to the fact that the resource allocation strategy uses estimates for the storage space needed in the source and destination tile of an edge (α_{src} and α_{dst}) that is bound to the interconnect. The minimal storage space distributions for a binding-aware SDFG can be found using the method presented in Section 7.8.

This algorithm computes all optimal trade-offs between the storage space of the edges and the throughput of the graph under the given binding and scheduling constraints. When used in the design flow, the method considers only different storage space allocations for edges whose source and destination actor are bound to different tiles. These edges are modeled with the SDFG shown in Figure 6.4(c). Only changes to the number of initial tokens α_{src} and α_{dst} are considered, as these model the storage space allocated in the source and destination tile of edges that are bound to the interconnect. Furthermore, the method, when used in the design flow, is not allowed to increase α_{src} and α_{dst} above the values assigned to them in the 4th step of the flow. This guarantees that the minimal storage distributions that are found by the method always fit within the storage space allocated by the resource allocation strategy.

Consider again the resource-aware SDFG shown in Figure 9.2 and Table 9.4 and the NoC-based MP-SoC architecture shown in Figure 9.3. The tile binding and scheduling phase binds the actors a_1 and a_2 to the tile t_1 and actor a_3 to tile t_2. However, it fails to find a TDMA time slice allocation which satisfies the throughput constraint. Using 100% of the timewheels on t_1 and t_2, it reaches

a throughput of 0.0084 iterations/time-unit which is below the throughput constraint of 0.0085 iterations/time-unit. As a result, the design flow returns to step 3 of the design flow and selects the next minimal storage distribution from the trade-off space shown in Figure 9.5. Using the storage distribution $\langle 2, 3 \rangle$, a new resource-aware SDFG is constructed and bound to the MP-SoC architecture. The actor to tile binding is identical to the binding found with the previous storage distribution. The throughput constraint is then satisfied when 75% of the time-wheels of the tiles t_1 and t_2 are allocated to the application. Step 10, which optimizes the storage space allocations, finds no reduction in the storage space assigned to the edges. The estimated storage assignment is already optimal given all scheduling and binding constraints.

9.8 NoC Routing and Scheduling

In the previous phase of the design flow, the actors and edges are bound to the resources inside the tiles of the NoC-based MP-SoC architecture. Furthermore, a static-order schedule has been created for every processor to which at least one actor of the resource-aware SDFG is bound. TDMA time-slices have also been allocated on these processors. The binding and scheduling of the resource-aware SDFG onto the platform graph is captured in a binding-aware SDFG.

The throughput of an application mapped to a predictable platform can be computed through a constrained self-timed execution of a binding-aware SDFG. This throughput can be guaranteed when two constraints are met. First, all resources used by the binding-aware SDFG should be available in the platform when the application is executed. Second, the tokens sent between the actors in the binding-aware SDFG during its constrained self-timed execution should be sent in the interconnect within the time bounds given by this execution. In other words, the constrained self-timed execution of a binding-aware SDFG determines the time at which the communication of a token over the interconnect can be started and when it should be completed. To guarantee the throughput of the system, a schedule must be constructed which sends all tokens within their timing constraints over the NoC from their source tile to their destination tile. The NoC routing and scheduling phase of the design flow solves this problem.

The first step of the NoC routing and scheduling phase (step 11) uses the technique presented in Section 8.9 to extract the timing constraints for all tokens that are sent during the constrained self-timed execution of the binding-aware SDFG. These tokens are then scheduled onto the NoC in step 12 of the flow. Any of the NoC routing and scheduling strategies presented in Section 8.6 can be used in this communication scheduling step.

The design flow has successfully mapped an application SDFG onto a NoC-based MP-SoC as soon as a feasible communication schedule is found. However, the bandwidth allocations that were made in the tile binding and scheduling phase of the design flow are only estimates to steer the tile binding process. These

estimates were made in the 6th step of the flow. The actual required bandwidth is known after the communication schedule is constructed (step 12). At this point in the flow, the actual slot allocations are known and the amount of bandwidth used by this streaming application SDFG can be computed. This information should be used to update the bandwidth that is available for other applications when they are mapped to the same MP-SoC architecture. Updating the available bandwidth of the architecture is done in the final step of the NoC routing and scheduling phase.

The MP-SoC configuration that is constructed by the design flow when it maps the streaming application of Figure 9.2 to the NoC-based MP-SoC architecture of Figure 9.3 is shown in Figure 9.4. The final phase of the design flow allocated in this architecture two consecutive time-slots on the links l_1, l_3 and l_6. These slots are used to send tokens from actor a_2, which is mapped on tile t_1 to actor a_3, which is mapped to tile t_2. The MP-SoC configuration uses two out of eight slots on the outgoing link of tile t_1 and on the incoming link of t_2. Basically, 2/8 of the incoming and outgoing bandwidth of the tiles is used. To takes this usage into account, the available bandwidth (i and o) of these tiles (see Table 9.2) is reduced from 96 bits/time-unit to 72 bits/time-unit.

9.9 Implementation

The SDF^3 tool-kit, introduced in Section 4.6, has been developed to test the algorithms and strategies presented in this thesis. SDF^3 offers a random SDFG generator that can generate streaming application SDFGs, memory-aware SDFGs and resource-aware SDFGs. The user can control the characteristics of the graphs by specifying bounds, averages and variances on various aspects of an SDFG. The tool contains implementations of all algorithms that are discussed in this thesis. These algorithms are available through command-line tools and a C/C++ API. This allows a user to experiment with existing techniques and develop novel algorithms on top of them.

The NoC-based MP-SoC design flow has also been implemented in SDF^3. To use the NoC-based MP-SoC design flow, a user must input a streaming application SDFG and a NoC-based architecture, both described in XML format, into SDF^3. The tool will automatically perform most of the steps in the design flow. Iterations to previous steps of the flow are also performed when needed. The only step that requires manual intervention is the first step of the design flow. In this step, the user must manually insert the SDF memory access model for the appropriate actors and specify the properties of the newly added actors and edges. At every step of the flow, the complete state of the design flow can be outputted in XML. The user can then manipulate the design decisions made by the flow and continue the flow on a next step. The tool also allows a user to by-pass part of the flow and replace this part with his own custom algorithms.

9.10 Summary

This chapter presents a design flow that maps an SDFG onto a NoC-based MP-SoC. The flow combines the various SDFG mapping and analysis techniques presented in this thesis into a coherent and complete design flow that maps a throughput-constrained SDFG to a NoC-based MP-SoC. It minimizes the resource usage while offering guarantees on the throughput of the application when mapped to the system. The starting point is an SDFG that models the application with a throughput constraint. The flow starts with modeling non-local memory access. Next, it dimensions the storage space for the edges in the graph. It derives from the selected storage space requirements of the edges and the throughput constraint of the graph a number of additional constraints on the edges of the SDFG. These constraints are used in the next phase of the flow to guide the binding and scheduling of actors and edges to the tiles of the MP-SoC. The design flow ends with scheduling the communication of the tokens on the NoC.

Chapter 10

Case Study

10.1 Overview

A case study has been performed to show that the developed design flow can be used to map a set of real multimedia application onto a NoC-based MP-SoC while providing throughput guarantees. The case study considers the following use-case scenario. A user is using his or her handheld to make a video conferencing call. The video stream that is captured with the handheld's camera is encoded using an H.263 encoder. The handheld is also running an H.263 decoder to display the video stream that it receives from the other party involved in the call. Furthermore, an MP3 decoder is running on the handheld. This MP3 decoder is responsible for decoding the received audio stream. The audio stream produced by the user is ignored in the case study. It is assumed that this audio stream is encoded in some custom hardware. The three other applications (i.e., the H.263 encoder, the H.263 decoder and the MP3 decoder) are executed simultaneously on the NoC-based MP-SoC hardware that is available inside the handheld. The design flow presented in the previous chapter is used in this case study to find a mapping of the applications onto the handheld hardware such that every application meets its throughput constraint.

The next section presents the three applications that are used in this case study. It introduces the SDFGs that model these applications along with their resource requirement. It also specifies the throughput constraints for the applications. Section 10.3 presents the hardware architecture that is targeted in this case study. The mapping of the applications onto the architecture is discussed in Section 10.4. Based on the results of the case study, extensions and changes to the design flow are discussed in Section 10.5.

10.2 Applications

10.2.1 H.263 encoder

The case study presented in this chapter considers a handheld that is used to make a video call. The H.263 [70] video compression standard has been specifically designed for this purpose. It aims at compressing low resolution video streams for transmission over a phone network. In this case study, it is assumed that the handheld compresses a video stream with frames that have a resolution of 174 by 144 pixels (QCIF resolution). An H.263 encoder divides a frame in a set of macro blocks (MBs). A macro block captures all image data for a region of 16 by 16 pixels. The image data inside a MB can be subdivided into 6 blocks of 8 by 8 data elements. Four blocks contain the luminance values of the pixels inside the MB. Two blocks contain the chrominance values of the pixels inside the MB. A frame with QCIF resolution contains 99 MBs that consist, in total, of 594 blocks. In this case study it is assumed that the handheld's camera can capture 15 frames per second. The H.263 encoder must encode all frames in real-time. So, it should be able to encode, on average, one frame each 67ms. This throughput requirement must be guaranteed when the application is executed in the handheld.

Figure 10.1 shows an SDFG that models an H.263 encoder. This model was already introduced in Section 2.3. The H.263 encoder from Figure 10.1 is based on the SDFG presented in [110]. The SDFG from [110] assumes that the motion estimation actor can only output a complete frame at a time and not the 99 MBs that this frame is composed of. The model of [110] uses a separate actor to divide the frame into 99 MBs. The SDF model of Figure 10.1 assumes that the motion estimation actor can directly output the MBs. This is a reasonable assumption as the division of a frame into its MBs is trivial. It essentially implies a merge of the two actors used in [110].

Telenor Research has made a C-based implementation of an H.263 encoder [119]. In [80], this implementation is used to derive the worst-case execution times of the actors in the H.263 encoder SDF model of [110]. Unfortunately, it is not specified which processor architecture is assumed when computing these worst-case execution times. Based on analysis of part of the source code with CTAP (see Section 2.4), it seems that the reported execution times are valid for a processor architecture similar to the ARM7. The goal of the case study presented in this chapter is to demonstrate the applicability of the predictable design flow. For this purpose, it is not needed that the worst-case execution times are completely accurate. They should only be representative for a processor architecture similar to the one used in the platform that is presented in the next section. For this reason, the worst-case execution times as reported in [80] are used in the case study as the execution times of the actors in the SDFG of Figure 10.1. These execution times are shown in Table 10.1. The table shows also the worst-case stack sizes of the actors and the size of the tokens communicated on the edges. The worst-case stack sizes have been computed using CTAP and the H.263 encoder

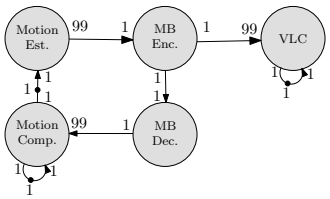


Figure 10.1: SDFG model of an H.263 encoder.

Actor	Execution time [cycles]	Stack size [bytes]
Motion Est.	382419	316352
Motion Comp.	11356	2796
MB Enc.	8409	2216
MB Dec.	6264	864
VLC	26018	1356

Edge	Token size [bytes]
VLC self-edge	1024
Motion Comp. self-edge	38016
Motion Comp. to Motion Est.	38016
Others	384

Table 10.1: Worst-case resource requirements for an H.263 encoder.

implementation from [119]. The token sizes are obtained through manual code analysis.

10.2.2 H.263 decoder

The handheld that is considered in this case study is also running an H.263 decoder. This decoder is used to decompress the video stream that the handheld receives from the other party involved in the call. The decoder reverts all operations performed by the encoder in order to reconstruct the original video stream. It is assumed that the H.263 decoder that is used in this case study operates on frames with QCIF resolution. Furthermore, it is assumed that the received video stream has a maximal frame rate of 15 frames per second. Sufficient resources should be reserved in the handheld to allow the decoder to support this frame rate.

An SDFG that models an H.263 decoder is shown in Figure 10.2. The variable length decoder (VLD) actor is responsible for decompressing the bitstream. It performs the inverse operation of the VLC actor in the H.263 encoder model (see Figure 10.1). On each firing, the VLD actor produces decompressed data (encoded MBs) for a complete video frame. The inverse quantization (IQ) and

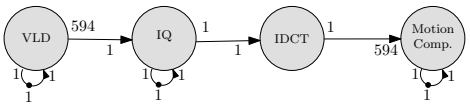


Figure 10.2: SDFG model of an H.263 decoder.

Actor	Execution time [cycles]	Stack size [bytes]
VLD	26018	1356
IQ	559	50
IDCT	486	50
Motion Comp.	10958	1000

Edge	Token size [bytes]
VLD self-edge	1024
IQ self-edge	64
Motion Comp. self-edge	38016
Others	64

Table 10.2: Worst-case resource requirements for an H.263 decoder.

inverse discrete cosine transformation (IDCT) actors revert the MB encoding. Together they perform the same transformation on the data as the MB decoding actor in the H.263 encoder. The IQ and IDCT actors operate on a single block of encoded pixel data instead of a complete MB. This allows smaller memory requirements for these actors when compared to the MB decoding actor in the H.263 encoder. The motion compensation (Motion comp.) actor takes a group of 594 blocks to reconstruct the original video frame. To do this, the Motion Comp. actor needs also the previously decoded frame. This frame is stored on the self-edge of the actor. The token on the self-edge of the VLD and IQ actors model respectively the Huffman decoding table and dequantization table that are stored between subsequent firings of these actors. A token on the other edges (i.e. all non self-edges) represents a block of encoded pixel data with a size of 8 by 8 pixels. These tokens requires thus 64 bytes of storage space.

A C-based implementation of an H.263 decoder is available from Telenor Research [119]. Using the techniques described in Section 2.4, the worst-case execution times, stack sizes and token sizes have been analyzed for this implementation. These results are summarized in Table 10.2.

10.2.3 MP3 decoder

The MPEG-1 layer 3 (MP3) audio compression format is widely used to compress audio streams [83]. The decoder that is needed to decompress this format has been introduced in Section 2.2. An SDFG to model an MP3 decoder has been presented

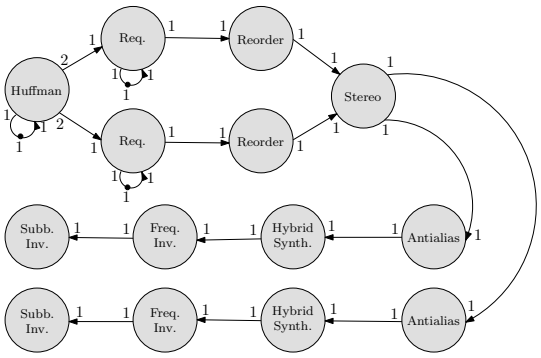


Figure 10.3: SDFG model of an MP3 decoder.

Actor	Execution time [cycles]	Stack size [bytes]
Huffman	151977	6068
Req.	72695	104
Reorder	34684	2352
Stereo	53602	68
Antialias	409	636
Hybrid Synth.	7414	10
Freq. Inv.	4912	16
Subb. Inv.	1865001	3736

Edge	Token size [bytes]
Huffman self-edge	1024
Req. self-edge	64
Others	576

Table 10.3: Worst-case resource requirements for an MP3 decoder.

in Section 2.3. This SDFG is also shown in Figure 10.3. Similar to the H.263 decoder, the tokens on the self-edges of the graph model data that is needed in subsequent firings (i.e. these tokens model global data). The tokens send over the other edges represent a (partly processed) frame of 576 audio samples. Each sample has a size of 1 byte. So, these tokens have a size of 576 bytes. Since all actors operate on a frame, the SDFG is said to operate on the frame level. A frame represents an audio sequence with a duration of 10ms. This duration determines the throughput constraint for the actors. It implies that on average a frame should be produced once every 10ms.

The predictable design flow uses the worst-case execution time and stack-size to allocate sufficient resources for an application to meet its throughput constraint when executed on a NoC-based MP-SoC. These worst-case figures have been de-

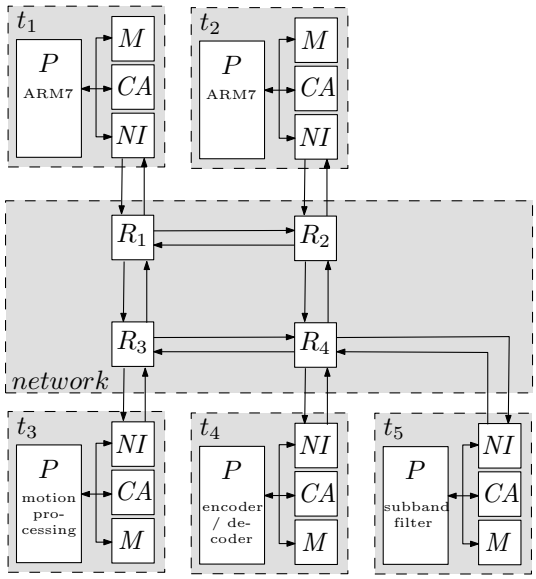


Figure 10.4: NoC-based MP-SoC architecture.

rived from an MP3 decoder implementation [83] using CTAP (see Section 2.4). The worst-case execution time and stack-size of the actors in the SDFG from Figure 10.3, which models an MP3 decoder, are repeated in Table 10.3.

10.3 Hardware Architecture

The handheld that is considered in this case study contains a NoC-based MP-SoC architecture that follows the platform template presented in Chapter 3. Its structure is shown in Figure 10.4. This hardware architecture consists of five tiles with different processing and storage capabilities. The specifications of the tiles are summarized in Table 10.4. The tiles t_1 and t_2 are generic processing tiles containing an ARM7 processor. Tile t_3 contains a variable length encoder/decoder accelerator. This accelerator can speed-up the execution of the VLC, VLD and Huffman actors that are present in the SDF models of the three multimedia applications. It is assumed that executing these actors on the accelerator requires only half the number of cycles as executing them on an ARM7 processor. Another accelerator is available in tile t_4. This accelerator contains special purpose hardware to perform motion estimation and motion compensation. It can execute the Motion Est. and Motion Comp. actors, reducing their execution time by 50% when compared to an ARM7. Tile t_5 contains a subband inverter. This accelerator can be used by the Subb. Inv. actor contained in the MP3 decoder model.

Table 10.4: Properties of the tiles in the architecture of Figure 10.4.

	pt	w [time-units]	m [bytes]	c	i/o [bits/time-unit]
t_1	ARM7	100000	$200{\cdot}10^3$	8	96/96
t_2	ARM7	100000	$200{\cdot}10^3$	8	96/96
t_3	Encoder/Decoder	100000	$500{\cdot}10^3$	8	96/96
t_4	Motion processing	100000	$500{\cdot}10^3$	8	96/96
t_5	Subband filter	100000	$500{\cdot}10^3$	8	96/96

Table 10.5: Properties of the NoC in the architecture of Figure 10.4.

N	sz_{flit} [bits]	sz_{ph} [bits]	T_{reconf} [time-units]
8	96	32	32

This accelerator also reduces the execution time of the actor by 50% when compared to an ARM7. All processors inside the architecture operate on a 500MHz frequency. The NoC, which interconnects the tiles, uses a flit-clock with the same frequency. A flit consists of 96 bits (see Table 10.4). Assuming a word-size of 32 bits for the NoC, which matches with the word-size of the processors, the NoC needs three cycles to send one flit. Therefore, it must operate at a core frequence which is three times as high. In other words, the NoC is running at a frequency of 1.5GHz. The communication assist (CA) and network interface (NI) inside each tile support up-to 8 connections. These connections can be used to receive 96 bits/time-unit (one flit) and also send one flit per time-unit. The three tiles containing an accelerator each have 500k bytes of storage space. This allows these tiles to buffer large tokens (e.g., a frame). The less compute intensive actors are typically also the less data intensive actors. These actors can be executed on the tiles with an ARM core. Tiles t_1 and t_2 have therefore a more limited storage space.

The design flow schedules individual flits into a single time-unit. This implies that the time-unit that is used in the design flow is equal to a single flit-clock cycle. The flit-clock has a frequency of 500MHz. Therefore, a time-unit has a duration of $1/500\mu$s (=2ns). Using this duration, the size of the TDMA time wheels inside the processors can be converted to wall-clock time. These time wheels have a size of 100000 time-units (see Table 10.4). So, one full rotation the time wheels occurs once every 0.2ms. The duration of the time-unit is also important as the throughput constraints of the applications have to be converted from wall-clock time to the abstract time-unit that is used inside the flow. Consider the H.263 decoder as an example. Its timing constraint requires the production of 15 frames per second. The throughput as used in the design flow is given by Definition 11. It considers the number of iterations per time-unit. During one iteration of the H.263 decoder model (see Figure 10.2), the graph produces one frame. So, 15

frames/second is equivalent to 15 iterations/second. Converting the seconds to the abstract time-unit, the throughput constraint of the H.263 decoder becomes $3{\cdot}10^{-8}$ iterations/time-unit. The throughput constraint of the H.263 encoder (15 frames/second) and MP3 decoder (10 ms/frame) can be converted in the same way, resulting in constraints of respectively $3{\cdot}10^{-8}$ iterations/time-unit and $2{\cdot}10^{-7}$ iterations/time-unit.

10.4 Mapping

Section 10.2 introduces the three applications that are used in this case study. The targeted hardware architecture is presented in the previous section. This section discusses the mapping of the applications onto this architecture. This mapping is performed using the design flow that is presented in the previous chapter.

The design flow maps a single application at a time onto the architecture. When mapping this application, the flow takes into account the resources that are already used by other applications. However, it cannot change the mapping of these applications. Therefore, selecting the order in which applications are handled is important, as this may influence the resource usage and/or number of applications that can be mapped to the architecture. One specific ordering has been chosen in this case study to map the applications onto the architecture. This ordering is based on the memory requirements of the applications. Both the stack-size of the actors and the storage space requirements of the edges are considered when estimating, by hand, these memory requirements. Based on these estimates, the H.263 encoder is the first application to be mapped onto the NoC-based MP-SoC. The second application that is bound and scheduled on the architecture is the H.263 decoder. The MP3 decoder is the last application to be mapped.

The tile binding and scheduling strategy presented in Chapter 6 is used in the design flow. This strategy uses a cost function to steer the binding of the actors of an application onto the various resources in the hardware architecture. The experimental evaluation presented in Section 6.8 shows that it is important to minimize the communication on the interconnect. This is due to the fact that when data is sent between tiles, the time wheels need to synchronize. The time spent on synchronization can become almost as large as a full time wheel rotation. This introduces large potential delays for tokens communicated between actors that are bound to different tiles. To accommodate for these delays, the design flow will allocate large time slices on the processors. This leads, in turn, to a poor resource usage as time wheels are quickly occupied. For this reason, a cost function $(1, 0, 2, 0)$ is used that minimizes the communication overhead. The cost function also aims at balancing the processing load. These two objectives allow the tile binding step (step 7) to make a trade-off between the communication overhead and the use of accelerators present in the hardware architecture. The memory usage is ignored in this cost function as the potential bindings have similar

memory requirements. The cost function also ignores the latency overhead as all possible connections in the NoC have a similar latency.

Three different NoC routing and scheduling strategies can be used in the communication scheduling step of the flow (step 12). The experimental evaluation presented in Chapter 8 shows that both the H.263 decoder and MP3 decoder can be handled with the greedy strategy. This strategy is the fastest of the three heuristics presented in Chapter 8. Based on these results, this strategy is used in the case study. The greedy strategy allows the use of a detour when selecting the routes through the NoC. Using a detour, routes larger than those with the minimal length are considered by the strategy. This can be useful in case all minimal routes are already occupied. For this experiment, a detour of 3 links is used. This detour allows the communication scheduling strategy to explore all possible routes between the tiles in the NoC when scheduling the communication.

As mentioned, the design flow starts with mapping the H.263 encoder onto the hardware architecture shown in Figure 10.4. The flow starts with the modeling of non-local memory accesses in the graph. This is needed when an actor uses tokens that are too large to fit into the memory of the tile to which the actor is bound. The tokens in this application are small when compared to the storage space available inside the tiles. Therefore, it is assumed that no tokens need to be stored in a non-local tile. So, no changes are made to the graph in the first step of the flow. In the second step, the storage-space/throughput trade-off space is computed. This space contains, as shown in Section 7.6, 20 different trade-off points. The first trade-off point from this space is selected in the storage distribution selection step (step 3) as this point meets the throughput constraint of the application. The first phase of the flow is then finished by estimating the storage space of the edges, when their source and destination actors are bound to different tiles. The constraint refinement phase uses this information to compute the latency and bandwidth constraints of the edges. This completes the construction of the resource-aware SDFG that is used in the tile binding and scheduling phase. This phase starts with binding the actors to the various tiles contained in the hardware architecture. The tile binding strategy binds actor MB Enc. to tile t_1, actor MB Dec. to t_2, actor VLC to t_3 and the Motion Est. and Motion Comp. actor to t_4. Following this binding, static-order schedules are constructed for the actors bound to the various tiles. The schedule on tile t_4 contains two states, all other schedule contain a single state. The flow continues with allocating TDMA time-slices in step 9. This requires a total of 16 throughput calculations of the bound and scheduled graph before time slices are found with which the throughput constraint is met and that also minimize the time wheel usage. Next, the flow tries to optimize the storage space allocated to the edges of the graph. It turns out that the used storage space is already minimal given the throughput constraint of the application. At this point, the last phase of the design flow is started. The flow extracts the timing constraints for the tokens that must be communicated via the interconnect (step 11) and it schedules these tokens on the links of the NoC (step 12). A total of 792 tokens are sent in the transient phase. Scheduling these

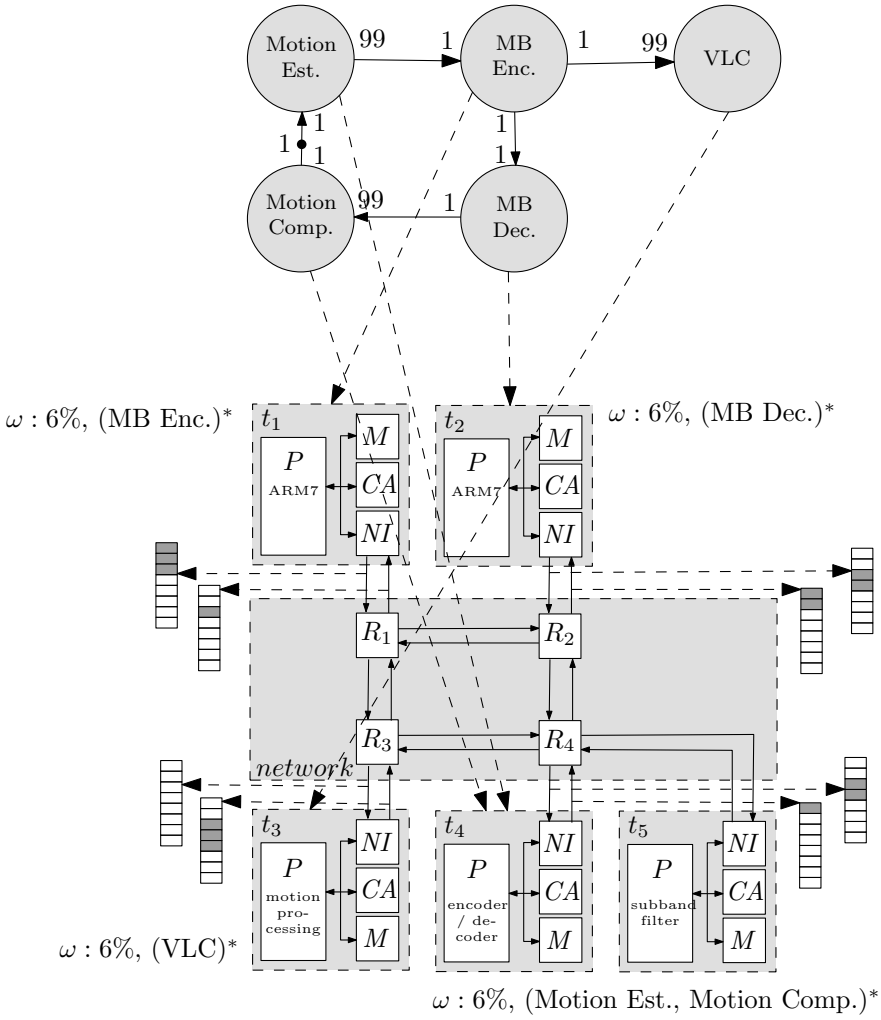


Figure 10.5: MP-SoC configuration of the H.263 encoder.

Table 10.6: Run-time of the design flow steps.

	H.263 enc.	**H.263 dec.**	**MP3 dec.**
1. Model non-local memory accesses [ms]	0	0	0
2. Compute storage-space / throughput trade-offs [ms]	2	1611	143
3. Select storage distribution [ms]	0	0	0
4. Estimate storage distribution per connection	1	0	0
5. Estimate latency constraints [ms]	0	0	1
6. Compute bandwidth constaints [ms]	0	0	0
7. Bind SDFG to tiles [ms]	1	5	6
8. Constrcuct static-order schedule per tile [ms]	15	110	3
9. Allocate TDMA time-slices [ms]	227	675	43
10. Optimize storage space allocations [ms]	2	26	2
11. Communication constraint extraction [ms]	37	59	3
12. Communication scheduling [ms]	86	199	2
13. Update bandwidth allocations [ms]	2	3	0
Total [ms]	**415**	**2688**	**203**

tokens takes 86ms. The periodic phase contains 395 tokens, which are scheduled in 22ms. As a final step, the flow computes the bandwidth usage and updates the available resources for the next application to be mapped onto the hardware architecture of Figure 10.4. This completes the mapping of the H.263 encoder onto the hardware architecture. The resulting MP-SoC configuration is shown in Figure 10.5. This MP-SoC configuration is found by SDF^3, which implements the design flow, within 415ms when executed on a P4 at 3.2GHz. The run-time used for the various steps in the design flow is shown in Table 10.6. The resource usage of the H.263 encoder when mapped to the architecture is shown in Table 10.7. This table shows also the resource usage of the other applications considered in this case study.

The H.263 decoder is the second application that is mapped onto the hardware architecture of Figure 10.4. When mapping this application, the design flow takes into account the resources that are already used by the H.263 encoder. The mapping of the H.263 decoder is done in a similar way as the H.263 encoder. The tile binding step (step 7) decides to map both the IQ and IDCT actors onto tile t_1. This leads to an unbalanced processing load for the identical tiles t_1 and t_2, but it avoids the communication of tokens over the interconnect. This shows one of the trade-offs made by the cost function in the tile binding strategy. The TDMA time wheel optimization step (step 9) performs 20 throughput computations, each requiring between 18ms and 71ms. It is interesting to note that the H.263 decoder needs much smaller time slices than the H.263 encoder to achieve the same frame rate (see Table 10.7). This is due to the fact that the decoder allows pipelining of the frames that are processed, while the encoder can only process a single frame at a time. Similar to the H.263 encoder, the storage-space allocated to the de-

Table 10.7: Resource usage of the applications.

	H.263 encoder				
	t_1	t_2	t_3	t_4	t_5
#actors bound	1	1	1	2	0
#channels bound	3	2	1	3	0
time slice [time-units]	6250	6156	5879	5963	0
memory [bytes]	41000	20448	244172	305656	0
#connections	3	2	1	2	0
input bw [bits/time-unit]	12	24	36	24	0
output bw [bits/time-unit]	36	24	0	12	0
#states in schedule	1	1	1	2	0
	H.263 decoder				
	t_1	t_2	t_3	t_4	t_5
#actors bound	2	0	1	1	0
#channels bound	3	0	1	1	0
time slice [time-units]	1916	0	71	47	0
memory [bytes]	38386	0	43468	191080	0
#connections	2	0	1	1	0
input bw [bits/time-unit]	12	0	0	24	0
output bw [bits/time-unit]	24	0	12	0	0
#states in schedule	2	0	1	1	0
	MP3 decoder				
	t_1	t_2	t_3	t_4	t_5
#actors bound	0	12	0	0	2
#channels bound	0	14	0	0	2
time slice [time-units]	0	13957	0	0	75000
memory [bytes]	0	19892	0	0	4888
#connections	0	2	0	0	2
input bw [bits/time-unit]	0	0	0	0	24
output bw [bits/time-unit]	0	24	0	0	0
#states in schedule	0	86	0	0	2

Table 10.8: Combined resource usage of the applications.

	t_1	t_2	t_3	t_4	t_5
#actors bound	3	13	2	3	2
#channels bound	6	16	2	4	2
time slice [time-units]	8166	20113	5950	6010	75000
memory [bytes]	79386	40340	287640	496736	4888
#connections	5	4	2	3	2
input bw [bits/time-unit]	24	24	36	48	24
output bw [bits/time-unit]	60	48	12	12	0

coder cannot be optimized after it is bound and scheduled on the architecture. The communication scheduling step (step 12) has to schedule 2376 tokens onto the interconnect. This takes a total of 199ms. In total, SDF^3 spends 2688ms on mapping this application onto the hardware architecture. The largest part of this time is spent in step 9 (time wheel optimization) and step 3 (computing the storage-space/throughput trade-off space). It takes 1.6s to compute the complete storage-space/throughput trade-off space that contains 183 minimal storage distributions. These results differ from the run-time (53 minutes) and number of minimal storage distributions (3255) reported for the same application in Section 7.6. The reason is that the H.263 decoder used in this case study operates at QCIF resolution (174 by 144 pixels), while the H.263 decoder consider in Section 7.6 works at CIF resolution (348 by 288 pixels). The VLD and Motion Comp. actors operate at a complete frame and have therefore a different execution time in both models. As a result, a much smaller trade-off space needs to be searched in this case study then in Section 7.6.

The last application that is bound and scheduled on the handheld's hardware architecture is the MP3 decoder. To avoid communication overhead, in particular the costly reservations needed to guarantee throughput in the absence of time wheel synchronization (see Section 10.5), most of the actors are bound to tile t_2. Only the actors modeling the subband inversion are bound to an accelerator (tile t_5). As a result, only a very limited number of 18 token communications must be scheduled onto the interconnect. SDF^3 computes the MP-SoC configuration for this application in 203ms. At this point, all three applications are bound and scheduled onto the NoC-based MP-SoC architecture of Figure 10.4. The combined resource usage of the applications when bound to the architecture is shown in Table 10.8. Compared to the available resources in the architecture (see Table 10.4), it shows that only about half of the available resources are used. This is due to the fact that the architecture has not been optimized for this specific set of applications. The data in Table 10.8 could be used to perform such an optimization.

SDF^3 used a total of 3.3s to find a mapping for all three applications. This shows that the predictable design flow and its implementation allow a fast mapping of real multimedia applications onto a heterogeneous NoC-based MP-SoC. The chosen resource allocations and schedules allow guarantees to be provided on the throughput of individual applications independent of other applications executed simultaneously on the architecture.

10.5 Discussion on the Design Flow

This section discusses modifications and extensions of the design flow. Studying the exact impact of these changes is left as future research.

The current design flow assumes that the relative position of the time slices on the various processors is unknown. To analyze the worst-case throughput

correctly, the design flow assumes that tokens that are sent from one tile to another arrive just after the time slice of the application on the receiving tile has passed. The tokens can only be used after a full time wheel rotation minus the reserved time slice. When large time wheels are used, the worst-case waiting time that is taken into account can become quite large. For example, on a processor with a time wheel of 10ms of which 10% is allocated to an application, the waiting time is equal to 9ms. To accomodate for these worst-case waiting times, the time slice allocation step (step 9) may have to allocate large time slices. The waiting time, and thus the large time slices, can be avoided when the relative position of the time wheels on the processors is known. When this relative position is known, it becomes possible to compute the exact amount of time that tokens have to wait before they can be used on the receiving tile. The disadvantage of this approach is that it requires that specific slices are allocated to an application instead of just a fraction of the time wheel. This may make it harder to map multiple applications to a hardware architecture as the time slice allocations may conflict. Another solution to solve this problem is to allow the design flow to resize the time wheels. Reducing a time wheel reduces the waiting time of the communicated tokens. Note that a reduction of a time wheel does not affect the compositionality of the design flow. It is still possible to provide throughput guarantees on individual applications as long as the sum of the time slices allocated to all applications mapped to a processor does not exceed the size of the time wheel.

When the communication scheduling step (step 12) fails to find a valid schedule, the flow iterates back to step 3 and increases the storage space allocations of the edges. It also computes new latency constrains for these edges. Currently, the latency assigned to the edges decreases when the storage space is increased. Increasing the storage space combined with decreasing the latency increases the freedom in the time wheel allocation step (step 9). However, reducing the latency puts additional constraints on the communication scheduling (step 12). In the situation that the flow fails to find a valid communication schedule, it is okay to increase the storage space allocated to the edges. This adds potential scheduling freedom. The latency should however not be decreased; it should be increased. Note that when an edge is not part of a cycle in the application graph, its latency can be made arbitrarily large while still meeting the throughput constraint of the application. This requires only that sufficient storage space is allocated to it. So, the communication scheduling problem can always be relaxed for these edges when sufficient storage space is available.

The run-times as reported in the case study show that it is feasible to complete the design flow within seconds. Many throughput computations are performed during the flow. Throughput computation are for example performed to find the storage-space/throughput trade-off space (step 3 and step 10) and to allocate TDMA time slices (step 9). A single throughput computation requires typically only milliseconds. The design flow currently uses latency and bandwidth estimates that are based on a throughput estimate and the throughput constraint. Considering the run-time needed for a single throughput computation, it might

be feasible to add additional steps after step 7 (tile binding) and step 10 (storage-space optimization) in which the latency and bandwidth requirements of the edges are dimensioned using the throughput analysis technique that is used in the design flow while taking into account the design decisions that have already been taken. This may lead to latency and bandwidth constraints that fit better with the tile binding. For example, it may lead to decreased time slice allocations on the processors and/or to increased latencies for edges bound to connections. The former is due to decreased latency and bandwidth constraints and leads to a lower resource usage. The latter relaxes the communication scheduling problem (step 12).

10.6 Summary

This chapter presents a case study in which a set of multimedia applications are mapped onto a NoC-based MP-SoC platform. This mapping is performed using the predictable design flow presented in the previous chapter. This design flow, in turn, uses all techniques presented in this thesis. The case study shows that the design flow can be used to map real applications onto a NoC-based MP-SoC platform, with a total run-time of only a few seconds, while providing throughput guarantees on the individual applications. The chapter presents, based on the results of the case study, also several potential changes and additions to improve the quality of the design flow.

Chapter 11

Conclusions and Future Work

11.1 Conclusions

Consumers expect that more and more functionality is integrated into novel multimedia devices. At the same time, they expect a seamless and reliable behavior of these devices. To manage this increasing design complexity, a design flow is needed that guarantees predictable behavior of the resulting system. This implies that it must map an application to a platform in such a way that the timing behavior of the application, when running on the platform, can be guaranteed, independent of other applications running on the platform. This requires that the timing behavior of the hardware, the software, as well as their interaction can be predicted.

To accommodate the increasing number of applications that are integrated into multimedia devices, system designers are starting to use heterogeneous multiprocessor Systems-on-Chip (MP-SoCs). An MP-SoC offers a good computational efficiency for many applications. A Network-on-Chip is often suggested as interconnect in these systems. It offers scalability and guarantees on the timing behavior when communicating data between various processing and storage elements. Combining this with a predictable resource arbitration strategy for the processors and storage elements gives a predictable platform. A predictable system requires also that the timing behavior and resource usage of an application, when mapped to a predictable platform, can be analyzed and predicted. For this purpose, the Synchronous Dataflow (SDF) Model-of-Computation is used in this thesis. It fits well with the characteristics of streaming multimedia applications, it can capture many mapping decisions, and it allows design-time analysis of timing and resource usage. The third aspect that is required to obtain a predictable system is the use of a predictable design flow. This design flow must allocate resources for streaming applications, modeled as SDF graphs (SDFGs), in NoC-based MP-SoC platforms such that the timing behavior of each individual

application can be guaranteed. This thesis studies several of the steps that are needed for such a predictable design flow in detail. It also embeds the proposed techniques into a coherent design flow.

An important aspect that influences the throughput of an SDFG is the storage space that is allocated for the edges of the graph. Chapter 7 proposes a technique to find the complete trade-off space between the storage-space allocation for the edges of a graph and the maximal throughput that can be realized under these storage constraints. The experimental results show that, using this technique, the complete storage/throughput trade-off space of real multimedia applications can often be computed in seconds or even milliseconds. An approximation technique that provides throughput guarantees and that has an analytical bound on the buffer size over-estimation is also presented. This approximation technique can be used when the run-time of the exact algorithm becomes too long. The experiments show that it can drastically improve the run-time needed for the exploration of the trade-off space with only very limited overestimation of the storage space.

The storage space requirements of the edges and actors of an SDFG might be larger than the storage space available in the memory that is close to a processor. The solution to this problem is to embed large, shared memories in the platform that are accessed through the NoC. An SDFG model is presented in Chapter 5 to model accesses to this memory. It allows modeling of arbitrary access patterns to the memory and analysis of the timing behavior of the various system components (i.e. processor, memory, NoC) involved in the memory operations. This chapter also outlines a technique to extract the SDF memory access model from the source code of an application.

The allocation of resources for throughput-constrained SDFGs is studied in Chapter 6. The proposed resource allocation technique binds and schedules an SDFG onto a heterogeneous multi-processor system while providing throughput guarantees, even in the context of resource sharing. The technique can deal with multi-rate and cyclic dependencies between actors without converting it to a homogeneous SDFG. Building on an efficient technique to calculate throughput of a bound and scheduled SDFG, it becomes possible to perform resource allocation for real multimedia applications much faster than when existing resource allocation techniques are used.

After binding and scheduling the actors to the MP-SoC platform, it remains to schedule the communication onto the NoC. Chapter 8 presents three different scheduling strategies that minimize resource usage by exploiting all scheduling freedom offered by NoCs while guaranteeing that the timing constraints are met. The experimental results show that these strategies outperform existing NoC scheduling techniques. Furthermore, a technique is presented to extract the timing constraints on the communication from a bound and scheduled SDFG. This connects the communication scheduling to the tile binding and scheduling mentioned before.

Chapter 9 proposes a predictable design flow that maps an application, modeled with an SDFG, onto a predictable platform. This design flow combines all

the techniques presented in this thesis. The objective of the flow is to minimize the resource usage while offering throughput guarantees on the application when mapped to the platform. A case study is performed in which a set of multimedia applications are mapped onto a NoC-based MP-SoC. It shows that the design flow, SDFG mapping techniques, and SDFG analysis techniques presented in this thesis enable a mapping of a streaming application onto a NoC-based architecture that has a predictable timing behavior. This makes it the first complete design flow that maps a time-constrained SDFG to a NoC-based MP-SoC while providing throughput guarantees.

The predictable design flow and all techniques presented in this thesis have been implemented in SDF^3. The result is a versatile tool that can be used to design multi-processor systems with a predictable timing behavior. It can also serve as a starting point for future research on predictable design flows. The tool and its source code are freely available from `http://www.es.ele.tue.nl/sdf3`.

11.2 Open Problems and Future Research

The design flow presented in this thesis offers an interesting starting point for research into the design of predictable systems. However, in order to increase its applicability in designing real systems, several issues need to be researched further:

- A technique is outlined in Chapter 5 to extract the memory access model presented in this chapter automatically from a source code fragment. This technique is currently not implemented in SDF^3. The tool should be extended with this technique to allow a fully automatic design flow that can explore alternative memory mappings to a multi-level memory architecture with both shared and private memory modules without user intervention.

- The ordering of the steps in the design flow has not been extensively tested. More research is needed into the ordering of the steps. Potentially this may also lead to the introduction of new steps in the flow. For example, an additional step may be introduced to reduce the storage-space allocation of edges further. The static-order schedule of the actors of an application on a processor makes it possible to perform life-time analysis on the data stored in the edges between those actors. Using this technique, the storage space requirements of the edges can be reduced further. Section 10.5 discusses more options for changes to the design flow based on the results of the case study presented in Chapter 10.

- The SDF Model-of-Computation is not suitable for modeling dynamic behavior. A first step to address this issue, is to extend the techniques presented in this thesis to the Cyclo-Static Dataflow Model-of-Computation.

This Model-of-Computation can express some dynamism and it seems possible to extend all techniques presented in this thesis to CSDF. A more general solution seems to be the extension of the design flow and techniques to support scenarios, as proposed in [48, 89]. When using scenarios, an application is partitioned into a set of SDFGs that each describe a different behavior (scenario) of the application. The design flow must then map this set of graphs to a platform while providing timing guarantees. To exploit scenarios in a system, techniques are also needed to perform run-time resource allocation in order to allow run-time switching between scenarios. Novel analysis techniques are necessary to provide timing guarantees on a context with switches.

- The presented design flow focuses on streaming applications. Often modern streaming applications contain also a (small) control part. Therefore, it might be interesting to study an extension of the design flow to deal with event-driven reactive behavior and control processing. The SDF or CSDF based version of Reactive Process Networks (RPN) [43] might be an interesting candidate to allow a predictable design flow for systems that have both (dynamic) streaming and reactive, control-oriented components. In addition, the RPN Model-of-Computation is sufficiently rich to allow the integration of scenarios. As a prerequisite, it is necessary to develop analysis and synthesis techniques for RPN, in particular the SDF or CSDF based version of it.

- A fixed platform is input to the design flow. The flow does not consider changing scheduler settings such as the number of TDMA slots on the links of the NoC or the size of the TDMA time wheels on the processors. Changing these scheduler settings may reduce the resource requirements of applications. In certain situations, these changes can be made without affecting the composability of the resulting mappings (see Section 10.5). When a single iteration of the design flow is sufficiently, a fast a design-space exploration algorithm can be build on top of the design flow to explore the architecture space. The design flow could then be used both in a compilation and synthesis approach.

- The design flow presented in this thesis focuses on providing throughput guarantees. Other system properties and requirements like latency and code size are not taken into account. Extension of the flow to take also these aspects into account would allow a more extensive exploration of the system design space.

- The current flow focuses on providing hard guarantees. Many multimedia applications require only soft guarantees. Designing a system with soft guarantees might reduce resource usage (e.g., smaller buffer sizes). It may also reduce the requirements for the platform. For example, a NoC can be used

that provides best-effort connections [107] instead of guaranteed throughput connections and the processor utilization may improve as unreserved TDMA slots can be used. Novel analysis techniques are needed to reason about soft guarantees.

- Currently the flow focuses on minimizing the resource usage which is typically also beneficial for the the power and energy consumption of a system. Novel systems will have strict power and energy budgets. Therefore, the design flow should be extended to take these constraints into account. This requires the development of novel system-level and system wide power and energy models.

- A feasibility study should be performed in which the back-end of the flow developed in this thesis is extended to map applications onto a real platform, e.g. a NoC-based MP-SoC embedded into an FPGA. This feasibility study should demonstrate the use of the design flow in building predictable systems.

Bibliography

[1] E. Aarts, R. Harwig, and M. Schuurmans. *Ambient Intelligence*, pages 235–250. The Invisible Future: The Seamless Integration of Technology in Everyday Life. McGraw-Hill, 2002.

[2] J. Absar and F. Catthoor. Analysis of scratch-pad and data-cache performance using statistical methods. In *Conference on Asia South Pacific design automation, ASP-DAC 06, Proceedings*, pages 820–825. IEEE, 2006.

[3] B. Ackland, A. Anesko, D. Brinthaupt, S.J. Daubert, A. Kalavade, J. Knobloch, E. Micca, M. Moturi, C.J. Nicol, J.H. O'Neill, J. Othmer, E. Sackinger, K.J. Singh, J. Sweet, C.J. Terman, and J. Williams. A single chip 1.6 billion 16-b MAC/s multiprocessor DSP. *IEEE Journal of Solid-State Circuits*, 35(3):412–424, March 2000.

[4] M. Adé, R. Lauwereins, and J.A. Peperstraete. Data minimisation for synchronous data flow graphs emulated on DSP-FPGA targets. In *34th Design Automation Conference, DAC 97, Proceedings*, pages 64–69. ACM, 1997.

[5] K. Altisen, G. Gößler, and J. Sifakis. A methodology for the construction of scheduled systems. In *International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems, FTRTFT 00, Proceedings, volume 1926 in LNCS*, pages 106–120. Springer-Verlag, 2000.

[6] G.M. Amdahl. Validity of the single processor approach to achieve large scale computing capabilities. In *American Federation of Information Processing Societies Conference, AFIPS 67, Proceedings*, pages 483–485. Thomson Book Company, 1967.

[7] T. Amnell, E. Fersman, L. Mokrushin, P. Pettersson, and W. Yi. Times: a tool for schedulability analysis and code generation of real-time systems. In *Formal Modeling and Analysis of Timed Systems, FORMATS 03, Proceedings, volume 2791 in LNCS*, pages 60–72. Springer-Verlag, 2003.

[8] ARM. ARM7TDMI processor. `http://www.arm.com/products/CPUs/ARM7TDMI.html`.

[9] O. Avissar, R. Barua, and D. Stewart. An optimal memory allocation scheme for scratch-pad-based embedded systems. *Transactions on Embedded Computing Systems*, 1(1):6–26, November 2002.

[10] A. Avižienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *Transactions on Dependable and Secure Computing*, 1(1):11–33, January 2004.

[11] F. Baccelli, G. Cohen, G.J. Olsder, and J.-P. Quadrat. *Synchronization and linearity: an algebra for discrete event systems*. Wiley, 1992.

[12] F. Balarin, M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, C. Passerone, A. Sangiovanni-Vincentelli, E. Sentovich, K. Suzuki, and B. Tabbara, editors. *Hardware-Software Co-design of Embedded Systems: The POLIS Approach*. Kluwer Academic Publishers, June 1997.

[13] N. Bambha, V. Kianzad, M. Khandelia, and S.S. Bhattacharyya. Intermediate representations for design automation of multiprocessor DSP systems. *Design Automation for Embedded Systems*, 7(4):307–323, November 2002.

[14] R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, and P. Marwedel. Scratchpad memory: a design alternative for cache on-chip memory inembedded systems. In *10th International Conference on Hardware-Software Codesign, CODES 02, Proceedings*, pages 73–78. ACM, 2002.

[15] T. Basten, L. Benini, A. Chandrakasan, M. Lindwer, J. Liu, R. Min, and F. Zhao. Scaling into ambient intelligence. In *Conference on Design Automation and Test in Europe, DATE 03, Proceedings*, pages 76–81. IEEE, 2003.

[16] T. Basten, M.C.W. Geilen, and H.W.H. de Groot, editors. *Ambient Intelligence: Impact on Embedded System Design*. Kluwer Academic Publishers, November 2003.

[17] M. Bekooij, R. Hoes, O. Moreira, P. Poplavko, M. Pastrnak, B. Mesman, J.D. Mol, S. Stuijk, V. Gheorghita, and J. van Meerbergen. *Dynamic and Robust Streaming in and between Connected Consumer-Electronic Devices*, chapter Dataflow Analysis for Real-Time Embedded Multiprocessor System Design, pages 81–108. Springer, May 2005.

[18] M. Bekooij, O. Moreira, P. Poplavko, B. Mesman, M. Pastrnak, and J. van Meerbergen. Predictable multiprocessor system design. In *International Workshop on Software and Compilers for Embedded Systems, SCOPES 04, Proceedings*, pages 77–91. Springer, 2004.

[19] L. Benini and G. de Micheli. Networks on chips: A new SoC paradigm. *IEEE Computer*, 35(1):70–78, January 2002.

[20] L. Benini and G. De Micheli. *Multiprocessor Systems on Chips*, chapter Networks on Chip: A new Paradigm for component-based MPSoC Design, pages 49–80. Morgan Kaufmannn, 2004.

[21] S. Bhattacharyya, P. Murthy, and E.A. Lee. Synthesis of embedded software from synchronous dataflow specifications. *Journal on VLSI Signal Processing Systems*, 21(2):151–166, June 1999.

[22] S.S. Bhattacharyya. *Compiling Dataflow Programs for Digital Signal Processing*. PhD thesis, UC Berkeley, July 1994.

[23] S.S. Bhattacharyya, P.K. Murthy, and E.A. Lee. *Software Synthesis from Dataflow Graphs*. Kluwer Academic Publishers, 1996.

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

[25] S.Y. Borkar, H. Mulder, P. Dubey, S.S. Pawlowski, K.C. Kahn, J.R. Rattner, and D.J. Kuck. Platform 2015: Intel processor and platfrom evolution for the next decade. Technical report, Intel, 2005.

[26] R.J. Bril, C. Hentschel, E.F.M. Steffens, M. Gabrani, G. van Loo, and J.H.A. Gelissen. Multimedia QoS in consumer terminals. In *Workshop on Signal Processing Systems, Proceedings*, pages 332–343. IEEE, 2001.

[27] J.T. Buck. *Scheduling dynamic dataflow graphs with bounded memory using the token flow model*. PhD thesis, UC Berkeley, 1993.

[28] M. Coenen, S. Murali, A. Rădulescu, K. Goossens, and G. De Micheli. A buffer-sizing algorithm for networks on chip using tdma and credit-based end-to-end flow control. In *4th International Conference on Hardware-Software Codesign and System Synthesis, CODES+ISSS 06, Proceedings*, pages 130–135. ACM, 2006.

[29] F. Commoner, A.W. Holt, S. Even, and A. Pneuli. Marked directed graphs. *Journal of Computer and System Sciences*, 5(5):511–523, October 1971.

[30] D.E. Culler, J.P. Singh, and A. Gupta. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishers, 1999.

[31] W.J. Dally and C.L. Seitz. The torus routing chip. *Journal of distributed computing*, 1(4):187–196, December 1986.

[32] W.J. Dally and B. Towles. Route packets, not wires: on-chip interconnection networks. In *38th Design Automation Conference, DAC 01, Proceedings*, pages 684–689. ACM, 2001.

[33] A. Dasdan. Experimental analysis of the fastest optimum cycle ratio and mean algorithms. *ACM Transactions on Design Automation of Electronic Systems*, 9(4):385–418, October 2004.

[34] A. Dasdan and R.K. Gupta. Faster maximum and minimum mean cycle algorithms for system performance analysis. Technical Report ICS-TR-97-07, UC Irvine, 1997.

[35] R.P. Dick, D.L. Rhodes, and W. Wolf. TGFF: task graphs for free. In *International Conference on Hardware/Software Codesign, CODES 98, Proceedings*, pages 97–101. IEEE, 1998.

[36] S. Dutta, R. Jensen, and A. Rieckmann. Viper: A multiprocessor SoC for advanced set-top box and digital tv systems. *IEEE Design and Test of Computers*, 18(5):21–31, September 2001.

[37] C. Erbas, S. Cerav-Erbas, and A.D. Pimentel. Multiobjective optimization and evolutionary algorithms for the application mapping problem in multiprocessor system-on-chip design. *IEEE Transactions on Evolutionary Computation*, 10(3):358–374, June 2006.

[38] R. Ernst, J. Henkel, Th. Benner, W. Ye, U. Holtmann, D. Herrmann, and M. Trawny. The COSYMA environment for hardware/software cosynthesis of small embedded systems. *Microprocessors and Microsystems*, 20(3):159–166, May 1996.

[39] A. Ferrari and A. Sangiovanni-Vincentelli. System design: Traditional concepts and new paradigms. In *International Conference on Computer Design, ICCD 99, Proceedings*, pages 2–12. IEEE, 1999.

[40] O.P. Gangwal, A. Rădulescu, K. Goossens, S. González Pestana, and E. Rijpkema. *Dynamic and Robust Streaming In and Between Connected Consumer-Electronics Devices*, volume 3 of *Philips Research Book Series*, chapter Building Predictable Systems on Chip: An Analysis of Guaranteed Communication in the AEthereal Network on Chip, pages 1–36. Springer, 2005.

[41] E.R. Gansner and S.C. North. An open graph visualization system and its applications to software engineering. *Software: Practice and Experience*, 30(11):1203–1233, August 2000.

[42] M.R. Garey and D.S. Johnson. *Computers and interactability: a guide to the theory of NP-completeness*. W.H. Freeman and Co., 1979.

[43] M.C.W. Geilen and T. Basten. Reactive process networks. In *4th International Conference on Embedded Software, EMSOFT 04, Proceedings*, pages 137–146. ACM, 2004.

[44] M.C.W. Geilen, T. Basten, and S. Stuijk. Minimising buffer requirements of synchronous dataflow graphs with model-checking. In *42nd Design Automation Conference, DAC 05, Proceedings*, pages 819–824. ACM, 2005.

[45] A.H. Ghamarian, M.C.W. Geilen, T. Basten, B.D. Theelen, M.R. Mousavi, and S. Stuijk. Liveness and boundedness of synchronous data flow graphs. In *6th International Conference on Formal Methods in Computer Aided Design, FMCAD 06, Proceedings*, pages 68–75. IEEE, 2006.

[46] 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 *6th International Conference on Application of Concurrency to System Design, ACSD 06, Proceedings*, pages 25–36. IEEE, 2006.

[47] A.H. Ghamarian, S. Stuijk, T. Basten, M.C.W. Geilen, and B.D. Theelen. Latency minimization for synchronous data flow graphs. In *10th Euromicro Conference on Digital System Design, DSD 07, Proceedings*, pages 189–196. IEEE, 2007.

[48] S.V. Gheorghita, T. Basten, and H. Corporaal. Application scenarios in streaming-oriented embedded system design. In *International Symposium on System-on-Chip, SoC 06, Proceedings*, pages 175–178. IEEE, 2006.

[49] S.V. Gheorghita, T. Basten, and H. Corporaal. Profiling driven scenario detection and prediction for multimedia applications. In *6th International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation, IC-SAMOS 06, Proceedings*, pages 63–70. IEEE, 2006.

[50] S.V. Gheorghita, T. Basten, and H. Corporaal. Scenario selection and prediction for dvs-aware scheduling. *Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology*, July 2007. `http://dx.doi.org/10.1007/s11265-007-0086-1`.

[51] S.V. Gheorghita, S. Stuijk, T. Basten, and H. Corporaal. Automatic scenario detection for improved WCET estimation. In *42nd Design Automation Conference, DAC 05, Proceedings*, pages 101–104. ACM, 2005.

[52] C.J. Glass and L.M. Ni. The turn model for adaptive routing. In *19th International Symposium on Computer architecture, Proceedings*, pages 278–287. ACM, 1992.

[53] K. Goossens, J. Dielissen, O.P. Gangwal, S. González Pestana, A. Rădulescu, and E. Rijpkema. A design flow for application-specific networks on chip with guaranteed performance to accelerate SoC design and verification. In *Conference on Design Automation and Test in Europe, DATE 05, Proceedings*, pages 1182–1187. IEEE, 2005.

[54] R. Govindarajan, G.R. Gao, and P. Desai. Minimizing buffer requirements under rate-optimal schedule in regular dataflow networks. *Journal of VLSI Signal Processing*, 31(3):207–229, July 2002.

[55] M. Gries. Methods for evaluating and covering the design space during early design development. *Integration, the VLSI Journal*, 38(2):131–183, December 2004.

[56] P. Guerrier and A. Greiner. A generic architecture for on-chip packet-switched interconnections. In *Conference on Design Automation and Test in Europe, DATE 00, Proceedings*, pages 250–256. IEEE, 2000.

[57] U. Hansmann, M.S. Nicklous, and T. Stober. *Pervasive computing handbook*. Springer-Verlag, January 2001.

[58] A. Hansson, M. Coenen, and K. Goossens. Channel trees: Reducing latency by sharing time slots in time-multiplexed networks on chip. In *International Conference on Hardware/Software Codesign and System Synthesis, CODES+ISSS 07, Proceedings*. IEEE, 2005.

[59] A. Hansson, K. Goossens, and A. Rădulescu. A unified approach to constrained mapping and routing on network-on-chip architectures. In *International Conference on Hardware/Software Codesign and System Synthesis, CODES+ISSS 05, Proceedings*, pages 75–80. IEEE, 2005.

[60] R. Heckmann and C. Ferdinand. Verifying safety-critical timing and memory-usage properties of embedded software by abstract interpretation. In *Conference on Design Automation and Test in Europe, DATE 05, Proceedings*, pages 618–619, 2005.

[61] J.L. Hennessy and D.A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 4th edition, 2007.

[62] R. Holsmark, M. Palesi, and S. Kumar. Deadlock free routing algorithms for mesh topology NoC systems with regions. In *9th Euromicro Conference on Digital System Design, DSD 06, Proceedings*, pages 696–703. IEEE, 2006.

[63] N. Holsti and S. Saarinen. Status of the Bound-T WCET tool. In *2nd International Workshop on Worst-Case Execution Time Analysis, WCET 02, Proceedings*, pages 36–41, 2002.

[64] J. Hu and R. Marculescu. Communication and task scheduling of application-specific networks-on-chip. *IEE Proceedings: Computers and Digital Techniques*, 152(5):643–651, September 2005.

[65] J. Hu and R. Marculescu. Energy- and performance-aware mapping for regular NoC architectures. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 24(4):551–562, April 2005.

[66] J. Hu, U.Y. Ogras, and R. Marculescu. System-level buffer allocation for application-specific networks-on-chip router design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systemss*, 25(12):2919–2933, December 2006.

[67] C.-T. Hwang, J.-H. Lee, and Y.-C. Hsu. A formal approach to the scheduling problem in high-level synthesis. *IEEE Transactions on Computer-Aided Design*, 10(4):464–475, April 1991.

[68] In-Stat. Consumer electronics sales. `http://www.itfacts.biz/index.php?id=P8224`.

[69] ITRS. International technology roadmap for semiconductors 2005 edition. Technical report, ITRS, 2005.

[70] ITU-T. Video coding for low bit rate communication. Technical report, ITU-T Recommendation H.263, 1996.

[71] ITU-T. Advanced video coding for generic audiovisual services. Technical report, ITU-T Recommendation H.264, 2005.

[72] A.A. Jerraya, A.Bouchhima, and F. Pétrot. Programming models and HW-SW interfaces abstraction for multi-processor SoC. In *43th Design Automation Conference, DAC 06, Proceedings*, pages 280–285. ACM, 2006.

[73] A.A. Jerraya and W. Wolf, editors. *Multiprocessor Systems-on-Chip*. Elsevier, September 2005.

[74] 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):589–604, 2005.

[75] G. Kahn. The semantics of a simple language for parallel programming. In *Information Processing 74, IFIP 74, Proceedings*, pages 471–475. North-Holland, 1974.

[76] R.M. Karp. A characterization of the minimum cycle mean in a digraph. *Discrete Mathematics*, 23(3):309–311, September 1978.

[77] R.M. Karp and R.E. Miller. Properties of a model for parallel computations: Determinancy, termination, queueing. *SIAM Journal of Applied Mathematics*, 14(6):1390–1411, November 1966.

[78] K. Keutzer, S. Malik, A.R. Newton, J.M. Rabaey, and A. 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):1523–1543, December 2000.

[79] B. Kienhuis, E.F. Deprettere, K.A. Vissers, and P. van der Wolf. An approach for quantitative analysis of application-specific dataflow architectures. In *International Conference on Application-specific Systems, Architectures and Processors, ASAP 97, Proceedings*, pages 338–349. IEEE, 1997.

[80] D. Kim. *System-Level Specification and Cosimulation for Multimedia Embedded Systems*. PhD thesis, Seoul National University, February 2003.

[81] H. Kopetz and N. Suri. Compositional design of real-time system: A conceptual basis for the specification of linking interfaces. In *6th International Symposium on Object Oriented Real-Time Computing, ISORC 03, Proceedings*, pages 51–60. IEEE, 2003.

[82] K. Kuchcinski. Constraint-driven scheduling and resource assignment. *ACM Transactions on Design Automation of Electronic Systems*, 8(3):355–383, July 2003.

[83] K. Lagerstrom. Design and implementation of an MPEG-1 layer III audio decoder. Master's thesis, Chalmers University of Technology, Sweden, May 2001.

[84] R. Lauwereins, P. Wauters, M. Ade, and J.A. Peperstraete. Geometric parallelism and cyclo-static data flow in GRAPE-II. In *5th International Workshop on Rapid System Prototyping, Proceedings*, pages 90–107. IEEE, 1994.

[85] E.A. Lee. Consistency in dataflow graphs. *IEEE Transactions on Parallel and Distributed Systems*, 2(2):223–235, April 1991.

[86] E.A. Lee and D.G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on Computers*, 36(1):24–35, January 1987.

[87] Y. Li and W. Wolf. Hierarchical scheduling and allocation of multirate systems on heterogeneous multiprocessors. In *European Design and Test Conference, ED&TC 97, Proceedings*, pages 134–139. IEEE, 1997.

[88] H. De Man. System design challenges in the post-PC era. In *37th Design Automation Conference, DAC 00, Proceedings*, page x. ACM, 2000.

[89] P. Marchal, C. Wong, A. Prayati, N. Cossement, F. Catthoor, R. Lauwereins, D. Verkest, and H. DeMan. Dynamic memory oriented transformations in the MPEG4 IM1-Player on a low power platform. In *1st International Workshop on Power-Aware Computer Systems, PACS 00, Proceedings*, pages 40–50. Springer-Verlag, 2000.

[90] R. Marculescu, U.Y. Ogras, and N.H. Zamora. Computation and communication refinement for multiprocessor SoC design: A system-level perspective. *ACM Transactions on Design Automation of Electronic Systems*, 11(3):564–592, July 2006.

[91] G. Martin. Overview of the MPSoC design challenge. In *43th Design Automation Conference, DAC 06, Proceedings*, pages 274–279. ACM, 2006.

[92] M. Matsumoto and T. Nishimura. Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation*, 8(1):3–30, January 1998.

[93] M. Millberg, E. Nilsson, R. Thid, and A. Jantsch. Guaranteed bandwidth using looped containers in temporally disjoint networks within the Nostrum network on chip. In *Conference on Design Automation and Test in Europe, DATE 04, Proceedings*, pages 890–895. IEEE, 2004.

[94] A. Moonen, M. Bekooij, and J. van Meerbergen. Timing analysis model for network based multiprocessor systems. In *15th annual Workshop of Circuits, System and Signal Processing, ProRISC 04, Proceedings*, pages 91–99. STW, 2004.

[95] O. Moreira, J.-D. Mol, M. Bekooij, and J. van Meerbergen. Multiprocessor resource allocation for hard-real-time streaming with a dynamic job-mix. In *11th Real Time and Embedded Technology and Applications Symposium, RTAS 05, Proceedings*, pages 332–341. IEEE, 2005.

[96] MPEG-2. Generic coding of moving pictures and associated audio. Technical report, ISO/IEC 13818-2.

[97] MPEG-4. Information technology coding of audio-visual objects. Technical report, ISO/IEC 14496-2.

[98] S. Murali, L. Benini, and G. De Micheli. An application-specific design methodolofy for on-chip crossbar generation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 26(7):1283–1296, July 2007.

[99] S. Murali, M. Coenen, A. Rădulescu, K. Goossens, and G. De Micheli. A methodology for mapping multiple use-cases onto networks on chip. In *Conference on Design Automation and Test in Europe, DATE 06, Proceedings*, pages 118–123. European Design and Automation Association, 2006.

[100] P.K. Murthy. *Scheduling Techniques for Synchronous Multidimensional Synchronous Dataflow*. PhD thesis, UC Berkeley, December 1996.

[101] P.K. Murthy and S.S. Bhattacharyy. Shared memory implementations of synchronous dataflow specifications. In *Conference on Design Automation and Test in Europe, DATE 00, Proceedings*, pages 404–410. IEEE, 2000.

[102] P.K. Murthy and E.A. Lee. On the optimal blocking factor for blocked, non-overlapped multiprocessor schedules. In *28th Asilomar Conference on Signals, Systems and Computers, Proceedings*, pages 1052–1057. IEEE, 1994.

[103] C. Neeb and N. Wehn. Designing efficient irregular networks for heterogeneous systems-on-chip. In *9th Euromicro Conference on Digital System Design, DSD 06, Proceedings*, pages 665–672. IEEE, 2006.

[104] Q. Ning and G.R. Gao. A novel framework of register allocation for software pipelining. In *20th Symposium on Principles of Programming Languages, Proceedings*, pages 29–42. ACM, 1993.

[105] U.Y. Ogras, J. Hu, and R. Marculescu. Key research problems in NoC design: A holistic perspective. In *International conference on Hardware/software codesign and system synthesis, CODES+ISSS 05, Proceedings*, pages 69–74. ACM, 2005.

[106] U.Y. Ogras and R. Marculescu. "it's a small world after all": NoC performance optimization via long-range link insertion. *IEEE Transactions on Very Large Scale Integration Systems*, 14(7):693–706, July 2006.

[107] U.Y. Ogras and R. Marculescu. Prediction-based flow control for network-on-chip traffic. In *43th Design Automation Conference, DAC 06, Proceedings*, pages 839–844. ACM, 2006.

[108] H. Oh, N. Dutt, and S. Ha. Memory optimal single appearance schedule with dynamic loop count for synchronous dataflow graphs. In *Conference on Asia South Pacific Design Automation, ASP-DAC 06, Proceedings*, pages 497–502. ACM, 2006.

[109] H. Oh and S. Ha. Efficient code synthesis from extended dataflow graphs. In *39th Design Automation Conference, DAC 02, Proceedings*, pages 275–280. ACM, 2002.

[110] H. Oh and S. Ha. Fractional rate dataflow model for efficient code synthesis. *Journal of VLSI Signal Processing*, 37(1):41–51, May 2004.

[111] P.G. Paulin, C. Pilkington, E. Bensoudane, M. Langevin, and D. Lyonnard. Application of a multi-processor SoC platform to high-speed packet forwarding. In *Conference on Design Automation and Test in Europe, DATE 04, Proceedings*, pages 58–63. IEEE, 2004.

[112] C.A. Petri. *Kommunikation mit automaten*. PhD thesis, Institute für instrumentelle Mathematik, 1962.

[113] A.D. Pimentel, C. Erbas, and S. Polstra. A systematic approach to exploring embedded system architectures at multiple abstraction levels. *IEEE Transactions on Computers*, 55(2):99–112, February 2006.

[114] J.L. Pino, S.S. Bhattacharyya, and E.A. Lee. A hierarchical multiprocessor scheduling system for DSP applications. In *29th Asilomar Conference on Signals, Systems and Computers, Proceedings*, pages 122–126. IEEE, 1995.

[115] B. Plateau and K. Atif. Stochastic automata network of modeling parallel systems. *IEEE Transactions on Software Engineering*, 17(10):1093–1108, October 1991.

[116] P. Poplavko, T. Basten, M. Bekooij, J. van Meerbergen, and B. Mesman. Task-level timing models for guaranteed performance in multiprocessor networks-on-chip. In *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems, CASES 03, Proceedings*, pages 63–72. ACM, 2003.

[117] W. Reisig and G. Rozenberg. *Advances in Petri Nets*, volume 1491 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.

[118] R. Reiter. Scheduling parallel computations. *Journal of the ACM*, 15(4):590–599, October 1968.

[119] Telenor Research. Tmn (h.263) encoder/decoder, version 1.7, June 1997.

[120] E. Rijpkema, K.G.W. Goossens, A. Rădulescu, 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*, 150(5):294–302, September 2003.

[121] S. Ritz, M. Willems, and H. Meyr. Scheduling for optimum data memory compaction in block diagramoriented software synthesis. In *International Conference on Acoustics, Speech, and Signal Processing, ICASSP 95, Proceedings*, pages 2651–2654. IEEE, 1995.

[122] A. Rădulescu, J. Dielissen, K. Goossens, E. Rijpkema, and P. Wielage. An efficient on-chip network interface offering guaranteed services shared-memory abstraction, and flexible network configuration. In *Conference on Design Automation and Test in Europe, DATE 04, Proceedings*, pages 878–883. IEEE, 2004.

[123] A. Rădulescu, J. Dielissen, S. González Pestana, O.P. Gangwal, E. Rijpkema, P. Wielage, and K. 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):4–17, January 2005.

[124] M. Ruggiero, A. Guerri, D. Bertozzi, F. Poletti, and M. Milano. Communication-aware allocation and scheduling framework for stream-oriented multi-processor systems-on-chip. In *Conference on Design Automation and Test in Europe, DATE 06, Proceedings*, pages 3–8. European Design and Automation Association, 2006.

[125] M.J. Rutten, J.T.J. van Eijndhoven, E.G.T. Jaspers, P. van der Wolf, O.P. Gangwal, A. Timmer, and E.-J.D. Pol. A heterogeneous multiprocessor architecture for flexible media processing. *IEEE Design & Test of Computers*, 19(4):39–50, July 2002.

[126] K. Salomonsen. Design and implementation of an MPEG/Audio layer III bitstream processor. Master's thesis, Aalborg University, Denmark, 1997.

[127] S.K. Shukla and R.K. Gupta. A model checking approach to evaluating system level dynamic power management policies for embedded systems. In *6th High-Level Design Validation and Test Workshop, HLDVT 01, Proceedings*, pages 53–57. IEEE, 2001.

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

[129] S. Sriram and E.A. Lee. Determining the order of processor transactions in statically scheduled multiprocessors. *Journal of VLSI Signal Processing*, 15(3):207–220, March 1997.

[130] T. Stefanov, C. Zissulescu, A. Turjan, B. Kienhuis, and E. Deprette. System design using Khan process networks: the Compaan/laura approach. In *Conference on Design Automation and Test in Europe, DATE 04, Proceedings*, pages 340–345. IEEE, 2004.

[131] S. Steinke, L. Wehmeyer, L. Bo-Sik, and P. Marwedel. Assigning program and data objects to scratchpad for energyreduction. In *Conference on Design Automation and Test in Europe, DATE 02, Proceedings*, pages 409–415. IEEE, 2002.

[132] S. Stuijk, T. Basten, M.C.W. Geilen, and H. Corporaal. Multiprocessor resource allocation for throughput-constrained synchronous dataflow graphs. In *44th Design Automation Conference, DAC 07, Proceedings*, pages 777–782. ACM, 2007.

[133] S. Stuijk, T. Basten, M.C.W. Geilen, A.H. Ghamarian, and B.D. Theelen. Resource-efficient routing and scheduling of time-constrained streaming communication on networks-on-chip. *Journal of Systems Architecture*. (to be published).

[134] S. Stuijk, T. Basten, M.C.W. Geilen, A.H. Ghamarian, and B.D. Theelen. Resource-efficient routing and scheduling of time-constrained network-on-chip communication. In *9th Euromicro Conference on Digital System Design, DSD 06, Proceedings*, pages 45–52. IEEE, 2006.

[135] S. Stuijk, T. Basten, B. Mesman, and M.C.W. Geilen. Predictable embedding of large data structures in multiprocessor networks-on-chip. In *8th Euromicro Conference on Digital System Design, DSD 05, Proceedings*, pages 388–395. IEEE, 2005.

[136] S. Stuijk, T. Basten, B. Mesman, and M.C.W. Geilen. Predictable embedding of large data structures in multiprocessor networks-on-chip (extended abstract). In *Conference on Design Automation and Test in Europe, DATE 05, Proceedings*, pages 254–255. IEEE, 2005.

[137] S. Stuijk, M.C.W. Geilen, and T. Basten. Exploring trade-offs in buffer requirements and throughput constraints for synchronous dataflow graphs. In *43th Design Automation Conference, DAC 06, Proceedings*, pages 899–904. ACM, 2006.

[138] S. Stuijk, M.C.W. Geilen, and T. Basten. SDF^3: SDF For Free. In *6th International Conference on Application of Concurrency to System Design, ACSD 06, Proceedings*, pages 276–278. IEEE, 2006.

[139] E. Teruel, J. Chrzastowski-Wachtel, M. Colom, and M. Silva. On weighted T-systems. In *13th International Conference on Application and Theory of Petri Nets, APN 92, Proceedings*, pages 348–367. Springer, 1992.

[140] 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 *4th International Conference on Formal Methods and Models for Co-Design, MEMOCODE 06, Proceedings*, pages 185–194. IEEE, 2006.

[141] R. Thid, I. Sander, and A. Jantsch. Flexible bus and NoC performance analysis with configurable synthetic workloads. In *9th Euromicro Conference on Digital System Design, DSD 06, Proceedings*, pages 681–688. IEEE, 2006.

[142] W. Thies, M. Karczmarek, and S. Amarasinghe. Streamit: A language for streaming applications. In *11th International Symposium on Compiler Construction, CC 02, Proceedings, volume 2304 in LNCS*, pages 179–196. Springer-Verlag, 2002.

[143] S. Udayakumaran and R. Barua. Compiler-decided dynamic memory allocation for scratch-pad based embedded systems. In *International Conference on Compilers, Architectures and Synthesis for Embedded Systems, CASES 03, Proceedings*, pages 276–286. ACM, 2003.

[144] S. Udayakumaran, A. Dominguez, and R. Barua. Dynamic allocation for scratch-pad memory using compile-time decisions. *Transactions on Embedded Computing Systems*, 5(2):472–511, May 2006.

[145] F. Vaandrager. *Lectures on Embedded Systems, LNCS 1494*, chapter Introduction, pages 1–3. Springer-Verlag, 1998.

[146] M. Verma, L. Wehmeyer, and P. Marwedel. Dynamic overlay of scratchpad memory for energy minimization. In *2nd International Conference on Hardware/software codesign and system synthesis, CODES+ISSS, Proceedings*, pages 104–109. ACM, 2004.

[147] Ogg Vorbis. Ogg vorbis rc3. Technical report, Xiph.org.

[148] L. Wehmeyer and P. Marwedel. Influence of memory hierarchies on predictability for time constrained embedded software. In *Conference on Design Automation and Test in Europe, DATE 05, Proceedings*, pages 600–605. IEEE, 2005.

[149] M. Weiser. Ubiquitous computing. *Computer*, 26(10):71–72, October 1993.

[150] M. Wiggers, M. Bekooij, P. Jansen, and G. Smit. Efficient computation of buffer capacities for multi-rate real-time systems with back-pressure. In *4th International Conference on Hardware-Software Codesign and System Synthesis, CODES+ISSS 06, Proceedings*, pages 10–15. ACM, 2006.

[151] M. Wiggers, M. Bekooij, and G. Smit. Efficient computation of buffer capacities for cyclo-static dataflow graphs. In *44th Design Automation Conference, DAC 07, Proceedings*, pages 658–663. ACM, 2007.

[152] W. Wolf. Multimedia applications of multiprocessor systems-on-chips. In *Conference on Design Automation and Test in Europe, DATE 05, Proceedings*, pages 86–89. IEEE, 2005.

Glossary

Acronyms and abbreviations

API	application program interface
BDF	boolean dataflow
BDFG	boolean dataflow graph
CA	communication assist
CIF	common intermediate format
CSDF	cyclo-static dataflow
CSDFG	cyclo-static dataflow graph
DCT	discrete cosine transform
DSP	digital signal processing
FPGA	field-programmable gate array
HSDFG	homogeneous synchronous dataflow graph
IDCT	inverse discrete cosine transform
IQ	inverse quantization
ITRS	international technology roadmap for semiconductors
KPN	Kahn process network
MB	macro block
MCM	maximum cycle mean
MCR	maximum cycle ratio
M	memory
MoC	model-of-computation
MPEG	motion pictures experts group
MP-SoC	multi-processor system-on-chip
NI	network interface
NoC	network-on-chip
P	processor
QCIF	quarter common intermediate format
RPN	reactive process network
R	router
SADF	scenario-aware dataflow
SADFG	scenario-aware dataflow graph

SDF	synchronous dataflow
SDFG	synchronous dataflow graph
TDMA	time-division multiple-access
VLC	variable length encoder
VLD	variable length decoder
WCRT	worst-case response time
XML	extensible markup language

Symbols and Notations

The list of symbols and notations is split into three categories. The first category contains the symbols and notations used for SDFGs. The second category contains symbols and notations used for tile binding and scheduling. This includes symbols and notations used to describe the resource requirements of SDFGs. The last category contains the symbols and notations used in the communication scheduling.

Synchronous Dataflow

a	actor
d	dependency edge
p	port
A	set of actors
D	set of dependency edges
$Ports$	set of ports
I	set of input ports
O	set of output ports
$Rate(p)$	rate of port p
$SrcA(d)$	source actor of a dependency edge d
$DstA(d)$	destination actor of a dependency edge d
$SrcP(d)$	source port of a dependency edge d
$DstP(d)$	destination port of a dependency edge d
$InD(a)$	set of all dependency edges connected to input ports of actor a
$OutD(a)$	set of all dependency edges connected to output ports of actor a
$Rd(a)$	number of tokens read by actor a from its edges on a firing
$Wr(a)$	number of tokens written by actor a to its edges on a firing
γ	repetition vector
δ	edge quantity which gives the distribution of tokens or storage space over the dependency edges
n	number of tokens on a dependency edge in the initial state
υ	set of remaining execution times of all actors in a given state

$\Upsilon(a)$	execution time of actor a
Th	throughput
λ	throughput constraint
σ	self-timed execution of an SDFG
Δ	(abstract) dependency graph
$step(d)$	step size of dependency edge d

Resource Allocation

t	tile in a platform graph
T	set of tiles in a platform graph
C	set of connections between tiles in a platform graph
$\mathcal{L}(c)$	latency of a connection c in the platform graph
pt	processor type
PT	set of all processor types
w	size of the processor's TDMA time wheel
$\Omega(t)$	time slice reserved by other applications on tile t
m_t	size of the memory inside tile t
c_t	number of connections supported by tile t
i_t	incoming bandwidth of tile t
o_t	outgoing bandwidth of tile t
$\Gamma(a, pt)$	resource requirements of actor a on processor type pt ($\Gamma(a, pt) = (\tau_{a,pt}, \mu_{a,pt})$)
$\tau_{a,pt}$	execution time of actor a on processor type pt
$\mu_{a,pt}$	memory requirement of actor a on processor type pt
$\Theta(d)$	resource requirements of dependency edge d ($\Theta(d) = (sz, \alpha_{tile}, \alpha_{src}, \alpha_{dst}, \rho, \beta)$)
sz	size of a token on dependency edge d
α_{tile}	storage-space constraint on a dependency edge when this edge is bound to a tile
α_{src}	storage-space constraint on the source tile when the dependency edge is bound to a connection in the platform graph
α_{dst}	storage-space constraint on the destination tile when the dependency edge is bound to a connection in the platform graph
ρ	latency constraint on a dependency edge when this edge is bound to a connection in the platform graph
β	bandwidth constraint on a dependency edge when this edge is bound to a connection in the platform graph
$\mathcal{B}(a) = t$	function gives binding of actor a to tile t
A_t	set of actors bound to tile t
$D_{t,src}$	set of dependency edges of which only the source port is bound to tile t

$D_{t,dst}$	set of dependency edges of which only the destination port is bound to tile t
$D_{t,tile}$	set of dependency edges of which both the source and destination port are bound to tile t
$\mathcal{S}(t) = (\omega_t, S_t)$	function gives schedule on tile t
ω_t	size of the TDMA time slice reserved for the application on tile t
S_t	static-order schedule on tile t
$\varrho(S_t, \kappa)$	the function gives the next position in a static-order schedule
ι	position of the processor's time wheels in a given state
κ	position of the static-order schedules in a given state

Communication Scheduling

u, v	vertex in the interconnect graph
V	set of vertexes in the interconnect graph
l_i	link l_i in the interconnect graph
L	set links in the interconnect graph
N	number of slots in the TDMA tables of the links
sz_{flit}	size of a flit
sz_{ph}	size of the packet header
T_{reconf}	time needed to reconfigure a slot in a NI
CS	set of communication scenarios
P_i	period of the communication scenario i
M	set of messages
m	message ($m = (u, v, s, n, \tau, \delta, sz)$)
s	stream identifier
n	sequence number of message in stream
τ	earliest start time of a message
δ	maximal duration of a message
sz	size of a message
E	set of scheduling entities
e	scheduling entity ($e = (t, d, r, st)$)
t	time-stamp
d	duration of a scheduling entity
r	the route used by a scheduling entity
st	the set of slots used by a scheduling entity
$\varphi(e)$	number of slots used by scheduling entity e
$\phi(e)$	number of reserved slots by scheduling entity e
$\pi(e)$	number of packets send by scheduling entity e
$\sigma(e, l_i, t)$	function indicates whether scheduling entity e uses the link l_i at time t

$\mathcal{U}(l_i, t)$	function indicates whether link l_i is used at time t by other communication scenarios
$\mathcal{S}(m) = e$	scheduling function assigns scheduling entity e to message m
R	set of routes
X	maximum detour

Samenvatting

Het ontwerp van nieuwe consumentenelektronica wordt voortdurend complexer omdat er steeds meer functionaliteit in deze apparaten geïntegreerd wordt. Een voorspelbaar ontwerptraject is nodig om deze complexiteit te beheersen. Het resultaat van dit ontwerptraject zou een systeem moeten zijn, waarin iedere applicatie zijn eigen taken binnen een strikte tijdslimiet kan uitvoeren, onafhankelijk van andere applicaties die hetzelfde systeem gebruiken. Dit vereist dat het tijdsgedrag van de hardware, de software, evenals hun interactie kan worden voorspeld.

Er wordt vaak voorgesteld om een heterogeen multi-processor systeem (MP-SoC) te gebruiken in moderne elektronische systemen. Een MP-SoC heeft voor veel applicaties een goede verhouding tussen rekenkracht en energiegebruik. On-chip netwerken (NoCs) worden voorgesteld als interconnect in deze systemen. Een NoC is schaalbaar en het biedt garanties wat betreft de hoeveelheid tijd die er nodig is om gegevens te communiceren tussen verschillende processoren en geheugens. Door het NoC te combineren met een voorspelbare strategie om de processoren en geheugens te delen, ontstaat een hardware platform met een voorspelbaar tijdsgedrag. Om een voorspelbaar systeem te verkrijgen moet ook het tijdsgedrag van een applicatie die wordt uitgevoerd op het platform voorspelbaar en analyseerbaar zijn. Het Synchronous Dataflow (SDF) model is erg geschikt voor het modelleren van applicaties die werken met gegevensstromen. Het model kan vele ontwerpbeslissingen modelleren en het is mogelijk om tijdens het ontwerptraject het tijdsgedrag van het systeem te analyseren. Dit proefschrift probeert om applicaties die gemodelleerd zijn met SDF grafen op een zodanige manier af te beelden op een NoC-gebaseerd MP-SoC, dat garanties op het tijdsgedrag van individuele applicaties gegeven kunnen worden.

De doorstroomsnelheid van een applicatie is vaak een van de belangrijkste eisen bij het ontwerpen van systemen voor applicaties die werken met gegevensstromen. Deze doorstroomsnelheid wordt in hoge mate beïnvloed door de beschikbare ruimte om resultaten (gegevens) op te slaan. De opslagruimte in een SDF graaf wordt gemodelleerd door de pijlen in de graaf. Het probleem is dat er een vaste grootte voor de opslagruimte aan de pijlen van een SDF graaf moet worden toegewezen. Deze grootte moet zodanig worden gekozen dat de vereiste doorstroomsnelheid van het systeem gehaald wordt, terwijl de benodigde opslagruimte geminimaliseerd

wordt. De eerste belangrijkste bijdrage van dit proefschrift is een techniek om de minimale opslagruimte voor iedere mogelijke doorstroomsnelheid van een applicatie te vinden. Ondanks de theoretische complexiteit van dit probleem presteert de techniek in praktijk goed. Doordat de techniek alle mogelijke minimale combinaties van opslagruimte en doorstroomsnelheid vindt, is het mogelijk om met situaties om te gaan waarin nog niet alle ontwerpbeslissingen zijn genomen. De ontwerpbeslissingen om twee taken van een applicatie op één processor uit te voeren, zou bijvoorbeeld de doorstroomsnelheid kunnen beïnvloeden. Hierdoor is er een onzekerheid in het begin van het ontwerptraject tussen de berekende doorstroomsnelheid en de doorstroomsnelheid die daadwerkelijk gerealiseerd kan worden als alle ontwerpbeslissingen zijn genomen.

Tijdens het ontwerptraject moeten de taken waaruit een applicatie is opgebouwd toegewezen worden aan de verschillende processoren en geheugens in het systeem. Indien meerdere taken een processor delen, moet ook de volgorde bepaald worden waarin deze taken worden uitgevoerd. Een belangrijke bijdrage van dit proefschrift is een techniek die deze toewijzing uitvoert en die de volgorde bepaalt waarin taken worden uitgevoerd. Bestaande technieken kunnen alleen omgaan met taken die een één-op-één relatie met elkaar hebben, dat wil zeggen, taken die een gelijk aantal keren uitgevoerd worden. In een SDF graaf kunnen ook complexere relaties worden uitgedrukt. Deze relaties kunnen omgeschreven worden naar een één-op-één relatie, maar dat kan leiden tot een exponentiële groei van het aantal taken in de graaf. Hierdoor kan het onmogelijk worden om in een beperkte tijd alle taken aan de processoren toe te wijzen en om de volgorde te bepalen waarin deze taken worden uitgevoerd. De techniek die in dit proefschrift wordt gepresenteerd, kan omgaan met de complexe relaties tussen taken in een SDF graaf zonder de vertaling naar de één-op-één relaties te maken. Dit is mogelijk dankzij een nieuwe, efficiënte techniek om de doorstroomsnelheid van SDF grafen te bepalen.

Nadat de taken van een applicatie toegewezen zijn aan de processoren in het hardware platform moet de communicatie tussen deze taken op het NoC gepland worden. In deze planning moet voor ieder bericht dat tussen de taken wordt verstuurd, worden bepaald welke route er gebruikt wordt en wanneer de communicatie gestart wordt. Dit proefschrift introduceert drie strategieën voor het versturen van berichten met een strikte tijdslimiet. Alle drie de strategieën maken maximaal gebruik van de beschikbare vrijheid die moderne NoCs bieden. Experimenten tonen aan dat deze strategieën hierdoor efficiënter omgaan met de beschikbare hardware dan bestaande strategieën. Naast deze strategieën wordt er een techniek gepresenteerd om uit de ontwerpbeslissingen die gemaakt zijn tijdens het toewijzen van taken aan de processoren alle tijdslimieten af te leiden waarbinnen de berichten over het NoC gecommuniceerd moeten worden. Deze techniek koppelt de eerder genoemde techniek voor het toewijzen van taken aan processoren aan de drie strategieën om berichten te versturen over het NoC.

Tenslotte worden de verschillende technieken die in dit proefschrift worden geïntroduceerd gecombineerd tot een compleet ontwerptraject. Het startpunt

is een SDF graaf die een applicatie modelleert en een NoC-gebaseerd MP-SoC platform met een voorspelbaar tijdsgedrag. Het doel van het ontwerptraject is het op een zodanige manier afbeelden van de applicatie op het platform dat de doorstroomsnelheid van de applicatie gegarandeerd kan worden. Daarnaast probeert het ontwerptraject de hoeveelheid hardware die gebruikt wordt te minimaliseren. Er wordt een experiment gepresenteerd waarin drie verschillende multimedia applicaties (H.263 encoder/decoder en een MP3 decoder) op een NoC-gebaseerd MP-SoC worden afgebeeld. Dit experiment toont aan dat de technieken die in dit proefschrift worden voorgesteld, gebruikt kunnen worden voor het ontwerpen van systemen met een voorspelbaar tijdsgedrag. Hiermee is het voorgestelde ontwerptraject het eerste traject dat een met een SDF-gemodelleerde applicatie op een NoC-gebaseerd MP-SoC kan afbeelden, terwijl er garanties worden gegeven over de doorstroomsnelheid van de applicatie.

Acknowledgments

This thesis would not have reached its current form without the guidance and support from many people. At the end of this work, I would like to express my sincere gratitude to all those who supported me while working on this PhD project.

In the past few years I have had the privilege of being coached by Twan Basten. His support and encouraging style of working have constituted an eminent incentive to complete this project. He invested a lot of time and effort in reading and commenting on all the manuscripts that I produced during my PhD project. These comments were of enormous value for the quality of the resulting publications. Furthermore, they often sparked ideas for new research directions. Twan also initiated the weekly PROMES meetings in which we discussed together with AmirHossein Ghamarian, Bart Theelen and Marc Geilen many new ideas around the PhD topics of Amir and me. These meetings were always very useful as they stimulated an intense discussion on the problems at hand and often led to new insights that could be used in my research.

I would like to thank Jef van Meerbergen and Marco Bekooij for inviting me to the Hijdra meetings at Philips research. These meetings increased my understanding of the problems that designers are facing when developing novel embedded multimedia systems. The discussion that we had during those meetings helped in defining the research problem that is addressed in this thesis.

Also, I would like to thank my promoter, Henk Corporaal, for the useful discussions, difficult questions and critical comments that I received throughout my PhD project. My thanks also extend to Jef van Meerbergen for being my second promoter, and Axel Jantsch, Radu Marculescu and Ralph Otten for being part of the PhD core-committee. Your thorough review of, and constructive comments on, the draft version of this thesis were very helpful. Also Kees Goossens is thanked for being part of the PhD committee.

The last few years I had the pleasure to work in the electronic systems group. I really enjoyed the nice atmosphere and discussions that we had over the coffee

breaks and lunches. I would like to thank all members of the electronic systems group for the great time we had together.

I would also like to thank my family and friends for their interest in my project and the much needed relaxation. I would especially like to thank my parents without whom I would not have been able to achieve this result.

Sander Stuijk
September 2007

Curriculum Vitae

Sander Stuijk was born in Breda, The Netherlands, on August 3, 1979. After finishing the "Atheneum" (secondary school) at the Onze Lieve Vrouwe Lyceum in Breda in 1997, he started studying electrical engineering at the Eindhoven University of Technology in Eindhoven. The research for his M.Sc. thesis was concerned with the analysis of concurrency in multimedia applications. He received his M.Sc. degree with honors in 2002.

In October 2002, he started working towards a Ph.D. degree within the electronic systems group at the department of electrical engineering of the Eindhoven University of Technology. His research was funded by the NWO within the PROMES project. It has led among others to several publications and this thesis.

Sander is currently a post-doc researcher, continuing his research in the electronic systems group at the electrical engineering department of the Eindhoven University of Technology.

List of Publications

First author

- S. Stuijk, T. Basten, M.C.W. Geilen, A.H. Ghamarian, and B.D. Theelen. Resource-efficient routing and scheduling of time-constrained streaming communication on networks-on-chip. In *Journal of Systems Architecture*, Elsevier, (to be published).

- S. Stuijk and T. Basten. Analyzing Concurrency in Streaming Applications. In *Journal of Systems Architecture*, Elsevier, `http://dx.doi.org/10.1016/j.sysarc.2007.05.002`, (published online).

- S. Stuijk, T. Basten, M.C.W. Geilen and H. Corporaal. Multiprocessor resource allocation for throughput-constrained synchronous dataflow graphs. In *44th Design Automation Conference, DAC 07, Proceedings*, pages 777–782. ACM, 2007.

- S. Stuijk, T. Basten, M.C.W. Geilen, A.H. Ghamarian and B.D. Theelen. Resource-efficient routing and scheduling of time-constrained network-on-chip communication. In *9th Euromicro Conference on Digital System Design, DSD 06, Proceedings*, pages 45–52. IEEE, 2006.

- S. Stuijk, M.C.W. Geilen and T. Basten. Exploring trade-offs in buffer requirements and throughput constraints for synchronous dataflow graphs. In *43th Design Automation Conference, DAC 06, Proceedings*, pages 899–904. ACM, 2006.

- S. Stuijk, M.C.W. Geilen and T. Basten. SDF^3: SDF For Free. In *6th International Conference on Application of Concurrency to System Design, ACSD 06, Proceedings*, pages 276–278. IEEE, 2006.

- S. Stuijk, T. Basten, B. Mesman and M.C.W. Geilen. Predictable embedding of large data structures in multiprocessor networks-on-chip. In *8th Euromicro Conference on Digital System Design, DSD 05, Proceedings*, pages 388–395. IEEE, 2005.

- S. Stuijk, T. Basten, B. Mesman and M.C.W. Geilen. Predictable embedding of large data structures in multiprocessor networks-on-chip (extended abstract). In *Conference on Design Automation and Test in Europe, DATE 05, Proceedings*, pages 254–255. IEEE, 2005.

- S. Stuijk and T. Basten. Analyzing Concurrency in Computational Networks (extended abstract). In *1st International Conference on Formal Methods and Models for Codesign, MEMOCODE 03, Proceedings*, pages 47–48. IEEE, 2003.

- S. Stuijk, T. Basten and J. Ypma. CAST - A Task-Level Concurrency Analysis Tool (extended abstract). In *3rd International Conference on Application of Concurrency to System Design, ACSD 03, Proceedings*, pages 237–238. IEEE, 2003.

Co-author

- A.H. Ghamarian, S. Stuijk, T. Basten, M.C.W. Geilen, and B.D. Theelen. Latency minimization for synchronous data flow graphs. In *10th Euromicro Conference on Digital System Design, DSD 07, Proceedings*, pages 189–196. IEEE, 2007.

- A.H. Ghamarian, M.C.W. Geilen, T. Basten, B.D. Theelen, M.R. Mousavi, and S. Stuijk. Liveness and boundedness of synchronous data flow graphs. In *6th International Conference on Formal Methods in Computer Aided Design, FMCAD 06, Proceedings*, pages 68–75. IEEE, 2006.

- 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 *4th International Conference on Formal Methods and Models for Co-Design, MEMOCODE 06, Proceedings*, pages 185–194. IEEE, 2006.

- 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 *6th International Conference on Application of Concurrency to System Design, ACSD 06, Proceedings*, pages 25–36. IEEE, 2006.

- M. Pasternak, P.H.N. de With, S. Stuijk and J. van Meerbergen. Parallel Implementation of Arbitrary-Shaped MPEG-4 Decoder for Multiprocessor Systems. In *International Conference on Visual Communications and Image Processing, VCIP 06, Proceedings*, pages 60771I-1 - 60771I-10. IST Society for Imaging Science and Technology, 2006.

- M.C.W. Geilen, T. Basten, and S. Stuijk. Minimising buffer requirements of synchronous dataflow graphs with model-checking. In *42nd Design Automation Conference, DAC 05, Proceedings*, pages 819–824. ACM, 2005.

- S.V. Gheorghita, S. Stuijk, T. Basten, and H. Corporaal. Automatic scenario detection for improved WCET estimation. In *42nd Design Automation Conference, DAC 05, Proceedings*, pages 101–104. ACM, 2005.

- M. Bekooij, R. Hoes, O. Moreira, P. Poplavko, M. Pastrnak, B. Mesman, J.D. Mol, S. Stuijk, V. Gheorghita, and J. van Meerbergen. *Dynamic and Robust Streaming in and between Connected Consumer-Electronic Devices*, chapter Dataflow Analysis for Real-Time Embedded Multiprocessor System Design, pages 81–108. Springer, May 2005.