Model-driven scheduling of dynamic streaming applications on MPSoCs

Model-Driven Scheduling of Dynamic Streaming Applications on MPSoCs

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 24 oktober 2013 om 16.00 uur

door

Morteza Damavandpeyma

geboren te Roodbar, Iran

Dit proefschrift is goedgekeurd door de promotoren:

prof.dr. H. Corporaal
en
prof.dr.ir. T. Basten

Copromotor:
dr.ir. S. Stuijk

CIP-DATA LIBRARY TECHNISCHE UNIVERSITEIT EINDHOVEN

Damavandpeyma, Morteza

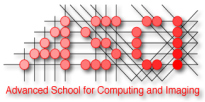
Model-Driven Scheduling of Dynamic Streaming Applications on MPSoCs
/ by Morteza Damavandpeyma. - Eindhoven : Technische Universiteit Eindhoven, 2013.
A catalogue record is available from the Eindhoven University of Technology Library
ISBN: 978-90-386-3460-9
NUR 959
Trefw.: multiprogrammeren / elektronica ; ontwerpen / multiprocessoren /
ingebedde systemen.
Subject headings: data flow graphs / electronic design automation /
multiprocessing systems / embedded systems.

Model-Driven Scheduling of Dynamic Streaming Applications on MPSoCs

Committee:

prof.dr. H. Corporaal (promotor, TU Eindhoven)
prof.dr.ir T. Basten (promotor, TU Eindhoven)
dr.ir. S. Stuijk (copromotor, TU Eindhoven)
prof.dr.ir. A.C.P.M. Backx (chairman, TU Eindhoven)
prof.dr.ir. M.J.G. Bekooij (University of Twente, NXP semiconductors)
prof.dr.ir. C.H. van Berkel (TU Eindhoven, ST Ericsson)
prof.dr.-Ing. C. Haubelt (University of Rostock)
dr. A.D. Pimentel (University of Amsterdam)

The work in this thesis is supported by the Technology Foundation STW, applied science division of NWO and the technology programme of the Ministry of Economic Affairs within the NEST project 10346.



This work was carried out in the ASCI graduate school.
ASCI dissertation series number 287.

Printing: Printservice Technische Universiteit Eindhoven

Abstract

Model-Driven Scheduling of Dynamic Streaming Applications on MPSoCs

Consumer electronics have evolved tremendously in recent decades by the ever-increasing advances in transistor fabrication technology. Nowadays, even further integration of computational and storage elements into a single silicon chip is feasible. Multi-processor systems-on-chips (MPSoCs) provide high performance, power efficient and reliable products at low-cost. Platform-based design is a well-known approach to construct a system using an MPSoC; compared to alternative design strategies, a platform-based strategy has lower design cost with a shorter time-to-market at lower risk. In a platform-based design approach, applications should be mapped to an MPSoC platform. For this purpose, model-based design approaches are introduced. In a model-based design approach, applications are captured by using some abstract model; analysis of these models can reveal information about the applications in early design stages to accelerate the construction of a system running on an MPSoC. This thesis focuses on applications from the multimedia and Digital Signal Processing (DSP) domain executing on an MPSoC. Such applications are repetitively applying deterministic transformations on an input stream of indefinite length, for example, a video decoder that decodes a video stream. These applications are often referred to in literature as streaming applications. Since the 1980s, synchronous dataflow (SDF) graphs are being used to model streaming applications. However, traditional SDF graphs are not able to accurately capture modern dynamic applications with different working modes, also called scenarios. Scenario-aware dataflow (SADF) graphs have been introduced to tackle this issue. We focus in this thesis on applications described by SADF graphs, addressing several challenges in mapping these applications to MPSoCs following a model-driven approach.

Streaming applications are expected to meet specific performance requirements, for example throughput and latency. These requirements can be evaluated by analyzing the underlying SADF models. To utilize all resources available on an MPSoC and provide better performance (e.g., higher throughput), an application is decomposed (or partitioned) into several tasks, allowing those tasks to run in parallel. The act of assigning processing elements to the tasks of an application is called binding. The order according to which tasks will be executed on

a single processing element is indicated by the scheduling decisions. Binding and scheduling decisions can influence the performance of the design. In this thesis, we assume a given binding and platform. The main subject of investigation is how to schedule a dynamic application on an MPSoC in a way that a strict timing requirement is being assured, and taking into account that modern MPSoCs allow to adapt frequencies and voltages of their processing elements at run-time in order to provide a trade-off between performance and energy consumption.

We start our work with the state of the art mapping techniques from the SDF^3 toolkit. The scheduler of SDF^3 assumes that enough on-chip memory is available in MPSoCs to store all application code and data. As a first contribution in this thesis, we refine this assumption by considering the availability of both off-chip and on-chip memories in a system, taking into account the impact of the prefetching on the performance when scheduling an application. The design decisions (binding, scheduling, etc.) must be considered when a specific analysis (e.g., throughput calculation) is being performed on a dataflow model. A second contribution in this thesis are two techniques to include mapping decisions into SADF graphs. These techniques allow faster and more accurate analysis. The timing properties of an SADF model may be changed by a modification in the frequencies setting of the underlying MPSoC platform. We employ a parametric SADF model to capture the effect of changing processor frequencies. The parametric SADF model uses expressions to represent the timing properties in terms of processor frequencies. The throughput of the parametric SADF is required to be known to decide on the best frequency and voltage settings. As execution times are not constant, the classical throughput analysis techniques cannot be used. The third contribution in this thesis is a parametric throughput analysis technique for SADF graphs. The outcome of our parametric throughput analysis is a set of expressions representing the critical timing cycles in the application. As a final contribution, we utilize the information extracted by our parametric throughput analysis to determine the frequencies and voltages of an MPSoC such that it minimizes the energy consumption while guaranteeing a certain throughput requirement. We combine the resulting pro-active dynamic voltage and frequency scaling (DVFS) technique with a reactive online approach to further exploit the timing slack identified at run-time.

Contents

Chapter 1

Introduction

1.1 Era of Embedded Systems

General purpose computers, such as personal computers, are used to serve various functions. In contrast, *embedded systems* are special-purpose computers developed to perform dedicated tasks, often with specific timing requirements [5]. An embedded system is a combination of hardware, software and often even some mechanical parts.

The Apollo spacecraft was one of the first important designs in which a modern embedded system is deployed by the *MIT Instrumentation Laboratory* in the 1960s. The *Apollo Guidance Computer* was developed to carry out the computation and electronic interfaces for guidance, navigation, and control of the Apollo spacecraft [37]. The Intel 4004 microprocessor developed for digital calculators, introduced in 1971, can also be considered as an early form of today's embedded systems. Over the last three decades, the continuous advances in silicon technology have made it possible to integrate more transistors into a single chip; such a technology is called *Very-Large-Scale-Integration* (VLSI). These VLSI devices can serve more complex functions. The mass production of VLSI circuits has made microprocessors cheap enough to be used in any computation-intensive device. Many of these devices serve one or a few dedicated functions; hence, they are considered embedded systems.

Nowadays embedded electronic devices are an inevitable part of human life. The presence of embedded systems is tangible from a simple alarm clock to a complex spacecraft. Many household appliances, such as washing machines, dishwashers, microwave ovens, vacuum cleaners, etc, comprise embedded systems to provide efficient and low cost products. Automotive companies are replacing mechanical parts with electronic devices to provide more reliable and efficient vehicles. The aviation industry takes advantage of advanced navigation and guidance embedded systems to assure safer aeroplanes. Embedded devices also make medical equipment more capable, easier to use and easier to integrate into hospital systems. These new technologies enable remote supervision and patient monitor-

ing for long-term treatment and medication. Recently emerged smartphones and tablets are a prominent example of embedded systems; such products gather multiple applications from multimedia and digital signal processing (DSP) domains in a single device. According to *International Data Corporation (IDC)* [45], vendors will ship 918.6 million smartphones in 2013; this number is estimated to reach 1.5161 billion in 2017. All of these shows how our lives are tightly coupled with products developed around embedded systems. In this thesis, we target applications from multimedia and DSP domains; such applications repeatedly impose deterministic transformations on an input stream of indefinite length. Video, audio and image processing are prominent examples. These applications are often referred to in literature as *streaming applications*.

1.2 Challenges in Embedded Systems Development

Nowadays, multiple microprocessors, DSPs, reconfigurable FPGAs (field programmable gate arrays), memories, sensors and actuators can all be integrated into a single chip; such a system is often called a *system on chip* (SoC). SoCs provide low-cost, compact, light, low-power, low-energy and efficient products; these nice properties make SoCs a suitable way of implementing embedded applications. However, the design complexity also grows with the increased integration of these components.

To be competitive in the consumer electronics market, the electronics industry tries to introduce alluring products with multiple functionalities; for instance, current handheld phones include an MP3 player, video display, GPS (global positioning system) navigation system, web browser, and camera besides their traditional usage (i.e., making phone calls). Different applications may use shared resources (e.g., microprocessors, memories, communication links, etc). Such resource sharing should not affect the *functionality* of the product. Functionality means that a task should be executed according to its design specification. Applications may require different qualities that must be taken into account in the development phase. Usually, temporal behavior is considered as one of those quality metrics; for instance, a video decoder should be able to deliver a video stream with 20 fps (frames-per-second). These devices are often battery-powered; hence, it is crucial to develop low-power and low-energy consuming products. The performance of an application can also be lowered to save more battery energy (e.g., the quality of a video stream can be reduced in a video decoder application). The design, verification, and test of the mentioned composite systems is a complicated task which further increases the development complexity. Moreover, time-to-market is also a critical issue in the development of these devices. It is vital for the electronics industry to employ design strategies such that it can quickly deliver/upgrade products while still respecting all required functional and temporal requirements. Moreover, the development cost of embedded systems is rapidly rising by the growth in the design complexity. Keeping these costs under control is another challenge faced by the electronics industry.

The following summarizes the important challenges in the development of

embedded systems:

- Managing the ever-increasing design complexity.
- Reducing the time-to-market.
- Providing flexible and upgradable designs.
- Realizing low-power/low-energy systems.
- Satisfying the temporal and functional requirements.
- Handling the rise in design cost.

In the next subsection, we introduce promising strategies that help to meet the mentioned challenges.

1.3 Successful Strategies in Embedded Systems

This section introduces existing successful strategies in developing embedded systems. In this thesis, we extend these strategies with novel analysis and modeling techniques for dynamic streaming applications.

1.3.1 Platform-Based Design

In the past, an electronics company was taking all responsibility and burden of developing a product from the specification phase to the final manufacturing phase. Such a development cycle may not be practical considering the increasing Non-Recurring Engineering (NRE) and design costs, tremendous rise in design complexity and ever-shortening time-to-market. The refinement of an architecture for a single application is costly and time consuming. Nowadays, VLSI industries are using pre-designed and pre-characterized components instead of full custom designs. A design philosophy using such pre-designed components is called *platform-based design* [26, 47]. In this philosophy, the steps required to explore and refine the underlying architecture can be skipped. This promotes the re-use concept in system design which is called *intellectual property* (IP) creation. The IPs or platforms are verified and tested beforehand. This saves design and verification time. The platform-based design strategy also prorates the costs involved in Non-Recurring Engineering (NRE) in making masks and manufacturing setups by using the same masks as often as possible in different products and in different companies.

The cost of a product depends on its silicon area and the NRE cost per product sample. Hence, selecting and designing a platform is a crucial issue. A platform with low flexibility (or applicability) which is utilized for a limited set of applications efficiently uses the silicon area and hence its silicon cost can be lowered. But, a less flexible platform has limited applicability and such a platform cannot attract many applications from other domains; as a result, the NRE cost per single chip may increase. The other extreme is developing a flexible platform.

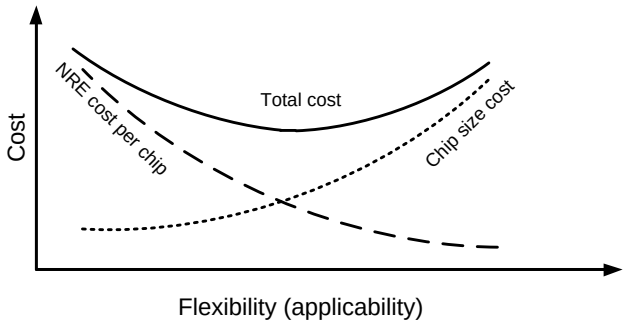


Figure 1.1: Cost versus flexibility of a platform.

By using a flexible platform, the production volume can be increased to prorate the NRE and design cost; however, a flexible platform requires more silicon area and this can increase the cost of a single chip. Figure 1.1 graphically depicts both extremes; in this figure, cost versus flexibility of an imaginary platform is shown. The middle region in this figure shows under how much flexibility the total production cost per single chip can be minimized. This figure is not based on real values and it is included to graphically highlight the concept of choosing a platform. In this thesis, we assume that the platform, which is a multi-processor system-on-chip (MPSoC), is chosen for the class of streaming applications.

Using a platform-based design is a way to deal with the design complexity; it also reduces the time-to-market and design cost by employing the re-use concept. Systems developed using a platform are easier to maintain and upgrade. The applications running on a platform are often expected to satisfy a certain temporal requirement. The achievable temporal performance depends not only on the application properties, but also on the underlying platform. The *model-based design* concept, introduced in Section 1.3.2, uses an integrated model to accommodate both application and platform properties into a single abstract model. The design decisions should also be captured in the model. Analyzing such a universal model can be a way to formally determine the achievable temporal performance. The next subsection is devoted to discuss these related topics.

1.3.2 Model-Based Design

In traditional approaches, system engineers provide a detailed specification of a system; hardware and software engineers are responsible to implement the provided specification. There is always a chance that an implemented system does not follow the given specification. This can happen because a specification may not include all relevant details or errors may be introduced due to misinterpretation of the specification. Moreover, the traditional approach is slow and time consuming and it requires interaction between system and hardware/software engineers. *Model-based design* approaches (e.g., [9, 10, 54, 69, 86]) are introduced to address such issues. In a model-based approach, a system is expressed in terms

of a well-defined model. For a model, some intrinsic properties are defined. Functional and temporal behavior of the system are described and can be evaluated either by a formal mathematical method or by a simulation-based method. The system described by such models can be directly used as the input to an automatic synthesis/compilation trajectory. Different properties of a system can be determined by analyzing the model of the system. For a model, the *analyzability* is defined by its ability to capture certain properties of the system using the model. Models may also differ in capturing different system behaviors; the *expressiveness* expresses the potential of a model in describing system behaviors. In literature several models were introduced for different purposes. Dataflow models have been used for DSP software synthesis since several decades ago [9, 49, 63, 67]. In [73, 85], detailed surveys on existing dataflow models are presented. A model-based approach can be automated; this can eliminate extra implementation time and designer mistakes. As a result, a better and faster design-space exploration can be possible using a model-based approach [11, 65, 68].

In this thesis, we use scenario-aware dataflow (SADF) [74] graphs to model applications. SADF graphs are mainly introduced to model dynamic applications with multiple operating modes, called scenarios. Each scenario of an SADF is described using a synchronous dataflow (SDF) graph [9, 49, 63, 67]. The scenario occurrence order is captured using a finite state machine (FSM).

Model-based design is a way to manage the design complexity by abstracting applications and the platform. This helps to reduce the development cost and time-to-market. Using a high-level model enables further flexibility in exploring different alternatives in the design trajectory to provide a flexible and upgradable design. The high-level model can formally be analyzed to determine the temporal performance of the system. For this purpose, some temporal guarantees must be assured by the application and the underlying platform; the *predictable design* concept, explained in Section 1.3.3, is used to maintain this temporal behavior.

1.3.3 Predictable Design

A system is called *predictable* if it assures a reasonable bound on performance of an application (e.g., a lower bound on the throughput of the application and/or an upper bound on the latency of the application). Consider a video decoder application; an implementation of a video decoder is often required to process a specific number of video frames within a second (i.e., frames-per-second) to provide an acceptable user-perceived quality.

Predictable system design is concerned with specifying a suitable hardware platform and an appropriate software design methodology. The hardware platform - in this thesis an MPSoC - must provide timing guarantees for each of its operations; for example, the *worst-case* time required to fetch a data element from a memory is required to be known in advance. These considerations can disallow using performance enhancing architectural elements such as cache memories and pipelines which cause a considerable non-determinism and hinder providing timing guarantees; however, on-going research in embedded systems tries to relax this issue by introducing new schemes or elements. For instance, scratchpad memo-

ries (SPMs) are identified as an alternative to cache memories, even though using SPMs in a system requires extra efforts such as memory profiling. In this thesis, we assume that the underlying MPSoC platform has predictable resources. Previous research has shown that a predictable platform is feasible to realize [38, 68]. Employing a predictable hardware platform is a necessity for a predictable system but it is not enough. An embedded application which is being executed on a platform should also be predictable. For example, worst-case processor cycles required to process a task should be known a priori. The model employed to describe an application must be monotone [63] since worst-case knowledge is used to capture the temporal behavior of the application. Monotonicity states that a reduction in the execution time of an arbitrary actor in the application graph cannot reduce the throughput of the application [63]. A monotone model is essential for a model-based design approach since in reality the execution times of tasks are often below their worst-case values. The SADF model which is the baseline of this thesis respects the monotonicity property [6, 31].

1.4 Problem Statement

A modern streaming application may run in different modes during its execution; various processing steps may be employed depending on the input data stream or the availability of the resources on the underling hardware platform. Such dynamism is essential to be captured in a model to provide a better understanding of the application in later design steps. In previous sections, SADF graphs are mentioned as a *Model-of-Computation* for dynamic applications. An application described by such a dynamic model is often mapped to a MPSoC platform. A platform usually has several processing elements (or processors); to obtain better temporal performance, the application is bound to the multiple processing elements. The execution of actors (from the SADF model) mapped to the same processing element can statically be ordered using a schedule. MPSoCs usually have limited on-chip memories; for this reason, remote memories are used besides the internal local memories. The remote-local memory transactions must also be scheduled accordingly. All these decisions influence the final achievable temporal performance. Moreover, the total energy consumption can be affected by such decisions. Dynamic voltage and frequency scaling (DVFS) [14] can be used as a technique to provide a trade-off between performance and the energy consumption. In the rest of this subsection, we enumerate some of the existing issues on running dynamic streaming applications on MPSoCs. The goal is ensuring a certain timing behavior when running a dynamic application on an MPSoC; at the same time, we try to provide a solution with a low energy consumption.

SDF graphs are a special case of SADF graphs with a single scenario; they have been studied extensively in recent literature. For example, the SDF^3 toolkit [71] implements most of the analysis techniques available for SDF graphs in mapping a static application onto a hardware platform. In this thesis, we employ the SDF^3 toolkit as our baseline design flow. Binding decisions are taken using the SDF^3 toolkit. Tasks, also called actors, mapped to a single processing element can be ordered using the scheduler implemented within the SDF^3 toolkit.

However, the scheduler from SDF^3 assumes that enough (code and data) on-chip memories are available in the underlying hardware platform to accommodate all memory objects of an application. This assumption limits the applicability of the SDF^3 design flow; either it can be used for applications with small memory requirements or it can be used for platforms with large on-chip memories. In a more realistic case, off-chip memories must also be used. Only the active memory objects are required to be present in on-chip memories at run-time. The memory objects may need to be read/writen from/to off-chip memories to/from on-chip memories during the application life time. The scheduler for such a system requires to specify all transactions between off-chip memories and on-chip memories besides determining the actors' execution order. Direct-Memory-Access (DMA) units are also a common element in current platforms. DMA units are used to improve the performance of a system by prefetching memory objects. In this way, computation and data communication can be overlapped to shorten the memory access latency. Considering prefetching of memory objects using a DMA unit can further increase the complexity of the scheduler. In this thesis, we present a scheduler that considers all mentioned aspects. The resulting scheduling (design) decisions must be encoded in the model. The only available technique [3] to model mapping into an SDF/SADF graph requires transformation of an SDF graphs (or each scenario graph of an SADF) to a so-called *homogeneous SDF* (HSDF) which is a sub-class of SDF graphs [67]. This transformation often leads to an exponential increase in the size of the graph that may make later analysis infeasible or impractical in practice. There is a second drawback to the technique from [3]; the original graph structure is lost due to the conversion to an HSDF. As a result, common buffer sizing techniques, that are determining the required buffer space for data communicated between different actors, cannot find the minimal buffer size for the original graph. A novel technique is needed to model mappings in an SDF/SADF graph. This technique should limit the increase in the number of actors such that analysis times do not increase too much when analyzing the graph with its schedules. The technique should also preserve the original graph structure as this enables accurate analysis of graph properties such as buffer sizes. This thesis presents two techniques that satisfy both requirements. The presented techniques can be used in any model-based design-flow that models mapping decisions into an SDF/SADF graph (e.g., [9, 10, 54, 69, 86]). Timing analysis of such a schedule-extended MoC provides insight into the achievable temporal properties of the mapped application.

The actor execution times may not be fixed during the design trajectory. For example, actor execution times can change by applying a DVFS optimization. Throughput of an SDF/SADF graph, i.e., the average number of actor firings per time unit, is an important metric to determine the performance of a system. Most of the existing throughput calculation techniques are only applicable to models with fixed execution times except [33]. Ref [33] introduces a parametric throughput analysis technique for SDF graphs. The technique finds throughput expressions for a parameterized SDF in which actors can have a linear function of some parameters as their execution time. These parameters have a specified time interval. However, applying SDF throughput analysis to applications with

a dynamic behavior may result in a loose bound on the worst-case throughput. This may lead to an over-allocation of resources. So, a new technique is required to determine throughput expressions when the underling application model is an SADF. In this thesis, a parametric throughput analysis technique is introduced for SADF graphs. The proposed parametric throughput analysis technique is used to find the frequency points for a platform with DVFS possibility.

Streaming applications, such as signal processing and multimedia applications, are often expected to meet strict timing requirements (e.g., a throughput constraint). Furthermore, energy consumption is an important design criterion for such applications. DVFS is a commonly used technique to develop low power/energy implementations. Hence, a multi-processor DVFS controller can be developed for such streaming applications to provide a timing guarantee while minimizing the energy consumption. We employ a parametric SADF model to capture the effect of changing processor frequencies. The parametric SADF model uses linear expressions to represent the timing properties in terms of processor frequencies. The throughput of the parametric SADF is required to be known in the process of constructing the DVFS controller; for this purpose, our parametric throughput analysis technique can be used. We utilize the information extracted by our parametric throughput analysis to determine the frequencies and voltages of an MPSoC in such a way that it minimizes the energy consumption while guaranteeing a throughput requirement. We combine the resulting pro-active dynamic voltage and frequency scaling (DVFS) technique with a reactive online approach to further exploit the timing slack identified at run-time.

1.5 Contributions

The main contributions of this thesis are listed below:

- A technique is presented to schedule an application on an MPSoC that contains a limited on-chip memory. The proposed scheduling technique explores the trade-off between executing actors in a code-driven (i.e., executing parallel actors) or data-driven (i.e., executing pipelined actors) manner to optimize temporal properties of the application. The technique also considers prefetching when choosing a suitable execution order. An earlier version of this work was published in [20].

- The design decisions (binding, scheduling, etc.) must be considered when a specific analysis (e.g., throughput calculation) is being performed on a dataflow model. For this purpose, we present two techniques to include mapping decisions directly into SADF graphs. This work was published in [19, 21].

- The SADF model is parameterized by associating the execution time of each actor in the model with a linear function of some parameters. A throughput function exists for such a parametric SADF model; evaluating this function for a specific parameter point reveals the throughput value of the SADF model for that parameter point. This thesis presents a technique to find

the throughput function of a parameterized SADF model. This work was published in [23].

- The parametric SADF model is used to accommodate processor frequency settings of an MPSoC platform. In this thesis, we present a technique to select a suitable multi-processor DVFS point for each mode (scenario) of a dynamic application described by an SADF. The resulting DVFS controller assures strict timing guarantees while minimizing energy consumption. This work was published in [22].

1.6 Thesis Overview

This thesis is organized as follows. The two dataflow models-of-computation which are used used throughout this thesis (i.e., SDF and SADF) are defined in Chapter 2. The same chapter also introduces the notations and concepts related to dataflow scheduling. Chapter 3 presents our ILP-based technique for dataflow scheduling. In Chapter 4, we introduce two techniques to directly model periodic static-order schedules and single appearance schedules into SADF graphs. Chapter 5 develops a parametric throughput analysis technique for a parameterized SADF graphs. Our parametric throughput analysis technique is employed in Chapter 6 to design an energy-aware DVFS controller for a dynamic application described by an SADF graph. A strict throughput guarantee is assured for a system that uses our DVFS controller. The same chapter explains how our DVFS controller is amended with a reactive online approach to further exploit the timing slack identified at run-time. Finally in Chapter 7, we conclude this thesis along with some recommendations for future work.

Chapter 2

Dataflow Preliminaries

This chapter introduces terminologies and definitions used in this thesis. The chapter also motivates the choice made in this thesis to model streaming applications with Scenario-Aware Dataflow Graphs. The current chapter also introduces a model transformation technique for dataflow graphs to speed up timing analysis.

2.1 Overview

Dataflow graphs, especially Synchronous Dataflow (SDF) [9, 49, 63, 67] graphs, are frequently being used to model DSP applications for concurrent implementation on parallel hardware. Moreover, the behavior of static streaming applications, such as DSP applications, can be captured using SDF graphs. One of the main properties that many streaming applications share is applying deterministic transformations on data stream. Consider a smart phone that may be used to play a movie from YouTube^{TM}; continuously, the video stream is being received over a medium using a network protocol like WLAN, 3G or 4G. At the same time, video and audio decoding are performed on the received data. Each of these steps can be modeled using an SDF graph. The algorithms and protocols used in such devices are getting tuned and revised to provide better bandwidth utilization, better performance, etc; for this purpose, algorithms are often adapting themselves to the data stream. For example, a video decoder applies different processes to different types of video frames. Hence, a single application may operate in different modes over its run-time. As a result a single (static) SDF may not be able to accurately model the behavior of a modern streaming application. Analyzing an SDF graph that captures all computations performed in all different modes of an application can lead to inaccurate results. Consequently, a model-driven design approach relying on those analysis results may incur resource over-allocation and/or extra energy consumption. Excess energy consumption for hand-held devices is not favorable since these devices are usually battery-powered. Scenario-aware dataflow (SADF) [74] graphs are introduced to provide a better model for dynamic applications as compared to SDF graphs.

Analysis techniques applied to such dynamic models can provide tighter performance results and prevent resource over-allocation when implementing dynamic applications. The focus of this thesis is on techniques and approaches for SADF graphs. In this chapter, we describe the SADF model which is the baseline of our work. Since an SADF uses SDF graphs to describe different modes (or scenarios) of an application, we first describe the SDF model in Section 2.2. Section 2.3 introduces the SADF model. The concepts related to the dataflow model in the context of multi-processor systems are discussed in Section 2.4. Max-Plus algebra can be used to mathematically describe the behavior of dataflow graphs. As we use some analyses based on Max-Plus algebra in this thesis, Section 2.5 is devoted to explaining the basics of Max-Plus algebra. Some modeling techniques (e.g., buffer size modeling) may add a large number of initial tokens to the graph. In Section 2.6, we introduce a technique to reduce the number of initial tokens in a dataflow graph without changing the timing behavior of the model. The last section concludes this chapter.

2.2 Synchronous Dataflow Graphs

A synchronous Dataflow (SDF) graph is a directed graph $G = (A, C)$. A node $a \in A$, called *actor*, represents a function (task) of the application. An edge $c \in C$, called *channel*, captures (data or control) dependencies between actors. An actor is connected to a channel via a *port*. We formally define an SDF as follows.

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 (∞). We assume a set $Ports$ of ports; each port $p \in Ports$ has a finite rate $Rate(p) \in \mathbb{N}$.

Definition 1. (Actor) *An actor a_i is a tuple (In, Out, τ_i) consisting of a set $In \subseteq Ports$ of input ports (denoted by $In(a_i)$) and a set $Out \subseteq Ports$ of output ports (denoted by $Out(a_i)$) with $In \cap Out = \emptyset$. A non-negative real value $\tau_i \in \mathbb{R}^+$ is used to represent the worst-case execution time of the actor a_i.*

In the thesis, the notation $\tau(a_i)$ is often used to refer to the worst-case execution time of a_i (i.e., τ_i). Note that the original definition of SDF does not include time; however, throughout this thesis, we consider timed SDF that includes worst-case execution times in the definitions of actors.

Definition 2. (SDF) *An SDF graph G is a tuple (A, C) consisting of a finite set A of actors and a finite set $C \subseteq Ports^2$ of channels. The channel source (destination) is an output (input) port of an actor. Each port of an actor is connected to only one channel and each channel end is connected to a single port. For every actor $a_i = (In, Out, \tau_i) \in A$, $InC(a_i)$ ($OutC(a_i)$) represents all channels connected to the ports in In (Out).*

Figure 2.1 shows an example of an SDF with three *actors* ($A = \{a_0, a_1, a_2\}$) and four *channels* ($C = \{c_0, c_1, c_2, c_3\}$). Actors communicate with *tokens* sent from one actor to another over the channels. Channels may contain *initial tokens*, depicted with a solid dot. The example SDF contains four initial tokens labeled

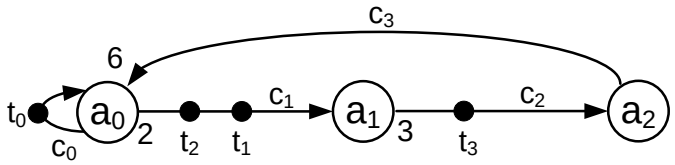


Figure 2.1: An example SDF graph.

t_0, t_1, t_2 and t_3. An essential property of SDF graphs is that every time an actor *fires* (executes) it consumes a fixed number of tokens from its input channels and produces a fixed number of tokens on its output channels. These numbers are called the *rates* (indicated next to the channel ends when the rates are larger than 1).

An actor can only fire (execute) if sufficient tokens are available on the channels from which it consumes. Tokens thus capture dependencies between actor firings. Such dependencies may originate from data dependencies, but also from for example dependencies on shared resources. When an actor a_i starts its firing, it removes $Rate(q)$ tokens from all input channels $(p, q) \in InC(a_i)$ and when it ends, it produces $Rate(p)$ tokens on all output channels $(p, q) \in OutC(a_i)$. The rates determine how often actors have to fire with respect to each other such that the distribution of tokens over all channels is not changed. This property is captured in the repetition vector of an SDF.

Definition 3. (Repetition Vector) *A repetition vector of an SDF graph* $G = (A, C)$ *is a function* $\gamma : A \to \mathbb{N}_0$ *such that for every channel* $(p, q) \in C$ *from* $a_i \in A$ *to* $a_j \in A$, $Rate(p) \cdot \gamma(a_i) = Rate(q) \cdot \gamma(a_j)$. *A repetition vector* γ *is called non-trivial iff for all* $a_i \in A, \gamma(a_i) > 0$. *An SDF graph is called consistent iff it has a non-trivial repetition vector. For a consistent graph, there is a unique smallest non-trivial repetition vector, which is designated as* the *repetition vector of the SDF graph.*

The repetition vector of the SDF graph shown in Figure 2.1 is equal to $(a_0, a_1, a_2) \to (1, 2, 6)$. This shows that the SDF graph is consistent as its repetition vector is non-trivial. Consistency and absence of deadlock are two important properties for SDF graphs which can be verified efficiently [8, 53]. Any SDF graph which is not consistent requires unbounded memory to execute or it eventually deadlocks. When an SDF graph deadlocks, no actor is able to fire, which is due to an insufficient number of initial tokens in a cycle of the graph. Any SDF graph which is inconsistent or deadlocks is not useful in practice. Therefore, we limit ourself to consistent and deadlock-free SDF graphs.

The special class of SDF graphs in which all port rates are equal to one is called the class of *homogeneous SDF* (HSDF) graphs [53]. Any consistent SDF graph can be converted to an HSDF graph [53, 67] that is equivalent from the timing perspective. The equivalent HSDF graph of our example SDF graph (shown in Figure 2.1) is shown in Figure 2.2; each actor in the SDF graph is repeated in the equivalent HSDF as often as indicated in the repetition vector of the SDF graph. For example, actors a_{1_1} and a_{1_2} are added to the equivalent

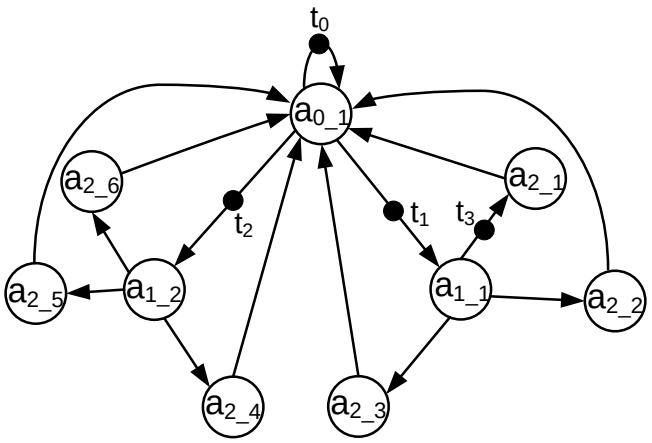


Figure 2.2: HSDF graph of the example SDF graph in Figure 2.1.

HSDF graph in Figure 2.2 in place of actor a_1 in the example SDF. The SDF to HSDF conversion may however lead to an exponential increase in the number of actors and it has an impact on the speed and possibly also the accuracy of the result of analysis techniques. As an example, consider applying a buffer sizing analysis algorithm (e.g., [70]) to our example SDF graph and its equivalent HSDF graph; the minimum buffer size required for deadlock-free execution of the SDF graph of Figure 2.1 and its equivalent HSDF graph in Figure 2.2 are 11 and 14 (tokens). Figure 2.3 shows an SDF graph which models a buffer allocation of $[c_1 : 2]$ $[c_2 : 3]$ $[c_3 : 6]$ into the SDF graph shown in Figure 2.1. Some new (dashed) channels are added to the initial SDF graph to model these buffer sizes. The dashed channels in Figure 2.3 are the same as normal channels in an SDF graph. The buffer modeling is performed using the technique proposed in [70] that places a back-edge with some initial tokens for each channel in the SDF graph. The number of initial tokens on the back-edge indicates the free space in the buffer allocated to the channel. In other words, a buffer size of $x \in \mathbb{N}$ for a channel that contains $y \in \mathbb{N}_0$ initial tokens is modeled by including a back-edge with $x - y$ initial tokens to the model. For example, the back-edge for channel c_2 contains $3 - 1 = 2$ initial tokens indicating a buffer size of 3 for channel c_2. A self-edge channel of an actor is not considered during the buffer size modeling because the memory object of the self-edge is an internal state and it is only accessed by the actor. Figure 2.4 depicts the HSDF graph in Figure 2.2 that models a buffer size of 1 for each channel excluding self-edges. Since 14 none self-edge channels exist in the HSDF graph of Figure 2.2, the common buffer sizing technique results in a minimum buffer size of 14; when the same buffer sizing technique is applied to the original SDF graph, a minimum buffer size of 11 is found. This example shows that a buffer sizing technique may result in over-allocation for HSDF graphs. A smart buffer sizing technique (e.g., using shared buffers for several channels) can

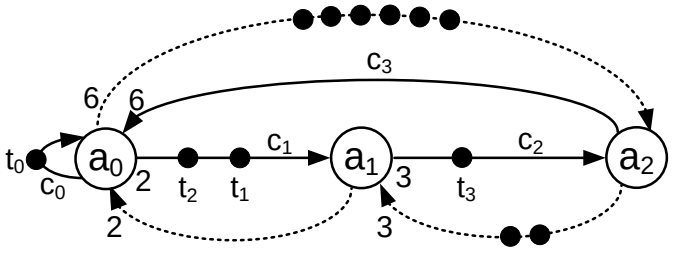


Figure 2.3: The SDF graph in Figure 2.1 with the back-edges encoding buffer size constraints.

be used to relax the over-allocation issue with HSDF graphs, but note that extra effort and more analysis time may be incurred. Hence, in this thesis, we focus on SDF-based analysis. Nevertheless, note that HSDF graphs may be favorable due to their regular structure (i.e., all rates are equal to 1). An actor exists in an HSDF graph to model a firing of an actor in the original SDF graph of the HSDF graph; this makes scheduling of HSDF graphs easier compared to scheduling of the original SDF graphs.

A firing of an actor leads to the consumption of tokens from its input channels and the production of tokens on its output channels. In order to capture the behavior of an SDF, we need to keep track of the distribution of tokens over the channels. The following concept is defined to measure quantities related to channels (e.g., the number of tokens present in channels).

Definition 4. (SDF State) *A state of an SDF (A, C) is a function* $\omega : C \longrightarrow \mathbb{N}_0$ *that returns the number of tokens stored in each channel. Each SDF has an initial state* ω_0 *denoting the number of tokens that are initially stored in the channels.*

An actor can only be fired if there are sufficient tokens in all of its input channels. An actor that satisfies this condition in a particular state is said to be enabled in this state.

Definition 5. (Enabled actor) *An actor* $a_i \in A$ *is called enabled in a state* ω_j *of SDF* $G = (A, C)$ *iff* $\omega_j(c) \geq Rate(q)$ *for each channel* $c = (p, q) \in InC(a_i)$.

Definition 6. (Actor firing) *The firing of an enabled actor* $a_i \in A$ *in state* ω_j *results in the transition from state* ω_j *to state* ω_{j+1} *and is denoted by* $\omega_j \xrightarrow{a_i} \omega_{j+1}$. *The relation between states* ω_j *and* ω_{j+1} *can be expressed as follow:* $\omega_{j+1}(c) = \omega_j(c) - Rate(q)$ *for each* $c = (p, q) \in InC(a_i) \setminus OutC(a_i)$, $\omega_{j+1}(c) = \omega_j(c) + Rate(p)$ *for each* $c = (p, q) \in OutC(a_i) \setminus InC(a_i)$, $\omega_{j+1}(c) = \omega_j(c) + Rate(p) - Rate(q)$ *for each* $c = (p, q) \in OutC(a_i) \cap InC(a_i)$ *and* $\omega_{j+1}(c) = \omega_j(c)$ *for each* $c = (p, q) \notin OutC(a_i) \cup InC(a_i)$;

Consider again our example graph shown in Figure 2.1. Its initial state ω_0 is equal to $(c_0, c_1, c_2, c_3) \rightarrow (1, 2, 1, 0)$. In this state, actors a_1 and a_2 are enabled.

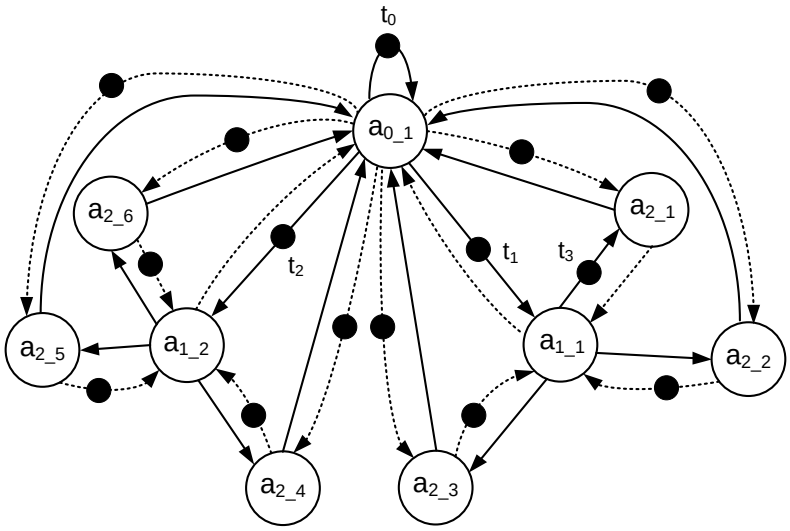


Figure 2.4: The HSDF graph in Figure 2.2 with the back-edges encoding buffer size constraints.

Firing actor a_1 would result in a transition from state ω_0 to a state $(1, 1, 2, 0)$. We use this concept of states and transitions to formalize the execution of an SDF.

Definition 7. (Execution) *An execution σ of an SDF graph $G = (A, C)$ is an infinite alternating sequence of states and transitions $\omega_0 \xrightarrow{a_i} \omega_1 \xrightarrow{a_j} \cdots$ starting from a designated initial state ω_0.*

Definition 8. (SDF graph Iteration) *Assume SDF graph $G = (A, C)$ has repetition vector γ. An SDF graph iteration is a set of actor firings such that for each $a \in A$, the set contains $\gamma(a)$ firings of a.*

2.3 Scenario-Aware Dataflow Graphs

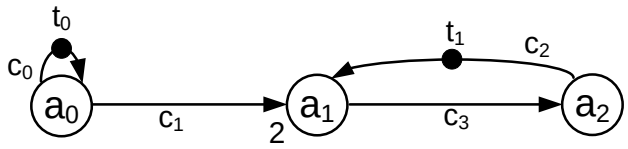
Streaming applications apply deterministic transformations on data streams. The data stream is often composed of consecutive data units. As an example, consider a video decoder application which processes a video stream; the input stream of a video decoder is a sequence of video frames. Each frame specifies a data unit. The operation used in a video decoder may deterministically change depending on the type of the current frame and/or the frame occurrence order. Hence, an application may run at different modes and require different resource usage (e.g., it may show a variation in required processor cycles, buffer sizes, etc). Using an SDF graph to describe such a dynamic application can result in an inaccurate model. Scenario-aware dataflow (SADF) [74] graphs are introduced to model such dynamic applications; an SADF uses a finite state machine (FSM) to capture any possible change in the operating mode of an application. Each operating

mode is called an *application scenario* of the SADF model. The behavior of the application when operating in a specific scenario mostly remains the same during the execution of that scenario; hence, each application scenario can be modeled using a specific SDF graph. A scenario can correspond to the processing of one or more data units or modeling of a behavior in the system (i.e., a reconfiguration operation). Hence each scenario is associated with a property called *progress*. The progress of a scenario is set to the number of data units processed by one iteration of the scenario; when no data unit is processed by an iteration of a scenario, the progress of the scenario is set to zero. In this thesis, the progress of a scenario is one whenever no progress value is mentioned for the scenario. The following definition formally introduces the SADF model.

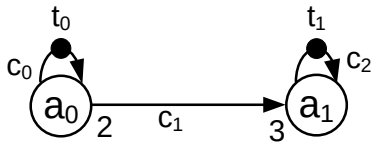
Definition 9. (SCENARIO-AWARE DATAFLOW (SADF)) *An SADF graph is composed of several SDF graphs and an FSM. The set Φ contains all scenarios required to describe the SADF. The behavior of the SADF graph in each scenario $\phi_i \in \Phi$ is described by an SDF graph SDF_i. Such an SDF is also referred to as scenario graph. The progress function $Prgs : \Phi \mapsto \mathbb{N}_0$ assigns a natural number including zero to each scenario $\phi \in \Phi$ that indicates the number of data units processed by one iteration of scenario ϕ. The FSM of the SADF graph is a tuple $FSM = (Q, Q_0, T, L)$ where Q is the set of all states in the FSM, the state $Q_0 \subseteq Q$ is the set containing the potential starting states of the FSM, the set $T \subseteq Q^2$ determines all possible state transitions and the state-labeling function $L : Q \mapsto \Phi$ specifies for each FSM state one scenario. This function returns the scenario required to describe the application when it operates in a specific state.*

Figure 2.5 contains an example SADF graph. Two scenarios exist in this SADF graph. Figure 2.5(a) and Figure 2.5(b) show the SDF graphs describing scenario ϕ_0 and ϕ_1 respectively. Consider the following timing properties for the SDF graph of scenario ϕ_0: $\tau(a_0) = 2$, $\tau(a_1) = 3$ and $\tau(a_2) = 1$. Consider $\tau(a_0) = 2$ and $\tau(a_1) = 2$ as the timing properties for the SDF of the scenario ϕ_1. The FSM of the SADF is shown in Figure 2.5(c). This FSM has three states represented by q_0, q_1 and q_2 ($Q = \{q_0, q_1, q_2\}$). The set $Q_0 = \{q_0\}$ specifies q_0 as the starting state of the FSM. The set $T = \{(q_0, q_0), (q_0, q_1), (q_1, q_2), (q_2, q_0)\}$ contains all state transitions in the FSM of the example SADF. The state-labeling function for the FSM of the example SADF is defined as follows: $L(q_0) = \phi_0$, $L(q_1) = \phi_1$ and $L(q_2) = \phi_1$. This FSM implies that the scenario ϕ_1 must be executed exactly twice whenever it is activated and afterwards the scenario ϕ_0 should be executed at least once. The progress function for the example SADF specifies $Prgs(\phi_0) = 1$ and $Prgs(\phi_1) = 1$.

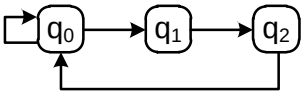
The initial tokens in the SDF of each scenario (scenario graph) capture the dependencies between subsequent iterations of the same scenario graph (as is also the case in an SDF) or iterations of different scenario graphs. The relation between the initial tokens in different scenario graphs is established through the token labels. For example, two tokens (i.e., t_0 and t_1) exist across the scenario graphs in Figure 2.5(a) and Figure 2.5(b). These types of tokens are referred to as *persistent tokens* in this thesis. Consider the sequence $q_0q_1q_2q_0$ as an example FSM state sequence; this sequence, by using the state-labeling function, implies $\phi_0\phi_1\phi_1\phi_0$



(a) SDF_0 to describe behavior of scenario ϕ_0. The timing properties are $\tau(a_0) = 2$, $\tau(a_1) = 3$ and $\tau(a_2) = 1$.



(b) SDF_1 to describe behavior of scenario ϕ_1. The timing properties are $\tau(a_0) = 2$ and $\tau(a_1) = 2$.



(c) FSM of the SADF. The state-labeling function specifies $L(q_0) = \phi_0$, $L(q_1) = \phi_1$ and $L(q_2) = \phi_1$. The progress function specifies $Prgs(\phi_0) = 1$ and $Prgs(\phi_1) = 1$.

Figure 2.5: SADF with two scenarios.

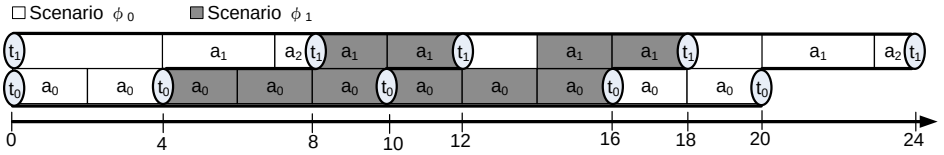


Figure 2.6: The execution related to the scenario sequence $\phi_0\phi_1\phi_1\phi_0$ for the SADF of Figure 2.5.

as a scenario sequence for our example SADF. Figure 2.6 depicts the execution related to the above scenario sequence; as it is shown, multiple iterations can overlap and provide a pipelined execution across consecutive scenario executions.

2.4 Dataflow Scheduling

In a multi-processor system, multiple actors may be bound to the same processor. These actors may be enabled at the same time. In such a situation, a schedule is needed to determine the order in which these enabled actors are fired on the processor. The fixed port rates make it possible to statically schedule SDF graphs with a finite schedule per processor that orders the actor firings for that processor and which is repeated indefinitely. Such a schedule is called a periodic static-order schedule (PSOS). Note that in a multi-processor system, a separate static-order

schedule should be constructed for each processor. Each schedule should only include actors bound to this specific processor. The following definition is used to formally specify a PSOS.

Definition 10. (Periodic Static-Order Schedule (PSOS)) *A PSOS is a finite ordered list of (a sub-set of) actors in an SDF graph $G = (A, C)$. A PSOS is denoted by $s_i = \langle \alpha_1 \alpha_2 \dots \alpha_n \rangle^*$ where each $\alpha_j|_{1 \leq j \leq n}$ is a sub-schedule that represents an actor from A and $n \in \mathbb{N}$ is the length of the schedule s_i, represented by $n = |s_i|$. The set A_i contains all actors that appear at least once in s_i ($A_i \subseteq A$).*

A PSOS can be represented in a compact format, called a looped schedule (LS). The following defines the LS term precisely.

Definition 11. (Looped Schedule (LS)) *A looped schedule, $s_i = \langle (\alpha_1)^{\beta_1} (\alpha_2)^{\beta_2} \cdots (\alpha_m)^{\beta_m} \rangle^*$, is defined as a successive execution of α_1 repeated β_1 times followed by α_2 repeated β_2 times and so on, where each $\alpha_j|_{j \in \mathbb{N}}$ is either an actor firing or a (nested) looped schedule and $\beta_j \in \mathbb{N}|_{j \in \mathbb{N}}$.*

Definition 12. (Single Appearance Schedule (SAS)) *A LS in which each actor appears only once is called a* single appearance schedule (SAS).

Assume that the SDF graph of Figure 2.1 is mapped to a platform with two processors (P_0 and P_1). Actor a_0 is mapped to the first processor (P_0) with the PSOS $s_0 = \langle a_0 \rangle^*$ and actors a_1 and a_2 are mapped to the second processor (P_1) with the PSOS $s_1 = \langle a_1 (a_2)^2 a_1 (a_2)^4 \rangle^*$. PSOS s_0 is a SAS; but, PSOS s_1 is not a SAS.

Definition 13. (PSOS Iteration) *Given a PSOS $s_i = \langle \alpha_1 \alpha_2 \dots \alpha_n \rangle^*$ that schedules actors in $A_i \subseteq A$. A PSOS iteration is a sequence of actor firings following the actor order specified in s_i starting from actor α_1 and ending with actor α_n with a length equal to $|s_i|$ and including only actors from A_i.*

The actor firing order in an execution $\sigma = \omega_0 \xrightarrow{a_x} \omega_1 \xrightarrow{a_y} \cdots$ can be captured using a list $\langle a_x, a_y, \cdots \rangle$ where the j^{th} element in this list is the actor which is fired in the transition from ω_{j-1} to ω_j. The notation $orderList(\sigma, A_i)$ represents the mentioned list where actors which do not belong to A_i are omitted. For example, in the SDF graph of Figure 2.1, consider an execution σ that results in list $\langle a_1, a_2, a_2, a_1, a_2, a_2, a_2, a_2, a_0 \rangle$ with $A_1 = \{a_1, a_2\}$; then $orderList(\sigma, A_1) = \langle a_1, a_2, a_2, a_1, a_2, a_2, a_2, a_2 \rangle$. We say that the corresponding execution of an SDF graph satisfies a PSOS when the SDF graph is executed according to the PSOS. We use the following to formalize this term.

Definition 14. (Satisfaction) *Let σ be an execution of an SDF graph (A, C) and s_i a PSOS for actors $A_i \subseteq A$. If it exists, let σ' be the prefix of σ such that it contains exactly $\gamma(a_i)$ occurrences of actor $a_i \in A_i$; σ' covers σ precisely up to the point that one PSOS iteration is executed. Execution σ satisfies PSOS s_i iff σ' exists and the ordered list $orderList(\sigma', A_i)$ corresponds to the order specified in s_i.*

When an execution of a consistent and deadlock-free SDF graph satisfies the specified PSOSs, the channels of the SDF graph need bounded memories (according to Theorem 1 from [30]). The number of actor appearances of a given actor in the PSOS is a fraction or multiple of its repetition vector entry. Formally, each actor a_i in the PSOS should appear $r \cdot \gamma(a_i)$ times in the PSOS (with $r = \frac{u}{v}$ where $u, v \in \mathbb{N}$) and the value r is identical for all actors in the PSOS [34]. This follows from the SDF property that firing each actor as often as indicated in the repetition vector results in a token distribution that is equal to the initial token distribution. In this thesis, the term *normalized PSOS* is used to refer to a PSOS with r equal to 1.

Definition 15. (NORMALIZED PSOS) *A PSOS s_i is called normalized iff each actor $a_j \in A_i$ appears $\gamma(a_j)$ times in one iteration of the PSOS s_i.*

2.5 Max-Plus Algebra for Dataflow Graphs

This subsection introduces Max-Plus algebra in the context of dataflow graphs. Section 2.5.1 explains the basics of Max-Plus algebra. In Sections 2.5.2 and 2.5.3, we discus the usage of Max-Plus algebra in throughput calculation of SDF and SADF models respectively.

2.5.1 Characteristic Matrix

One iteration of an SDF graph resets the token distribution to their initial places. These tokens may be reproduced at different points in time. A *token timestamp* vector is defined to specify the production time of tokens. The notation θ_k ($k \in \mathbb{N}$) is used to accommodate the production time of the tokens needed in the k^{th} iteration of the graph. Consider θ_0 as the initial token timestamp vector of the SDF graph. Assume that all entries in θ_0 are set to zero. For each SDF graph, a characteristic Max-Plus matrix $M|_{n \times n}$ ($n = |\theta_0|$) exists that can be used to calculate timestamp vectors [2]. An entry $M[i,j] \in M$ corresponds to the minimum time distance from the j^{th} token in the previous iteration to the i^{th} token in the current iteration. The characteristic Max-Plus matrix can be determined using the technique presented in [29]. Consider $\tau(a_0) = 2$, $\tau(a_1) = 4$ and $\tau(a_2) = 3$ time units as execution times for the three actors in our example SDF (see Figure 2.1). The characteristic matrix for our example SDF is:

$$M = \begin{array}{c} \\ t_0 \\ t_1 \\ t_2 \\ t_3 \end{array} \begin{array}{c} \begin{array}{cccc} t_0 & t_1 & t_2 & t_3 \end{array} \\ \begin{pmatrix} 2 & 9 & 9 & 5 \\ 2 & 9 & 9 & 5 \\ 2 & 9 & 9 & 5 \\ -\infty & 4 & -\infty & -\infty \end{pmatrix} \end{array}$$

As an example, the matrix M specifies that the minimum time distance from token t_3 in the k^{th} iteration to token t_2 in the $(k+1)^{th}$ iteration is 5 time units via $a_2 - a_0$ (5 time units is the result of $\tau(a_2) + \tau(a_0)$). When the i^{th} token is not dependent on the j^{th} token, then $M[i,j]$ is set to $-\infty$.

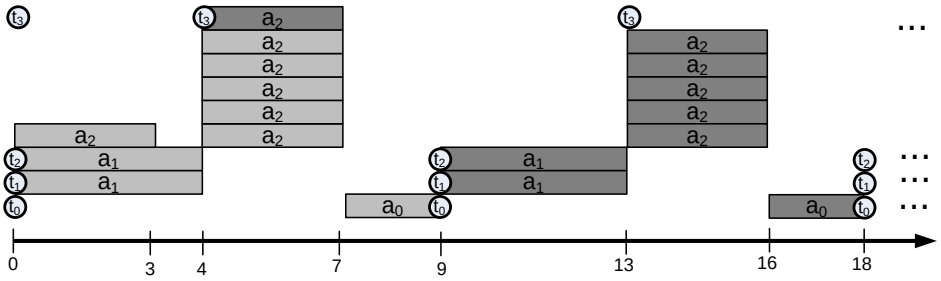


Figure 2.7: Execution of the example SDF graph.

The evolution of the token timestamp vector can be determined by using Max-Plus matrix multiplication as follows:

$$\theta_{k+1} = M \ \cdot \ \theta_k \tag{2.1}$$

Similar to conventional algebra, Max-Plus algebra operates on real numbers (i.e., $\mathbb{R}$). However, the role of addition and multiplication operators in Max-Plus algebra are different from their traditional usage in conventional algebra. In Max-Plus algebra, addition plays the role of a maximum operator (denoted by $\oplus$) and multiplication plays the role of an addition operator (denoted by $\otimes$). The next example shows how the timestamp vector θ_1 is computed using Equation 2.1:

$$\begin{aligned}
\theta_1 = M \ \cdot \ \theta_0 &= \begin{pmatrix} 2 & 9 & 9 & 5 \\ 2 & 9 & 9 & 5 \\ 2 & 9 & 9 & 5 \\ -\infty & 4 & -\infty & -\infty \end{pmatrix} \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} \\
&= \begin{pmatrix} (2 \otimes 0) \oplus (9 \otimes 0) \oplus (9 \otimes 0) \oplus (5 \otimes 0) \\ (2 \otimes 0) \oplus (9 \otimes 0) \oplus (9 \otimes 0) \oplus (5 \otimes 0) \\ (2 \otimes 0) \oplus (9 \otimes 0) \oplus (9 \otimes 0) \oplus (5 \otimes 0) \\ (-\infty \otimes 0) \oplus (4 \otimes 0) \oplus (-\infty \otimes 0) \oplus (-\infty \otimes 0) \end{pmatrix} \\
&= \begin{pmatrix} \max\{2+0, 9+0, 9+0, 5+0\} \\ \max\{2+0, 9+0, 9+0, 5+0\} \\ \max\{2+0, 9+0, 9+0, 5+0\} \\ \max\{-\infty+0, 4+0, -\infty+0, -\infty+0\} \end{pmatrix} = \begin{pmatrix} 9 \\ 9 \\ 9 \\ 4 \end{pmatrix}
\end{aligned}$$

Any timestamp vector θ_k ($k \in \mathbb{N}$) can be determined by iteratively performing the Max-Plus multiplication of Equation 2.1. Figure 2.7 shows the execution of the example SDF of Figure 2.1 for two iterations; actor firings related to the first iteration are shown with light gray color and actor firings related to the second iteration are shown with dark gray color. From Figure 2.7, the evolution of the token timestamp vector can also be obtained (see Figure 2.8).

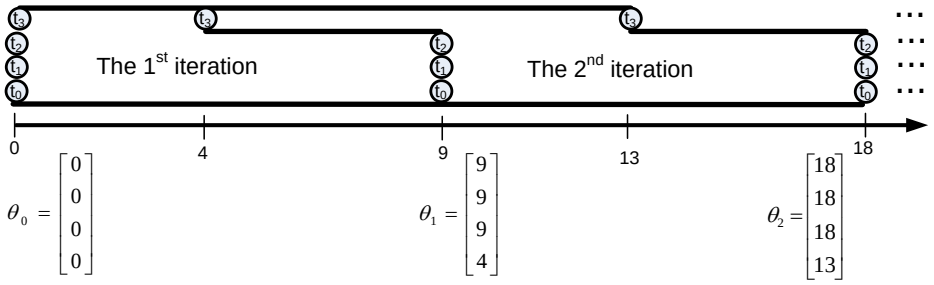


Figure 2.8: The evolution of the timestamp vector for the example SDF graph.

2.5.2 Max-Plus Algebra for SDF Throughput Analysis

The technique presented in [32] explains how the throughput of an SDF graph can be determined using the Max-Plus characteristic matrix of the graph; to calculate throughput, a Max-Plus automaton graph (MPAG) is built using the characteristic Max-Plus matrix. The corresponding MPAG for the example SDF is shown in Figure 2.9. In an MPAG, a node is created for each initial token in the SDF and if the value $M[i,j]$ of the characteristic matrix is not equal to $-\infty$, an edge with weight $M[i,j]$ is added from the node of the j^{th} token to the node of the i^{th} token. The $-\infty$ value for an element $M[i,j]$ means that there is no dependency from the j^{th} token to the i^{th} token. A cycle in an SDF graph which is limiting the throughput is called a *critical cycle* of the SDF graph. A critical cycle of the SDF graph can be obtained by applying a maximum cycle ratio (MCR) analysis [87] on the MPAG. We use the YTO algorithm [87] to perform MCR analysis. The input of the YTO algorithm is a directed graph; each edge of this graph associated with two values called weight and delay. The MCR analysis finds a cycle in the directed graph which maximizes the sum of cycle edge weights over the sum of cycle edge delays; this value corresponds to the inverse of the throughput. The MPAG can be used directly as an input graph to the YTO algorithm. The cost assigned to each edge of the MPAG graph is considered as weight of that edge in the YTO algorithm; the delay of each edge in the input graph of the YTO algorithm is assumed to be 1 because each edge corresponds to one iteration of the SDF. There are 3 cycles in the MPAG graph which are equally critical: the self-edge on node t_1, the self-edge on node t_2 and the cycle $t_1 - t_2$. Any of these cycles determines the throughput which is equal to 1/9 iterations/time unit in our example SDF.

2.5.3 Max-Plus Algebra for SADF Throughput Analysis

Throughput of an SADF can be calculated using the Max-Plus algebra introduced in Section 2.5.1. As for an SDF, characteristic matrices can be determined for each scenario graph. The corresponding matrices for scenarios ϕ_0 and ϕ_1 of the SADF graph shown in Figure 2.5 are as follows:

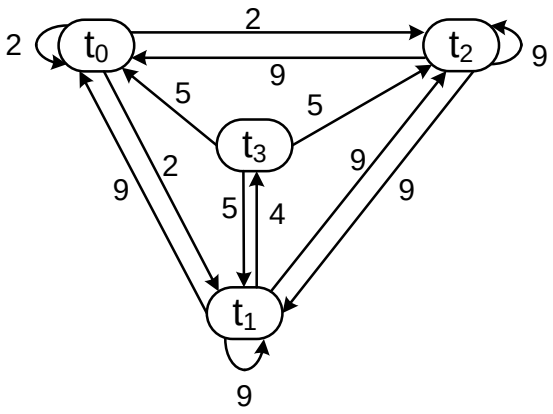


Figure 2.9: MPAG of the example SDF.

$$M_{\phi_0} = \begin{pmatrix} 4 & -\infty \\ 8 & 4 \end{pmatrix} \qquad M_{\phi_1} = \begin{pmatrix} 6 & -\infty \\ 8 & 4 \end{pmatrix}$$

Each characteristic matrix can be translated to an MPAG (see Section 2.5.1). Reference [32] explains how to combine the MPAGs of all scenarios of an SADF to a single MPAG. Figure 2.10 shows the MPAG for our example SADF. In short, a node is added to the MPAG for each token in the scenario graph of an FSM state (e.g., node q_0/t_0 for token t_0 in the scenario graph of state q_0 in Figure 2.10). The notation $M_{L(q_i)}$ denotes the characteristic matrix of the corresponding scenario of the FSM state q_i (i.e., scenario $L(q_i)$). If $M_{L(q_i)}[y, x]$ is not equal to $-\infty$ and there is a state transition from the state q_j in the FSM to the state q_i in the FSM, an edge with weight $M_{L(q_i)}[y, x]$ is added from node q_j/t_x to node q_i/t_y in the MPAG. Using MCR analysis on this MPAG, the critical timing cycle (or cycles) of the SADF can be determined. The constructed MPAG graph can be used as an input graph to the MCR (YTO) algorithm; the cost assigned to each edge of the MPAG graph is considered as weight of that edge in the YTO algorithm; the progress value assigned to the source scenario of each edge in the MPAG is used as the delay value of that edge in the YTO algrithm. The delay values in the case of SDF are assumed to be 1 representing the processing of one data unit in one iteration. However, in the case of a SADF graph, the delay values can get 0 progress representing the processing of no data unit in one scenario iteration (e.g., a scenario that models a reconfiguration) or a higher progress value representing the processing of multiple data units in one scenario iteration. In our example, two critical cycles are found; the self-edges on the nodes q_1/t_0 and q_2/t_0 in Figure 2.10 with the length of 6 are identified as critical cycles in our example SADF ($MCR = 6$). Any of these cycles determines the throughput of the SADF, which for our example is equal to 1/6 iterations/time unit. Each of these cycles corresponds to 3 times a firing of actor a_0 in scenario ϕ_1.

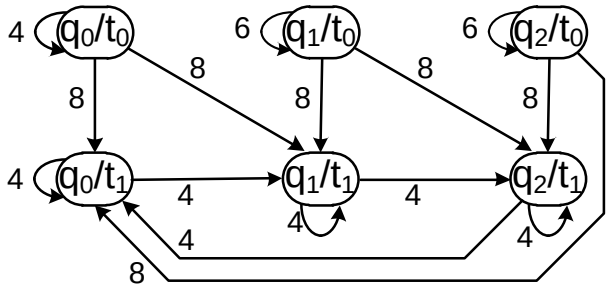


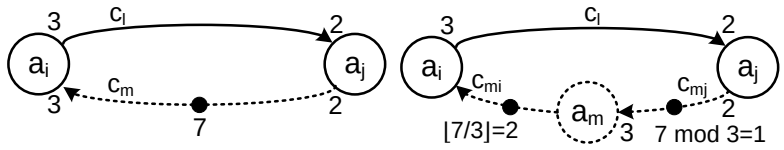
Figure 2.10: Max-Plus automaton graph of the example SADF.

2.6 Model Transformation

The number of nodes in an MPAG is equal to the sum of the number of initial tokens in the scenario graphs of all FSM states of the SADF. Initial tokens are often used to model buffer sizes [68] or schedules in dataflow graphs (see Chapter 4). Consider as an example a common method to model buffer sizes in the SDF as sketched in Figure 2.11(a). The dashed channel in Figure 2.11(a) models a limit on the buffer size of channel c_l (i.e., the 7 tokens on c_m limit the number of tokens that can be present simultaneously in c_l to 7). Note that multiple tokens on a channel are represented by placing a number next to a single solid dot instead of placing multiple solid dots; the number next to the single solid dot indicates the number of tokens on the channel. Dataflow models that contain implementation aspects such as buffer sizes may therefore contain a considerable amount of initial tokens. As a consequence, the MPAG may become large which in turn may lead to a long run-time for a throughput analysis technique. In this subsection, we introduce two types of model transformations to reduce the number of initial tokens in the dataflow graph without changing the timing behavior of the actors. Two token reduction techniques for SDF graphs are presented in Section 2.6.1. Section 2.6.2 explains under which assumptions and how the proposed token reduction techniques can be applied to SADF graphs. Our token reduction technique has been published in [23].

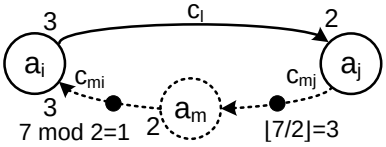
2.6.1 Token reduction for SDF Graphs

Consider again the SDF shown in Figure 2.11(a). After applying our model transformation (explained below), the graph is transformed to the one shown in Figure 2.11(b) reducing the number of initial tokens from 7 to 3. In Figure 2.11(a), 3 tokens are consumed from c_m in each firing of a_i. So, initial tokens of a channel can be grouped based on the consumer actor's rate (e.g., rate 3). We call this value *grouping factor*, represented with gf. Our model transformation replaces any channel c_m from some actor a_j with production rate v to some actor a_i with consumption rate w and with n initial tokens with the following constructs: (1) an actor a_m with zero execution time; (2) a channel c_{mj} with ($n \mod gf$)



(a) Original.

(b) Consumer side grouping.



(c) Producer side grouping.

Figure 2.11: Modeling buffer with size 7 for channel c_l.

initial tokens from a_j to a_m with production rate v and consumption rate gf, (3) a channel c_{mi} with $\lfloor n/gf \rfloor$ initial tokens from a_m to a_i with production 1 and consumption rate w/gf. Here, w is consider as the grouping factor (i.e., gf).

The proposed graph transformation does not affect the timing behavior of the original actors in the graph. In the original graph, actor a_i can fire once whenever at least w tokens exist in channel c_m ($w = 3$ in our example). The n initial tokens that are present on channel c_m all have a corresponding entry in the timestamp vector θ_k, i.e., $\theta_k(t_1) \cdots \theta_k(t_n)$ denote the production times of these tokens in the k^{th} iteration. Firing actor a_i consumes w tokens at once from channel c_{mi}. Among this group of w tokens, the token which has the largest timestamp influences the start time of the actor firing. In other words, $\max\{\theta_k(t_1) \cdots \theta_k(t_w)\}$ influences the first firing of a_i of the $(k+1)^{th}$ iteration of the graph. Understanding this principle enables us to group the n initial tokens as much as possible. We can form $\lfloor n/w \rfloor$ groups of tokens and the remaining (i.e., $n \mod w$) tokens should preserve their individual timestamps.

Grouping initial tokens, on the producer actor side of a channel is also possible. Figure 2.11(c) shows the resulting graph after applying initial tokens grouping to the producer side of the channel c_m. In this case, the production rate of actor a_j on channel c_m (i.e., rate 2) specifies the grouping factor.

2.6.2 Token reduction for SADF Graphs

The introduced token reduction techniques are also applicable to channels of an SADF where the initial tokens of channels only models intra-scenario dependencies. A token in each of the scenario graphs added to model an inter-scenario dependency uses the same token labeling in all scenario graphs in the SADF. When tokens on channels model inter-scenario dependencies, the following considerations are required. The token reduction techniques are also applicable to SADF channels that contain inter-scenario token labelings when the rates on the source and destination side of the channels are identical across all scenarios. In

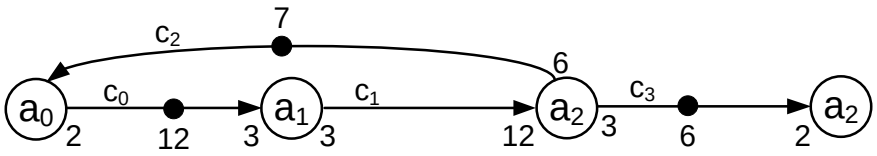
case of different rates for a channel on its destination side for some scenarios, the token grouping on the consumer side cannot preserve the token timestamp information; so, this token reduction cannot be applied. In case of different rates for a channel on its source side for some scenarios, the token grouping on the producer side gets limited to the minimum amount of the tokens produced into the channel in one iteration across all scenario iterations. In this situation, the grouping factor is determined based on the greatest common divisor (GCD) of the channel's source side rate in all scenarios. Consider an actor a_j which in each of its firings produces $v_1, \cdots, v_s$ tokens in channel c_m in scenarios $\phi_1, \cdots, \phi_s$ respectively. The grouping factor is specified with $GCD(v_1, \cdots, v_s)$; in this way timestamp information of tokens can be preserved and the transformation does not affect the timing behavior of the original actors of the SADF.

As an example, consider an SADF graph with two scenario graphs depicted in Figure 2.12(a) and Figure 2.12(b). Applying our token reduction technique to this SADF results in the scenario graphs shown in Figure 2.12(c) and Figure 2.12(d). The rates on the destination side of the channel c_0 are different in the scenario graphs SDF_1 and SDF_2; hence, only token grouping on the producer actor side of c_0 is possible. This transformation replaces c_0 in the original graphs with an actor a_4 and two channels c_{00} and c_{01} (see Figure 2.12(c) and Figure 2.12(d)). Similar token grouping is performed for c_2. Note that our token grouping is not applicable to c_3 because rates on both source and destination rates of this channel are different across the scenario graphs SDF_1 and SDF_2. In total, our model transformation reduces the number of initial tokens from 25 to 14.

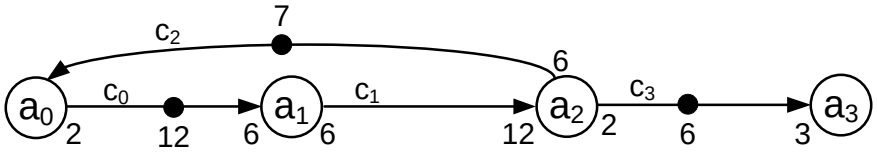
2.6.3 Impact of initial token reduction

We evaluate our token reduction technique on benchmark SDF graphs with buffer sizes identical to the buffers sizes used in the experiments of [33]. The token reduction techniques (i.e., consumer side grouping and producer side grouping introduced in Section 2.6) are applied to the graphs. The second column of Table 2.1 shows the number of initial tokens in the original SDF graphs. Using the consumer side tokens grouping reduces the initial tokens to the amount specified in the third column. Applying the producer side initial token grouping could further reduce the initial tokens (see the fourth column).

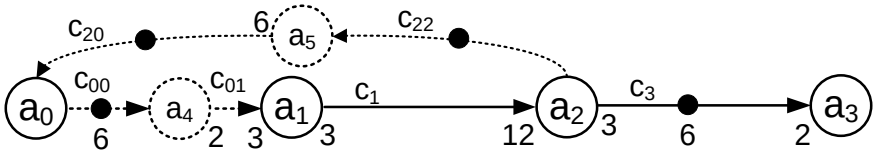
Using the proposed token reduction techniques makes the MPAG-based throughput analysis faster for the buffer-aware graphs; for example in the case of the H.263 decoder (without any token reduction), the throughput analysis lasts 7010 ms. The same analysis for the graph on which consumer side token reduction is applied requires 1910 ms. The analysis time further reduces to 20 ms for the graph on which both token reductions are applied. Table 2.2 contains the throughput analysis times for the benchmark graphs and the transformed graphs. The results confirm that our token reduction techniques are beneficial in order to reduce throughput analysis times. We should also mention that token reductions are performed in a negligible amount of time (< 1 ms) for all benchmark graphs.



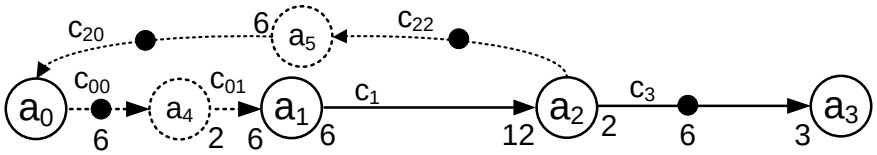
(a) Scenario graph SDG_1.



(b) Scenario graph SDG_2.



(c) Scenario graph SDG_1 after token reduction.



(d) Scenario graph SDG_2 after token reduction.

Figure 2.12: Token reduction applied to an SADF with two scenario graphs SDG_1 and SDG_2.

Table 2.1: Number of initial tokens.

Benchmark	Original	Consum. Optimization	Cons.+Prod. Optimization
H.263 decoder [72]	1193	600	7
H.263 encoder [62]	304	206	10
Modem [9]	54	44	35
MP3 decoder [72]	30	28	28
MP3 playback [81]	1983	106	106
Samplerate conv. [9]	38	14	14
Satellite receiver [64]	1564	791	48

Table 2.2: Throughput analysis times (in milliseconds).

Benchmark	Original	Consum. Optimization	Cons.+Prod. Optimization
H.263 decoder [72]	7010	1910	20
H.263 encoder [62]	390	160	10
Modem [9]	10	<1	<1
MP3 decoder [72]	<1	<1	<1
MP3 playback [81]	35050	4270	4270
Samplerate conv. [9]	<1	<1	<1
Satellite receiver [64]	12860	5200	230

2.7 Summary

In this chapter, we introduce the traditional dataflow model, i.e., the SDF model and a more recent dataflow model for dynamic applications, i.e., the SADF model. The limitations of SDF graphs in modeling dynamic streaming applications were highlighted; an SADF graph, which is a suitable model to capture dynamic behavior of modern streaming applications, was presented with some formal definitions and explanatory examples. Max-Plus algebra, as a strong mathematical tool to analyze dataflow graphs, is also discussed. Performance of an application is quantifiable by measuring the throughput of the model of the application. The available throughput calculation techniques - based on Max-Plus algebra - for dataflow graphs were explained. A model transformation technique is also introduced to provide dataflow graphs with less initial tokens. Reducing the number of initial tokens has a positive effect on the analysis speed, for analysis of properties such as throughput and buffer sizes.

Chapter 3

Dataflow Scheduling

The ever increasing performance gap between processors and memories is one of the biggest performance bottlenecks for computer systems. In this chapter, we propose a technique that schedules an application, modeled with an SADF graph, on a multi-processor system-on-chip (MPSoC) that contains a limited on-chip memory. The proposed scheduling technique explores the trade-off between executing actors in a code-driven (i.e., executing parallel actors) or data-driven (i.e., executing pipelined actors) manner to either minimize the run-time (latency) or maximize the throughput of the application. Our static scheduler identifies those actor sequences in which it is useful to use a code-driven execution and those actor sequences that benefit from a data-driven execution. We extend the proposed technique to consider prefetching when choosing a suitable order. The technique is implemented using an integer linear programming framework. We have published an early version of this work in [20]. Our initial work in [20] was only optimizing the latency and it was only applicable to cycle-free task graphs; in this chapter, we extend our scheduling technique to deal with more general models such as HSDF, SDF and SADF. Moreover, schedules are constructed to also obtain better throughput for the application.

3.1 Overview

MPSoCs are used to fulfill the increasing demand for computational performance of emerging applications. MPSoCs offer a promising solution to the ever-increasing digital electronics market desire for more sophisticated and integrated applications. Nowadays, MPSoCs are used in different electronic devices such as consumer appliances, medical and navigation systems, industrial equipment, etc. Most of these devices run applications that perform complex processing operations on streams of input data. The performance of these devices depends on the performance of the processing elements as well as on the performance of the memory system. Processor performance has been improving by 60% per year [41]. However, memory access times have improved by less than 10% per year [41]. The

resulting performance gap between processor and memories encouraged designers to put more effort into this crucial issue. On-chip memories have been introduced to alleviate this concern. These memories limit the number of off-chip (remote) memory accesses. On-chip (local) memory can be used as caches or as scratch-pad memories (SPMs). SPMs have become an efficient replacement for caches in novel embedded systems, thanks to their lower energy/area cost and better predictability [4]. Due to limitations in the size of local memories (i.e., SPMs), an application (code and data) can only be partially loaded to the SPM. Therefore, applications must be split into several smaller actors (tasks). Instead of loading a whole application to local memory, which requires a large memory space, actors are loaded consecutively one-by-one based on their scheduling order. After the completion of one actor, its code can be discarded from the local memory and its data can be written back to the remote memory such that free space is created for another actor.

Since the focus of this thesis is on dynamic streaming applications, we explain how a dynamic application described by an SADF graph can be scheduled on an MPSoC. The performance of iterative streaming applications is often measured in terms of number of processed data units in a specific amount of time, called throughput. Each iteration of a streaming application processes a data unit. For example, in case of a video decoder, one iteration of a video decoder handles one video frame. The interval between the starting of two consecutive iterations is called the *iteration interval*. The iteration interval has an inverse relation with the throughput of the application. To obtain better throughput, it is desirable to have schedules with shorter iteration interval. Our technique aims to construct schedules with a small iteration interval (or high throughput). Moreover, the objective of our scheduling technique can be adapted to minimize the time needed to execute the application for one iteration, called latency. Latency specifies the response time of the application to a given input. To achieve any of the mentioned objectives, our technique determines an actor execution order that minimizes the total access time on the local and remote memories. It does this by considering the ratio between the code size of the actors and the amount of data that is needed by the actors. Data-intensive actors [28] are scheduled in a data-driven manner and code-intensive actors [28] are scheduled in a code-driven manner.

We implement the proposed scheduling technique using an integer linear programming (ILP) framework. The technique generates a compact ILP formulation. We apply our technique to an application from the multimedia domain (an MP3 decoder) and on a synthetic graph from a closely related paper. The experimental results show that our technique reduces the run-time of the MP3 decoder and the synthetic graph by 7% and 29% respectively when compared to a commonly used technique.

Nowadays, MPSoCs are equipped with direct memory access (DMA) controllers. DMA was devised to liberate processors from transferring data between different memories in a memory hierarchy. Using a DMA unit, the transfer of code or data to/from a local on-chip memory and the execution of an actor on a processor can be overlapped. At the end of this chapter, we show how our technique can be extended to take prefetching into account during the scheduling.

In contrast to scheduling techniques that consider prefetching opportunities in a post-processing step after construction of the schedule, our technique is able to find schedules that make better use of prefetching. The experimental results show that our extended technique reduces the run-time of the MP3 decoder and the synthetic graph by 8% and 32% respectively when compared to a commonly used technique that considers prefetching in a post-processing step.

This chapter is organized as follows. Section 3.2 discusses related work on scheduling in MPSoCs. Section 3.3 explains how we deal with dynamic applications described by SADF graphs. Section 3.4 presents a motivating example. Section 3.5 describes our target MPSoC platform. Section 3.6 specifies the application properties needed in our scheduling technique. Section 3.7 describes the proposed scheduling technique. Section 3.8 extends the proposed technique with the notion of prefetching. Section 3.9 contains an experimental evaluation. Conclusions are drawn in Section 3.10.

3.2 Related Work

There is a rich literature on mapping applications onto MPSoCs [43, 50, 69]. Mostly, this work proposes mapping algorithms followed by a scheduling technique to meet design constraints such as performance, energy consumption, communication cost, or memory usage. In this thesis, we rely on existing mapping methods like the one proposed in [69] and we focus on the scheduling problem. In this chapter, we explore two groups of related work. The first group considers the trade-off between code and data in a pipelined parallel system during actor (task) scheduling. The second group considers code and data prefetching while scheduling the actors (tasks).

As mentioned before, our scheduling technique considers the trade-off between the code-driven and data-driven execution of parallel applications that need to be scheduled on an MPSoC. A similar problem is studied in [84] where the authors propose an evolutionary algorithm to find an optimal schedule for pipelined parallel task graphs. The algorithm does not consider the effect of memory operations. Memory access times have a large impact on the performance of streaming applications. For this reason, and in contrast to [84], we consider memory access times in our scheduling technique. In [52], a technique is presented to generate a data-parallel schedule from applications modeled as SDF graphs. Similar to [84], the authors do not consider the overhead of moving code and data between on-chip and off-chip memory. Furthermore, they implicitly assume the availability of unlimited on-chip memory. Our scheduling technique alleviates both of these issues.

In [16], a prefetching and partitioning technique is presented to minimize the execution time of nested loops by iteratively re-timing the instructions of the loops. The approach maps the instructions of the nested loops to multiple processing units with the objective of increasing parallelism. It only considers the effect of partitioning on the prefetching efficiency. It does not consider the effect of scheduling on the prefetching efficiency in a single partition. When compared

to [16], one aim of our work is finding suitable schedules in order to lower the latency or boost the throughput of applications by maximizing the amount of the possible code/data prefetching in MPSoCs.

Much research has been done on optimizing SPM behavior. In [88], a heuristic is presented to partition variables of an application such that they can be mapped to the on-chip memories of an MPSoC; the heuristic performs task scheduling while considering the effect of scheduling on the variable partitioning. The heuristic assumes that the time needed to access the off-chip memory can be hidden completely through prefetching. In our work, we consider the situation in which off-chip accesses are not negligible. The authors of [42] investigate the use of prefetching for SPM memories. Based on profiling information, they add software prefetching commands inside the application source code to prefetch instructions. Our work is different from this type of work because we use prefetching at an actor level granularity to prefetch both code (instructions) and data. Furthermore, our technique results in a predictable solution because it is computed based on the dependencies extracted from the dataflow models of the applications and not based on an approach like profiling that may cause miss-predictions during the prefetching phase.

The most relevant work to our scheduling techniques is [27] which proposes an ILP-based solution to map an application modeled with a task graph onto a Cell processor. The authors assume that the application code fits in the local memory and they solve the problem of mapping data objects to memories. Scheduling is left to run-time and no design-time analysis is suggested in [27]. Our technique provides a design-time approach for the scheduling problem; it finds an efficient actor order to minimize the run-time of applications without incurring any run-time overhead. The authors of [27] compare their technique with a few well-known heuristics. We evaluate our techniques using the same heuristics.

The most recent related work was published in [17]. It presents a scheduling technique for SDF graphs based on evolutionary algorithms; the technique of [17] takes prefetching into account to minimize the latency of the application. To attain a compact ILP formulation, our scheduler assumes that the memory object of only one actor can be prefetched during the execution of the running actor; the scheduler of [17] has relaxed this restriction. However, in contrast to [17] which focuses on latency, the objective of the scheduler presented in this chapter can be set in a way to either minimize the latency or minimize the iteration interval of the application.

3.3 Scheduling Strategy

An SADF graph employs an SDF graph to capture the behavior of an application operating in a specific scenario. The application may operate in different scenarios. The place where a scenario switch happens is easily distinguishable in an SADF execution. Hence, in principle, a schedule can be constructed for each scenario graph (which is an SDF graph) of the SADF regardless of the scenario occurrence order. However, considering the scenario occurrence order in scheduling each scenario graph may reduce the remote memory access. For example, an

actor may fire in different scenarios; the code memory object of this actor may be required to be loaded to the on-chip memory in two consecutive scenarios; hence, an effective scheduling strategy will try to preserve this memory object in the on-chip memory from the first scenario to the second scenario. In this way, there is no need to reload the same memory object in the second scenario. However, the exact scenario occurrence order is only known in run-time. In this chapter, we address the scheduling problem for each scenario independently, assuming that a scenario repeats indefinitely; this assumption implies that if the current active scenario repeats, the generated schedules will also consider across scenario optimizations. However, in case of different consecutive scenarios, the generated schedules do not consider across scenario interactions.

The fixed port rates of an SDF graph make it possible to statically schedule the SDF graph with a finite schedule per processor. Such a schedule orders the actor firings on the underlying processor. This type of schedules, which are called periodic static-order schedules (PSOSs), can be repeated indefinitely (see Chapter 2). A separate PSOS should be constructed for each processor. Each PSOS only includes actors bound to this specific processor. In the following sections, we introduce a technique to find PSOSs for an SDF graph. Our technique does not only specify a firing order for actors mapped to a specific processor; we also determine the ordering required for memory transactions. Since each actor in an SDF graph may fire several times in one iteration, we use the equivalent HSDF graph of the SDF graph in our scheduling technique. Conversion of an SDF graph to an equivalent HSDF graph is a straight-forward procedure; but, this may cause exponential growth in the graph size. However, this conversion is inevitable when we are exploring a suitable order for all firings of an actor. The generated schedules can be encoded into the S(A)DF model using the technique proposed in Chapter 4. Any subsequent analysis or optimization in a model-driven design flow can then be performed at the S(A)DF level after finding and encoding the right schedules.

In the rest of this chapter, we initially find the fully static schedules which determine absolute start times of actors and memory operations. Then, we derive PSOSs from the fully static schedules.

3.4 Motivation and Preliminaries

We assume that each processor has its own local memory to store code and data. An actor can start its execution on a processor if its code and input data are available in the local memory and when there is enough space in the local memory to store all output produced by the actor. We also assume that a DMA unit is available for each processor.

We show the effect of different scheduling strategies and prefetch-aware scheduling on the performance of an application when running on an MPSoC with a simple example. Figure 3.1 shows a sample HSDF graph of an artificial application. The number close to an actor is the code size of the actor and the number close to a channel is the size of the data that needs to be communicated between the actors. The number inside each actor represents the function of the actor. The actors with the same function number have the same code. For sim-

plicity, we assume in this example that all actors have the same execution time. Assume that actors a_0-a_7 are mapped to one processing element and the remaining actors are mapped to another processing element. For the sake of brevity, we only discuss the scheduling of actors a_0-a_7. Consider the situation in which the processing element, which has to execute these actors, has 40KB of local code memory and 40KB of local data memory. Here, we assume that the remote memory access times have a linear relation with the amount of memory objects to be transferred (i.e., if transferring 100 bytes takes x time units, then transferring 1000 bytes takes $10x$ time units).

Figure 3.2 shows four alternative schedules for (one iteration of) our example graph. Schedule A is a code-driven schedule with prefetching, B is a data-driven schedule with prefetching, C is a combined code and data driven (hybrid) schedule without prefetching, and D is a hybrid prefetch-aware schedule. The blue bars in Figure 3.2 indicate when a processing element (pe) is busy executing an actor. The red bars show the activation of DMA. The green bars indicate that the processing element and DMA are active simultaneously. Schedules C and D are constructed using the scheduling techniques proposed in this chapter. In this example, the goal of our scheduling techniques is to find schedules that minimize the runtime of applications (i.e., that minimize the execution time of a single HSDF iteration). However, this objective can be modified to obtain a lower iteration interval. Generally, our scheduling strategy explores alternative schedules that use a combination of code and data-driven scheduling while taking the impact of prefetching into account.

In Figure 3.1, the horizontal arrow shows the direction of consecutive pipelined actors and the vertical arrow show the direction of parallel actors that may use the same code (i.e., execute the same function). The actors in our example graph can be executed in a *code-driven* or *data-driven* manner. In a code-driven schedule, the code needed for subsequent actors will be reused as much as possible. Hence, the code needs to be loaded only once from remote memory. In a data-driven schedule, the data needed for subsequent actors can remain in the local on-chip memory. In other words, scheduling actors in a data-driven manner avoids moving data between the local and remote memory. Schedule C in Figure 3.2 uses a combination of both code- and data-driven scheduling. Actors a_2 and a_5 are scheduled in a code-driven manner while actors a_0, a_1, a_3, a_4, a_6, and a_7 are scheduled in a data-driven manner. In general, a code-driven strategy gives better performance for actors that have a large code size and a data-driven strategy

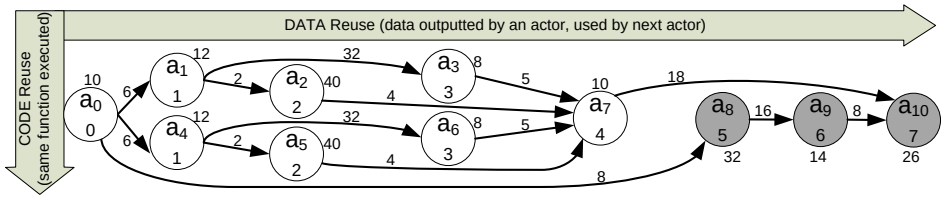


Figure 3.1: An example HSDF graph.

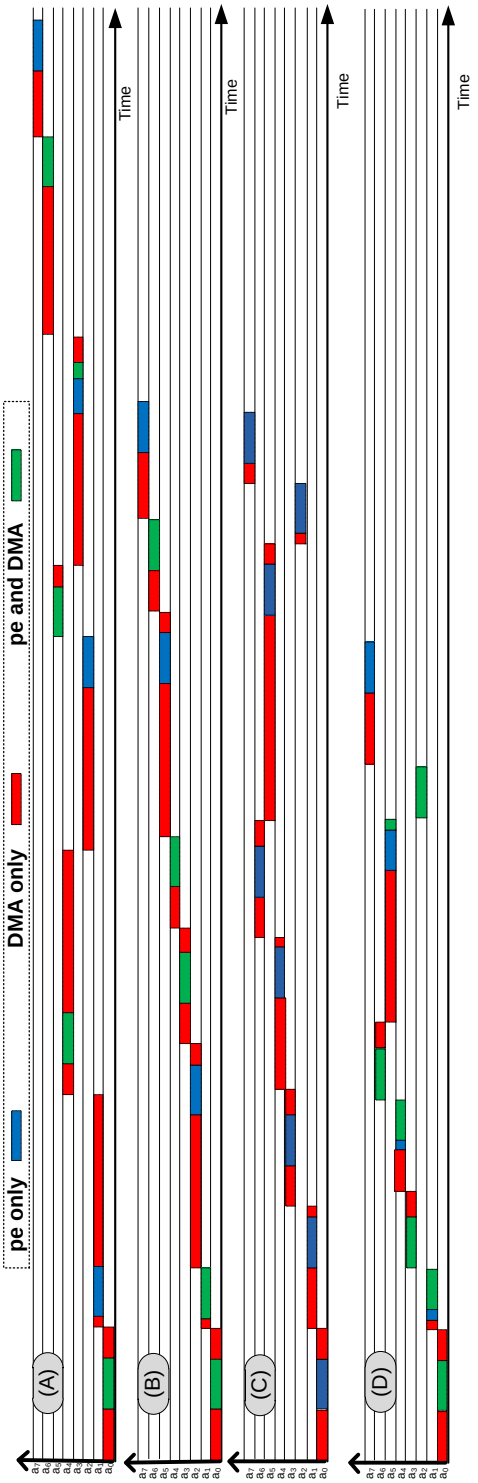


Figure 3.2: Different schedules: (A) code-driven (B) data-driven (C) hybrid (D) hybrid prefetch-aware.

gives better performance for actors that operate on large data objects.

Schedule D is an extension of schedule C that is optimized for prefetching. It can be seen in Figure 3.2 that schedule D is able to keep the processing element and DMA active simultaneously for a longer period of time compared to schedules A and B (i.e., the total size of the green bars in schedule D is larger than the total size of the green bars in schedules A and B). This example shows that the order in which actors are scheduled may have a noticeable impact on the amount of code and data that can be prefetched. This, in turn, has an impact on the overall completion time of the schedule or the resulting iteration interval. Schedule D also provides the smallest iteration interval among all schedules in Figure 3.2.

3.5 MPSoC Platform Template

Figure 3.3 shows the (abstract) MPSoC platform template that is targeted in this work. The platform template consists of a set of processing tiles (PTs) that are interconnected through a shared bus or network-on-chip. Let $|PT|$ denotes the number of processing tiles in the platform. Each processing tile contains a processing element (pe), a code memory (CM), a data memory (DM), and a direct memory access (DMA) unit. The DMA unit can work independently from the processing element and it has direct memory access on the CM and DM as well as the remote memory. A real world example of this type of architecture is the Cell processor [46]. Each processing tile is specified by a pair $pt_i = (mc, md)$ where mc specifies the capacity of the code memory (in bytes) and md specifies the capacity of the data memory (in bytes). We use a constant H to model the per-word read/write latency of the remote memory in terms of clock cycles. Without loss of generality, we assume that the local memory can be accessed within one clock cycle. This is similar to the assumption made in [15]. We also assume that the DMA units of all tiles can work in parallel with each other without causing interference on the interconnect and remote memory. Partitioning remote memory into multiple banks is a common solution [24] to realize this assumption.

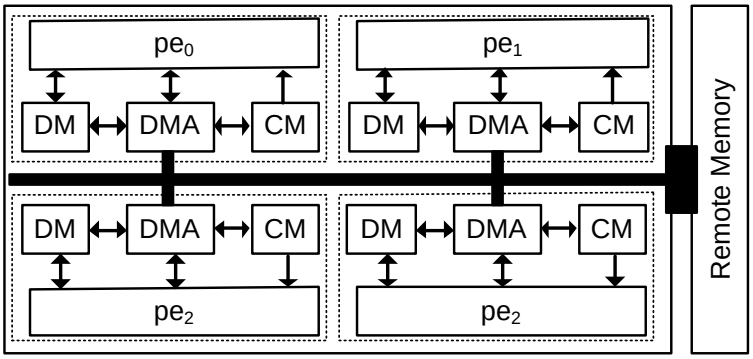


Figure 3.3: Target MPSoC platform template.

3.6 Application Specification and Modelling

An application scenario of an SADF graph described by an SDF graph can be converted to an equivalent HSDF graph [53, 67] denoted by $G = (A, C)$ where A is the set of actors and C is the set of channels (according to Definition 2.2). We assume that the mapping of actors to processing tiles is given. The following functions are used to specify properties of an actor: η, κ, τ and ψ. The notation $\eta(a_i)$ specifies the processing tile to which the actor a_i is mapped, $\kappa(a_i)$ is the code size of the actor a_i (in bytes), $\tau(a_i)$ is the execution time of the actor a_i (in cycles), and $\psi(a_i)$ is the function identifier of the actor a_i. Actors that have the same function identifier require the same code to be executed. When such actors are executed immediately after each other on the same processing element, then we only need to load the code of these actors prior to the execution of the first actor. The following functions are used to specify properties of a channel: ϵ and ϑ. The notation $\epsilon(c_i)$ indicates the communication delay in cycles and is equal to the number of required processor clock cycles to transfer a data object from one local memory to another local memory. The notation $\vartheta(c_i)$ specifies the size of a token (in bytes) getting transferred via c_i.

Our technique initially adds some extra actors and channels to the input HSDF graph. These elements are added to include the mapping information to the given graph. Initially, we add two dummy actors for each processing element in the platform. Consider $a_{first,i}$ and $a_{last,i}$ as the dummy actors added for processing element pe_i. Three sets of dependencies are created for each processing element in the platform as follows: (1) a dependency is created by placing a channel from the dummy actor $a_{first,i}$ to each actor mapped to pe_i; (2) a dependency is created by placing a channel from each actor mapped to pe_i to the dummy actor $a_{last,i}$; (3) a channel with one initial token is placed from $a_{last,i}$ to $a_{first,i}$. The dummy actor $a_{first,i}$ is the first actor able to fire on the processing element pe_i and the dummy actor $a_{last,i}$ is the last actor to fire on the same processing element. The added initial token for processing element pe_i will reset to its initial place after all actors mapped to pe_i have completed their firing. The time between consumption and production of this initial token represents the duration of one iteration on processing element pe_i. Figure 3.4 depicts the graph shown in Figure 3.1 after adding all dummy elements (shown with dotted lines). Actors $a_{first,1}$ and $a_{last,1}$ are added for processing element pe_1 and actors $a_{first,2}$ and $a_{last,2}$ are added for processing element pe_2. The dummy actors have an execution time and code size zero. A dummy actor added for a processing element is assumed to be mapped on that processing element. The dummy channels have a size and delay of zero.

3.7 Hybrid Scheduling

In this section, we provide an integer linear programming (ILP) formulation to solve the scheduling problem. The proposed scheduling technique explores the trade-off between executing actors from the graph in a code or data-driven manner. The scheduler identifies actors with a large code size and consecutive actors

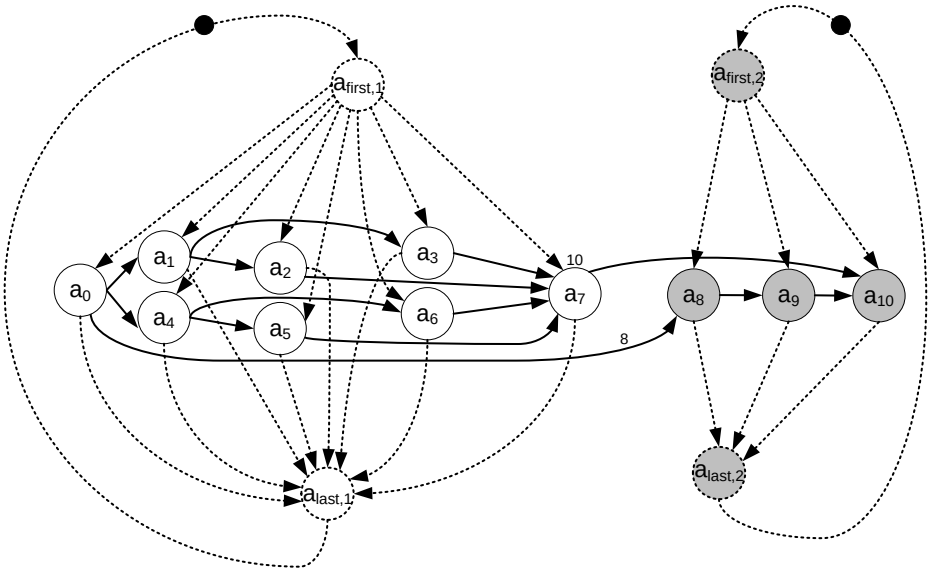


Figure 3.4: The HSDF graph of Figure 3.1 amended with mapping information.

with a large communication data size. The scheduler chooses a proper scheduling order for all actors in the graph to optimize the desired objective. The objective can be set either to minimize the latency or to minimize the iteration interval (inverse of the throughput) of the application. We call the proposed technique *Hybrid ILP (HybILP)*. All necessary elements to form the ILP formulation are explained in the following subsections.

3.7.1 ILP Variables

We consider two groups of ILP variables in our formulation. The first set (denoted by χ) models the start times of the actors in the graph. These start times also form the final solution of the scheduling problem. The second set (denoted by π) captures the ordering of the actors that are running on the same processing element.

χ_i start time of the actor a_i

$$\pi_{ij} = \begin{cases} 1 & \text{if actor } a_j \text{ scheduled immediately after actor } a_i \\ 0 & \text{otherwise} \end{cases}$$

Two other variables are also considered in our ILP formulation: *II* represents the iteration interval of the application and *LAT* represents the latency of the application. The variable *LAT* is an auxiliary variable in our ILP formulation which has the following property:

$$\forall 1 \leq i \leq |PT| \rightarrow \chi_{last,i} \leq LAT \tag{3.1}$$

Based on the set of π variables (described above), we define a notation to model the initialization time ($I\!T$) of an actor which is the time needed to complete all required operations before the actor can be executed. Figure 3.5 shows a simple graph with four actors. Assume that actor a_j will be executed immediately after actor a_i on the same processing element. The following operations are necessary before the execution of a_j:

- WD_{ij}: Write the necessary output data of the actors executed before actor a_j to the remote memory (O1 and O2 in Figure 3.5).
- RD_{ij}: Read the necessary input data of actor a_j from the remote memory (O3 and O4 in Figure 3.5).
- RC_{ij}: Read the necessary code of actor a_j from the remote memory (O5 in Figure 3.5).

In some situations, it is not necessary to perform all these initialization operations. Analysis of the dependencies between actors reveals when certain operations can be skipped:

- It is unnecessary to read/write the intermediate data between consecutive actors from/to the remote memory, when the intermediate data can be used immediately by the next scheduled actor (i.e., $O2$ and $O3$ in Figure 3.5).
- It is unnecessary to load the code of an actor from the remote memory if its function is the same as the previously executed actor since the code of the actor already exists in the local memory ($O5$ in Figure 3.5 can be skipped if a_i and a_j execute the same function).

Assume that actors a_i and a_j in the example execute the same function. The initialization time of actor a_j, i.e., $I\!T_j$, is then equal to:

$$I\!T_j \texttt{ = (Time of } O1\texttt{) + (Time of } O4\texttt{)} \tag{3.2}$$

We define a term denoted by $distance(a_i, a_j)$ that expresses the number of actors - mapped to the same processing element as a_i and a_j - that exist in a simple

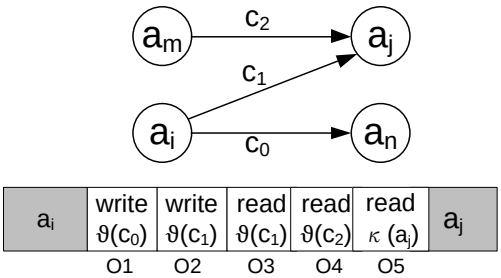


Figure 3.5: Initialization steps.

path from a_i to a_j. In case of multiple paths from a_i to a_j, the path with the highest value represents the distance. As an example, consider the HSDF graph shown in Figure 3.6; in this figure, actors filled with gray color are mapped to processing element pe_1 and the rest of the actors are mapped to processing element pe_2. Consider actor a_6 in Figure 3.6; this actor should be scheduled on processing element pe_1 directly either after a_1 or after a_4 where both $distance(a_1, a_6)$ and $distance(a_4, a_6)$ are equal to zero. When there is no path from a_i to a_j, the distance is set to -1. For example $distance(a_3, a_5)$ in Figure 3.6 is set to -1. In the case of HSDF graphs, existing tokens on a channel make the first firing of the consumer side actor of the channel independent from the firing of the producer side actor of the channel. Hence, we ignore channels with initial tokens when we calculate the distance values. Note that in deadlock-free graphs, cycles of channels without initial tokens do no exist, which means that defining distance as the path with the highest value is well defined.

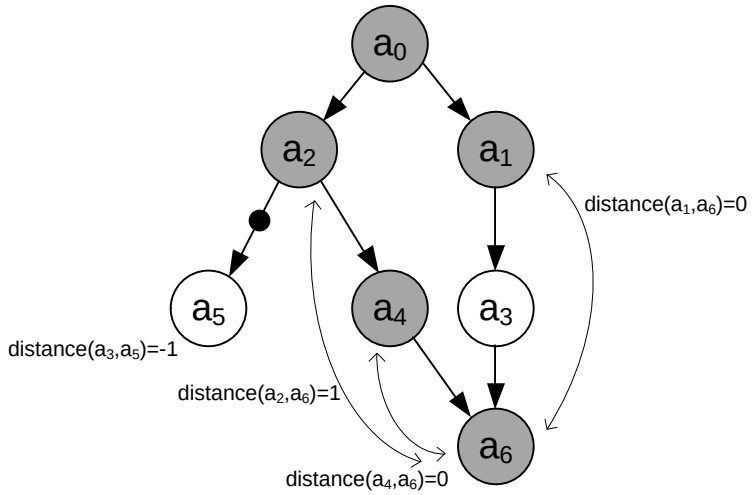


Figure 3.6: An example HSDF graph.

Using the distance concept, we can define the initialization time of an actor a_j as follows:

$$IT_j = \sum_{a_i \in A | \eta(a_j) = \eta(a_i) \wedge distance(a_i, a_j) \leq 0} \pi_{ij} \cdot IT_{ij} \tag{3.3}$$

In Equation 3.3, only actors mapped to the same processing element as a_j with a distance of zero or -1 are considered as an actor that can fire directly before a_j. In Equation 3.3, IT_{ij} is the initialization time of actor a_j when its previous actor is known to be actor a_i. IT_{ij} can be computed as follows:

$$IT_{ij} = H \cdot (WD_{ij} + RC_{ij} + RD_{ij}) \tag{3.4}$$

IT_{ij} is computed by multiplying the size of the memory objects to be transferred to/from the remote memory (i.e., summation of WD_{ij}, RC_{ij}, and RD_{ij})

with the latency of the remote memory (H). In the remainder of this subsection, it is shown how each part of Equation 3.4 can be computed. We assume that the size of the actors in the graph and the size of the local memories are of the same order of magnitude. In other words, the code size of the actors and the intermediate data between the actors are assumed to be similar in size. Therefore, we do not consider the situation in which the memory objects of two different actors can be placed in the local memory at the same time. These assumptions are in line with the objective of this work, which is optimizing the memory behavior of an application running on an MPSoC with limited local memories.

3.7.1.1 Size of the output data

The size of the output data produced by the actor a_i which needs to be written to the remote memory is given by the next equation. It is assumed that actor a_j is scheduled immediately after actor a_i.

$$WD_{ij} = \underbrace{\sum_{c \in OutC(a_i)} \vartheta(c)}_{\alpha} - \underbrace{\sum_{c \in OutC(a_i) \cap InC(a_j)} \vartheta(c)}_{\beta} \tag{3.5}$$

In Equation 3.5, α is equal to the size of all output data produced by the actor a_i and β is the size of all data produced by actor a_i and used by actor a_j. The ILP solver could decide to schedule actors that communicate large data objects between each other in a consecutive order to get a lower initialization time (i.e., a data-driven scheduling strategy could be selected).

3.7.1.2 Size of the code

The size of the code needed to be fetched from memory to execute actor a_j, which is scheduled immediately after actor a_i, is as follows:

$$RC_{ij} = \begin{cases} 0 & \psi(a_i) = \psi(a_j) \\ \kappa(a_j) & \text{otherwise} \end{cases} \tag{3.6}$$

Equation 3.6 could force the ILP solver to schedule large actors with the same functionality in a consecutive order (i.e., a code-driven scheduling strategy could be selected).

3.7.1.3 Size of the input data

The size of the input data that needs to be read from remote memory to execute actor a_j, which is scheduled immediately after actor a_i, is as follows:

$$RD_{ij} = \underbrace{\sum_{c \in InC(a_j)} \vartheta(c)}_{\alpha} - \underbrace{\sum_{c \in OutC(a_i) \cap InC(a_j)} \vartheta(c)}_{\beta} \tag{3.7}$$

In Equation 3.7, α is equal to the size of all input data needed for actor a_j and β is equal to the intermediate data between actor a_i and actor a_j. As

before, the ILP solver could decide to schedule the actors a_i and a_j immediately after each other if their intermediate data is large (i.e., a data-driven scheduling strategy could be selected).

3.7.2 ILP Objective Function

The goal of our optimization is either minimizing the latency (i.e., LAT) or minimizing the iteration interval (i.e., II) of the input graph. Depending on the desired goal, one of the following equations can be used as the objective function of our ILP formulation. Equation 3.8 should be used to obtain schedules with minimum latency and Equation 3.9 should be used to obtain schedules with minimum iteration interval.

$$\texttt{Objective: Minimize } LAT \tag{3.8}$$

$$\texttt{Objective: Minimize } II \tag{3.9}$$

3.7.3 ILP Constraints

This subsection introduces the constraints that are used in our ILP formulation. These constraints force the ILP solver to satisfy essential properties of the application (i.e., data dependencies) and intrinsic properties of the scheduling problem (i.e., avoid resource conflicts).

3.7.3.1 Data dependency constraints

The dependencies between actors should be considered in our ILP formulation; this can be done by including ILP constraints suggested in Equation 3.10. These constraints are based on Equation 7 presented in [59]. The notation $\omega_0(c)$ represents the number of initial tokens in channel c.

$$\forall c = (a_i, a_j) \in C \rightarrow \chi_i + \tau(a_i) + \epsilon(c) + II_j \leqslant \chi_j + \omega_0(c) \cdot II \tag{3.10}$$

The constraint states that the start times of actors a_i and a_j should allow sufficient time to execute actor a_i, to transfer data from actor a_i to actor a_j and to initiate actor a_j. The actor a_j can fire $\omega_0(c) \cdot II$ time units earlier than its allowed time if $\omega_0(c)$ initial tokens exist on the channel c.

3.7.3.2 Resource conflict constraint

Two actors cannot be executed on the same processing element at the same time. Any pair of actors that has this possibility should be identified; then, some ILP constraints should be considered for that pair of actors to prevent such a resource conflict. Two actors can compete for a processing element if no dependency exists between them. This condition can be verified by checking whether or not a path with no initial token exists between two actors in the initial HSDF graph. If a path

with no initial token exists between two actors, the constraints originating from dependencies in the graph (i.e., Equation 3.10) can prevent the resource conflict between those two actors. If no path exist between two actors a_i and a_j, two ILP constraints should be added to our formulation according to Equation 3.11. One constraint considers that a_i fires before a_j and the other constraint considers the contrary. However, only one of these statements finally should hold. This is ensured by adding other constraints introduced later (i.e., Equation 3.13 and Equation 3.14). In Equation 3.11, MAXINT is a large integer value that exceeds the sum of the execution times of all actors in the given graph. The following constraint ensures sufficient time between firing of a_i and a_j if the ILP-solver decides to schedule a_j immediately after a_i (i.e., π_{ij} is set to 1 by the ILP-solver).

$$\begin{aligned}\forall a_i, a_j \in A \ \wedge \ \eta(a_i) = \eta(a_j) \ \wedge \ distance(a_i, a_j) = distance(a_j, a_i) = -1 \\ \rightarrow \chi_i + \tau(a_i) + \epsilon(c) + \pi_{ij} \cdot \mathit{\Pi}_{ij} \leqslant \chi_j + (1 - \pi_{ij}) \cdot \texttt{MAXINT}\end{aligned} \tag{3.11}$$

As an example, consider again the HSDF graph shown in Figure 3.6. When there is no path between actors a_i and a_j, both $distance(a_i, a_j)$ and $distance(a_j, a_i)$ are equal to -1. In this case a_i and a_j are able to compete with each other and their firings should be ordered. For example in Figure 3.6, both $distance(a_1, a_2)$ and $distance(a_2, a_1)$ are equal to -1. Hence two ILP constraints using Equation 3.11 should be added for a_1 and a_2.

3.7.3.3 Other constraints

The next equation enforces positive start times for all actors.

$$\forall a_i \in A \rightarrow \chi_i \geqslant 0 \tag{3.12}$$

The next equation enforces that only one actor is allowed to be scheduled immediately after each actor. Actors with distance -1 or 0 from actor a_i can be scheduled directly after a_i; the equality of Equation 3.13 ensures only one of those actors is scheduled directly after a_i.

$$\forall a_i \in A \rightarrow \sum_{a_j \in A | \eta(a_i) = \eta(a_j) \wedge distance(a_i, a_j) \leq 0} \pi_{ij} = 1 \tag{3.13}$$

The next equation enforces that only one actor is allowed to be scheduled immediately before each actor. Actors with distance -1 or 0 to actor a_i can be scheduled directly before a_i; the equality of Equation 3.14 ensures only one of those actors is scheduled directly before a_i.

$$\forall a_i \in A \rightarrow \sum_{a_j \in A | \eta(a_i) = \eta(a_j) \wedge distance(a_j, a_i) \leq 0} \pi_{ji} = 1 \tag{3.14}$$

3.8 Hybrid Prefetch-Aware Scheduling

In this section, we refine the HybILP scheduler of Section 3.7 to consider prefetching. We call this technique *Hybrid-Prefetch-ILP* (*HybPrefILP*). To attain a compact ILP formulation, we assume that the memory object of only one actor can

be prefetched during the execution of the running actor. This assumption reduces the number of prefetching options which leads to a smaller ILP formulation. As a result of this limiting assumption, the outcome of our scheduler may become sub-optimal, but it is practical. However, two limitations for prefetching often prevent the loading of many memory objects into the local memory. The first limitation is related to the available free space in the local memory. The second limitation comes from the limited time that is available to perform prefetching. Usually, these limitations are barriers to prefetch more than one actor in a realistic application. Hence, this assumption does not typically affect the quality of the solution significantly. The scheduling technique of [17] can be used when larger on-chip memories are available.

We refine the HybILP technique by changing the elements of Equation 3.4. The size of the output data that should be written back to the remote memory (WD_{ij}) is independent from prefetching, but the size of the code (RC_{ij}) and data (RD_{ij}) that must be fetched depends on the amount of code and data that is already prefetched. We introduce two new constants RC'_{ij} and RD'_{ij} that capture respectively the size of code and data that need to be fetched after the prefetching has ended. These constants replace RC_{ij} and RD_{ij} in Equation 3.4. All other parts of the ILP formulation in the HybILP technique can be used without any change.

3.8.1 Size of the code

Assume that actor a_j is scheduled after actor a_i. The size of the code that needs to be fetched from the memory after the execution of actor a_i has ended (i.e., RC'_{ij}) depends, amongst others, on the amount of free space that is left in the local code memory. This code space limitation (CSL) is given by:

$$CSL_{ij} = [\texttt{mc of } pt_{\eta(a_j)}] - \kappa(a_i) \tag{3.15}$$

The notation $[\texttt{mc of } pt_{\eta(a_j)}]$ represents the code memory capacity of the processing element to which a_j has been mapped (i.e., the processing element $pt_{\eta(a_j)}$).

RC'_{ij} depends also on the execution time of the previous actor as this influences the time during which the DMA and processing element can run in parallel (i.e., code can be prefetched). This temporal limitation (TL) is given by:

$$TL_{ij} = \left\lfloor \frac{\tau(a_i)}{H} \right\rfloor \tag{3.16}$$

We assume that data prefetching is done after code prefetching. The total amount of the code that can be prefetched does not only depend on the space and temporal limitations discussed above. It is also limited by the code size of the actor (i.e., RC_{ij} as defined in Equation 3.6). The code prefetched amount (CPA) is given by:

$$CPA_{ij} = \min(RC_{ij}, CSL_{ij}, TL_{ij}) \tag{3.17}$$

So, the total size of the code that needs to be fetched, when executing actor

a_j directly after actor a_i, is equal to:

$$RC'_{ij} = RC_{ij} - CPA_{ij} \tag{3.18}$$

3.8.2 Size of the input data

Assume once more that actor a_j is scheduled after actor a_i. The amount of data that needs to be fetched from remote memory to execute actor a_j (i.e., RD'_{ij}) depends on the amount of data that could be prefetched during the execution of actor a_i. The data prefetched amount depends on the amount of free space in the local data memory. This so-called data space limitation (DSL) is given by:

$$DSL_{ij} = [\texttt{md of } pt_{\eta(a_j)}] - \sum_{c \in InC(a_i) \cup OutC(a_i)} \vartheta(c) \tag{3.19}$$

The notation $[\texttt{md of } pt_{\eta(a_j)}]$ represents the data memory capacity of the processing element to which a_j has been mapped.

The data prefetched amount depends also on the running time of the actor which is executed directly before actor a_j and the amount of the time that the DMA was busy with transferring the code; the latter needs to be considered as we assume that data prefetching is done after code prefetching. This so-called data temporal limitation (DTL) is given by:

$$DTL_{ij} = TL_{ij} - CPA_{ij} \tag{3.20}$$

The data prefetched amount is limited by the data space limitation, the data temporal limitation, and the actual data memory requirement of the actor (i.e., RD_{ij} as defined in Equation 3.7). Hence, the data prefetched amount is equal to:

$$DPA_{ij} = \min(RD_{ij}, DSL_{ij}, DTL_{ij}) \tag{3.21}$$

So, the total size of the data that needs to be fetched, when executing actor a_j directly after actor a_i, is equal to:

$$RD'_{ij} = RD_{ij} - DPA_{ij} \tag{3.22}$$

3.8.3 Post-processing

The HybPrefILP scheduler uses the assumption that pre-fetching of memory objects for actor a_j can only be started when its direct predecessor in the schedule (e.g., actor a_i) has started. In practice it may sometimes be possible to start the prefetching for actor a_j earlier. This may lead to a shorter completion time of the schedule. Consider as an example the schedules shown in Figure 3.7. All actors in this figure are mapped to a single processing tile with 30KB for each of the code and data memories. Schedule A is generated using our HybPrefILP scheduler. This schedule is sub-optimal as the prefetching for actor a_2 is only

Figure 3.7: Post-processing of a schedule.

started when actor a_1 starts its execution. In practice, it might be possible to start the prefetching of a_2 earlier. Schedule B uses the property that prefetching can be started earlier (i.e., when a_0 is executing). Extending our ILP formulation such that it also considers options in which the prefetching is started earlier would result in a large increase in the number of variables. Therefore, we have decided to keep our current assumption with respect to the start time of prefetching operations. However, we have included a post-processing step that optimizes the schedule. This post-processing step tries to fill the free space of the local memory by prefetching memory object of subsequent actors based on the actor order generated by the HybPrefILP scheduler.

Algorithm 1 contains the proposed post-processing step after generating the schedules. This algorithm accepts a schedule s_x generated using the HybPrefILP scheduler for a processing tile pt_x. The algorithm prints the possible prefetchings during the execution sequence indicated by s_x. Note that this algorithm verifies whether the time limit and on-chip memory space allows prefetching more memory objects or not. The data dependencies may also limit the prefetching; at run-time, it can be checked whether or not a data object is produced and is available to be prefetched.

In Algorithm 1, we use *cpc* and *dpc* to indicate up to which point in s_x the code objects and data objects are prefetched respectively. For example, *cpc* equal to 6 indicates that the code objects of the actors from the currently running actor in s_x till the 6^{th} actor in s_x are available in the code memory. *codeBudget* and *dataBudget* specify the free space in the code and data memories respectively. Lines 1-4 in Algorithm 1 initialize these variable. The for-loop of lines 5-32 iterates for each element in the input schedule s_x; variable i specifies the currently running actor. For each actor in s_x, first it is checked whether or not the code object of the i^{th} actor in s_x exists in the code memory (line 6); if not, the related code memory should be loaded to the code memory and the related variables are updated accordingly (lines 7-9). Similar operations are performed for the data objects of the i^{th} actor in s_x (lines 10-13). *prefBudget* specifies the prefetching limit imposed by the execution time of the currently running actor (line 15). The inner for-loop in lines 16-30 checks how many actors can be prefetched while $s_x[i]$ is running. Lines 17-22 checks for the prefetching possibility of code objects and lines 23-30 do the similar check for data objects. After completion of the i^{th} actor in s_x, the allocated memory space to this actor can be released (lines 31-32).

Algorithm 1: Prefetch post-processing

input : PSOS s_x
input : PT pt_x
output: Will be printed out

```
1  cpc = 0                                          /* Code prefetched counter */
2  dpc = 0                                          /* Data prefetched counter */
3  codeBudget = [mc of pt_x]
4  dataBudget = [md of pt_x]
5  for i ← 1 to |s_x| do
6      if i > cpc then
7          PRINT "Load the code object of s_x[i]"
                                  /* Reserve the code memory space of s_x[i] */
8          codeBudget = codeBudget − κ(s_x[i])
9          cpc = i
10     if i > dpc then
11         PRINT "Load the data object of s_x[i]"
                                  /* Reserve the data memory space of s_x[i] */
12         dataBudget = dataBudget − Σ_{c∈InC(s_x[i])∪OutC(s_x[i])} ϑ(c)
13         dpc = i
14     PRINT "s_x[i] can execute"
15     prefBudget = ⌊τ(s_x[i]) / H⌋
16     for j ← cpc + 1 to |s_x| do
                  /* Possible to prefetch the code object of s_x[j]?  */
17         if prefBudget > 0 & codeBudget > 0 then
18             PRINT "Prefetch the code object of s_x[j] while s_x[i] is running"
19             prefBudget = prefBudget − κ(s_x[j])
20             codeBudget = codeBudget − κ(s_x[j])
21             if prefBudget ≥ 0 & codeBudget ≥ 0 then
22                 cpc = cpc + 1
                   /* Possible to prefetch the data objects of s_x[j]?  */
23         if prefBudget > 0 & dataBudget > 0 then
24             PRINT "Prefetch the data object of s_x[j] while s_x[i] is running"
25             prefBudget = prefBudget − Σ_{c∈InC(s_x[j])∪OutC(s_x[j])} ϑ(c)
26             dataBudget = dataBudget − Σ_{c∈InC(s_x[j])∪OutC(s_x[j])} ϑ(c)
27             if prefBudget ≥ 0 & dataBudget ≥ 0 then
28                 dpc = dpc + 1
29         else
30             break                    /* No more prefetching is possible */
                                /* Release the memory space allocated to s_x[i] */
31     codeBudget = codeBudget + κ(s_x[i])
32     dataBudget = dataBudget + Σ_{c∈InC(s_x[i])∪OutC(s_x[i])} ϑ(c)
```

At first glance, the complexity of Algorithm 1 appears to be $O(n^2)$ ($n = |s_x|$) because of two nested loops. A precise look at the inner loop at line 16 reveals that the whole block (lines 16-30) iterates at most $n-1$ times in a complete execution of the algorithm; because each iteration of the block represents the prefetching of the memory objects of one actor and in total memory objects of $n-1$ actors may be prefetched in one execution of s_x. Hence, the inner loop cannot affect the complexity of the algorithm; as a result, the complexity of Algorithm 1 is limited to $O(n)$.

3.8.4 PSOS Generation

The obtained schedules using our technique are fully static schedules which determine the exact starting time of each actor. However, the later analyses in a typical model-driven design flow often require PSOSs. A PSOS can be generated from a fully static schedule by ordering the actors based on their start time. For example, consider the fully static schedule D in Figure 3.1; sorting the actors of this schedule based on their start time results in the PSOS $PSOS_D = \langle a_0 a_1 a_3 a_4 a_6 a_5 a_2 a_7 \rangle^*$. The PSOSs are often needed to be captured in the model. The PSOSs found for an HSDF graph can be modeled using the technique presented in [3]. The equivalent HSDF graph of a scenario graph usually has many actors and channels. Hence, it is preferable to directly perform the analysis on the SADF graph. To enable this possibility, we developed a schedule modeling technique to directly model PSOSs in SADF graphs. The next chapter explains our schedule modeling technique in detail.

3.9 Experimental Results

The proposed scheduling technique takes an application graph and MPSoC platform as input. It generates an ILP formulation which is solved using CPLEX [44]. CPLEX is executed on a Linux platform with an Intel® Core^{TM} i7 running at 2.67GHz and 4GB of internal memory. Large execution times are often mentioned as an important drawback of using an ILP-based solution. Thanks to our compact ILP formulation, which avoids unnecessary variables and constraints, the execution time of the ILP solver when looking for a schedule never was more than a minute for the selected graphs in our experiments (for multi-processor cases). For example, the HybILP technique needs a run-time of 0.76 seconds to schedule an MP3 decoder. The HybPrefILP requires 0.96 seconds to schedule the same application while also considering prefetching. However, in single-processor cases, more pairs of actors are considered in Equation 3.11 and more variables are required in the ILP formulation; as a result, the run-time of the ILP solver can raise to hours in single-processor cases.

To make a comparison between our technique and related work, we implement the greedy CPU (G-CPU) scheduling technique from [27]. We also implement the *heterogeneous earliest finish time* (HEFT) scheduling technique [78] which is a commonly used heuristic-based mapping and scheduling technique. The existing techniques find schedules that are minimizing the latency of the applications.

To the best of our knowledge, our approach is the first work which considers off-chip memory interactions while scheduling a streaming application on a multi-processor platform in order to provide better throughput. Hence, we are only able to assess our technique versus related approaches when latency has been chosen as the target optimization.

We use three different models to evaluate our scheduling technique. The first graph models an MP3 decoder. This application is a frequently used application from the multimedia domain. We manually extract a DAG for this application (see Figure 3.8). We estimate the execution time of all actors and their memory requirements when they would be executed on an ARM7TDMI core. An SADF model of an MP3 decoder is also used in our experiments. The scenarios of this SADF are identical to the scenarios of the model given in [32], while we used the profiling information acquired for the ARM7TDMI core. A second DAG is taken from [27]. It is a synthetic graph with 50 actors. In our experiments, we assume that the latency of the remote memory is 100 times larger than the local memories and access to the local memory will take one clock cycle. These assumptions are in line with realistic values for memory access latencies [36].

3.9.1 MP3 Decoder DAG

MP3 decoding is a frame-based algorithm that transforms a compressed stream of data into pulse code modulation (PCM) data. Figure 3.8 shows the DAG of the MP3 decoder. As the graph of this application is composed of two symmetric sub-graphs, we map each sub-graph onto a single processing element. This mapping is depicted in Figure 3.8; the upper sub-graph is mapped to the first processing element (pe_0) and the lower sub-graph is mapped to the second processing element (pe_1).

We apply the HybILP, HybPrefILP, HEFT and G-CPU scheduling techniques to the graph of the MP3 decoder. The HEFT and G-CPU result in the same outcome; the result of the HybILP and HybPrefILP scheduling is different from these techniques. HEFT and G-CPU schedule all actors of the MP3 decoder in a code-driven order. The HybILP and HybPrefILP schedule part-a of the graph (shown with an arrow in Figure 3.8) in a data-driven order. In part-a, the size of the intermediate data is larger than the code size of the actors. By using a data-driven strategy, the data outputted by one actor is consumed immediately by the next actor. Therefore, there is no need to store/load this large intermediate data to/from the remote memory. The code sizes of the actors are larger than the size of the intermediate data in part-b of the MP3 decoder graph. The HybILP and HybPrefILP schedule part-b in a code-driven order. This decision leads to a reduction in the number of off-chip memory accesses because with the code-driven strategy each actor only needs to be loaded once from the remote memory. As a result of these scheduling strategy decisions, HybILP achieves a schedule with a 7% shorter execution time (in terms of cpu cycles) when compared to HEFT and G-CPU.

It is possible to apply prefetching to the schedules that are obtained using HEFT (or G-CPU). The order of actor firings is taken from the obtained sched-

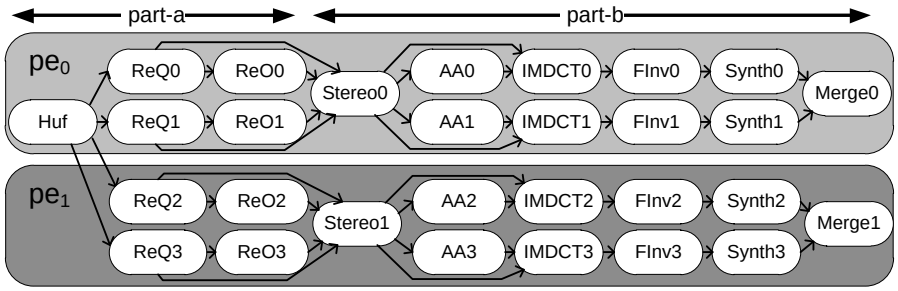


Figure 3.8: The graph of an MP3 decoder.

ules. The actors are executed according to these schedules; during the execution of an actor, it is checked whether or not it is possible to prefetch the memory objects of the actors following the currently running actor. In this scheme, the prefetching acts as a post-processing step and it cannot affect the actor firing order to provide shorter schedules. In contrast, our HybPrefILP technique is able to find shorter schedules by taking prefetching into account at the same time that our hybrid scheduler (i.e., HybILP) orders the actors. The HybPrefILP technique finds a schedule with an 8% shorter execution time when compared to extended versions of HEFT and G-CPU in which prefetching is considered after obtaining the schedules. These results show that our scheduling technique is able to construct schedules that are significantly faster when compared to existing well-known scheduling techniques. The schedule constructed by our hybrid prefetch-aware technique is 11% faster when compared to a the non-prefetch-aware schedule constructed by our technique. This shows the advantage of using prefetching to decrease the run-time of applications.

3.9.2 MP3 Decoder SADF

We have applied our scheduling technique to an SADF model of the MP3 decoder application to demonstrate the applicability/scalability of our scheduling technique. We construct an SADF model for the MP3 decoder application using the scenario information for the MP3 decoder SADF graph from [32] and the profiling information from the MP3 decoder DAG (i.e., execution times and memory sizes).

Applying our scheduling technique on all scenario graphs of the MP3 decoder SADF graph results in similar schedules for all scenario graphs. This is due to the similar memory behavior for this application across different scenarios. The analysis time of our scheduler scales with the number of scenarios in this application. For example, the MP3 decoder SADF model used in our experiment has five scenarios. HybPrefILP technique takes 4.9 seconds to schedule the MP3 decoder SADF bound to a platform with two ARM7TDMI cores; the mentioned analysis time is almost 5 times the analysis time required to schedule the equivalent MP3 decoder DAG; this factor conforms to the number of scenarios in the application. Optimizing schedules for both throughput and latency yields the same result for the MP3 decoder SADF; this originates from the fact that the application model

is partitioned evenly on two processing elements (see Figure 3.8) except for actor *Huf* in the model and two partitions can run in parallel independently after firing of actor *Huf*.

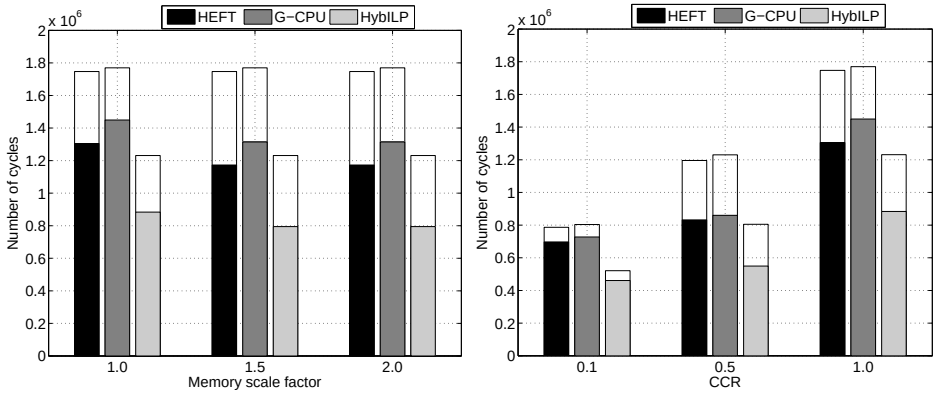
3.9.3 Synthetic DAG

To further verify the effectiveness of our proposed technique we select the synthetic DAG used in [27]. The mapping decisions computed by HEFT are used as input to our scheduling technique. We evaluate our technique for three different experimental set-ups: first, different sizes of local memories; second, different amounts of communication-to-computation ratios (CCRs) in the graph; third, different numbers of processors in the platform.

The size of the local memory can affect the amount of prefetching. We determine the necessary amount of local memory which is required to execute the graph. Figure 3.9(a) shows the execution time of the graph (in the number of processor cycles) for different sizes of the local memory, for two processors, and a CCR of 1.0. The memory scale factor in Figure 3.9(a) is the scaling factor of the necessary amount of the local memory. Scaling local memory size from 1.0 to 1.5 times decreases the run-time of the application when using HybPrefILP by 10%, when using HEFT with prefetching by 10%, and when using G-CPU with prefetching by 9%. Scaling local memory size from 1.5 to 2.0 times does not further reduce the run-time of the application. This is due to another limitation of prefetching which is the time limitation (see Equation 3.16). These results show that the effect of prefetching is similar in all these three scheduling techniques. However, overall, the HybPrefILP gives a schedule which is 32% and 39% faster compared to HEFT with prefetching and G-CPU with prefetching respectively for all local memory scaling factors in Figure 3.9(a).

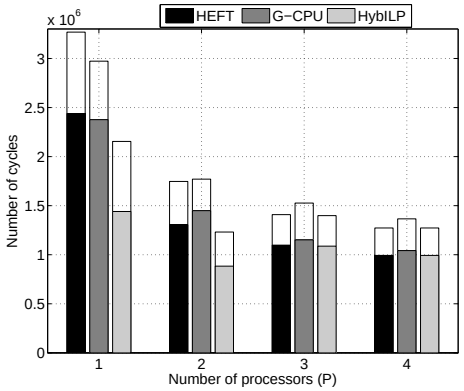
The number of off-chip memory accesses determines the required amount of communication in a multi-processor system. Hence, we explore the effectiveness of our proposed scheduling (HybILP) for different amounts of communication to computation ratios. For this purpose, we derive several graphs from the original graph by scaling the size of the memory objects. CCR is a term for digitizing the amount of communication in an application. A larger CCR implies a larger number of remote memory accesses. Figure 3.9(b) shows the execution time of the graph (in the number of processor cycles) for different CCRs when the graph is mapped to a platform with two processors and the memory scale factor is 1.0. The required amount of processor cycles increases by increasing the CCR of the graph. This is due to the fact that more remote accesses are needed in a graph with larger CCR. Comparing the outcome of HybILP (HybPrefILP) with the outcome of HEFT (HEFT with prefetching) and G-CPU (G-CPU with prefetching) in Figure 3.9(b) confirms that our technique outperforms common heuristic techniques in different CCRs.

By increasing the number of the processing tiles in the MPSoC, the required amount of the processor cycles decreases (see Figure 3.9(c)). This shows the existence of parallel actors in the graph that enable actors to execute in a concurrent way on a multi-processor system. For the selected DAG, HybILP (HybPrefILP)



(a) configuration [2-1.0-X]

(b) configuration [2-X-1.0]



(c) configuration [X-1.0-1.0]

Figure 3.9: Execution time of a DAG in different configurations ([# of processors][CCR][memory scale factor]). Each filled bar shows the execution time of a technique when considering prefetching and the stacked bar represents the extra execution time of the same technique without prefetching.

gives better efficiency to HEFT (HEFT with prefetching) and G-CPU (G-CPU with prefetching) by using a platform with fewer processors. For example, the execution time of the HybILP schedule on a platform with two processors is smaller than the execution time of HEFT and G-CPU schedules on a platform with four processors. This means that HybILP requires less computation resources compared to HEFT and G-CPU. Increasing the number of processors up to three (or four) does not reduce the execution time of the graph significantly while using HybILP (HybPrefILP) for scheduling the selected DAG; this is due to the limited parallelism in the DAG and on-chip communication overhead. This effect is not visible in case of using the HEFT or G-CPU heuristics because these heuristics find longer schedules when compared to HybILP in case of two processors and increasing the number of processors gives a chance to these heuristics to find shorter schedules. However, the schedules found by HEFT or G-CPU with 4 processors are still longer than the schedule found by HybILP with 2 processors.

The outcome of the proposed technique for the selected DAG is a hybrid schedule (i.e., it uses the combination of code- and data-driven scheduling for actors in the graph) which reduces the amount of remote accesses for large code elements and large intermediate data elements between the actors. When comparing the outcome of HybILP with HEFT and G-CPU, our technique achieves a 29% and 30% respectively shorter execution time for the selected DAG in a nominal experimental configuration (with two processors, a CCR equal to 1.0, a memory scale factor equal to 1.0). By extending HybILP to HybPrefILP, the execution time of the DAG in the nominal experimental configuration reduces by 32% and 39% compared to HEFT with prefetching and G-CPU with prefetching respectively.

3.10 Summary

The performance of applications when running on an MPSoC is affected by the time spent on fetching code and data from remote memories. We present a scheduling technique to improve the memory behavior of an MPSoC with limited on-chip memories. We formulate our approaches using an ILP framework.

The proposed technique, HybILP, makes a trade-off between loading code or data to reduce the run-time of the application or enhance its throughput. In essence, it chooses the most suitable scheduling strategy for a series of actors in a graph. Actors with dominant code size are scheduled with a code-driven scheduling strategy and actors that exchange large amounts of data are scheduled with a data-driven scheduling strategy. Our technique uses a compact ILP formulation which requires limited time for an ILP solver. To further refine the result of HybILP, we extend it to HybPrefILP. The HybPrefILP technique takes the overhead of prefetching into account when scheduling an application onto an MPSoC.

We evaluated our scheduling technique with a synthetic graph, taken from recent related work [27], and a common multimedia application (an MP3 decoder). We achieve a reduction in run-time of 8% compared to a common heuristic solution for the MP3 decoder application and 32% for the graph from [27]. These results demonstrate the advantage of our hybrid prefetch-aware scheduling technique.

We also applied our scheduling technique to an SADF model of an MP3 decoder application to show the applicability and scalability of our approach when the model is an SADF. As expected, the analysis time scales with the number of scenarios in an SADF model. The obtained schedules can be modeled directly in the S(A)DF model using our schedule modeling technique introduced in next chapter.

Chapter 4

Modeling Dataflow Schedules

Dataflow graphs can be extended with scheduling decisions, allowing analysis to obtain properties like throughput or buffer sizes for the scheduled graphs. Analysis times depend strongly on the size of the graph. SADF graphs can be statically scheduled using static-order schedules. The only generally applicable technique to model a static-order schedule in an SADF graph is to convert its scenario graphs, represented by SDF graphs, to the equivalent HSDF graphs. This may lead to an exponential increase in the size of the graph and/or to sub-optimal analysis results (e.g., for buffer sizes in multi-processors). We present techniques to model two types of static-order schedules, i.e., periodic schedules and periodic single appearance schedules, directly in an S(A)DF graph. Experiments show that both techniques produce more compact graphs compared to the technique that relies on a conversion to HSDF. This results in reduced analysis times for performance properties and tighter resource requirements. This work was published in [19, 21].

4.1 Overview

Model-based design-flows (e.g., [9, 10, 54, 69, 86]) model binding and scheduling decisions into the model. This enables the analysis of performance properties (e.g, throughput [34]) or resource requirements (e.g., buffer sizes [72]) under resource constraints. Many dataflow analysis algorithms, e.g., throughput calculation or buffer sizing, are straightforward when a single processor platform is used. For instance, the buffer sizes can be determined by executing the model according to a given schedule. However, in a multi-processor environment, analysis algorithms are not trivial because of the inter-processor communication, amongst other reasons. An S(A)DF can be bound to a multi-processor platform. Each processor in the platform executes a set of actors from the S(A)DF; the firings of actors bound to a processor are required to be sequentialized. For this purpose, a finite periodic schedule (i.e., PSOS) can be constructed. PSOSs only specify the firing order of actors. This separates them from fully static schedules, which determine absolute start times of actors (e.g., schedules generated using the technique of

[7]). Traditionally, for DSP software synthesis, a sub-set of all periodic static-order schedules is considered. This sub-set contains so-called *single appearance schedules* (SAS) [9]. In a SAS, the functional code of the actors is included in a nested loop structure such that each piece of code occurs only once. This minimizes the code size potentially at the cost of additional buffer memory needed to implement the channels. A model-based design-flow usually uses PSOSs (or a sub-set of PSOSs such as SASs) for an application modeled with an S(A)DF. In this way timing (throughput) and memory usage (buffers) can be analyzed.

There is only one technique [3] known to model PSOSs in an S(A)DF. This technique requires a conversion of the model to an HSDF graph. Consider the SDF graph depicted in Figure 4.1. Figure 4.2 (without the colored edges) shows the equivalent HSDF graph of the SDF graph in Figure 4.1. The technique of [3] sequentializes the actor firings by inserting a channel between each pair of consecutive actors in a schedule. At the end of a schedule, it adds a channel with one initial token from the last to the first actor in the schedule. This ensures an indefinite execution of the graph according to the schedule. To model PSOSs $s_0 = \langle a_0(a_2)^2\rangle^*$ and $s_1 = \langle(a_1)^5(a_3)^3a_1(a_3)^3\rangle^*$, the technique of [3] adds in total 15 channels to the HSDF graph of the example graph (the green edges for s_0 and the blue edges for s_1 in Figure 4.2). For example, s_0 indicates an indefinite sequence of one firing of a_0 followed by two firings of a_2. This order is enforced in the HSDF graph of Figure 4.2 by the green edges between the actors a_{0_1}, a_{2_1}, and a_{2_2}.

The SDF to HSDF conversion can lead to an exponential increase in the size of the graph. For example, such a conversion for an H.263 decoder [72] (with QCIF resolution) increases the graph size from 4 actors to 1190 actors. Note that the number of actors in the resulting HSDF graph highly depends on how the application is modeled in the original SDF. The run-time of analysis algorithms depends amongst others on the size of the graph. As a result, the run-time of many S(A)DF analysis algorithms may increase drastically when modeling PSOSs in the graph using the technique from [3]. For example, the buffer sizing algorithm from [72] takes less than 1 ms on the SDF of an H.263 decoder. Modeling a schedule into this SDF using the technique from [3], the same analysis lasts 1330 ms. Analysis algorithms are usually repeated more than once in an iterative design-flow. As an example, for the SDF graph of an H.263 decoder, the design-flow from [69] performs 8 throughput calculations on the SDF graph to obtain the desired binding. Hence, it is vital to maintain a compact *schedule-extended graph*, i.e., a graph in which schedules are modeled explicitly, to provide a fast and practical design flow. There is a second drawback to the technique from [3]. The original graph structure is lost due to the conversion to an HSDF graph. A single channel in an S(A)DF corresponds to a set of channels in the HSDF

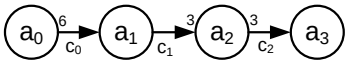


Figure 4.1: An example SDF.

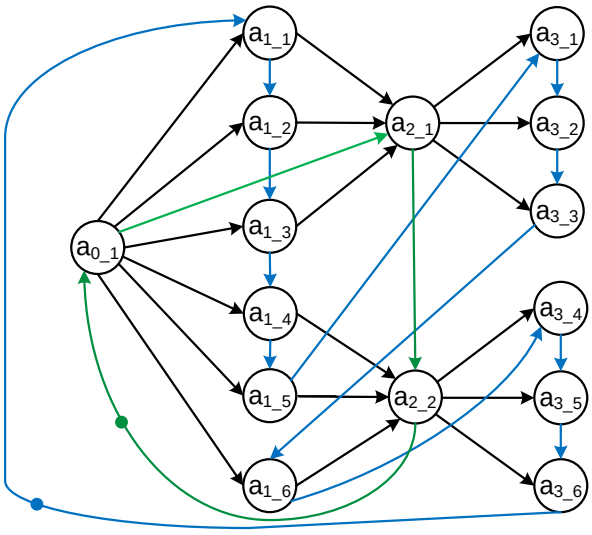


Figure 4.2: PSOSs $s_0 = \langle a_0(a_2)^2\rangle^*$ and $s_1 = \langle (a_1)^5(a_3)^3a_1(a_3)^3\rangle^*$ modeled in the SDF of Figure 4.1 using the technique from [3]; each a_{i_j} actor in the HSDF is an instance of SDF actor a_i.

graphs. In Figure 4.2, for example, the six edges between actor a_{0_1} and the a_{1_j} actors correspond to the single edge between a_0 and a_1 in the SDF graph of Figure 4.1. As a result, common buffer sizing techniques cannot find the minimal buffer size for the original S(A)DF. The H.263 decoder buffer sizes are for example overestimated by 49% when applying the technique of [72] to the HSDF graph.

A novel technique is needed to model PSOSs in an S(A)DF graph. This technique should limit the increase in the number of actors such that analysis times do not increase too much when analyzing the S(A)DF graph with its schedules. The technique should also preserve the original graph structure as this enables accurate analysis of graph properties such as buffer sizes. In this chapter, we presented a schedule modeling technique, called DSM, to model any PSOS directly in an S(A)DF graph. In addition, a second schedule modeling technique, called SASM, is introduced that is limited to SASs, but that results in more compact models compared to the first technique when modeling SASs.

DSM and SASM can be used in any model-based design-flow that models PSOSs into the S(A)DF graph (e.g., [9, 10, 54, 69, 86]). Conversion to an HSDF graph may be inevitable at some steps of a design trajectory. For example, multi-processor scheduling may require such conversions, although some techniques exist that can find schedules for SDFs without any conversion to HSDFs. For example, the technique presented in [80] solves the buffer sizing and scheduling problems simultaneously at the SDF level. It is not the conversion from SDF to HSDF itself that is problematic though. The problem is that analyses or optimizations on large HSDFs may be time consuming (e.g., throughput analysis) or inaccurate (e.g., buffer sizing). With our techniques, obtained schedules can be annotated back

to the original SDF; hence, the later analysis and optimization can be performed on the schedule-extended SDF. Besides the already mentioned analyses, also for example dynamic voltage scaling can be directly applied to a schedule-extended SADF model of an application mapped to a multi-processor platform (see Chapter 6). Note that code generation is another step which requires an S(A)DF to HSDF conversion; this conversion can be delayed until all (or most of the) prior analyses are carried out on the S(A)DF. Ultimately, the proposed techniques may save significant amounts of analysis time in a multi-processor design flow and they may lead to more accurate results.

The remainder of this chapter is structured as follows. The next section discusses related work. Section 4.3 explains our approach in modeling PSOSs in SADF graphs. Sections 4.4 and 4.5 present our techniques to model PSOSs and SASs in an SDF graph. Sections 4.6 and 4.7 contain the formal proofs of the correctness of DSM and SASM respectively. We evaluate our technique by applying it to several realistic applications in Section 4.8. Section 4.9 concludes the chapter.

4.2 Related Work

The technique from [3] is the only available technique to model PSOSs in an SDF, through a conversion to an HSDF. As already explained, this technique may result in a long run-time for analysis algorithms and/or inaccurate results from these algorithms. Our techniques alleviate both shortcomings of the technique from [3].

The work in [82] models the effect of a budget scheduler or preemptive TDMA scheduler on the temporal behavior of the SDF, either by computing an accurate worst-case response time or, more precisely, by introducing additional actors to model the timing impact as a latency-rate model. In contrast, for non-preemptive schedules, such as PSOSs, we focus on the ordering of actor firings; their execution time remains the same. We force an SDF to follow the PSOSs selected for each processor. This allows SDF analysis to obtain properties like throughput or buffer sizes for the scheduled SDF. This is also true for the models of [82]. However, for non-preemptive schedules, our results are tighter and our techniques require less analysis time. The authors of [76] have shown that an SDF can be used to consider an application with resource sharing possibilities; they perform buffer sizing under a throughput constraint considering a given schedule for the actors using a shared resource. For shared resource analysis, they use event-models [40] which is based on realtime-calculus [77]. Our approach differs from [76], since modeling schedules directly into SDFs enables us to use dedicated analysis for dataflow graphs. Moreover, the technique of [76] can only handle an SDF with limited types of cycles, such as cycles formed by self-edges or the back edges modeling the buffer capacity between two actors. However, staying in dataflow domain, as is done in our technique, does not impose such a limitation on the graph structure.

Ref [83] uses some new (custom) components, e.g., $if-then-else$, to model schedules in an SDF. These components are not supported by the common model-

based design-flows using SDFs (e.g., [9, 10, 54, 69, 86]) and cannot be modeled in an SDF using the basic elements of an SDF (i.e., actors and channels). Our techniques eliminate the need for any new (custom) component. As a result, any analysis technique for SDFs is directly applicable to the schedule-extended SDF.

4.3 Modeling Schedules in SADF Graphs

As discussed before in Chapter 2, an SADF graph uses an SDF graph to describe the behavior of a scenario in the model. A set of PSOSs can be devised for each scenario in the SADF graph by using a static scheduling technique (e.g., the scheduling techniques presented in Chapter 3). In the rest of this chapter, we present two schedule modeling techniques, i.e., DSM and SASM, to model PSOSs and SASs directly in an SDF graph. These techniques are also used to model schedules generated for a scenario of an SADF directly into a scenario graph of the SADF, which is an SDF graph.

An actor a_i in a PSOS should appear $r \cdot \gamma(a_i)$ times in the PSOS (with $r = \frac{u}{v}$ where $u, v \in \mathbb{N}$) and the value r is identical for all actors in the PSOS [34]. This is imposed by the SDF property that firing each actor as often as indicated in the repetition vector results in a token distribution that is equal to the initial token distribution. As stated before, the term *normalized PSOS* refers to a PSOS with r equal to 1. We limit ourselves in the remainder to PSOSs in which r is a *unit fraction* (i.e., $r = \frac{u}{v}$ with $u = 1$ and $v \in \mathbb{N}$), although our schedule modeling techniques can also be directly applied to model other PSOSs (i.e., in which $u \in \mathbb{N}$) in case of SDF graphs. PSOSs generated for a scenario graph of the SADF should cover one iteration of the scenario graph (i.e., r equal to 1); this is the only limitation of our schedule modeling techniques for SADF graphs when compared to the schedule modeling techniques for SDF graphs.

4.4 Modeling Periodic Static-Order Schedules

In this section, we introduce a technique to model PSOSs in an SDF. We first illustrate all ingredients of our approach through a running example, and then discuss the algorithm and the main steps in the algorithm in more detail. Note that a schedule is correctly modeled if and only if any execution of the schedule-extended graph satisfies the schedule and if any execution of the original graph that satisfies the schedule is still feasible in the schedule-extended graph.

4.4.1 Running example

Here we briefly introduce our approach in modeling a PSOS in an SDF using a running example. For this purpose consider the example SDF shown in Figure 4.1 and the PSOS $s_1 = \langle (a_1)^5 (a_3)^3 a_1 (a_3)^3 \rangle^*$ which is a schedule for a_1 and a_3. Our approach captures this schedule in the SDF in three steps, that (i) remove auto-concurrency, (ii) avoid inter-iteration execution, and (iii) enforce the correct scheduling decisions.

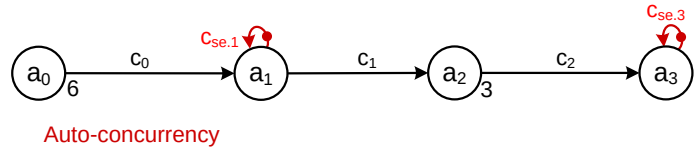
In the example SDF, a single firing of a_0 produces 6 tokens in channel c_0; then 6 firings of a_1 can be performed simultaneously. This simultaneous firing of an actor is called *auto-concurrency*; in practice, this corresponds to executing multiple instances of a function (task) in parallel. Auto-concurrency for an actor cannot be handled in a real hardware platform, unless more than one processor is allocated for that actor. Here, we focus on the case that an actor is mapped to one processor. Hence, auto-concurrency must be removed from the model. To sequentialize firings of an actor, a self-edge with one initial token must be added to that actor; this way auto-concurrency can be removed for that actor. Figure 4.3(a) shows the example SDF of Figure 4.1 in which any auto-concurrency related to a_1 and a_3 is removed by adding two self-edges (shown in red).

In one PSOS iteration, each actor must fire a specific number of times. Actors must not be able to get fired more than the number of times indicated by the PSOS. Consider the following situation in the example SDF of Figure 4.1. Two firings of a_0 provide 12 tokens in channel c_0; this number of tokens is enough for 12 firings of the actor a_1. The first 6 firings of a_1 belong to the first iteration and the second 6 firings belong to the second iteration. The second 6 firings of a_1 can occur before the completion of the first PSOS iteration of s_1; in this case, the resulting execution does not satisfy the given PSOS s_1. This situation is called *inter-iteration execution*. To prevent inter-iteration execution related to s_1, we create a dependency from the last actor appearing in s_1 to the first actor appearing in s_1; see the blue elements in Figure 4.3(b). This dependency limits the firing of the first actor to a number of firings required in one PSOS iteration.

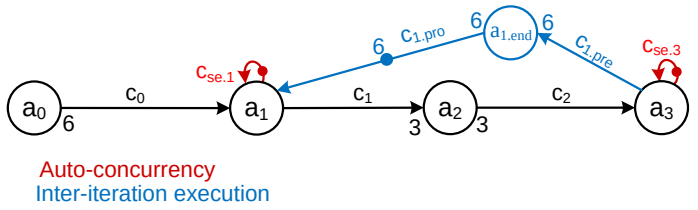
In the SDF of Figure 4.3(b), consider the case that actors a_0, a_1 and a_2 have fired 1, 3 and 1 times, respectively. The initial token distribution is changed to the distribution shown in Figure 4.3(c). In this graph, both a_1 and a_3 from PSOS s_1 are enabled; but, only the firing of a_1 must be granted at this point to form an execution that satisfies the given PSOS s_1. We call such a state in which several actors are enabled a *decision state*. Later on, a precise definition is given for this concept. According to schedule s_1, actor a_3 must get enabled after 5 firings of a_1. For this purpose, a dependency is created from a_1 to a_3 (shown with green actor and channels in Figure 4.3(d)); this new dependency prevents a_3 from getting enabled unless a_1 has completed 5 firings. Another decision state can be found after a_1 has completed 5 firings; once again, both a_1 and a_3 from PSOS s_1 are enabled at this point; but, the firing of a_3 must take place. For this purpose, a dependency is created from a_3 to a_1 (see Figure 4.3(e)). This new dependency prevents a_1 from getting enabled from the current state onwards unless a_3 has completed 3 firings. The 5 initial tokens on the added input channel to a_1 ensure that the first 5 firings of a_1 can take place as planned. The SDF of Figure 4.3(e) shows the final solution that models the PSOS s_1 in the SDF of Figure 4.1.

4.4.2 The algorithm

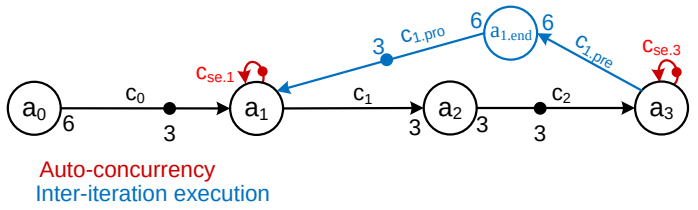
Algorithm 2 captures our technique, called decision state modeling (DSM). DSM accepts an SDF and one or several PSOSs as its input. In the algorithm $n + 1$ ($n \in \mathbb{N}_0$) is the number of processors (or input PSOSs). DSM ensures that actor



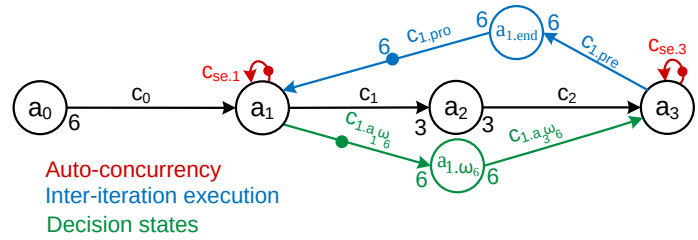
(a) Auto-concurrency has been removed.



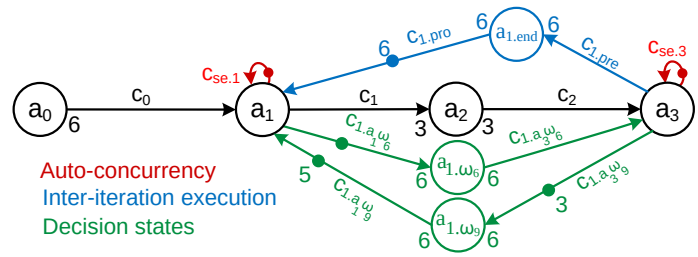
(b) Auto-concurrency and inter-iteration execution have been removed.



(c) SDF of Figure 4.3(b) after 1, 3 and 1 firings of a_0, a_1 and a_2 resp., leading to a decision state in which both actors a_1 and a_3 are enabled.



(d) Creating a dependency for the first decision state.



(e) Creating a dependency for the second decision state.

Figure 4.3: Step-by-step modeling of the PSOS s_1 in the SDF of Figure 4.1.

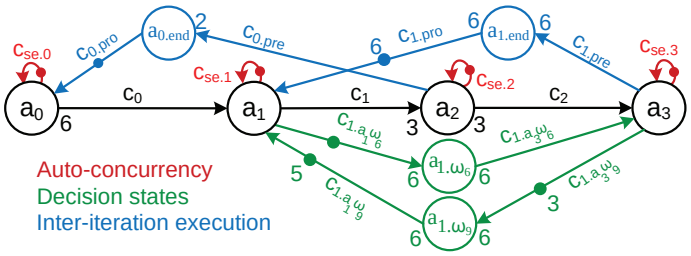


Figure 4.4: PSOSs s_0 and s_1 modeled in the SDF of Figure 4.1 using DSM.

firings of each PSOS follow the specified order in that PSOS; the output of DSM is a new SDF that models the provided PSOSs in the input SDF. Figure 4.4 depicts the corresponding SDF of Figure 4.1 which models the PSOSs $s_0 = \langle a_0(a_2)^2\rangle^*$ and $s_1 = \langle(a_1)^5(a_3)^3a_1(a_3)^3\rangle^*$ using DSM. The remainder of this section discusses the three main steps of the algorithm - removing auto-concurrency, avoiding inter-iteration execution, enforcing correct decisions in decision states - in detail.

The description of some basic functions used in Algorithm 2 is as follows. The function *AA(G, a_{new})* is responsible to include the actor a_{new} in the SDF G. The function *AC(G, c_{new}, a_{src}, a_{dst}, srcRate, dstRate, initTok)* adds the channel c_{new} from source actor a_{src} to destination actor a_{dst}; the production rate of a_{src} on this channel is equal to *srcRate* and the consumption rate of a_{dst} on this channel is equal to *dstRate*; this channel is initialized with *initTok* initial tokens. The function $CNT(a_j, s_i)$ returns the number of times that the actor a_j is fired in one iteration of PSOS s_i. The function *BEF(a_k,j,s_i)* returns the number of times that a_k appears from the first position in the PSOS s_i to and including the j^{th} position in the PSOS s_i; the function *AFT(a_k,j,s_i)* returns the number of times that a_k appears from the $(j+1)^{th}$ position in the PSOS s_i to the last position in the PSOS s_i.

4.4.3 Auto-concurrency

An actor $a_i \in A$ can be enabled multiple times simultaneously in an SDF state ω_j if $\omega_j(c) \geq k \cdot Rate(q)$ for each channel $c = (p, q) \in InC(a_i)$ where $k \in \mathbb{N}, k \geq 2$. This is called auto-concurrency. The firings of actor a_i should occur sequentially according to the PSOS to which a_i belongs. This sequential execution can be enforced by adding a self-edge with one initial token to actor a_i (Line 1 in Algorithm 2). In Figure 4.4, channels $c_{se.0} - c_{se.3}$ (shown in red) are used to prevent any auto-concurrency in the SDF of Figure 4.1.

4.4.4 Inter-iteration execution

Consider actor a_0 in the SDF of Figure 4.1; the 1^{st} firing of a_0 belongs to the 1^{st} PSOS iteration of s_0 and the 2^{nd} firing of a_0 belongs to the 2^{nd} PSOS iteration of s_0 and so on. Since a_0 has no input channel, it can always be fired regardless of other actors in the graph. This behavior, which is called inter-iteration execution,

can prevent an SDF execution from satisfying the given PSOSs. Inter-iteration execution happens when one PSOS iteration has not been completed and an actor from that PSOS can proceed its firings beyond the current PSOS iteration. Lines 4-8 in Algorithm 2 are used to control this undesirable actor enabling. This part of the algorithm adds (per PSOS) one actor and two channels to create a dependency between the last and first actor appearing in the PSOS. The added components limit, within one PSOS iteration, the firing of the first actor in the PSOS (i.e., a_F) to the count of actor a_F (i.e., $CNT(a_F, s_i)$) in one iteration of the PSOS s_i. The next iteration of the PSOS s_i can only commence if the last actor in PSOS s_i (i.e., a_L) fires $CNT(a_L, s_i)$ times in one iteration of the PSOS s_i. In other words, the next iteration of a PSOS can only commence after the completion of the current iteration of this PSOS. In Figure 4.4, actor $a_{0.end}$ and channels $c_{0.pre}$ and $c_{0.pro}$ are added to prevent any inter-iteration execution in PSOS s_0. Actor $a_{1.end}$ and channels $c_{1.pre}$ and $c_{1.pro}$ are added to prevent any inter-iteration execution in schedule s_1. These elements are shown in our example in blue in Figure 4.4.

Algorithm 2: Decision State Modeling (DSM)

input : SDF $G(A, C)$, PSOSs $\{s_0, \cdots, s_n\}$
output: G extended with schedules $\{s_0, \cdots, s_n\}$

1 add a self edge with 1 initial token for each $a \in A$
2 $\{s'_0, \mu_0, \cdots s'_n, \mu_n\} \leftarrow$ `normalize`$(G, \{s_0, \cdots, s_n\})$
3 **for** $i \leftarrow 0$ **to** n **do**
`/* To control inter-iteration execution */`
4 $a_L :=$ *last actor in* s_i
5 $a_F :=$ *first actor in* s_i
6 `AA`$(G, a_{i.end})$
7 `AC`$(G, c_{i.pre}, a_L, a_{i.end}, 1, \texttt{CNT}(a_L, s_i), 0)$
8 `AC`$(G, c_{i.pro}, a_{i.end}, a_F, \texttt{CNT}(a_F, s_i), 1, \texttt{CNT}(a_F, s_i))$
`/* To control decision states */`
9 Ω, $pos \leftarrow$ `getDecisionStates`$(G, s'_i, \{s'_0, \cdots, s'_n\} \setminus s'_i)$
10 $\Omega \leftarrow$ `reduceDecisionStates`(Ω)
11 $\Omega \leftarrow$ `foldDecisionStates`(Ω, μ_i)
12 **foreach** $\omega_j \in \Omega$ **do**
13 `AA`(G, $a_{i.\omega_j}$)
14 **foreach** $a_k \in \Delta_j$ **do**
15 **if** a_k *is the actor of choice* **then**
16 `AC`(G, $c_{i.a_k\omega_j}$, a_k, $a_{i.\omega_j}$, 1, `CNT`(a_k, s_i), `AFT`$(a_k, pos[\omega_j], s_i)$)
17 **else**
18 `AC`(G, $c_{i.a_k\omega_j}$, $a_{i.\omega_j}$, a_k, `CNT`(a_k, s_i), 1, `BEF`$(a_k, pos[\omega_j], s_i)$)

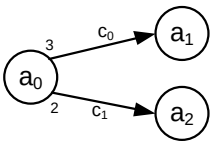


Figure 4.5: An example SDF.

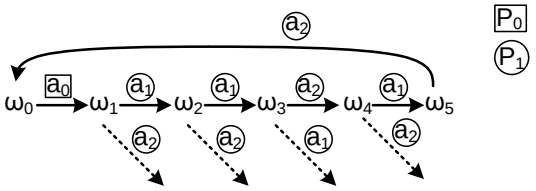


Figure 4.6: The state space of the SDF of Figure 4.5 when PSOSs $s'_0 = \langle a_0 \rangle^*$ and $s'_1 = \langle (a_1)^2 a_2 a_1 a_2 \rangle^*$ are used.

4.4.5 Decision states

This sub-section presents the third step of DSM. It first defines the concept of a decision state and then proceeds with the algorithm for identifying decision states; after explaining two optimization steps, it ends with the technique to enforce the appropriate schedule decisions.

4.4.5.1 Concept

Multiple different actors mapped to a single processor may be enabled in a specific state. Here, we describe such situations in an SDF execution. Consider the SDF in Figure 4.5. Assume that a_0 is mapped to processor P_0 with PSOS $s'_0 = \langle a_0 \rangle^*$ and a_1 and a_2 are mapped to processor P_1 with PSOS $s'_1 = \langle (a_1)^2 a_2 a_1 a_2 \rangle^*$. For brevity, we only discuss the actors mapped to processor P_1. The corresponding state space - for one SDF iteration - when executing our example SDF using the PSOSs s'_0 and s'_1 is visualized in Figure 4.6. In this figure, the actors mapped to processor P_0 (P_1) are surrounded by a square (circle). Auto-concurrency and inter-iteration execution are excluded using the constructs introduced in Section 4.4.3 and Section 4.4.4 resp. The periodic behavior of the PSOSs is obvious from the state space where one SDF iteration moves the graph to its initial state, i.e., ω_0 (see Figure 4.6). There are some states in Figure 4.6 in which more than one actor is enabled (e.g., $\omega_1 - \omega_5$) on processor P_1. In such a situation, the execution related to those actors can deviate from the specified PSOS. We use the following definition to formalize such a situation.

Definition 16. (Decision State) *Consider the PSOS s_i as a schedule for actors $A_i \subseteq A$ and an execution σ of an SDF (A, C) which satisfies PSOS s_i. A state $\omega_j \in \sigma$ is a decision state iff multiple different actors from A_i are enabled in ω_j.*

We use Ω to refer to the finite set containing all decision states occurring in

the execution of an SDF up-to one iteration of a PSOS. The following terminology describes the enabled actors in a decision state.

Definition 17. (Opponent Actor Set) *Let $\omega_j \in \Omega$ be a decision state for PSOS s_i. The opponent actor set Δ_j of the decision state ω_j is a finite set which contains all actors that are enabled in decision state ω_j and that belong to A_i.*

The finite set Δ_j represents the opponent actors in the decision state $\omega_j \in \Omega$. One of the enabled actors in a decision state ω_j, in line with the given PSOS s_i, should be selected to fire. The following is used to describe such an actor.

Definition 18. (Actor of Choice) *Consider the PSOS s_i which schedules actors $A_i \subseteq A$ and the opponent actor set Δ_j of the decision state ω_j in an execution σ of the SDF $G(A, C)$ which satisfies s_i. An actor $a_c \in \Delta_j$ is called the actor of choice of the decision state ω_j iff the firing of actor a_c in state ω_j is a necessity for the execution σ in order to satisfy the PSOS s_i. Since a PSOS specifies a fixed firing order, there can only be a single actor of choice in any decision state.*

Lines 9-18 in DSM show how we deal with non-deterministic execution due to decision states. DSM models the given PSOSs one-by-one iteratively. The ordering of PSOSs in DSM does not have any impact on the final behavior. In each iteration of the for-loop in line 3, we enforce the execution of the actors in the current schedule of interest (i.e., schedule s_i) to follow schedule s_i. The next sub-section explains how decision states of the schedule of interest are extracted. At the same time, a value is preserved for each decision state that captures the relative position of that decision state with respect to the beginning of the schedule of interest; the notation $pos[\omega_j]$ refers to this position for ω_j. For example, in the SDF of Figure 4.1, $pos[\omega_6] = 5$ since the relative position of ω_6 with respect to the beginning of the schedule of interest (i.e., s_1) is 5 (in Figure 4.7 consider 4 firings related to s_1 have been occurred before ω_6 and the 5^{th} actor firing related to s_1 is going to happen in ω_6). For each $\omega_j \in \Omega$ extracted for s_i, DSM adds an actor ($a_{i.\omega_j}$ in line 13) and one channel between the new actor $a_{i.\omega_j}$ and each opponent actor in the set Δ_j (lines 14-18 in Algorithm 2). The elements added in each decision state (e.g., ω_j) postpone the execution of the actors in $\Delta_j \setminus \{a_c\}$ to the state after decision state ω_j. Hence, a_c (i.e., the actor of choice) is the only actor which can be fired in the state ω_j.

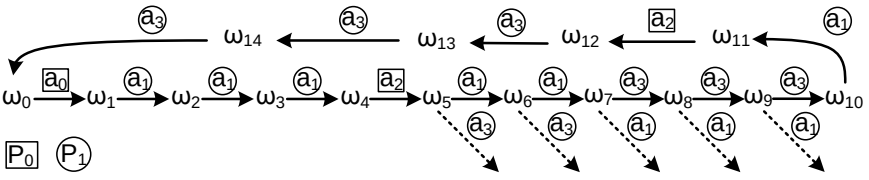


Figure 4.7: The state space of the SDF of Figure 4.1 when the PSOS $s_1 = \langle (a_1)^5 (a_3)^3 a_1 (a_3)^3 \rangle^*$ is the schedule of interest (i.e., s_c) in Algorithm 3.

Algorithm 3: Get Decision States

input : SDF G, PSOS s_c, PSOSs $\{s_{o1}, \cdots, s_{on}\}$
output: Decision state set Ω
output: relative positions *pos*

```
1 ω_j ← the initial state of G
2 for i ← 1 to |s_c| do
3     ω_j ← maxExec(G, ω_j, {s_o1, ··· , s_on})
4     if sizeof(enabledActors(G, ω_j, s_c)) >1 then
5         pos[ω_j] ← i
6         Ω ← Ω ∪ {ω_j}
7         Δ_j ← enabledActors(G, ω_j, s_c)
8     ω_j ← fire(G, ω_j, s_c[i])
```

4.4.5.2 Decision state identification

Algorithm 3 shows our technique to detect all decision states within PSOS s_c. Assume s_c is a PSOS for the actors mapped to processor P_c. Schedules $s_{o1} \cdots s_{on}$ are PSOSs for the other actors of the SDF mapped to the other processors (denoted by $P_{o1} \cdots P_{on}$). The output of Algorithm 3 is a set that contains all decision states for PSOS s_c. This algorithm also returns the relative positions of decision states with respect to the beginning of s_c. In Algorithm 3, the input schedules are normalized PSOSs. The function *normalize* (in line 2 of Algorithm 2) normalizes the input PSOSs. The function returns the normalized PSOSs along with their normalization factors. The normalized PSOS s'_x can be achieved by repeating μ_x times the input PSOS s_x (i.e., $s'_x = (s_x)^{\mu_x}$). μ_x is the normalization factor of s_x and can be calculated by dividing the repetition vector entry of an arbitrary actor in s_x by the count of that actor in the PSOS s_x (in our running example, μ_0 and μ_1 are 1).

After normalizing the input schedules, all situations that can lead to multiple actors (mapped to the same processor) being ready to fire must be discovered. An actor in the schedule of interest s_c could be affected by the execution of an actor in the other schedules as well as by another actor in s_c. Processors can run at different clock rates; these differences and inter-processor dependencies cause variation in the number of tokens on the *inter-processor channels* originating from the actors mapped to the other processors to the actors mapped to the processor of interest (i.e., P_c). The number of tokens on the input channels of an actor determines whether an actor is enabled or not. To determine any possible actor enabling within s_c, a *necessary and sufficient number of tokens* on all inter-processor channels entering to the actors mapped to processor P_c must be considered. We will now explain what necessary and sufficient number of tokens means in our algorithm. Each iteration of the schedule of interest s_c requires that the actors mapped to the other processors are fired up-to at most their repetition vector entry values. Hence, only executing one iteration of the other schedules $s_{o1} \cdots s_{on}$ is enough to provide a *necessary number of tokens* on

inter-processor channels entering to the actors mapped to processor P_c. More than one iteration for the other schedules $s_{o1} \cdots s_{on}$ may be feasible; this may cause an actor in s_c to be enabled more than its count in one iteration of s_c. The inter-iteration prevention constructs introduced in Section 4.4.4 control this undesired actor enabling. So, we only extract decision states within one iteration of the normalized schedule to provide a *sufficient number of tokens*.

Also, DSM does not impose any limitation between PSOSs since no dependency is created between actors mapped to different processors. PSOSs can independently be iterated if the dependencies in the SDF allow that. We allow the actors on the other processors to be executed (according to their schedules) as much as they can; the corresponding execution is named *maximal execution*. The maximal execution will stop at one point either due to a dependency on the actors on the processor P_c or because one PSOS iteration is completed. The SDF state (denoted by ω_j in Algorithm 3) should be kept during the operation of the algorithm. The maximal execution is represented by the function *maxExec* in Algorithm 3. After one maximal execution, the number of tokens on the inter-processor channels entering into the actors on the processor P_c determines any possible enabled actor. The preserved state (i.e., ω_j) will be added to the decision state set (Ω) if more than one actor on the processor P_c is enabled at this state (line 6 in Algorithm 3). The current position (i.e., i) in the schedule of interest s_c is also preserved for the discovered decision state (see line 5 in Algorithm 3). All enabled actors will be recorded as opponent actors of the state ω_j (line 7 in Algorithm 3). The execution of the actors on the processor P_c is continued by executing the enabled actor in line with s_c to determine all possible decision states (line 8 in Algorithm 3). The function $fire(G, \omega_j, s_c[i])$ fires the actor at the i^{th} position in the PSOS s_c. The process is repeated until a full iteration of the PSOS has been examined. In the end, the set Ω contains all possible decision states when executing s_c.

Figure 4.7 depicts the resulting state space from applying Algorithm 3 on the SDF of Figure 4.1 when s_1 is the schedule of interest (i.e., s_c). In the SDF of Figure 4.1, five consecutive decision states ($\Omega = \{\omega_5 \cdots \omega_9\}$) have been found for s_1 and no decision state has been found for s_0 (see Figure 4.7).

4.4.5.3 Redundant decision states

Here, we explain an optimization proposed in DSM to remove unnecessary decision states. DSM adds some components (per decision state) to create a dependency from the actor of choice of a decision state to the other opponent actors of that decision state. Such a dependency delays the firing of those opponent actors to the state after the decision state. The added components are explained in detail in Section 4.4.5.5.

It is possible to have several consecutive decision states which are delaying the firing of an actor to several states later. For example, three consecutive decision states ($\omega_7 - \omega_9$) exist in Figure 4.7 that all delay the firing of a_1; the added components in ω_7 delay the sixth firing of a_1 to ω_8; the added components in ω_8 delay the sixth firing of a_1 to ω_9; and so on. The latest decision state in the sequence of decision states $\omega_7 - \omega_9$ is enough to delay the firing of a_1 to

ω_{10}. Hence, the decision states $\omega_7 - \omega_8$ are redundant and can be removed from the decision state set Ω. The function *reduceDecisionStates* is responsible for removing redundant decision states. Note that it would be possible to perform this reduction during the decision state identification step. This optimization can significantly reduce the number of extra components in the final SDF. Decision state ω_5 is also redundant according to our optimization. So, only two decision states ω_6 and ω_9 are necessary to model s_1 in the SDF of Figure 4.1.

4.4.5.4 Decision state folding

In Algorithm 2, the input PSOSs are normalized to find all decision states. The normalization of PSOSs is required to explore sufficient states of an SDF. Consider PSOSs $s_2 = \langle a_0 \rangle^*$ and $s_3 = \langle a_2\ a_1 \rangle^*$ for our second example SDF in Figure 4.8. To obtain normalized PSOSs, μ_2 and μ_3 must be equal to 3 and 4 respectively. This leads to the following normalized PSOSs: $s'_2 = \langle (a_0)^3 \rangle^*$ and $s'_3 = \langle (a_2\ a_1)^4 \rangle^*$. Decision state identification for s'_3 results in 5 decision states. $\underbrace{\binom{a_2}{-}\binom{a_1}{a_2}}_{1^{st}\ 2^{nd}}\underbrace{\binom{a_2}{a_1}\binom{a_1}{a_2}}_{3^{rd}\ 4^{th}}\underbrace{\binom{a_2}{a_1}\binom{a_1}{a_2}}_{5^{th}\ 6^{th}}\underbrace{\binom{a_2}{-}\binom{a_1}{-}}_{7^{th}\ 8^{th}}$ shows the corresponding execution of s'_3. In construct $\binom{a_x}{a_y}$, a_x is the enabled actor in line with the PSOS and a_y is the other enabled actor if any at all. In this execution, the 1^{st}, 3^{rd}, 5^{th} and 7^{th} states are similar in behavior since the same actor, i.e., a_2, is expected to fire in all of those states.

Modeling a repetitive behavior for a PSOS s_i, also models this behavior for its normalized PSOS (i.e., $s'_i = (s_i)^{\mu_i}$). Using this property, we can merge decision states appearing in all μ_i repetitions of the PSOS s_i. We call this optimization *decision state folding* (line 11 in Algorithm 2). Folding groups the similar states. In our example, the 1^{st}, 3^{rd}, 5^{th} and 7^{th} states are grouped and represented with one state. Similarly, the 2^{nd}, 4^{th}, 6^{th} and 8^{th} states are grouped. So, the above execution shrinks to $\binom{a_2}{a_1}\binom{a_1}{a_2}$. After grouping all similar states in the original execution into a representative state, it is considered a decision state if any of the group members is a decision state. In practice, a decision state in a state of the new folded execution will be considered as a decision state for each of the equivalent states in the original execution. This cannot violate the execution according to the input PSOS because DSM ensures that only the actor of choice executes in a decision state. This optimization could reduce the number of decision states up to μ_i times in a normalized PSOS s'_i. The firings related to those actors enabled in the last state except the actor of choice of that state are supposed to happen in subsequent PSOS iterations; this is already ensured by preventing inter-iteration execution (see Section 4.4.4). Hence, after folding, the last state can be ignored as

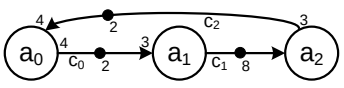


Figure 4.8: A third example SDF.

a decision state. In our third example, decision state folding reduces the number of decision states from 5 to 1 for the PSOS s_3.

4.4.5.5 Enforcing a schedule in decision states

In our first example SDF, only two actors are enabled in decision state ω_6 (i.e., $\Delta_6 = \{a_1, a_3\}$) (see Figure 4.7). Actor a_1 is the actor of choice in ω_6 and a_3 is the only opponent actor whose execution should be delayed to the state after ω_6. To enforce firing of a_1 and to prevent firing of a_3 in ω_6, DSM creates a dependency from a_1 to a_3 by adding actor $a_{1.\omega_6}$ and channels $c_{1.a_1\omega_6}$ and $c_{1.a_3\omega_6}$. The rates and initial tokens related to the new elements are set in such a way that the execution of the graph in other states are not affected. The actor $a_{1.\omega_6}$ is only responsible for decision state ω_6. So, $a_{1.\omega_6}$ needs to only fire once in an iteration. For this purpose, the port rates of $a_{1.\omega_6}$ on its channels (i.e., $c_{1.a_1\omega_6}$ and $c_{1.a_3\omega_6}$) are set to 6. The added dependency channels from the newly added actor in decision state ω_j (e.g., $a_{1.\omega_6}$ in ω_6) to the opponent actors which are not the actor of choice (e.g., a_3 in ω_6) only provide enough tokens for their execution in states $\omega_0 - \omega_{j-1}$ (e.g., 0 tokens for a_3 in $\omega_0 - \omega_5$); these actors cannot be enabled due to the lack of initial tokens in the newly added channels in the corresponding decision state (e.g., there will be no token in $c_{1.a_3\omega_6}$ in ω_6). Hence, only the actor of choice amongst the opponent actors of a decision state will be enabled in that state (e.g., only a_1 can fire in ω_6). The delayed actors in a decision state (e.g., ω_j) will have sufficient tokens on the incoming channel from the newly added actor for that decision state (i.e., $a_{i.\omega_j}$) after firing of the actor of choice in ω_j. For example, there will be 6 tokens in channel $c_{1.a_1\omega_6}$ after the firing of a_1 (i.e., the actor of choice) in decision state ω_6; hence, the actor $a_{1.\omega_6}$ immediately fires and then provides sufficient tokens for later firings of a_3. So, the delayed actor in decision state ω_6 (i.e., a_3) will no longer be blocked due to the absence of tokens in channel $c_{1.a_3\omega_6}$ after the decision state ω_6. The firing of actor a_1 after decision state ω_6 produces 1 token in channel $c_{1.a_1\omega_6}$ and the firings of actor a_3 after decision state ω_6 consumes 6 tokens from channel $c_{1.a_3\omega_6}$; as a result, the number of tokens in the new channels are reset to the initial values at the end of one iteration of the schedule s_1. Hence, the periodic behavior is also achievable for the added components.

DSM also adds actor $a_{1.\omega_9}$ and channels $c_{1.a_1\omega_9}$ and $c_{1.a_3\omega_9}$ to the graph for the other decision state ω_9. The components added in decision state ω_9 show similar behavior as the components added in ω_6.

4.5 Modeling Single Appearance Schedules

A well-known class of scheduling techniques are single appearance schedules (SASs) in which each actor appears exactly once in the LS form. This aspect makes SASs suitable to minimize code memory size. $s_2 = \langle (a_0 a_1)^5 a_2 \rangle^*$ is a PSOS and SAS for part of the SDF (i.e., actors a_0-a_2) in Figure 4.9.

DSM is able to model any PSOS. However, more compact graphs are possible for SASs. Algorithm 4 presents our single appearance schedule modeling (SASM)

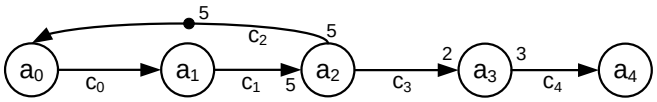


Figure 4.9: An example SDF.

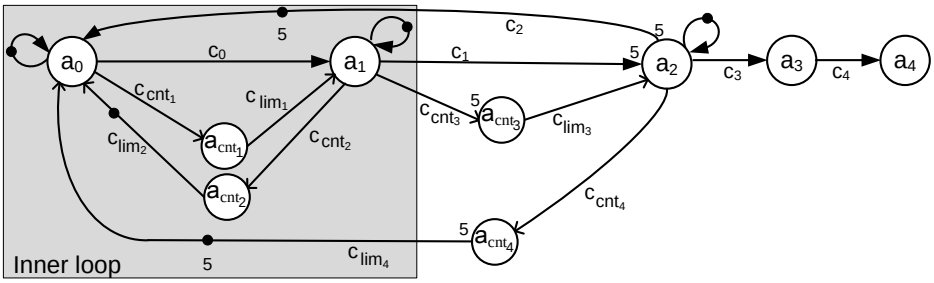


Figure 4.10: SDF of Figure 4.9 extended with $s_2 = \langle (a_0 a_1)^5 a_2 \rangle^*$ using SASM.

technique. Similar to DSM, SASM adds some extra actors and channels to the original SDF to model the given schedules. The original channels and actors in the schedule-extended SDF are preserved and distinguishable from the newly added elements by any of our techniques. Hence, both our techniques preserve the original structure of an SDF. This property can be beneficial when a resource allocation algorithm needs to be applied on the schedule-extended graph; a resource allocation algorithm can easily distinguish an original actor (or channel) from an actor (or channel) which is added to model the schedules.

We know that each actor appears only once in a SAS; this property can help us to model a SAS in an SDF in a smarter way than DSM does. An actor (or a nested inner LS) should be executed a specific number of times before another actor (or another nested inner LS) starts executing. In $s_2 = \langle (a_0 a_1)^5 a_2 \rangle^*$, the nested inner LS $(a_0 a_1)$ must be executed 5 times before a_2 starts firing. This type of execution control can be handled using a counter construct. The key idea of SASM is to implement a counter concept in the graph. Later, we explain how we implement such counters to model SASs in an SDF. Similar to DSM, auto-concurrency can be eliminated by adding a self-edge with 1 initial token to each actor in the SDF (see line 1 in Algorithm 4). The rest of Algorithm 4 deals with implementing the counter concept.

Figure 4.10 shows the graph of the SDF in Figure 4.9 which models the schedule s_2 using SASM. Schedule s_2 has a nested $a_0 a_1$; this can be replaced with α_{01} to form a looped schedule representation (i.e. $s_2 = \langle (\alpha_{01})^5 a_2 \rangle^*$ where $\alpha_{01} = a_0 a_1$). A counter in SASM is implemented by one actor a_{cnt_i} (e.g., a_{cnt_3} in Figure 4.10), a *counter channel* c_{cnt_i} (e.g., c_{cnt_3}) and a *limiter channel* c_{lim_i} (e.g., c_{lim_3}). A counter in SASM has two properties: first, it counts the number of times that element α_i (e.g., α_{01} above) is being executed; second, it limits the element α_{i+1} (e.g., a_2) to be executed to β_{i+1} times (e.g., 1 as the number of repetitions

Algorithm 4: SAS Modeling (SASM)

input : SDF $G(A, C)$, PSOS $s_i = \{(\alpha_1)^{\beta_1}(\alpha_2)^{\beta_2}...(\alpha_n)^{\beta_n}\}$
output: G extended with schedule s_i

```
1  add a self edge with 1 initial token for each a ∈ A_i
2  for i ← 1 to n do
3      if α_i is not an actor then
4          SASM (G, α_i)
5      AA(G, a_cnt_i)
                                        /* adding a counter channel */
6      AC(G, c_cnt_i, rightMost(α_i), a_cnt_i, 1, RN(rightMost(α_i), α_i^β_i), 0)
                                        /* adding a limiter channel */
7      if i = n then
8          AC(G, c_lim_i, a_cnt_i, leftMost(α_1), RN(leftMost(α_1), α_1^β_1), 1,
           RN(leftMost(α_1), α_1^β_1))
9      else
10         AC(G, c_lim_i, a_cnt_i, leftMost(α_n+1), RN(leftMost(α_n+1), α_n+1^β_n+1), 1,
           0)
```

of a_2 is one). The counter channel c_{cnt_i} (e.g. c_{cnt_3}) is placed from the rightmost actor in α_i (e.g., actor a_1 in α_{01}) to the actor a_{cnt_i} (e.g., a_{cnt_3}); the production rate of the rightmost actor in α_i on c_{cnt_i} is set to 1 and the consumption rate of a_{cnt_i} on c_{cnt_i} is set to the number of times that the rightmost actor in α_i can be executed in ${\alpha_i}^{\beta_i}$ (e.g., 5 as the number of times that a_1 can be executed in ${\alpha_{01}}^5$ is five). In Algorithm 4, the function *rightMost*(α_i) (*leftMost*(α_i)) returns the rightmost (leftmost) actor in element α_i (e.g. *rightMost*($(a_0a_1)^5$) returns actor a_1). The function *RN*(a,${\alpha_i}^{\beta_i}$) retrieves the count of a in element ${\alpha_i}^{\beta_i}$ (e.g. *RN*(a_0,$(a_0a_1)^5$) returns 5).

The limiter channel c_{lim_i} (e.g., c_{lim_3}) is placed from a_{cnt_i} (e.g., a_{cnt_3}) to the leftmost actor in the next element, i.e., α_{i+1} (e.g., a_2). SASM performs the placement of the counter constructs in a circular way. In other words, the next element of α_j is considered to be $\alpha_{(j+1) \mod n}$ where $j \in \mathbb{N} \wedge j \leq n$ for a LS $\{(\alpha_1)^{\beta_1}(\alpha_2)^{\beta_2}...(\alpha_n)^{\beta_n}\}$. The production rate of a_{cnt_i} on c_{lim_i} is set to the number of times that the leftmost actor in α_{i+1} can be executed in ${\alpha_{i+1}}^{\beta_{i+1}}$ (e.g., 1 as the number of times that a_2 can be executed in element a_2) and the consumption rate of the leftmost actor in the next element, i.e., α_{i+1}, from c_{lim_i} is set to 1. So, element α_{i+1} depends on actor a_{cnt_i} (because of c_{lim_i}) and actor a_{cnt_i} depends on α_i (because of the c_{cnt_i}); the β_i executions of α_i produce enough tokens on the counter channel c_{cnt_i} and then actor a_{cnt_i} can be fired. The firing of actor a_{cnt_i} provides enough tokens on limiter channel c_{lim_i} to only allow β_{i+1} executions for the next element α_{i+1}. Hence, by adding these components we enforce that α_{i+1} can be executed β_{i+1} times after α_i is executed β_i times.

The limiter channel of the counter construct added for the last element (i.e.,

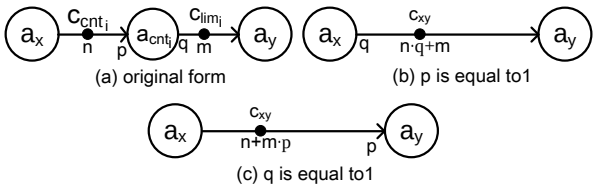


Figure 4.11: Optimization of SASM.

$(\alpha_n)^{\beta_n}$) in a LS $\{(\alpha_1)^{\beta_1}(\alpha_2)^{\beta_2}...(\alpha_n)^{\beta_n}\}$ is initialized with some initial tokens to prevent a deadlock in the graph. The number of initial tokens on the limiter channel is set to the count of the left most actor of α_1 in the first element of that LS (i.e., $(\alpha_1)^{\beta_1}$). Line 8 in SASM performs the token initialization. Inter-iteration execution cannot happen because SASM always creates a dependency from the last actor in the schedule to the first actor in the schedule.

In Figure 4.10, the actor a_{cnt_3} is added to count the number of times that the sequence a_0a_1 is executed. The consumption rate of actor a_{cnt_3} on its input channel is 5; this means that after 5 executions of sequence a_0a_1 the next actor (i.e., a_2) can be enabled. Also, the actor a_{cnt_3} limits the number of times that actor a_2 should get enabled; this can be done by choosing the value 1 as production rate of actor a_{cnt_3} on its output channel. In other words, the actor a_2 can only fire once because of the limitation imposed by actor a_{cnt_3}. The actor a_{cnt_1} (a_{cnt_2}) is added to ensure the single execution of actors a_1 (a_0) after the single execution of actor a_0 (a_1). The actor a_{cnt_4} is added to ensure that the sequence a_0a_1 can be executed 5 times after the actor a_2 is executed once.

SASM is applied recursively (line 4 in SASM) to model the nested LS α_i. For example, SASM(G, a_0a_1) will be called inside SASM(G, $(a_0a_1)^5a_2$); the result of the recursive call is shown in Figure 4.10 with a rectangle marking the inner-loop.

Some of the elements added by SASM can be removed without affecting the outcome. Consider Figure 4.11(a) which contains a counter actor and two channels that can be discarded in the following cases:

- The counter actor a_{cnt_i} can be removed if rate p is equal to 1. The counter actor and two channels in the original form are replaced with channel c_{xy} (see Figure 4.11(b)).

- The counter actor a_{cnt_i} can be removed if rate q is equal to 1. The counter actor and two channels in the original form are replaced with channel c_{xy} (see Figure 4.11(c)).

The newly replaced channel c_{xy} is only necessary if there is no other equivalent channel in the original SDF. The channels (p, q) and (p', q') which have the same source actor $a_x \in A$ and sink actor $a_y \in A$ are equivalent if the equation $\frac{Rate(p)}{Rate(p')} = \frac{Rate(q)}{Rate(q')} = \frac{\omega_0(c)}{\omega_0(c')}$ is true. Applying these optimizations on Figure 4.10 replaces all components added by SASM by channel c_{01} (see Figure 4.12). The SDF which models schedule s_2 in the SDF of Figure 4.9 with DSM and the HSDF-based

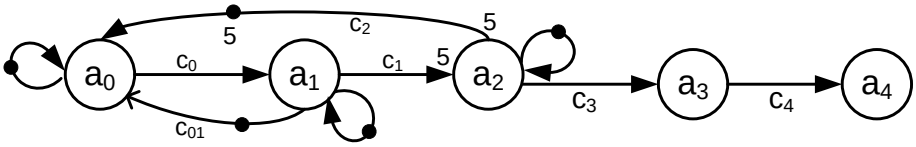


Figure 4.12: SDF of Figure 4.9 extended with $s_2 = \langle (a_0 a_1)^5 a_2 \rangle^*$ using optimized SASM.

techniques result in a graph with 10 (26) and 13 (21) actors (channels) resp. The SDF which models the same schedule with SASM only has 5 (9) actors (channels).

4.6 Correctness of DSM

This section discusses the correctness of DSM in modeling a single PSOS for a sub-set of the actors of the SDF. The extra actors and channels added by our techniques to model a PSOS (e.g., s_i) are only placed between actors of that schedule (i.e., A_i) and this is not imposing any restriction on the other actors of the SDF (i.e., actors in $A \setminus A_i$). Hence, if we can model a single PSOS in the SDF, then we can model multiple PSOSs by applying the algorithm multiple times.

DSM adds some actors, denoted by A_{s_i}, and channels, denoted by C_{s_i} to model the PSOS s_i in SDF $G(A, C)$. $G'(A', C')$ is the SDF that models s_i in G using DSM where $A' = A \cup A_{s_i}$ and $C' = C \cup C_{s_i}$.

The existence of a repetition vector for an SDF $G(A, C)$ ensures a balance between production and consumption rates. Hence, a *balance equation* can be defined as follow for each channel $c = (p, q) \in C$ where p and q are ports of actors a_p and a_q respectively: $Rate(p) \cdot \gamma(a_p) = Rate(q) \cdot \gamma(a_q)$. The existence of a repetition vector γ for an SDF ensures that each balance equation related to a channel in the SDF holds; under this situation the SDF is consistent. The following proposition shows the consistency of the schedule-extended SDF G'.

Proposition 1. *The SDF $G'(A', C')$ which models PSOS s_i in the consistent SDF $G(A, C)$ is consistent.*

Proof. The SDF G is consistent. Hence, a non-trivial repetition vector γ exists for G. We need to show that a non-trivial repetition vector γ' exists for G'. The balance equation related to each self-loop added by DSM (in line 1 of Algorithm 2) to remove auto-concurrency is always valid because the source and destination actor of the self-loop channel are identical with production and consumption rate equal to 1. The rates of the other channels added by DSM (for decision states or inter-iteration execution) share the following properties: (1) the newly added channel $c \in C_{s_i}$, which is added by DSM (in all lines 7, 8, 16 or 18 of Algorithm 2), is between an actor $a_p \in A (= A' \setminus A_{s_i})$ and an actor $a_q \in A_{s_i}$; (2) the rate of the new channel on the side of the actor a_q is equal to the count of the actor a_p in one iteration of s_i (i.e., $CNT(a_p, s_i)$); (3) the rate of the new channel on the side of the actor a_p is equal to 1. From these properties we conclude that $a_q \in A_{s_i}$, which is added by DSM (in both line 6 and 13 of Algorithm 2), fires

only once in each iteration of s_i. Consider a_q as the producer (see Figure 4.13(a)) of the newly added channel (i.e., $(a_q, a_p) \in C_{s_i}$); then, the only firing of a_q in one iteration of s_i provides $CNT(a_p, s_i)$ tokens for all firings of a_p in one iteration of s_i. Now reconsider actor a_q as the consumer (see Figure 4.13(b)) of the newly added channel (i.e., $(a_p, a_q) \in C_{s_i}$); as a result, all firings of actor a_p in one iteration of s_i provide $CNT(a_p, s_i)$ tokens for only one firing of a_q.

Let set A_i be the set of the actors in the PSOS s_i. Consider that each actor $a_p \in A_i$ appears $r \cdot \gamma(a_p)$ times in the PSOS s_i ($r = \frac{u}{v}$ where $u, v \in \mathbb{N}$) where the value r is identical for all actors in s_i. The appearance count of the actor $a_p \in A_i$ in s_i is represented by $CNT(a_p, s_i)$ and it is assumed to be equal to $\frac{u}{v} \cdot \gamma(a_p)$. We can write the above statement as follow: $CNT(a_p, s_i) \cdot v = \gamma(a_p) \cdot u$. From this equation we can conclude that u iterations of the SDF G cause v iterations of the PSOS s_i, leading to v firings of each of the actors added by DSM (i.e., actors from the set A_{s_i}). So, in u iterations of the SDF G (or v iterations of the PSOS s_i) the following equation holds for each channel $(a_p, a_q) \in C_{s_i}$ or $(a_q, a_p) \in C_{s_i}$ (where $a_p \in A_i$ and $a_q \in A_{s_i}$):

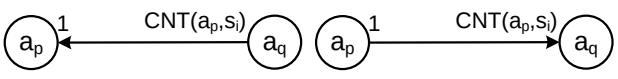
$$\underbrace{CNT(a_p, s_i)}_{Rate(a_q)} \cdot \underbrace{v}_{\gamma'(a_q)} = \underbrace{1}_{Rate(a_p)} \cdot \underbrace{\gamma(a_p) \cdot u}_{\gamma'(a_p)} \tag{4.1}$$

The entries of the repetition vector of the schedule-extended graph can be obtained using Equation 4.1. So a non-trivial repetition vector γ' can be found for the schedule-extended SDF G'. The relation between the repetition vector of the schedule-extended SDF and the repetition vector of the original graph is as follows: $\gamma'(a_q)$ (where $a_q \in A_{s_i}$) is equal to v and $\gamma'(a_p)$ (where $a_p \in A$) is equal to $\gamma(a_p) \cdot u$. □

The following proposition states that inter-iteration execution for actors of a PSOS s_i is impossible in the SDF G' which extends the original SDF G with PSOS s_i using DSM.

Proposition 2. *PSOS inter-iteration execution is impossible for any actor appearing in PSOS $s_i = \langle \alpha_1 \alpha_2 \dots \alpha_n \rangle^*$ in the SDF $G'(A', C')$, which models PSOS s_i in the SDF $G(A, C)$ using DSM.*

Proof. Let set A_i be the set of the actors in the PSOS s_i. In one iteration of a PSOS s_i, an actor $a_p \in A_i$ could be enabled more often than its designated number (i.e., $CNT(a_p, s_i)$ times). DSM prevents this by creating a dependency from the last actor appearing in s_i (i.e., actor $a_L = \alpha_n$) to the first actor appearing in s_i (i.e., actor $a_F = \alpha_1$) (see lines 4-8 in DSM). This dependency is created by



(a) a_q is producer (b) a_q is consumer

Figure 4.13: Extra actor a_q added by DSM in different situations.

inserting a new actor $a_{i.end}$ and two channels $c_{i.pre}$ and $c_{i.pro}$. The source (destination) actor of the channel $c_{i.pre}$ is a_L ($a_{i.end}$) with rate 1 ($CNT(a_L, s_i)$). So, one firing of $a_{i.end}$ needs $CNT(a_L, s_i)$ tokens available in $c_{i.pre}$; for this purpose actor a_L should fire $CNT(a_L, s_i)$ times. The destination actor of the channel $c_{i.pro}$ is a_F; the consumption rate of a_F from $c_{i.pro}$ is 1. The source actor of the channel $c_{i.pro}$ is $a_{i.end}$; the production rate of $a_{i.end}$ to $c_{i.pro}$ is $CNT(a_F, s_i)$. So the firing of a_F related to one iteration of PSOS s_i needs $CNT(a_F, s_i)$ tokens available in $c_{i.pro}$; for this purpose actor $a_{i.end}$ should fire once. The channel $c_{i.pro}$ is initialized with $CNT(a_F, s_i)$ number of tokens; this number of tokens provides enough tokens for actor a_F to fire $CNT(a_F, s_i)$ times in one iteration of PSOS s_i. The subsequent firing of actor a_F depends on the firing of the actor $a_{i.end}$ and the firing of the actor $a_{i.end}$ demands $CNT(a_L, s_i)$ times a firing of actor a_L. Hence, the firing of a_F belonging to the subsequent iteration of s_i can be performed only after a_L finishes all of its firings belonging to the current iteration of s_i (i.e., after completion of the current iteration of s_i). This holds because actor a_F is the first actor which should be fired in an iteration of s_i and other actors in s_i cannot get enabled before the first firing of a_F. This second fact is guaranteed by adding decision state constructs (i.e., lines 12-18 in DSM) for any possible decision state and Proposition 6 below. □

Even after eliminating inter-iteration execution from the SDF, multiple actors from a schedule may be enabled in an SDF iteration. We call such a state a *decision state*. The following proposition shows that analyzing only one SDF iteration is enough to identify all possible decision states.

Proposition 3. *Executing an SDF $G(A, C)$ for one iteration is enough to find all decision states within a PSOS s_i.*

Proof. Let the set A_i be the set of the actors in s_i and $A_o = A \setminus A_i$ the remaining actors in A. Consider inter-processor channels, which are defined as channel originating from the actors in A_o to the actors in A_i denoted by $C_{ipc} = \{(a_p, a_q) \in C | a_p \in A_o \wedge a_q \in A_i\}$. The execution of an actor $a_p \in A_o$ where $(a_p, a_q) \in C_{ipc}$ up-to its entry in the repetition vector of the SDF produces $\gamma(a_p) \cdot Rate(a_p)$ tokens in the corresponding channel $(a_p, a_q) \in C_{ipc}$. Actor $a_q \in A_i$ consumes those produced tokens within one iteration of the normalized PSOS s_i (because $\gamma(a_p) \cdot Rate(a_p) = \gamma(a_q) \cdot Rate(a_q)$). Hence, actors in A_i can receive the required tokens from all inter-processor channels C_{ipc} for one iteration of the normalized PSOS s_i. This means that all possible decision states related to PSOS s_i are detectable.

Actors in A_o could possibly fire more than the number mentioned above (i.e., corresponding value in vector γ) if the channel dependencies in the SDF allow additional firings of these actors. This could cause more than enough tokens (for one iteration of the normalized PSOS s_i) in channels C_{ipc}. This could enable an actor in A_i more than its designated number in one iteration of the normalized PSOS s_i. To avoid this undesirable actor enabling, some constructs are added to the graph to prevent inter-iteration execution (see Proposition 2). As a result, extra tokens produced by further firing of the actors A_o cannot enable any actor in A_i more than its designated value in one iteration of the normalized PSOS s_i.

So, executing actors in the SDF up-to their repetition vector entry (i.e., one SDF iteration) is enough to find all decision states within s_i. □

The identified decision states may be redundant. Proposition 4 discuses the proposed decision state reduction in the DSM.

Proposition 4. *Let σ be an execution of an SDF (A, C) and a PSOS s_i which schedules actors $A_i \subseteq A$. In the execution σ, consider y consecutive decision states $\omega_{x+1}, \omega_{x+2}, \cdots, \omega_{x+y}$. Assume that $a_o \in A_i$ is an opponent actor in each of these decision states but not the actor of choice in any of them. It is sufficient to only consider the last decision state ω_{x+y} to postpone the firing of the opponent actor a_o in those consecutive decision states to the state $\omega_{x+y+1} \in \sigma$.*

Proof. The purpose of the components added by DSM (i.e., lines 13-18 in Algorithm 2) in a decision state $\omega_j \in \Omega$ is to prevent any opponent actor a_o which is not the actor of choice in decision state ω_j from getting enabled in that state and as such to postpone that firing to the state ω_{j+1}. It is assumed that the opponent actor a_o is enabled in the consecutive decision states $\omega_{x+1}, \omega_{x+2}, \cdots, \omega_{x+y}$. Suppose that the opponent actor a_o was fired e times before the first decision state (i.e., ω_{x+1}) where $0 \leq e < \gamma(a_o)$. The components added by DSM in the last decision state ω_{x+y} prevent the opponent actor a_o from getting enabled for the $(e+1)^{th}$ time in decision state ω_{x+y}. As a result, the opponent actor a_o can also not be enabled in states $\omega_{x+1}, \omega_{x+2}, \cdots, \omega_{x+y-1}$ after adding DSM components for decision state ω_{x+y}. So, the components added by DSM in the last decision state of consecutive decision states are enough to prevent the firing of an opponent actor which is not the actor of choice in those consecutive decision states. □

Decision state folding overlaps the consecutive repetitions of the designated PSOS in an SDF iteration to reduce the number of decision states. The following proposition states that decision state folding does not dismiss any decision state.

Proposition 5. *Consider PSOS $s_i = \langle \alpha_1 \alpha_2 \dots \alpha_n \rangle^*$ for the sub-set of actors A_i from SDF (A, C); assume s_i is repeated μ_i times to form the corresponding normalized PSOS ($s'_i = \langle (s_i)^{\mu_i} \rangle^*$) to identify decision states related to PSOS s_i. After decision state folding all decision states are preserved.*

Proof. Normalization can be done by repeating s_i μ_i times (μ_i is the normalization factor of s_i). Decision state identification is applied to the normalized PSOS $s'_i = \langle (s_i)^{\mu_i} \rangle^* = \langle \underbrace{\alpha_1 \alpha_2 \dots \alpha_n}_{1^{st}} \underbrace{\alpha_1 \alpha_2 \dots \alpha_n}_{2^{nd}} \cdots \underbrace{\alpha_1 \alpha_2 \dots \alpha_n}_{\mu_i{}^{th}} \rangle^*$. Decision point constructs are added based on the given PSOS s_i. Folding groups the identified decision states of the normalized PSOS s'_i in the following manner: the state related to the firing of an actor α_j in s_i is considered as decision state if a decision state is found at least in one of the μ_i states that α_j fires in the execution related to s'_i.

The corresponding state related to firing of the actor α_j in s_i which is considered as a decision state imposes a decision state to all μ_i corresponding states of actor α_j in s'_i. So, no decision state will be lost after decision state folding and the

only effect is introducing unnecessary decision state controlling. We need to show that this extra controlling does not affect the execution of the SDF according to the schedule. The construct added in a decision state is used to guarantee execution of the actor of choice of that decision state. Even if a state is not a decision state, the added constructs for that unnecessary decision state only ensure that the only enabled actor in that state (i.e., the actor of choice) can be fired. This does not violate the actor firing order according to the PSOS in that state. □

DSM adds some components per decision state to enforce the firing of the enabled actor in a decision state which is in line with the given PSOS (i.e., actor of choice). Proposition 6 explains how those components can guarantee the firing of the actor of choice in the decision state.

Proposition 6. *The PSOS s_i is a schedule for actors $A_i \subseteq A$ from SDF (A,C). Let $\omega_j \in \Omega$ be a decision state within PSOS s_i and $\Delta_j \subseteq A_i$ the set of the opponent actors in decision state ω_j. The actor of choice in decision state ω_j (denoted by actor a_c) is the only actor which can fire in decision state ω_j among all actors in Δ_j after applying DSM. Also, one iteration of the resulting schedule-extended SDF resets the tokens to their initial positions (i.e., the periodic behavior of the original SDF is also preserved).*

Proof. We need to show that the opponent actors $\Delta_j \setminus \{a_c\}$ can not be enabled in the decision state ω_j after applying DSM. Accordingly, the actor of choice a_c in the decision state ω_j is the only actor which can fire in ω_j among all actors in Δ_j.

In the DSM technique, actor $a_{i.\omega_j}$ is added for each decision state $\omega_j \in \Omega$ within the PSOS s_i. An opponent actor $a_k \in \Delta_j \setminus \{a_c\}$ in the decision state ω_j is dependent on the new actor $a_{i.\omega_j}$ because of the added channel $c_{i.a_k\omega_j}$; this channel is initialized with $BEF(a_k, pos[\omega_j], s_i)$ tokens. The rate of the channel $c_{i.a_k\omega_j}$ on the side of a_k ($a_{i.\omega_j}$) is equal to one ($CNT(a_k, s_i)$). The new actor $a_{i.\omega_j}$ is also dependent on the actor of choice a_c of the decision state ω_j because of the added channel $c_{i.a_c\omega_j}$; this channel is initialized with $AFT(a_c, pos[\omega_j], s_i)$ tokens. The rate of the channel $c_{i.a_c\omega_j}$ on the side of a_c ($a_{i.\omega_j}$) is equal to one ($CNT(a_c, s_i)$).

An opponent actor $a_k \in \Delta_j \setminus \{a_c\}$ fires $BEF(a_k, pos[\omega_j], s_i)$ times before decision state ω_j and every time the opponent actor a_k consumes one token from $c_{i.a_k\omega_j}$. So, the $BEF(a_k, pos[\omega_j], s_i)$ firings of a_k before ω_j consume all tokens which were available in channel $c_{i.a_k\omega_j}$. Hence, the opponent actor $a_k \in \Delta_j \setminus \{a_c\}$ cannot fire in ω_j. Firings of the opponent actor $a_k \in \Delta_j \setminus \{a_c\}$ from decision state ω_j onward will be dependent on the firing of $a_{i.\omega_j}$ to provide the required tokens in channel $c_{i.a_k\omega_j}$. As mentioned before, $a_{i.\omega_j}$ depends on the actor of choice a_c. So, the opponent actor $a_k \in \Delta_j \setminus \{a_c\}$ cannot fire from decision state ω_j onward, unless the actor of choice a_c of the decision state ω_j fires. Hence, the actor of choice a_c is the only actor among the other opponent actors in ω_j which can fire. The actor of choice a_c is fired $BEF(a_c, pos[\omega_j], s_i)$ times by state ω_j and this results in $BEF(a_c, pos[\omega_j], s_i)$ tokens being produced in channel $c_{i.a_c\omega_j}$; as channel $c_{i.a_c\omega_j}$ is initialized with $AFT(a_c, pos[\omega_j], s_i)$ tokens, the number of tokens in this channel is $BEF(a_c, pos[\omega_j], s_i) + AFT(a_c, pos[\omega_j], s_i) = CNT(a_c, s_i)$ after

firing a_c in decision state ω_j. So, there will be sufficient tokens (for one firing of $a_{i.\omega_j}$) on the only channel leading to $a_{i.\omega_j}$ after firing of the actor of choice a_c in decision state ω_j. Then, firing of $a_{i.\omega_j}$ consumes all $CNT(a_c, s_i)$ tokens that are present in channel $c_{i.a_c\omega_j}$ and it produces $CNT(a_k, s_i)$ tokens in channel $c_{i.a_k\omega_j}$ ($a_k \in \Delta_j \setminus \{a_c\}$); therefore, the opponent actors $\Delta_j \setminus \{a_c\}$ are not any more dependent on $a_{i.\omega_j}$ in the remainder of the current iteration of s_i.

The firings of the opponent actor $a_k \in \Delta_j \setminus \{a_c\}$ after decision state ω_j consumes $AFT(a_k, pos[\omega_j], s_i)$ tokens from channel $c_{i.a_k\omega_j}$; as a result, at the end of the PSOS iteration, the number of tokens on this channel returns to its initial value which is $BEF(a_k, pos[\omega_j], s_i)$ (because $BEF(a_k, pos[\omega_j], s_i) = CNT(a_k, s_i) - AFT(a_k, pos[\omega_j], s_i)$). The actor of choice a_c fires $AFT(a_c, pos[\omega_j], s_i)$ times after decision state ω_j and the number of tokens in $c_{i.a_c\omega_j}$ returns to $AFT(a_c, pos[\omega_j], s_i)$. These initial token resettings at the end of the PSOS iteration ensure the periodic behavior for the added components in each decision state. Thus, in a decision state only the actor of choice (which is in line with the given schedule) amongst all opponent actors of the decision state can fire and this eliminates any non-determinism because of the decision state. □

The following theorems state the correctness of DSM in modeling a single PSOS for a sub-set of the actors of the SDF.

Theorem 1. *Consider PSOS s_i as a schedule for actors $A_i \subseteq A$ from SDF $G\ (A, C)$. Assume $G'(A', C')$ is the SDF that models s_i in G using DSM. For any execution σ' of $G'(A', C')$ it holds that σ satisfies s_i where it is assumed that σ is the execution of $G(A, C)$ with $orderList(\sigma, A) = orderList(\sigma', A)$.*

Proof. Proposition 6 states that in a decision state of s_i, an enabled actor of the decision state which is in line with s_i is the only actor able to fire in that state among all enabled actors in A_i. So, the order of s_i is the only possible order of actor firing for those actors of G' in the set A_i. Proposition 2 implies that the next PSOS iteration cannot interfere. Hence, for any execution σ' of $G'(A', C')$, $orderList(\sigma', A_i)$ has the form of $(s_i)^\kappa$ where $\kappa \in \mathbb{N}$ (i.e., infinite repetition of s_i). It is assumed that $orderList(\sigma, A) = orderList(\sigma', A)$; as $A_i \subseteq A$, we can conclude that $orderList(\sigma, A_i) = orderList(\sigma', A_i)$. Hence, $orderList(\sigma, A_i)$ also has the form of $(s_i)^\kappa$ and this form satisfies s_i; in other words, σ satisfies s_i. □

Theorem 2. *Consider PSOS s_i as a schedule for actors $A_i \subseteq A$ from SDF $G(A, C)$. Assume $G'(A', C')$ is the SDF that models s_i in G using DSM. For any execution σ of G that satisfies s_i it holds that there is exactly one σ' that is an execution of $G'(A', C')$ such that $orderList(\sigma, A) = orderList(\sigma', A)$.*

Proof. It is assumed that the firing order of actors belonging to the set A in execution σ' has the same actor firing order as in execution σ and execution σ satisfies s_i. We should show that there is precisely one execution with the property of execution σ' and that is a valid execution for G'. In a precise way, the actor firing order related to the set A in execution σ is a possible actor firing order for the original actors (i.e., actors not added by DSM) of G' when σ' is an execution of

G'. Actor a_x from $orderList(\sigma, A)$ belongs either to A_i or to $A_o = A \setminus A_i$ ($x \in \mathbb{N}$). The state transition $\omega_x \xrightarrow{a_x} \omega_{x+1}$ in execution σ is related to the firing of a_x. The state transition $\omega'_y \xrightarrow{a_y} \omega'_{y+1}$ in execution σ' is related to the firing of a_y. The difference between states from execution σ and σ' is only in the extra channels added by DSM (i.e., C_{s_i}). Any channel from C_{s_i} is connected to an actor from A_i; in other words, it is not connected to any actor from A_o. Incoming channels of an actor determine whether that actor can fire or not. Consider ω'_y from execution σ' has the same content of state ω_x from execution σ for all channels in C. Hence, each $a_x \in A_o$ that fires in ω_x of execution σ can also fire in ω'_y of execution σ' (i.e., a_y is a_x) because the content of the state related to channels C_{s_i} has no influence on actor enabling for any actor from A_o. But, when an actor a_x belongs to A_i, the content of the state related to channels C' could manipulate the actor firing order. It is assumed that firings of actor a_x in execution σ satisfy the PSOS s_i; actor a_x can also fire in the corresponding state (i.e., ω'_y) of execution σ' (i.e., a_y is a_x) because components added by DSM force the firing of the actor which is in line with the given s_i (i.e., a_x) among all actors in A_i (see Proposition 6). So, the firing order of actors from $A' \setminus A_{s_i}$ in execution σ' follows the same firing order as it is indicated in execution σ. Each actor $a \in A_{s_i}$ also has a single possible firing order in each PSOS iteration; if a is added to control the actor firing in a decision state, it fires before the actor of choice in the decision state (see Proposition 6) and if it is added for the inter-iteration prevention, it fires at the end of the PSOS iteration (see Proposition 2). Hence, there exists only one possible firing order in execution σ' for each actor $a \in A_{s_i}$. So, all actors from A' have exactly one firing order in execution σ' where $orderList(\sigma, A) = orderList(\sigma', A)$. □

The size of the schedule-extended graph (e.g., G') is dependent on the number of decision states found in the given schedule (e.g., s_i). In this section, decision state identification for a sub-set of actors of G (i.e., $A_i \subseteq A$) which belong to the schedule of interest (i.e., s_i) is explained regardless of existing other schedules for the rest of the actors in the SDF (i.e., actors in $A_o = A \setminus A_i$). In our implementation, we consider other possible schedules designated for the rest of the actors (i.e., actors in A_o) to reduce the number of the decision states and as a result the size of the schedule-extended graph. As we explained in Section 4.4.5.2, actors which do not belong to s_i should fire according to their schedule (if there is any) to perform their maximal execution. Firing of any actor $a_o \in A_o$, which belongs to another PSOS s_j ($j \neq i$), in function *maxExec* of the DSM algorithm should satisfy the schedule to which a_o belongs. In other words, a_0 in function *maxExec* of DSM fires if (1) a_o is enabled and (2) its firing satisfies s_j. Without the second condition, firing of a_o could enable an actor from s_i and lead to unnecessary decision states. So, considering other schedules in the function *maxExec* of DSM algorithm removes such redundant decision states.

4.7 Correctness of SASM

This section discusses the correctness of SASM in modeling a periodic static-order SAS for a sub-set of the actors of the SDF. We can model multiple SASs by applying the algorithm multiple times.

SASM adds some actors, denoted by A_{s_i}, and channels, denoted by C_{s_i} to model the SAS s_i in SDF $G(A, C)$. $G'(A', C')$ is the SDF that models s_i in G using SASM where $A' = A \cup A_{s_i}$ and $C' = C \cup C_{s_i}$.

Between any (nested) LSs α_u and α_v, where $v = (u \mod n) + 1$, in a LS SAS $s_i = \{(\alpha_1)^{\beta_1}(\alpha_2)^{\beta_2}...(\alpha_n)^{\beta_n}\}$, SASM adds one counter actor a_{cnt_u} and two channels c_{cnt_u} and c_{lim_u}. The following propositions shows how these components control the actor firings in α_u and α_v.

Proposition 7. *For any (nested) LSs α_u and α_v, where $v = (u \mod n) + 1$, in LS $s_i = \{(\alpha_1)^{\beta_1}(\alpha_2)^{\beta_2}...(\alpha_n)^{\beta_n}\}$, SASM ensures that α_v can only be executed for β_v times after α_u is executed for β_u times.*

Proof. SASM creates a dependency from the right-most actor (RMA) in α_u to the left-most actor (LMA) in α_v by adding one counter actor a_{cnt_u} and two channels c_{cnt_u} and c_{lim_u}. The channel c_{cnt_u} is placed from the RMA in α_u to a_{cnt_u}; the production rate of the RMA in α_u on channel c_{cnt_u} is set to one and consumption rate of a_{cnt_u} on channel c_{cnt_u} is set to the number of times that the RMA in α_u is needed to fire in β_u executions of α_u. Hence, actor a_{cnt_u} can fire once after the looped schedule α_u is executed for β_u times (the 1^{st} behavior). The channel c_{lim_u} is placed from a_{cnt_u} to the LMA in α_v; the production rate of a_{cnt_u} on channel c_{cnt_u} is set to the number of times that the LMA in α_v is needed to be fired in β_v executions of α_v and the consumption rate of the LMA in α_v on channel c_{cnt_u} is set to one. Hence, the looped schedule α_v can be executed for β_v times after one firing of a_{cnt_u} (the 2^{nd} behavior). It is assumed that SASM is recursively applied to each (nested) looped schedules in s_i. Considering both of the explained 1^{st} and 2^{nd} behaviors provides that α_v can only be executed for β_v times after α_u is executed for β_u times. □

SASM only places a specific number of initial tokens (i.e., the number of times that the LMA of α_1 should fire in ${\alpha_1}^{\beta_1}$) on the last limiter channel (i.e., c_{lim_n}) to prevent deadlock in execution; the following proposition explains how this prevents inter-iteration execution in the schedule-extended graph.

Proposition 8. *Inter-iteration execution is impossible for any actor appearing in SAS s_i in the SDF $G'(A', C')$.*

Proof. Considering the behavior described in Proposition 7 for all pairs of consecutive (nested) looped schedules in s_i ensures that the next schedule iteration cannot be started before the current schedule iteration ends. The initial token placed on the last limiter channel (i.e., c_{lim_n}) only provides tokens for β_1 executions of the looped schedule α_1; the executions of α_1 related to the current schedule iteration consumes all tokens on the last limiter channel and this prevents the next iteration from being executed until completion of the current iteration

(i.e., α_n is executed for β_n times). □

The following theorems state the correctness of SASM. These theorems are similar to the theorems given in Section **??**.

Theorem 3. *Consider SAS* $s_i = \{(\alpha_1)^{\beta_1}(\alpha_2)^{\beta_2}...(\alpha_n)^{\beta_n}\}$ *as a schedule for actors* $A_i \subseteq A$ *from SDF* $G\ (A, C)$. *Assume* $G'(A', C')$ *is the SDF that models* s_i *in* G *using SASM. For any execution* σ' *of* $G'(A', C')$ *it holds that* σ *satisfies* s_i *where it is assumed that* σ *is the execution of* $G(A, C)$ *with* $orderList(\sigma, A) = orderList(\sigma', A)$.

Proof. Considering Proposition 7 for all pairs of consecutive (nested) LSs in s_i guarantees that the order specified in s_i is the only possible order of actor firing for those actors of the SDF G' in the set A_i. Proposition 8 implies that subsequent SAS iterations cannot interfere. Hence, for any execution σ' of SDF $G'(A', C')$, $orderList(\sigma', A_i)$ has the form of $(s_i)^{\kappa}$ where $\kappa \in \mathbb{N}$ (i.e., infinite repetition of s_i). It is assumed that $orderList(\sigma, A) = orderList(\sigma', A)$; as $A_i \subseteq A$, we can conclude that $orderList(\sigma, A_i) = orderList(\sigma', A_i)$. Hence, $orderList(\sigma, A_i)$ also has the form of $(s_i)^{\kappa}$ and this form satisfies s_i; in other words, σ satisfies s_i. □

Theorem 4. *Consider SAS* s_i *as a schedule for actors* $A_i \subseteq A$ *from SDF* $G(A, C)$. *Assume* $G'(A', C')$ *is the SDF that models* s_i *in* G *using SASM. For any execution* σ *of* $G(A, C)$ *that satisfies* s_i *it holds that there is exactly one* σ' *that is an execution of* $G'(A', C')$ *such that* $orderList(\sigma, A) = orderList(\sigma', A)$.

Proof. It is assumed that the firing order of actors belonging to the set A in execution σ' is the same as in execution σ and σ satisfies s_i. We should show that there is precisely one execution with the property of execution σ' and that is a valid execution for G'. The actor firing order related to the set A in execution σ (i.e., $orderList(\sigma, A)$) is a possible actor firing order for the original actors (i.e., actors not added by SASM) of G' when σ' is an execution of G'. Actor a_x from $orderList(\sigma, A)$ belongs either to A_i or to $A_o = A \setminus A_i$ ($x \in \mathbb{N}$). The state transition $\omega_x \xrightarrow{a_x} \omega_{x+1}$ in execution σ is related to the firing of a_x. The state transition $\omega'_y \xrightarrow{a_y} \omega'_{y+1}$ in execution σ' is related to the firing of a_y. The difference between states from execution σ and σ' is only in the channels added by SASM (i.e., C_{s_i}). Any channel from C_{s_i} is connected to an actor from A_i; in other words, it is not connected to any actor from A_o. Incoming channels of an actor determine whether that actor can fire or not. Consider state ω'_y from execution σ' has the same content of state ω_x from execution σ for all channels in C. Hence, each $a_x \in A_o$ that fires in ω_x of execution σ can also fire in ω'_y of execution σ' (i.e., a_y is a_x) because the content of the state related to channels C_{s_i} has no influence on actor enabling for any actor from A_o. But, when an actor a_x belongs to A_i, the content of the state related to channels C' could manipulate the actor firing order. As we assume firings of a_x in execution σ satisfy s_i, a_x can fire in the corresponding state (i.e., ω'_y) of execution σ' (i.e., a_y is a_x) because components added by SASM force the firing of the actor which is in line with the given SAS s_i among all actors in A_i (considering Proposition 7 for all pairs

of consecutive nested LSs) and it is assumed that a_x is in line with s_i. So, the firing order of actors from $A' \setminus A_{s_i}$ in execution σ' follows the same firing order as indicated in σ. Each counter actor $a_{cnt_x} \in A_{s_i}$ also has a single possible firing order in each schedule iteration; it fires once after the LS α_x executed for β_x times (see proof of Proposition 7). Hence, there exists only one possible firing order in σ' for each actor $a \in A_{s_i}$. So, all actors from A' have exactly one firing order in σ' where $orderList(\sigma, A) = orderList(\sigma', A)$. □

4.8 Experimental Results

In this section, we evaluate our techniques experimentally. We first explain the experimental setup. We then evaluate our techniques in terms of the sizes of the schedule-extended graphs, comparing our techniques to that of [3]. We further consider the throughput analysis time when analyzing the schedule-extended graphs obtained by different techniques. Note that the throughput that is achievable for a given schedule is independent of the way it is encoded. It is the analysis time itself that is of interest. Finally, we look at the accuracy of buffer sizing analysis. The accuracy of obtained buffer requirements does depend on the way schedules are encoded.

4.8.1 Experimental setup

The DSM and SASM techniques have been integrated in the SDF^3 [71] dataflow tool set. We use a set of DSP and multimedia applications (see the first column of Table 4.1) to assess our DSM and SASM techniques.

In our experiments, applications are bound to a multi-processor platform using the technique of [69]. A PSOS determines the actor firing order and as such it influences the enabled actors in a state; as a result, the number of decision states can be different for different PSOSs. The size of the schedule-extended graph using DSM depends on the number of decision states in the given schedules. We use a list scheduling approach from [25] to determine PSOSs for the applications. We use two different variations to verify DSM in different situations. The first list schedule uses forward priorities (Lfp) and the second one uses reverse priorities (Lrp). Actors closer to the inputs of the graph have higher priority in the Lfp schedules compared to actors closer to the outputs of the graph and vice-versa in Lrp schedules.

The scheduling technique presented in [60] is used to derive SASs for our benchmark applications. The technique in [60] also minimizes the required buffer sizes when determining a SAS. However, the technique in [60] cannot directly be used for multi-processors. We have utilized the technique of [60] to find SASs for a multi-processor platform. Initially the binding technique from [69] is used to bind the SDF to a multi-processor platform. Then, the technique of [60] is applied to the SDF to derive a SAS for all actors in the SDF. This SAS is decomposed into some smaller SASs using the binding information; each of the smaller SASs is a schedule for one processor in the platform. Consider an example SDF with

Table 4.1: Size of the schedule-extended SDFs

	Orig. size		# act. (# chan.)			Reduction compared to the HSDF-based technique	
Benchmark	**# act. (# chan.)**	**Scheduling**	**HSDF-based**	**DSM**	**SASM**	**DSM**	**SASM**
H.263 dec. [72]	4 (6)	Lfp	1190 (2973)	6 (14)	NA	99% (99%)	NA
		Lrp	1190 (2972)	598 (1198)	NA	49% (59%)	NA
		SAS	1190 (2972)	598 (1198)	4 (12)	49% (59%)	99% (99%)
H.263 enc. [62]	5 (7)	Lfp	201 (596)	7 (16)	NA	96% (97%)	NA
		Lrp	201 (499)	105 (212)	NA	47% (57%)	NA
		SAS	201 (499)	106 (214)	5(15)	47% (57%)	97% (97%)
MP3 dec. [72]	14 (21)	Lfp	911 (2849)	27 (61)	NA	97% (97%)	NA
		Lrp	911 (2327)	400 (807)	NA	56% (65%)	NA
		SAS	911 (2327)	400 (807)	14 (44)	56% (65%)	98% (98%)
modem [9]	16 (35)	Lfp	48 (115)	22 (63)	NA	54% (45%)	NA
		Lrp	48 (128)	35 (89)	NA	27% (30%)	NA
		SAS	48 (128)	35 (89)	16 (61)	27% (30%)	66% (52%)
samplerate conv. [9]	6 (11)	Lfp	612 (1639)	12 (29)	NA	98% (98%)	NA
		Lrp	612 (1784)	157 (319)	NA	74% (82%)	NA
		SAS	612 (1865)	238 (481)	12 (31)	61% (74%)	98% (98%)
satellite rec. [64]	22 (48)	Lfp	4515 (11638)	41 (108)	NA	99% (99%)	NA
		Lrp	4515 (12820)	1223 (2472)	NA	72% (80%)	NA
		SAS	4515 (15270)	3673 (7372)	24(91)	18% (51%)	99% (99%)
MP3 playback [81]	4 (8)	Lfp	10601 (37531)	5298 (10600)	NA	50% (71%)	NA
		Lrp	10601 (37529)	5296 (10596)	NA	50% (71%)	NA
		SAS	10601 (37530)	5297 (10598)	6 (16)	50% (71%)	99% (99%)
bipartite [64]	4 (8)	Lfp	73 (341)	8 (20)	NA	89% (94%)	NA
		Lrp	73 (359)	26 (56)	NA	64% (84%)	NA
		SAS	73 (350)	17(38)	9(22)	76% (89%)	87% (93%)
channel eq. [57]	21 (40)	Lfp	41 (100)	32 (83)	NA	22% (17%)	NA
		Lrp	41 (95)	27 (73)	NA	34% (23%)	NA
		SAS	41 (93)	30 (79)	21(65)	27% (15%)	48% (30%)

5 actors denoted by $a_0 - a_4$. Assume a_0, a_1 and a_3 are bound to processor P_1 and a_2 and a_4 are bound to processor P_2. Applying the technique of [60] to this imaginary SDF gives the SAS $s_0 = \langle(a_0(a_1{}^2a_2a_3{}^4)^3)^2a_4{}^5\rangle^*$ for the whole SDF. This SAS can be decomposed using the binding information to form a SAS for each of the processor in the platform. Only considering actors bound to P_1 in s_0 results in $s_{01} = \langle a_0(a_1{}^2a_3{}^4)^3\rangle^*$ which is a SAS for the actors bound to P_1. Similarly, a SAS $s_{02} = \langle a_2{}^6a_4{}^5\rangle^*$ can be extracted from s_0 to order actors bound to P_2. This way we utilize the technique of [60] for multi-processor platforms. The optimality of the generated schedules from the performance or buffer sizing perspective is debatable. However, we use this adapted SAS technique merely to provide some near-optimal inputs to evaluate our SASM schedule encoding technique versus the existing technique. Our techniques do not affect the quality of the scheduling result itself.

4.8.2 Comparison on graph sizes

Table 4.1 contains the size of the schedule-extended graphs using the HSDF-based and DSM techniques to model Lfp and Lrp schedules for a single core platform (see the first two rows of each application line). Using schedules generated by Lfp, the number of decision states is less than when Lrp is used, except in the channel equalizer and MP3 playback applications. By using Lfp scheduling, actors closer to inputs have higher priority compared to actors closer to outputs. This leads to consecutive execution of an actor followed by consecutive execution of another actor with lower priority and so on. Thanks to our optimization in DSM, considering only one decision state before a context switch will be sufficient (e.g., decision state ω_9 in Figure 4.7) and the number of decision states can be reduced significantly. Usually SDF actors closer to outputs are dependent on actors closer to inputs; this dependency can prevent an actor from being executed consecutively in a graph scheduled by Lrp. As a result, the number of context switches in a graph scheduled by Lrp will typically be larger compared to Lfp. Hence, the effectiveness of the decision state optimization in DSM decreases and extra elements are required to model the schedules in the graph. The exceptions in the channel equalizer and MP3 playback are due to the existence of a cycle in the SDF; the cycle can increase the number of context switches in the schedule and as a result, Lfp could result in the same or a higher number of decision states in DSM compared to Lrp. However, DSM always outperforms the HSDF-based technique regardless of the input schedule in our experiments. The number of actors (channels) using DSM is 66% (71%) lower compared to the HSDF-based technique on average and 22% (17%) lower in the worst-case observed in our experiments. The average case refers to the mean value of the obtained results and the worst-case reports the smallest graph size reduction (i.e., reduction in numbers of actors and channels compared to the HSDF-based technique).

SASs are a suitable class of schedules that minimize code memory size. DSM is able to model any arbitrary schedule in an SDF. SAS can be modeled using DSM; however, it is possible to consider the intrinsic property of SASs when modeling a SAS in an SDF. Our second technique, SASM, uses the fact that each

Table 4.2: The throughput analysis time (in milliseconds)

		Type of the schedules / The used schedule modeling technique							
		SAS			**Lfp**		**Lrp**		
Benchmark	**# of processors**	**HSDF-based**	**DSM**	**SASM**	**HSDF-based**	**DSM**	**HSDF-based**	**DSM**
H.263 dec.	2	36540	< 1	< 1	118720	< 1	120710	< 1
H.263 enc.	2	380	< 1	< 1	690	< 1	400	< 1
MP3 dec.	3	13900	320	10	17660	< 1	13980	1380
modem	3	10	< 1	< 1	< 1	< 1	< 1	< 1
samplerate conv.	3	3970	110	< 1	3140	< 1	3880	< 1
satellite rec.	2	not finished in 3 days	12414400	130	not finished in 3 days	280	not finished in 3 days	675700
MP3 playback	2	not finished in 3 days	< 1	< 1	not finished in 3 days	10780	not finished in 3 days	< 1
bipartite	2	30	< 1	< 1	40	< 1	50	< 1
channel eq.	3	< 1	< 1	< 1	< 1	< 1	< 1	< 1

actor appears only once in the looped schedule form. SASM models the counter concept in the graph in order to force actors to be executed a specific number of times. The third row of each application line in Table 4.1 contains the size of the schedule-extended graphs using the HSDF-based, DSM and SASM techniques to model SASs, generated by the technique developed in [60].

SASM results in a schedule-extended SDF with a limited number of extra actors and channels. For example, SASM only adds 2 (8) extra actors (channels) to the original graph of the MP3 playback application in order to model a SAS, while the HSDF-based technique adds 1057 (37522) extra actors (channels) to model the same schedule. The graphs obtained by SASM have 88% (85%) and 48% (30%) less actors (channels) compared to the HSDF-based technique on average and in the worst-case among the benchmark applications. Using the DSM technique to model the same SASs results in 46% (57%) and 27% (15%) less actors (channels) compared to the HSDF-based technique on average and in the worst-case. Our results confirm that the techniques proposed in this chapter achieve a more compact schedule-extended graph compared to the available technique.

4.8.3 Comparison on analysis times

The time required to perform an analysis on an SDF depends on the size of the graph and the number of cycles in the graph. As an example, the throughput analysis of [34] is performed on the schedule-extended graphs using our techniques and the HSDF-based technique. Our experiments are performed to evaluate the impact of the graph size on the analysis time of a common analysis technique. Note that other techniques (e.g., YTO [87]) can be employed to calculate throughput of HSDFs, but the size of the schedule-extended HSDFs is such that our conclusions remain the same. The benchmark graphs are mapped onto multi-processor platforms with two or three processors. Table 4.2 contains the throughput analysis times when SASs, list forward priority (Lfp) schedules and list reverse priority (Lrp) schedules are used as input schedules. The results show the superiority of SASM over DSM and the HSDF-based technique. Note that the SDF to HSDF conversion is fast; the numbers reported for the HSDF-based technique in Table 4.2 are related to the run-time of the throughput analysis from [34]. In our experiment, the run-time of a throughput calculation for HSDFs is long independent of the analysis technique used (i.e., state-space [34], YTO [87], etc.).

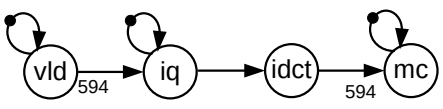
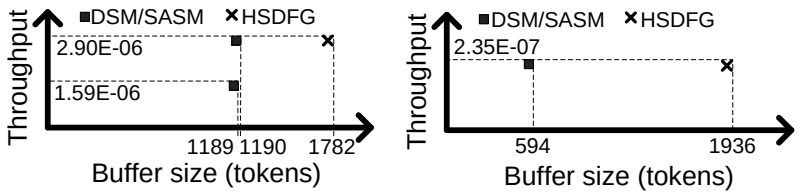


Figure 4.14: SDF of H.263 decoder application.

4.8.4 Comparison on buffer sizes

To further analyze the effectiveness of our techniques, the buffer sizing algorithm from [72] is applied to the schedule-extended SDFs of the H.263 decoder and MP3 decoder. Figure 4.14 depicts the SDF of the H.263 decoder. Besides the compactness of the schedule-extended graph, our techniques preserve the original structure of an SDF (when ignoring the added actors and channels), allowing accurate buffer sizing, which is not guaranteed for the state of the art technique. The H.263 decoder is mapped to a platform with two processors. The actors *vld* and *iq* are mapped to the first processor with a PSOS $\langle vld(iq)^{594}\rangle^*$ and the actor *idct* and *mc* are mapped to the second processor with a PSOS $\langle(idct)^{594}mc\rangle^*$. The analysis time for buffer sizing on the schedule-extended H.263 decoder is less than 1 ms when using DSM (or SASM) to model the schedules. The same analysis takes 1330 ms when using the technique from [3] to model the same schedules in the same graph. Figure 4.15(a) shows the Pareto space of throughput and buffer size when modeling the schedules with DSM (or SASM) and the HSDF-based technique [3]. In this experiment, the schedules are first modeled in the graph; then, the buffer sizing technique of [72] is applied. A single channel in an SDF corresponds to a set of channels in the equivalent HSDF. As a result, the buffer sizing technique cannot find the minimal buffer size when applying it on the equivalent HSDF. Our experiments show these inaccuracies. Applying buffer sizing on the graph which models the schedules using the technique from [3] results in 49% overestimation in required buffers compared to applying the same buffer sizing technique on the graph which models the schedules with one of our techniques. Note that the maximal achievable throughput is independent of the way schedules are encoded. The analysis results confirm this. Only the computed buffer sizes differ. For instance in both cases of Figure 4.15, the maximal throughput for the given schedules is always achievable, also by using the HSDF-based schedule modeling technique; the latter suggests the need for larger buffers though. Figure 4.15(b) shows results for the MP3 decoder. We use the mapping and scheduling from [32] for a platform with 3 processors. The analysis time on the graph which models the schedule using one of our techniques is 594 ms while 141610 ms is required to perform the same analysis on the graph using the technique from [3]. Using the technique from [3] results in 226% overestimation in buffer size compared to using our techniques.

Modeling a PSOS in an SDF using DSM requires execution of one complete SDF iteration. The number of states in one iteration could be exponential in the number of actors in the graph. However, for all real-world SDFs used in our experiments, the execution time of DSM is below 1 ms. SASM also models SASs based on the structure of the schedules in their looped form; as each actor appears once in a SAS, the complexity of SASM depends on the number of actors in the graph. Similar to DSM, the execution time of SASM is always below 1 ms in our experiments. The complexity of our techniques relates to the length of the SDF iteration and the number of processing tiles in the platform (i.e., $|PT|$). Hence, the complexity of our techniques is bounded to $O(|PT| \cdot \sum_{a \in A} \gamma(a))$.



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

Figure 4.15: Pareto space of schedule-extended graphs modeled by DSM and HSDF-based techniques (the scales of the two graphs are different).

4.9 Summary

We present two techniques, DSM and SASM, to model PSOSs and SASs directly in an S(A)DF. The resulting graphs are much smaller (often much less than half the size) than graphs resulting from the state of the art technique that requires conversion of the model to an HSDF graph. This results in a speed-up of analysis techniques. Computing the trade-off between buffering and throughput for multi-processor platforms, for example, becomes several orders of magnitude faster. Moreover, properties like buffer sizes can be analyzed more accurately. The techniques have been integrated in the SDF^3 tool set available at http://www.es.ele.tue.nl/sdf3. This allows easy integration of the techniques in multi-processor design flows. In the following chapters, we use the proposed schedule modeling techniques to amend the application model with scheduling decisions.

Chapter 5

Parametric Throughput Analysis

5.1 Overview

The timing behavior of an application depends on its binding, scheduling, buffer allocation, etc. Dynamic voltage and frequency scaling (DVFS), which is a commonly used technique to reduce the energy consumption, has also a direct influence on the timing behavior of an application. Common design space exploration (DSE) frameworks [11, 65, 68] perform several throughput calculations to determine the performance of multiple solutions in the design space. Moreover, at run-time, throughput calculation might be required when a run-time parameter (e.g., frequency of a processor) is changed. Both at design-time and at run-time, throughput analysis must be performed as fast as possible. To address this challenge, [33] introduces a parametric throughput analysis technique for SDF graphs. The technique finds throughput expressions for a parameterized SDF graph in which actors can have parameters as their execution time. These parameters have a specified time interval. The combination of all parameters of the SDF forms a multi-dimensional parameter space. A divide and conquer technique is used to determine all throughput regions in the parameter space. Each throughput region corresponds to a critical timing cycle in the SDF graph. For each region a throughput expression is discovered using a state-space exploration technique. The discovered throughput expressions can be used in any DSE framework or run-time manager to quickly compute the throughput of an SDF when the concrete values for all parameters are known. An evaluation of these throughput expressions (instead of a complete throughput analysis) can be done quickly while providing the same result.

When all actors have a fixed actor execution time, throughput analysis can be performed using the techniques from [34] (in case of SDF) or [32] (in case of SADF). However, when executing a design-time mapping flow, the actor execution times are only fixed near the end of the flow. Existing mapping flows re-evaluate

the throughput of an SADF whenever a mapping decision changes the actor execution times. The impact of this design decision can typically only be assessed after a throughput analysis is performed. Since many design alternatives must be evaluated, it is crucial to have a fast throughput analysis technique. Moreover, existing state-of-the-art throughput analysis techniques provide limited information to steer the design decisions (i.e., the critical cycle can be extracted, but it is often not possible to determine how design decisions influence this cycle or any of the other cycles in the graph).

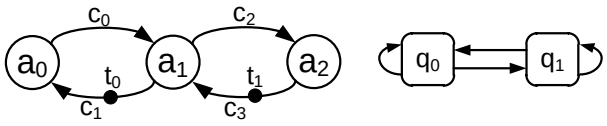
The existing technique from [33] is not directly applicable to SADF graphs. Applying SDF throughput analysis on applications with a dynamic behavior may result in a loose bound on the worst-case throughput. So, a new technique is required to determine throughput expressions for dynamic applications. This chapter presents such a parametric SADF throughput analysis technique. We use several real-world applications to evaluate our technique. Our experiments show that our technique is also better scalable than the one from [33]. The throughput expressions found using our technique can be applied to solve practical problems; in Chapter 6, we demonstrate how our parametric throughput analysis technique can be used to determine the lowest multiprocessor frequency setting under an application throughput constraint.

The remainder of this chapter is structured as follows. Section 5.2 contains the related work. Section 5.3 demonstrates how we parameterize an SADF model. Section 5.4 explains the divide and conquer approach from [33]. Section 5.5 explains how we find a throughput expression for a parameter point using a Max-Plus automaton graph. We evaluate our technique on several realistic applications in Section 5.6. Section 5.7 concludes. This chapter is based on the paper published as [23].

5.2 Related Work

The technique from [33] performs throughput analysis for SDF graphs when the execution time of the model is parameterized. SADF graphs are proposed to refine SDF graphs for dynamic applications. In [32], a Max-Plus-based SADF throughput analysis is introduced. Neither the technique from [32] nor [33] can determine throughput for parameterized SADF graphs. We propose a technique to enable throughput analysis for parameterized SADF graphs. Our technique can also be used to determine throughput expressions for parameterized SDF graphs. Similar to [33], we use a divide and conquer approach to determine throughput regions. In order to determine the throughput expression of a throughput region, we extend the MPAG-based analysis from [32] to a symbolic MPAG-based analysis.

Symbolic executions of the scenario graphs are used in our parametric throughput analysis technique to determine the Max-Plus matrices of an SADF. Symbolic executions may not be practical for large graphs. Approaches to symbolic execution for timing analysis like [39, 51] solve this issue by removing redundant expressions which are not affecting the critical paths. In our case, we only need to determine the Max-Plus characteristic matrices for a parameter point. We



(a) Scenario graph. (b) FSM.

	a_0	a_1	a_2	Parameter range
scenario ϕ_0	$6p_1$	$2.5p_2$	$2p_2$	$1 \leq p_1 \leq 5$
scenario ϕ_1	$1p_1$	$4.5p_2$	$1p_2$	$1 \leq p_2 \leq 5$

(c) Actor execution times in scenario ϕ_0 and ϕ_1.

Figure 5.1: SADF with two scenarios ϕ_0 and ϕ_1. The state-labeling function for the FSM is defined as follows: $L(q_0) = \phi_0$ and $L(q_1) = \phi_1$. The progress function specifies $Prgs(\phi_0) = 1$ and $Prgs(\phi_1) = 1$.

avoid complex redundant expression elimination by evaluating only the resulting terms for the given parameter point after each symbolic firing of the actors. In this way, only the terms which have the largest values amongst all terms propagate to the next step of the symbolic execution. Moreover, the techniques developed in [39, 51], which calculate timing expressions for integrated circuits, are limited to the single scenario cases and do not generalize to SADF.

5.3 Parametric SADF

A parametric SADF is identical to an SADF except that the actor execution times are not constant. Instead they are a function of a set P of parameters. Each parameter $p_i \in P$ can have any real value within an interval (i.e. $p_i \in I_i = [\min_i, \max_i] \subset \mathbb{R}$). Consider as an example the SADF shown in Figure 5.1. In this graph, the actor execution times are a function of two parameters p_1 and p_2. The parameterized execution times of our example parameterized SADF are shown in Figure 5.1(c). For simplicity, we assume in our example that the execution time of each actor depends only on a single parameter. Our technique (and implementation) can handle arbitrary linear expressions of the parameters to specify the execution time of each actor, i.e., $k_0 + \Sigma_{i=1}^{|P|} k_i \cdot p_i$ represents the general form of an actor execution time where each $k_i \in \mathbb{R}$ is a constant.

5.4 Parameter Space and Divide and Conquer Approach

Consider a parameterized SADF that uses a set P of different parameters in the actor execution times. These parameters form a $|P|$-dimensional parameter space which is a convex polyhedron, i.e., $\prod_{i=1}^{|P|} I_i$ where $I_i = [min_i, max_i]$ is the interval to which parameter $p_i \in P$ belongs. Figure 5.2 shows the 2-dimensional parameter

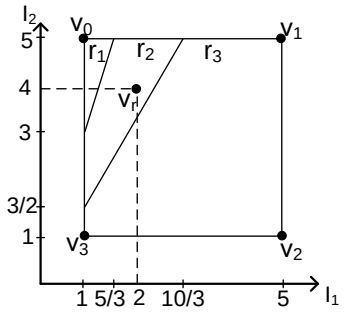


Figure 5.2: Throughput regions of the example SADF for the parameter space $I_1 \times I_2$. Expressions for the throughput regions: $MCR_1 = 5.5p_2$, $MCR_2 = 3p_1 + 4.5p_2$ and $MCR_3 = 6p_1 + 2.5p_2$.

space for our parametric example SADF. In this figure, the square $\nu_0 - \nu_1 - \nu_2 - \nu_3$ is the initial convex polyhedron. A throughput value $Th(\nu_i)$ can be assigned for each parameter point ν_i inside the parameter space. The throughput of parameter point ν_i can also be calculated by evaluating a throughput expression e_i for the given point ν_i; e.g., $e_r = \frac{1}{3p_1+4.5p_2}$ is the throughput expression for the parameter point $\nu_r : \{p_1 = 2, p_2 = 4\}$ and evaluating the expression e_r for ν_r results in the throughput amount $e_r(\nu_r) = \frac{1}{24}$ for the given parameter point. In [33], it is shown that such a throughput expression e_i can be used to calculate the throughput for all points in a convex sub-polyhedron in the initial parameter space polyhedron (see Proposition 5 from [33]). Hence, the initial parameter space is composed of one or several but a finite number of convex sub-polyhedrons and for each sub-polyhedron a throughput expression exists. Such a sub-polyhedron is called a *throughput region*.

As in [33], we use the same divide and conquer strategy to determine all throughput regions. The parameter space determines the initial polyhedron (e.g., the square in Figure 5.2). Initially, a random point ν_r is selected inside the polyhedron. As a first step, the throughput expression e_r for this point must be identified. The algorithm that does this is explained in the next sub-section. Once the throughput expression e_r has been found for this random point ν_r, the divide and conquer technique from [33] evaluates the throughput of each corner point ν_c in the initial polyhedron. When the throughput $Th(\nu_c)$ in a corner point ν_c is equal to the throughput found when evaluating the expression e_r for this point, then this point ν_c belongs to the same throughput region as point ν_r (see Proposition 8 from [33]); if this statement holds for all corner points of a polyhedron, the polyhedron will be identified as a throughput region with throughput expression e_r (see Corollary 6 from [33]). If $Th(\nu_c)$ is not equal to $e_r(\nu_c)$, then it holds that the corner point ν_c belongs to another throughput region than the random point ν_r. In that case, a new throughput expression e_c can be identified for the corner point ν_c. The hyperplane $e_r - e_c = 0$ cuts the initial polyhedron into two convex sub-polyhedrons. For each convex sub-polyhedron the divide and conquer strategy is performed recursively until all throughput regions

are identified. The interested reader is referred to [33] for a detailed description of the divide and conquer technique.

Algorithm 5: Throughput expression for a parameter point

input : SADF G
input : Parameter-Point ν_{r}
output: Throughput-Expression $\mathrm{e_r}$

```
1  n ← number of scenarios in G
2  sg_1 ··· sg_n ← scenario graphs of G
3  fsm ← FSM of G
4  for i ← 1 to n do
5      symG_i ← getSymG(sg_i, ν_r)
6  symMPAG ← getSymMPAG(symG_1 ··· symG_n, fsm)
7  MPAG ← evaluateSymMPAG(symMPAG, ν_r)
8  criticalCycle ← maximumCycleRatio(MPAG)
9  reward = 0
10 foreach channel c ∈ criticalCycle do
11     periodExp ← periodExp + getTerm(c, symMPAG)
12     reward = reward + delay(c)
13 e_r ← reward/periodExp
```

5.5 Throughput Expression for a Parameter Point

In [33], a time-based state-space exploration is used to find the throughput expression for a parameter point ν_r in the parameter space of an SDF. The technique is not directly applicable when more than one scenario in an application exists. Extending a time-based state-space exploration technique for a situation with multiple scenarios is not trivial because of the drastic growth in the number of states during the state-space exploration. In [32], two techniques to compute throughput of an SADF are presented, i.e., an iteration-based state-space exploration and an approach based on MPAG analysis. Inspired by [32], we use an approach based on MPAG analysis to perform parametric throughput analysis of an SADF. The complexity of the approach based on MPAG analysis depends less strongly on the number of parameters than the state-space approach. In an MPAG, iteration boundaries across all scenarios are distinguishable; this makes the throughput analysis for SADF graphs feasible. Algorithm 5 shows the pseudo-code of our algorithm to compute the throughput expression e_r of a parameter point ν_r in a parametric SADF G. In Section 2.5.3, it is explained how an MPAG can be used to compute the throughput when all actors have a known execution time (i.e., when all parameter values are fixed). We extend this MPAG to a symbolic MPAG that can be used to compute symbolic throughput expressions. As a first step, a symbolic Max-Plus characteristic matrix of each scenario graph must be computed (line 4-5). Next, these matrices must be combined into a symbolic

MPAG (line 6). As a third step, the symbolic MPAG is evaluated for a concrete parameter point ν_r. This results in a concrete MPAG (line 7) from which the critical cycle can be extracted using a maximum cycle ratio algorithm (line 8). The YTO algorithm [87] is used to perform MCR analysis. The input of the YTO algorithm is a directed graph; each edge of this graph is associated with two values called weight and delay (see Section 2.5.3). Note that the evaluated concrete value of each edge in the symbolic MPAG is used as weight of that edge in the YTO algorithm and the progress value assigned to the scenario on the source side of each edge in the symbolic MPAG is used as delay of that edge in the YTO algorithm. Using a relation between the edges in the concrete and symbolic MPAG, the critical cycle in the concrete MPAG can be translated into a symbolic cycle (expression) in the symbolic MPAG. This symbolic expression is the inverse of the symbolic throughput expression e_r. Lines 9-13 in Algorithm 5 are responsible to identify the symbolic critical cycle using the concrete critical cycle. The variable *reward* in the algorithm is used to count the number of data units processed in the critical cycle; this is done by summing up the delay values assigned to those edges of the MPAG in the critical cycle (see line 12). The symbolic expression of the critical cycle (i.e., periodExp) divided by the reward results in a symbolic MCR that is the inverse of the throughput expression.

Algorithm 6: Symbolic matrix extraction (**getSymG**)

input : SDF g
input : Parameter-Point ν_r
output: Symbolic Max-Plus Matrix symG

```
1 while one iteration of g is not completed do
2     foreach enabled actor a ∈ g within one iteration do
3         symbolicFire(a)
4         prune timetamps of produced tokens by firing a for the point ν_r
5 extract symG from the resulting timetamps
```

Algorithm 6 represents our approach to determine the symbolic Max-Plus matrix of a scenario graph, which is an SDF, for a specified parameter point. A symbolic execution of the given graph g - up to one iteration - is performed to determine the token timestamps (lines 1-4 in Algorithm 6). In the symbolic execution, the parameterized actor execution time expressions and a symbolic timestamp for each initial token are used. Figure 5.3 illustrates the symbolic execution of scenario ϕ_1 in our example SADF. Step $S.1$ shows the initial graph and steps $S.2$, $S.3$ and $S.4$ show the graph after consecutive firing of the actors a_0, a_1 and a_2 respectively. Each actor firing is performed symbolically (symbolicFire in Algorithm 6). For example the firing of actor a_0 consumes a token from channel c_1 and produces a token with timestamp $\tau_{a_0} + t_0 = p_1 + t_0$ on channel c_0. Comparing steps $S.1$ and $S.4$ shows that the graph returns to the initial token distribution after one iteration (as expected). The token timestamps in step $S.4$ contain the symbolic dependencies of the initial tokens at the end of this itera-

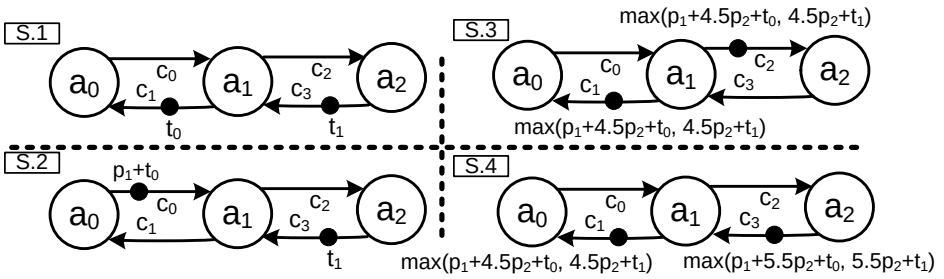


Figure 5.3: Symbolic execution of scenario graph ϕ_1 of our example SADF.

tion to the initial tokens at the end of the previous iteration. For example, the timestamp $max(p_1 + 4.5p_2 + t_0, 4.5p_2 + t_1)$ of the reproduced token t_0 in step $S.4$ implies that token t_0 depends on the production time of the tokens t_0 and t_1 in the prior iteration. The distance of t_0 in the current iteration to t_0 and t_1 in the previous iteration are at least $p_1 + 4.5p_2$ and $4.5p_2$ respectively; this information is used to construct the symbolic Max-Plus characteristic matrix of the scenario graph (line 5 in Algorithm 6). In our example, the matrices for scenarios ϕ_0 and ϕ_1 are equal to:

$$M_{\phi_0} = \begin{pmatrix} 6p_1 + 2.5p_2 & 2.5p_2 \\ 6p_1 + 4.5p_2 & 4.5p_2 \end{pmatrix} \qquad M_{\phi_1} = \begin{pmatrix} p_1 + 4.5p_2 & 4.5p_2 \\ p_1 + 5.5p_2 & 5.5p_2 \end{pmatrix}$$

In practice, it is often not feasible to perform a complete symbolic execution of one iteration of a scenario graph. This is caused by the number of terms that appear in the maximum (max) operator of the token timestamps. In our example, both max operators contain only two terms, one for each initial token. The number of terms may however grow rapidly when a graph contains more initial tokens and/or has a large repetition vector rendering a complete symbolic execution impractical. Ref. [39] encounters a similar issue when trying to identify timing expressions for the critical paths in an integrated circuit. This issue is solved by removing redundant expressions which are not affecting the critical paths. In our case, we only need to determine a symbolic Max-Plus characteristic matrix valid for a concrete parameter point (line 7 and 8 in Algorithm 5). This allows us to evaluate the maximum operation (in a token timestamp) for the given parameter point after each symbolic firing of an actor (line 4 in Algorithm 6). In this way, only the terms which have the largest values among all other terms in the maximum operator propagate to the next step of the symbolic execution. This prevents an explosion in the number of terms in the maximum operation. We construct the symbolic MPAG using the FSM of the SADF and the symbolic Max-Plus matrices $symG_1 \dots symG_n$ in the same way as the concrete MPAG is constructed in [32]. The only difference is that we use expressions instead of concrete numbers as edge weights. The function getSymMPAG in Algorithm 5 is used to construct the symbolic MPAG *symMPAG*. Figure 5.4 shows the symbolic MPAG for our example SADF.

To find the critical cycle, and hence the throughput expression, a maximum

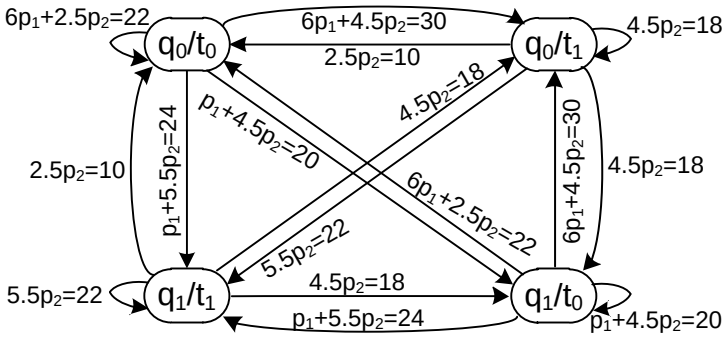


Figure 5.4: Symbolic MPAG of the example SADF.

cycle ratio (MCR) analysis must be performed on the symbolic MPAG. As the parameter point ν_r is known, we evaluate the term of each edge in the symbolic MPAG for the parameter values in ν_r. The function evaluateSymMPAG in Algorithm 5 is used for this purpose. The result of the evaluation for the parameter point $\nu_r : \{p_1 = 2, p_2 = 4\}$ is shown in Figure 5.4 as a number that follows the term of the edge. These numbers are used as a weight for the corresponding edges. Using the MCR analysis algorithm proposed in [87] (function maximumCycleRatio in Algorithm 5), the algorithm identifies a critical timing cycle (*criticalCycle*) which in turn is used to determine the throughput expression. Note that the progress value of each scenario in our example SADF is assumed to be one; as a result, all edges in the MPAG receive the delay value of one for the MCR analysis. The critical timing cycle in our example graph is the cycle between nodes q_0/t_1 and q_1/t_0 in Figure 5.4. The symbolic terms of each edge in the critical cycle are used to determine the throughput expression for the given parameter point (lines 9-13 in the algorithm). In our example, for the given point (i.e., $\nu_r : \{p_1 = 2, p_2 = 4\}$), $periodExp = 6p_1 + 9p_2$ and the throughput expression is $e_r = \frac{2}{6p_1+9p_2}$ (is shown by $MCR_2 = 3p_1 + 4.5p_2$ in Figure 5.2). This expression is used as an expression for a throughput region in the divide and conquer technique. Continuing the divide and conquer, different throughput expressions result for the corner points. The symbolic MCR for ν_0 and ν_2 are calculated as follows respectively: $MCR_1 = 5.5p_2$, $MCR_3 = 6p_1 + 2.5p_2$. The hyperplanes $MCR_1 - MCR_2 = 0$ and $MCR_3 - MCR_2 = 0$ divide the parameter space into three throughput regions (see Figure 5.2) for the example parameterized SADF.

5.6 Experimental Results

A comparison between our technique and an existing parametric throughput analysis technique for SDF graphs is made in Section 5.6.1. The performance of our technique on SADF graphs is evaluated in Section 5.6.2. All experiments are performed on an Intel core i7 (3 GHz) running Linux.

5.6.1 Comparison with [33] on SDF Graphs

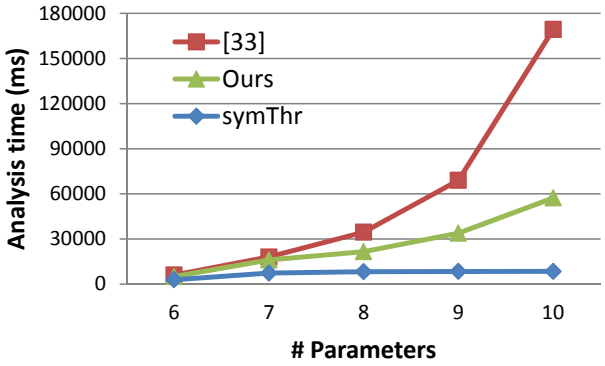
Since SDF graphs are a special case of SADF graphs, we can compare our technique for SADF throughput analysis to the only existing parametric throughput analysis technique for SDF graphs, i.e., [33]. In our comparison, we use the same set of SDF graphs (see Table 5.1) with the largest parameter range used in [33]. In this case, the parameterized actors show a variation in their execution time between their nominal value and 150% of this value. Table 5.1 shows the number of parameters ($\#p$) in each SDF, the number of throughput expressions ($\#exp$) and the run-times of the technique presented in [33] and our technique. Note that the model transformation introduced in Section 2.6 is applied to all benchmark graphs before performing our parametric throughput analysis. The results show that our technique is faster on almost all benchmark SDF graphs. The MP3 playback and modem SDF graphs are the only graphs on which our technique is slightly slower. This is due to the large number of initial tokens in the graphs which results in a large MPAG and in a long run-time of the MCR algorithm.

Table 5.1: Number of throughput regions and run-time SDF graphs.

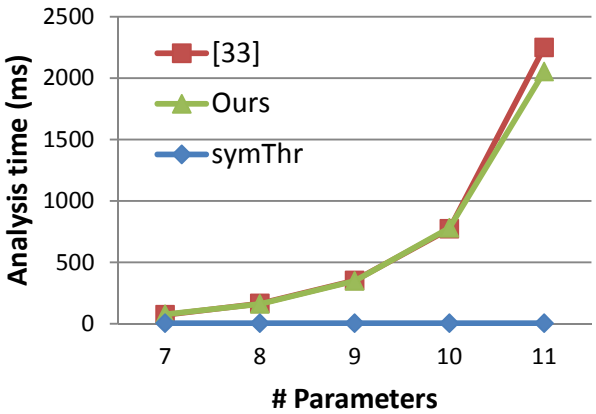
Benchmark	**#p**	**#exp.**	**[33] (ms)**	**Ours (ms)**
H.263 decoder [72]	4	1	94	80
H.263 encoder [62]	5	1	36	20
Modem [9]	7	1	96	116
MP3 decoder [72]	8	1	164	160
MP3 playback [81]	1	1	1348	1680
Samplerate conv. [9]	4	2	168	128
Satellite receiver [64]	9	3	69376	33478

The divide and conquer algorithm which is used in both [33] and in our technique performs two different kinds of throughput calculations. It needs to determine throughput expressions for some parameter points and compute concrete throughput values for corner points of throughput regions. The technique from [33] uses a state-space exploration to determine the throughput expressions and concrete throughput values. The state-space exploration needs to keep track of all parameters in its state-space exploration which makes its run-time dependent on the number of parameters. The size of the MPAG used in our technique is not dependent on the number of parameters. Therefore, the run-time of the MCR algorithm is also independent of the number of parameters. Only the function *evaluateSymMPAG* in Algorithm 5 depends on the number of parameters. However this function only evaluates a symbolic MPAG for a parameter point and this can be done in a negligible amount of time. As a result, the run-time of our technique is almost independent of the number of parameters in the graph. The blue line in Figure 5.5 shows the run-time of our symbolic throughput computation (i.e., Algorithm 5) when increasing the number of parameters. In this experiment, a parameter models the execution time of an actor. Increasing the number of parameters in this experiment implies that the execution times of more actors in the model are parameterized. The results shown in Figure 5.5 confirms the indepen-

dence of the run-time of our symbolic throughput computation of the number of parameters. Figure 5.5 also compares the overall run-time of the technique from [33] and our technique (including both the symbolic throughput computation as well as all concrete throughput computations at the corner points). When increasing the number of parameters, more concrete throughput computations need to be performed. Our concrete throughput analysis technique (which is in principle intended for the more expressive SADF model) is slower than the dedicated SDF technique used in [33]. As a result, part of the gains in our symbolic throughput computation are lost. However, the total analysis time of our approach is smaller than the analysis time of the technique of [33] in most of the benchmark graphs (see Table 5.1).



(a) Satellite receiver.



(b) MP3 decoder.

Figure 5.5: Execution time when varying number of parameters.

Table 5.2: Number of throughput regions and run-time SADF graphs.

Benchmark	#sce.	#p	#exp.	Time (ms)
MPEG4 decoder [75]	9	4	3	488
MP3 decoder [32]	5	8	3	2792
WLAN [58]	4	10	3	2342
Mapped MP3 decoder [32]	5	3	4	1252
Mapped WLAN [58]	4	3	10	196

5.6.2 Performance of our technique on SADF graphs

Our throughput analysis technique is tested on a set of SADF graphs described in the literature (see Table 5.2). Our benchmark consists of a set of realistic applications, i.e., an MPEG-4 decoder with 9 scenarios, an MP3 decoder with 3 scenarios, and a wireless LAN (WLAN) receiver with 4 scenarios. For the latter two applications, we use both a version in which each actor is mapped to a different processor and a version in which some actors are mapped to the same processor (labeled 'Mapped' in Table 5.2). In the MPEG-4 decoder, each actor is mapped to a different processor. The execution time of all actors is parametrized with a linear expression in which the execution time of the actor in the non-parameterized SADF model is multiplied with a parameter p_i that corresponds to the processor on which the actor is mapped. The actor execution times are varied between their nominal value and 500% of this value (i.e., $1 \leq p_i < 5$). These large parameter ranges allow us to show the scalability of our approach when the parameter ranges are large. Table 5.2 shows the number of throughput expressions ($\#exp$, corresponding to the number of throughput regions) of each SADF and the time used by our technique to discover these expressions. The experimental results show that our technique is able to handle realistic applications within a limited run-time.

5.7 Summary

SADF graphs with parameterized execution times enable analysis of implementation decisions that could change the timing property of the application across all scenarios. The only existing parametric throughput analysis technique can handle a less expressive MoC, i.e., the SDF model. Experimental results show that our technique outperforms this technique despite the fact that we can handle a more expressive MoC, i.e., SADF model. Moreover, our technique offers better scalability when the number of parameters increases. In the next chapter, we use our parametric SADF throughput analysis to devise a DVFS controller to minimize total energy consumption for a throughput-constrained application running on a multi-processor platform.

Chapter 6

Throughput-Constrained DVFS

Many streaming applications must provide timing guarantees (e.g., throughput) to assure their quality-of-service. For instance, a video decoder which is running on a mobile device is expected to deliver a video stream with a specific frame rate. Moreover, the energy consumption of such applications on handheld devices should be as low as possible. This chapter proposes a technique to select a suitable multiprocessor DVFS point for each mode (scenario) of a dynamic application described by an SADF. Our technique assures strict timing guarantees while minimizing energy consumption. The technique is evaluated by applying it to several streaming applications. It solves the problem faster than the state of the art technique for dataflow graphs. Moreover, the DVFS controller devised using the proposed technique is more compact and reduces energy consumption compared to the controller devised using the counterpart technique. An online approach is combined with our design-time DVFS technique to further lower the operating frequencies by exploiting the possible slack produced at run-time. An initial version of this work was published as [22].

6.1 Overview

Streaming applications, such as signal processing and multimedia applications, are often expected to meet certain timing requirements. Furthermore, energy consumption is an important design criterion for such applications. DVFS [14] is used to develop low power/energy implementations. This chapter presents a technique to determine for each scenario an energy-aware frequency setting while satisfying a throughput constraint. It is assumed that the application (SADF) is already mapped and scheduled to a platform with multiple processing elements. The technique presented in Chapter 4 is used to include mapping and scheduling information into the SADF model. The switching cost of DVFS is considered in

our analysis.

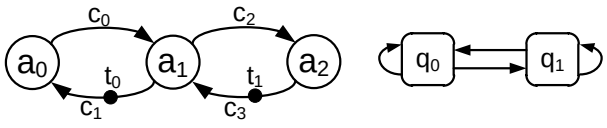
Ref. [89] addresses the same problem as we do. In the DVFS controller of [89] devised for an SADF, a power mode (i.e., DVFS operating point) is specified for each possible state of the application. Timestamps are used to distinguish between states. Timestamps capture the miss-aligned completion of the iterations on a platform with multiple processing elements. In [89], an initial state is selected as starting point; for each possible scenario transition from that state a low-power mode that satisfies the timing requirement for the upcoming iteration is considered as desired power mode for that specific scenario switch. This can lead to a new state or a recurrent state. In case of a new state, the exploration should be continued for the new state. The exploration is stopped if all discovered states are recurrent. This way, all possible states within the given power modes are traversed; the authors of [89] categorize their approach within the *state-space based* techniques. Our approach is distinguishable from [89] for several reasons:

1. In [89], only one iteration is considered in power mode selection which can result in a greedy slack distribution within just that iteration; this prevents fair slack distribution over multiple iterations. In contrast, we do power mode selection over all iterations involved in all *critical timing cycles*; a critical timing cycle is defined as a cycle that limits the throughput. In our approach, slack can be used across multiple iterations which generally provides higher energy savings.

2. The state-space-based DVFS technique can result in many distinguishable states. This makes the analysis a time consuming procedure. In our approach, the critical timing cycles are identified and resolved by choosing suitable frequency settings for the processing elements that execute the actors involved in those cycles; processing elements not involved in any critical cycle can operate at their lowest frequency. The analysis time of our technique on four realistic benchmark models is much smaller than the analysis time of the state-space based technique on the same applications.

3. Our DVFS controller is more compact than the one from [89] reducing the storage requirement. For a WLAN application, our technique only requires storing 4 sets of processor frequencies (equal to the number of scenarios) while the state-space based technique requires for the same application a controller with 277 frequency sets.

Our technique requires to identify which actor firings in which scenarios are part of the critical timing cycle and on which processing elements those actors are executed. For this reason, the SADF model is extended to a parametric SADF model to accommodate the processor clock cycle periods (i.e. inverse of the frequencies) in the model. In this model, instead of using concrete values, linear expressions (e.g., as shown in Figure 6.1(c)) provide the actor execution times in terms of some parameters (scale factors). As an example consider the parametric SADF depicted in Figure 6.1 with two scenarios ϕ_0 and ϕ_1; both scenarios use the same scenario graph shown in Figure 6.1(a). The timing properties of scenarios are specified according to Figure 6.1(c). In this example, parameters



(a) Scenario graph. (b) FSM.

	pe_1	pe_2		
scenario	a_0	a_1	a_2	scale factor sets
scenario ϕ_0	$10p_{1,1}$	$7p_{1,2}$	$9p_{1,2}$	$p_{1,1}, p_{1,2} \in \text{SFS}_1 = \{1,2,3,4,5\}$
scenario ϕ_1	$8p_{2,1}$	$3p_{2,2}$	$8p_{2,2}$	$p_{2,1}, p_{2,2} \in \text{SFS}_2 = \{1,2,3,4,5\}$

(c) Actor execution times in scenario ϕ_0 and ϕ_1.

Figure 6.1: Parametric SADF with two scenarios ϕ_0 and ϕ_1. The state-labeling function for the FSM is defined as follows: $L(q_0) = \phi_0$ and $L(q_1) = \phi_1$. The progress function specifies $Prgs(\phi_0) = 1$ and $Prgs(\phi_1) = 1$.

can get a value from the associated set; For example $p_{1,1}$ can be assigned with a value from set SFS_1. The FSM of the example SADF is shown in Figure 6.1(b). The processors can be explicitly modeled with some initial tokens. This makes it possible to identify when an iteration has ended on a processor to allow switching to another DVFS operating point. The timing expressions of critical cycles in a parametric SADF reveal which processors are involved in the critical cycles and how much their clock cycle periods (or frequencies) contribute to the length of these timing cycles. This information is used to choose energy-aware frequencies per application scenario to ensure a throughput-constrained solution. The frequency choices are made at design-time and are used at run-time to enable DVFS on iteration boundaries. Our technique is applied to four streaming applications: an MPEG4 video decoder [74], an MP3 audio decoder [32], a WLAN receiver [58] and the baseband (physical layer) processing of the Long Term Evolution (LTE) standard [55]. In all cases, our proposed technique is faster than the technique from [89] and it provides significant energy saving compared to the counterpart technique; furthermore, our technique constructs more compact DVFS controllers. We combine the resulting DVFS setting obtained from our design-time technique with a reactive online approach to further exploit the timing slack produced at run-time.

The remainder of this chapter is structured as follows. Section 6.2 discusses related work. Section 6.3 presents our design-time DVFS assignment technique. Section 6.4 shows how the results of our design-time DVFS technique can be used with a run-time DVFS approach. The experimental results are given in Section 6.5. Section 6.6 concludes.

6.2 Related work

Related DVFS approaches can be viewed from three different angles:

- Design policy.
- Application model.
- Solution granularity.

The *design policy* determines whether the DVFS choices are made at *run-time* [1, 18] or *design-time* [61, 66]; the first type predicts the workload and adjusts the supply voltage and the operating frequency of the underlying processing elements at run-time. The second ones assume that the workload of applications are known at design-time. Run-time approaches suffer from extra timing and energy overheads. Design-time approaches can result in pessimistic solutions if the behavior of the applications is not captured properly. The application models to devise a DVFS controller may be *static* or *dynamic*; in static models (e.g., SDF and task graphs) [61, 66], actors (tasks) use the worst case as their execution time (i.e., WCET). Dynamic models (e.g., SADF) [35, 89] capture execution time variation. Static models are simple and easy to analyze which make them suitable for static applications. Dynamic models are favorable for more dynamic applications (e.g., modern multimedia applications). The solution granularity determines how often a voltage and frequency switch can occur in a design. In a *fine-grained* solution [61, 66], a DVFS may happen before executing actors (tasks); in a *coarse-grained* solution [89], the DVFS may happen before executing iterations (e.g., processing a video frame). The overhead of DVFS, the size of the actors and the size of the iteration determine which granularity is suitable for a system.

Our technique analyzes SADFs to devise a coarse-grain DVFS controller at design time for dynamic throughput-constrained applications. SADFs are suitable to capture the dynamic behavior of applications. We offer a coarse-grained DVFS solution to avoid overhead because of the frequent voltage and frequency switches in fine-grained solutions. However, fine-grained solutions like [61] can be beneficial when the overhead is negligible. Some run-time approaches (e.g., [79]) use WCET information to perform better run-time power management. Our technique can also be used to provide initial information for such run-time techniques to further tune the final solution and loosen the run-time overhead. We demonstrate such a run-time refinement by considering an online DVFS approach at run-time.

Among the related work, [89] is the most similar to ours. The technique of [89] suffers from state-space explosion and it requires much time for the analysis; our technique finds solutions for all of our benchmark applications in seconds to minutes, depending on the number of scenarios and frequency points. Moreover, using [89] results in pessimistic solutions because of the greedy approach used in its voltage and frequency selection. Our technique considers all iterations involved in the critical timing cycles to effectively utilize slack; in this way, workloads are balanced across multiple iterations and frequencies can be lowered. As a result, our technique assures better solutions in terms of energy consumption. The DVFS controller devised by our technique is more compact than the DVFS controller of [89]; this can save considerable memory space to store the controller.

The authors of [86] develop a game theory-based technique to synthesize a controller for SADFs; they optimize throughput by modifying the scheduling policy. The resulting controller of [86] is optimized for a single design metric

(i.e., only throughput), while our technique reduces energy consumption under a throughput constraint. Note that our technique is a heuristic which provides sub-optimal solutions and the technique of [86] finds optimal solutions. Extending the game theory-based technique of [86] to consider two design metrics is a relevant, but challenging problem which requires further research. Ref. [35] presents a technique to detect application scenarios at design-time; it also devises a pro-active voltage scaling by predicting the scenario sequences. However, the technique of [35] is not applicable to multiprocessor platforms.

6.3 Off-line DVFS Approach

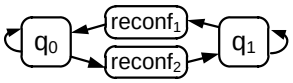
Power consumption in VLSI circuits depends linearly on the operating frequency and quadratically on the supply voltage of the processing elements [12]. While lowering the voltage supply, the maximal possible operating frequency also reduces. Hence, lowering the voltage and frequency could reduce the total energy consumption quadratically at linear time cost [13]. So, in our technique, the execution times are expressed linearly in terms of the processor clock cycle periods (i.e. inverse of the frequencies) and energy consumption is expressed quadratically in terms of the processor frequencies.

6.3.1 Overview

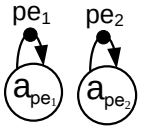
This section presents a multiprocessor frequency assignment technique for dynamic applications modeled by SADF graphs in such a way that a strict throughput requirement is guaranteed. In the following subsection, we show how to capture the timing delays resulting from DVFS in an SADF model. Our DVFS assignment technique starts with the minimum energy option; in other words, it assigns the lowest possible frequency (or the highest clock cycle period) to each processor running in an application scenario. The initial setting may violate the required throughput. So, the technique checks whether or not the initial setting satisfies the throughput; if the throughput is satisfied, the initial setting is reported as final solution. In the other case, our technique finds a critical timing cycle in the application which violates the throughput. Then, the frequency of the processors involved in the critical cycle are increased to make that cycle fit within the required throughput. Even after resolving the first critical cycle, some other timing cycles may exist which are violating the throughput. Those timing cycles are similarly resolved one after another until all of the timing cycle in the SADF model respect the required throughput. Section 6.3.3 discusses our technique in detail.

6.3.2 Modeling Voltage and Frequency Scaling in SADF

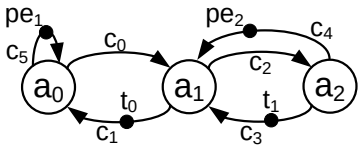
DVFS can be viewed upon as a reconfiguration at run-time. This can be effectively modeled in SADFs as a *reconfiguration scenario*. In the FSM of the SADF, an intermediate state must be placed before switching to another original FSM's state. Two *reconfiguration states* are added to the FSM of our example SADF



(a) Scenario FSM of reconfiguration.



(b) Reconfiguration SDF.



(c) Mapped scenario graph.

Figure 6.2: Modeling mapping and reconfiguration in the example SADF. The state-labeling function for the FSM is defined as follows: $L(q_0) = \phi_0$, $L(q_1) = \phi_1$ and $L(\text{reconf}_1) = L(\text{reconf}_2) = \phi_{\text{reconf}}$. The progress function specifies $Prgs(\phi_0) = 1$, $Prgs(\phi_1) = 1$ and $Prgs(\phi_{\text{reconf}}) = 0$.

to capture the reconfiguration steps (see Figure 6.2(a)). For a reconfiguration state, a reconfiguration scenario is defined. A reconfiguration scenario captures any step involved in the reconfiguration operation. For instance in our example SADF which is mapped to a platform with two processing elements, the reconfiguration requires setting the frequencies and voltages for two processing elements. The SDF of the reconfiguration scenario for each of the processing elements contains an actor with a self-edge; on that self-edge, a token with a label indicating the related processing element models the processing resource dependency between consecutive iterations (see Figure 6.2(b)). The execution time of the added actors are set to the DVFS delay. This way, the overhead of switching between different frequency points on iteration boundaries is considered. Mapping should also be modeled in the scenario graphs of the SADF. The technique presented in Chapter 4 is used for this purpose. Figure 6.2(c) shows such a modeling for the scenario graph of the example SADF; the processor tokens in this SDF establish the resource dependencies among all scenarios. This effect is shown graphically in Figure 6.3; in this figure, an example scenario transition from scenario ϕ_0 to scenario ϕ_1 is depicted. The time required to capture the DVFS delay is modeled by a reconfiguration scenario shown with ϕ_{reconf} in Figure 6.3. Hence processor tokens pe_1 and pe_2 can be released (to be used by ϕ_1) after the DVFS setting completion on both of the processing elements.

6.3.3 Clock cycle period settings under a throughput constraint

The used SADF is a parametric model. Algorithm 7 contains our heuristic technique to identify the scale factors (parameters) which result in energy savings under a throughput constraint. G represents the given parametric SADF and $Period$ represents the inverse of the required throughput. The set SFS of scale

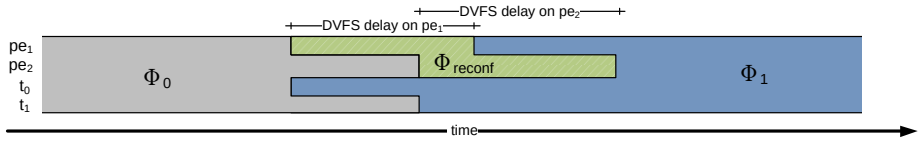


Figure 6.3: Processor tokens to model resource dependency across scenarios.

factors is also required as an input to the algorithm. Each $\text{SFS}_j \in \text{SFS}$ contains all concrete values that a parameter $p_{i,j}$ can obtain (i.e., all allowed scale factors for processor pe_j). *Solution* is a set which contains the final concrete values for all parameters (as the output of the algorithm). For the example SADF, 65 time units is specified as the timing constraint (i.e, $Period = 65$). The DVFS delay is assumed to be 1 time unit in our example.

As a first step, the existence of any valid solution for the given scale factors is checked by the function feasibilityCheck (lines 1-2 in Algorithm 7); this is checked by extracting the critical timing cycle for the case that all parameters (scale factors) are set to their minimum values (i.e. the highest processor frequencies). The problem is not feasible in case the length of the critical cycle is larger than the timing constraint (i.e., *Period*); otherwise, the analysis is continued to find the desired solution.

Initially, all parameters in the parametric SADF are set to their maximum value (lines 5-7 in Algorithm 7); this assures the lowest energy consumption, although this may not satisfy the timing requirement. The technique iteratively refines the initial parameter setting to obtain a parameter point that meets the timing constraint. The repetitive part of the algorithm (lines 8-25 in Algorithm 7) is composed of two main steps followed by a re-initialization step. In each of the main steps, a critical cycle of the SADF for the given parameter point is extracted by function getCriticalCycle; the technique presented in Chapter 5 is used for this. In case of multiple critical cycles with equal length, one of them is chosen arbitrarily. The other cycles can be processed in later repetitions of the algorithm. The extracted critical cycle must be resolved by choosing a proper scale factor for the parameters involved in the critical cycle; our scale factor selection approach, which is abstracted by function resolveCycle in the algorithm, picks a parameter point amongst all possible combinations of the involved parameters in the critical cycle to achieve the lowest energy consumption while still meeting the timing requirement. The complexity of resolveCycle depends on the number of parameters that contribute to the critical cycle (denoted by ρ) and the number of possible clock cycle (or frequency) points for each of the processing elements (denoted by π). So, the number of parameter points required to be verified by resolveCycle is equal to π^ρ because each parameter in the critical cycle can get π different values. The value of π depends on the platform property; the value of ρ depends on the application property and at worst case ρ can be equal to the number of parameters in the parametric SADF model. Hence, resolveCycle has an exponential complexity. However, our experiments on several real applications

Algorithm 7: Throughput-constrained DVFS for SADF

input : Parametric SADF G
input : Timing constraint $Period$
input : Scale factor sets SFS = $\{\text{SFS}_1 \dots \text{SFS}_n\}$
output: Scale factor set $Solution$

```
1  if feasibilityCheck(G, min_1 ... min_n, Period) ≠ "Feasible" then
2      return Solution ← ∅
3  n ← number of processing elements
4  m ← number of scenarios in G
5  for j ← 1 to n do
6      for i ← 1 to m do
7          pa_{i,j} = max_j
8  while true do
9      /* step 1 */
10     criticalCycleExpr_1 ← getCriticalCycle(G, pa)
11     if |criticalCycleExpr_1| > Period then
12         pb ← resolveCycle(criticalCycleExpr_1, pa, SFS, Period)
13     else
14         return Solution ← pa
15     /* step 2 */
16     criticalCycleExpr_2 ← getCriticalCycle(G, pb)
17     if |criticalCycleExpr_2| > Period then
18         pc ← resolveCycle(criticalCycleExpr_2, pb, SFS, Period)
19     else
20         return Solution ← pb
21     /* re-initialization */
22     for j ← 1 to n do
23         for i ← 1 to m do
24             if pb_{i,j} ≠ pc_{i,j} then
25                 pa_{i,j} = pc_{i,j}
```

reveal that only a few parameters contribute to the critical cycle (i.e., in practice ρ is small); hence, verifying all parameter points can be done quickly (refer to Section 6.5).

Figure 6.4 depicts the application of our algorithm to the example SADF. At step 1 of the 1^{st} repetition, the critical timing cycle with expression $10p_{1,1} + 7p_{1,2}$ is extracted when all parameters are initialized with a value of 5 time units. This cycle violates the required timing constraint of 65 time units. Hence, the parameters $p_{1,1}$ and $p_{1,2}$ must be set in a way that this cycle gets bounded within the required period. All parameter combinations of $p_{1,1}$ and $p_{1,2}$ are shown in Figure 6.4(a) (left-top); the points marked with solid black dots are not valid selections as they violate the timing constraint. Among the rest of the parameter points, $p_{1,1} = 3$ and $p_{1,2} = 5$ are selected since this choice assures the lowest energy consumption at this stage. In this example, we assume that the platform is homogeneous and processing elements consume equal amounts of energy when they are operating at the same frequency. The resulting parameter point after step 1 (i.e., parameter point $\overline{pb}$) is fed to step 2. The first critical cycle now has been resolved by step 1; the next critical cycle can be extracted in step 2. The new critical cycle found in step 2 can be resolved similarly as step 1. As shown in step 2 of the 1^{st} repetition in Figure 6.4, only parameter $p_{1,2}$ contributes to the second critical cycle; so, $p_{1,2}$ is reduced to value 4 in order to resolve the critical cycle found in step 2. The outcome of the second step is the parameter point $\overline{pc}$. Resolving a critical cycle by one step cannot enlarge other critical cycles found in prior steps; because in function resolveCycle, we restrict our choice to parameter points in which all parameters have values smaller than or equal to the ones in the input parameter point argument of resolveCycle.

Both of the main steps decrease parameters in order to shrink the identified critical cycles. The re-initialization step (lines 22-25 in Algorithm 7) provides an opportunity for parameters to avoid unnecessary frequency increase for some processing elements involved in critical timing cycles when possible. Consider that $\overline{pa}$ is the parameter point to be used in the subsequent algorithm repetition. The re-initialization step updates the parameters in $\overline{pa}$ with the values of the parameters whose values have been reduced in step 2. Parameters which were only reduced in step 1 of the current repetition keep their original values. In this way, they are reconsidered in the next algorithm repetitions. The reason for not changing the parameter values for those parameters that were only changed in the first step is that the critical cycle of the first step may have been affected by the adaptations made in the second step. As a result of the re-initialization step, the information obtained in one repetition is used in subsequent repetitions to provide better parameter point selection. In the second repetition of the algorithm for our example SADF, resolving the first critical cycle $10p_{1,1} + 7p_{1,2}$ is performed with the knowledge that parameter $p_{1,2}$ should get a value smaller than 5 because of the critical cycle $16p_{1,2}$; the solid red dots in Figure 6.4(c) display this effect. Hence, step 1 in the second repetition of the algorithm reduces $p_{1,1}$ to 4 (instead of 3 after step 1 of the 1^{st} repetition). The critical cycle identified by step 2 in the second repetition is already smaller than the required *Period* and the algorithm stops further analysis and reports the current parameter point (i.e., $\overline{pb}$ in the

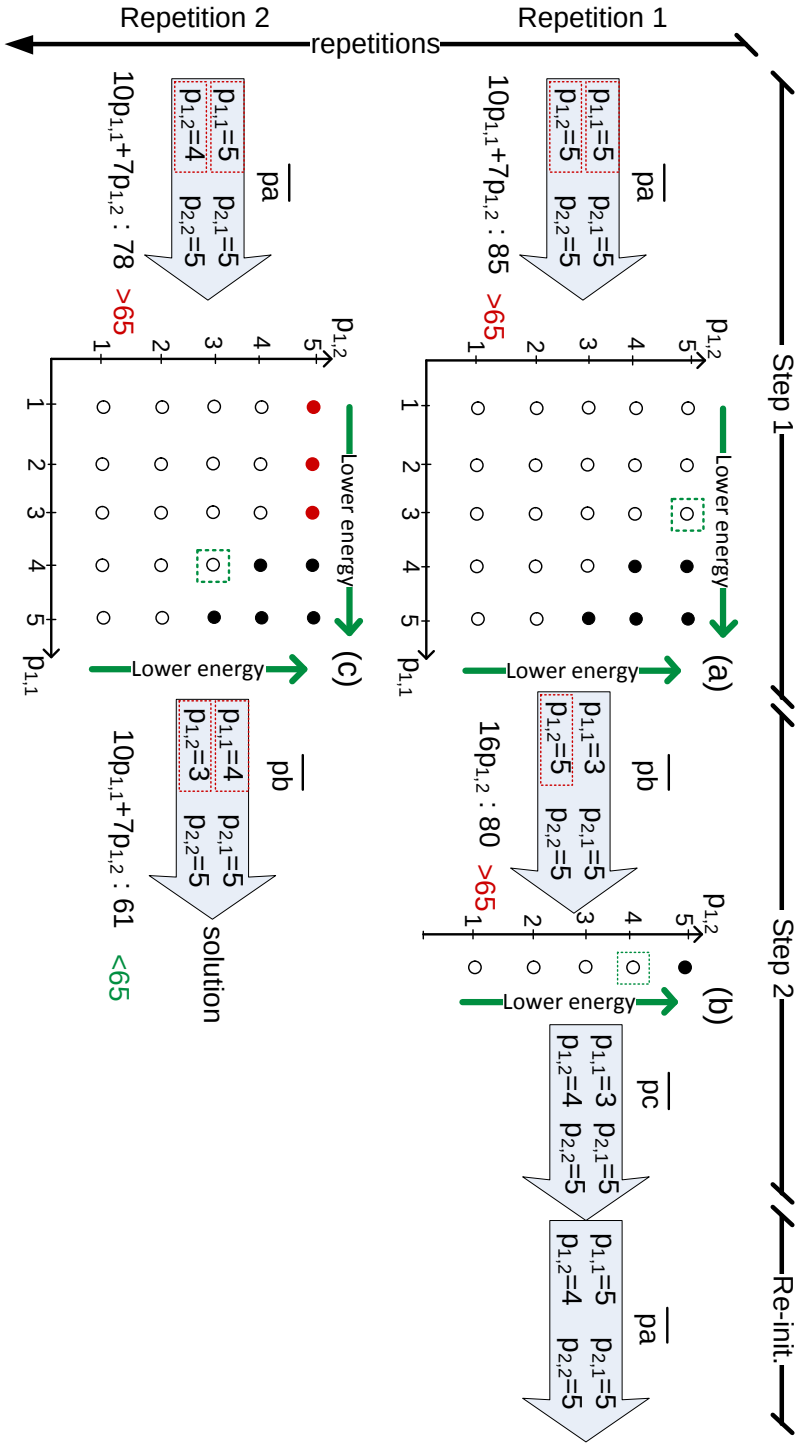


Figure 6.4: Applying our technique to the example SADF.

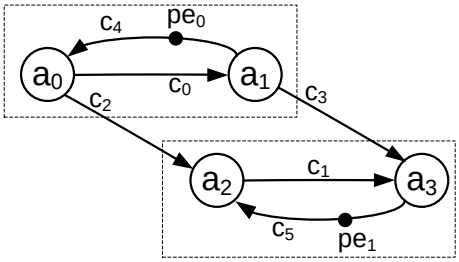
second repetition) as the final solution.

The algorithm stops either after step 1 or step 2 (line 15 or line 20 in the algorithm) whenever the length of the critical cycle in one of those steps is smaller than the timing requirement; if not, the re-initialization step fixes the values of those parameters which are reduced in step 2 for the subsequent repetitions. This ensures that by each repetition of the algorithm some parameters get smaller and eventually the timing requirement is met. In our algorithm, two main steps perform parameter point selection, each considering the information from one critical cycle. Increasing the number of main steps in our algorithm allows considering more critical cycles in our parameter point selection; however, more steps implies more analysis time. Empirically, the number of main steps is set to two in our heuristic.

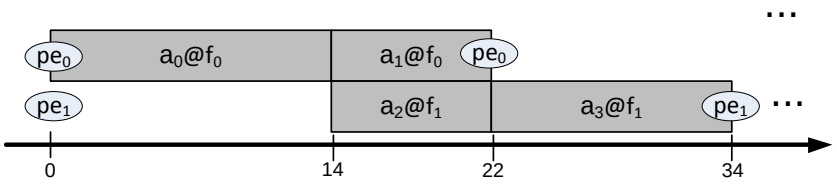
6.4 Online DVFS Approach

The proposed approach in Section 6.3 offers a design-time frequency assignment for each scenarios of an SADF graph. Our design-time approach uses the worst-case execution times for an actor running in a specific scenario. Using scenarios makes it possible to assign a better worst-case execution time value to each actor in the model; in other words, the worst-case execution time of an actor running in a specific scenario is a good estimation of the actual execution time of that actor at run-time. However, this may not hold for any actor. As an example, consider a *variable length decoding* (VLD) actor that receives data items with non-deterministic lengths; the length of a data item is only known after receiving all bits. Hence, a design-time DVFS approach using the worst-case execution time of a VLD actor may assign an unnecessarily high frequency to the underlying processing element. Some timing slack will be generated at run-time when the worst-case and actual execution times are different. Since this difference is only detectable at run-time, only an online (also called run-time) approach can exploit the slack to lower the frequencies and consequently lower the total energy consumption.

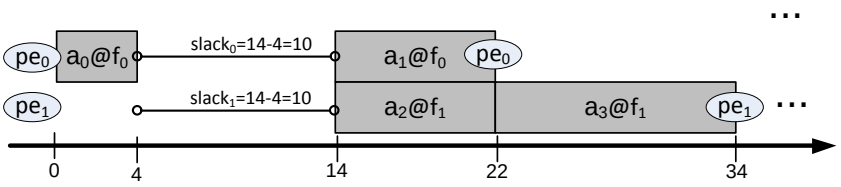
We use our design-time approach to derive the initial frequencies for all scenarios in the SADF graph. As an example, consider the SDF graph show in Figure 6.5(a) as a scenario graph of an SADF graph. Assume that actors a_0 and a_1 running on processing element pe_0 with design-time frequency setting f_0 and actors a_2 and a_3 running on processing element pe_1 with design-time frequency setting f_1. These frequencies are selected to ensure a throughput constraint of $1/22$ iterations per time unit. The frequencies f_0 and f_1 are selected based on the worst-case execution times; and when the underlying processing elements are running at these frequencies the worst-case execution times are as follows: $\tau(a_0) = 14$, $\tau(a_1) = \tau(a_2) = 8$ and $\tau(a_3) = 12$. One iteration of the SDF graph is shown in Figure 6.5(b) when actual execution times of the actors are equal to their worst-case values. However, in practice, an actor may fire before its expected worst-case value. Assume that the actual execution time of actor a_0 when running on pe_0 with frequency f_0 in one iteration is 4 time units. Figure 6.5(c) depicts this situation. The difference between the worst-case and the actual execution times



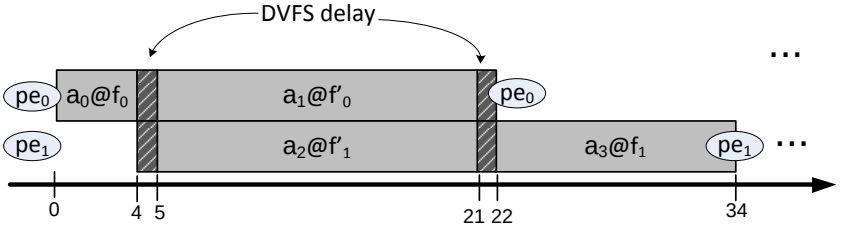
(a) A scenario graph (SDF).



(b) One iteration considering worst-case execution times with design-time frequencies.



(c) One iteration considering actual execution times with design-time frequencies.



(d) One iteration considering actual execution times with run-time frequencies. $f'_0 = f_0/2$ and $f'_1 = f_1/2$.

Figure 6.5: Demonstration of an online DVFS approach. Actors a_0 and a_1 running on processing element pe_0 with design-time frequency setting f_0. Actors a_2 and a_3 running on processing element pe_1 with design-time frequency setting f_1.

Table 6.1: The specification of benchmark applications.

Benchmark	**#sce.**	**#p.e.**	π	**#par**	**timing constraint**
MPEG4 dec.	9	4	2	36	20 frames/sec.
MP3 dec.	5	3	2	15	20 frames/sec.
WLAN	4	3	2	12	250k OFDM symbols/sec.
LTE	5	2	5	10	100 symbols/sec.

determines the generated slack. Hence, on each of the processing elements, a slack of $14 - 4 = 10$ time units is generated. This slack can be utilized to lower the operating frequencies of the underling processing elements. In our example, the frequencies of processing elements executing a_1 and a_2 can be reduced respectively from f_0 and f_1 (assigned by the design-time approach) to f_0' and f_1'. We assume that the DVFS delay is equal to 1 time unit. So considering the DVFS delay in our example leads to extra available $slack - 2 \times (DVFS\ delay) = 10 - 2 \times 1 = 8$ time units for execution of actors a_1 and a_2. Hence, at run-time, the frequency of pe_0 when executing a_1 can be reduced to $f_0' = f_0 \times 8/(8+8) = f_0/2$ without violating the throughput constraint. Similarly, at run-time, the frequency of pe_1 when executing a_2 can be reduced to $f_1' = f_1 \times 8/(8+8) = f_1/2$.

6.5 Experimental Results

The proposed DVFS technique for SADFs is compared to the related work (i.e., [89]) which uses an approach based on state-space exploration. This comparison is performed because [89] is the closest approach to our technique in the literature and we solve a similar problem for (scenario-aware) dataflow graphs. A set of realistic applications is used for this comparison: an MPEG4 video decoder [74], an MP3 audio decoder [32], a WLAN receiver [58] and the baseband (physical layer) processing of the Long Term Evolution (LTE) standard [55]. Table 6.1 shows for each of the applications the number of application scenarios ($\#sce.$), the number of processing elements ($\#p.e.$) in the platform to which the application is mapped, the number of the available frequency points (π), the number of parameters in the parametric SADF ($\#par$) and the timing constraint of each application. The overhead of DVFS is set to a value taken from [48, 56]. Hence, 10 ns is used as delay of DVFS in our experiments. All experiments are performed on an Intel core i7 (3 GHz) with 4GB of RAM running Linux.

6.5.1 Results of Off-line Approach

The complexity of both our DVFS technique and the state-space based technique from [89] is determined by the number of scenarios. Scenarios of an application can be clustered to form a less complex model. For instance, we clustered 9 scenarios in the MPEG4 decoder into a smaller model with 4 scenarios. Analyzing a smaller model can be done faster than the original model; but, analysis of such

Table 6.2: Experimental results.

	the technique from [89]		*our technique*			
Benchmark	**#states**	**analysis times (ms)**	**#rep**	**analysis times (ms)**	ρ	**energy gain**
MPEG4 dec. (9 scenarios)	over 300k	not finished in 3 days	4	104	2	N/A
MPEG4 dec. clustered (4 scenarios)	189415	47192700	2	16	2	34 %
MP3 dec.	over 700k	not finished in 3 days	9	416	3	N/A
MP3 dec. quantized	4370	162550	9	416	3	9%
WLAN	277	3796	3	10	8	10%
LTE	65	456	3	8	3	44%

a clustered model can deteriorate the accuracy of the analysis, in the end leading to suboptimal frequency settings. An approach to further limit the number of states distinguished by the state-space based technique from [89] is to quantize the execution time of the actors; but, timing quantization can also affect the accuracy of the model and as a result the outcome of the DVFS analysis.

Initially, we apply our technique and the technique from [89] to the original models of the benchmark applications. Results are shown in Table 6.2. Besides analysis times, the number of unique states identified for each of the benchmarks are reported when applying the state-space based DVFS. As our technique is a repetitive algorithm, the number of repetitions is reported for our technique. For our technique, the maximum number of parameters found in any of the critical cycles of the SADF model is also reported (ρ); the small values for ρ in our results show that the function *resolveCycle* can quickly find a suitable parameter point for critical cycles to make them fit within the throughput constraint while looking for an energy-efficient option. For the original model of the MPEG4 decoder and the MP3 decoder, the state-space based DVFS does not find solutions within a reasonable time (i.e., 3 days). The number of states of an application can drastically increase; hence, analyzing all states may not be a practical approach. Our technique finds solutions in a fraction of a second.

To make the state-space based technique applicable to the MPEG4 decoder, we perform scenario clustering; 9 scenarios in the original model are clustered into 4 separate scenarios. The results of applying our technique and the state-space based technique on the clustered model of the MPEG4 decoder are also shown in Table 6.2. For the MP3 decoder, the execution times of the actors are quantized to limit the state space. Our technique can be applied on both the MPEG4 decoder and MP3 decoder without any scenario clustering nor any timing quantization; the state-based technique only works on the coarser models. The results in Table 6.2 show that our technique is also on the coarser models much faster than the state-space based DVFS.

The memory required to store the DVFS controller devised by our technique depends on the number of scenarios in the SADF, while for the state-space based technique, the memory size depends on the number of discovered states. The results in Table 6.2 (the second column) show that more compact DVFS controllers are achievable by using our technique.

We also estimate the energy consumption of an SADF running on a platform with multiple processing elements. A long sequence of scenario iterations (i.e., 200k scenario iterations in our experiments) is fed to the DVFS controller devised by each of the two techniques. The energy consumption is calculated per iteration. Each experiment is performed 10 times with a different seed for the scenario sequence generator; the results reported in Table 6.2, last column, are averages of those 10 experiments. The results show that our technique offers solutions with less energy consumption compared to the technique from [89]. The reason why our technique offers lower energy consumption is because our algorithm considers critical cycles that may run across multiple iterations when assigning clock frequencies. In this way timing slack of one or several iterations can be effectively exploited through all iterations of a critical cycle. As a result,

operating frequencies of processing elements can get lowered.

Figure 6.6 gives some concrete energy results for the MPEG4 decoder. The figure confirms the energy savings obtained by our technique for the model with 4 scenarios. It also shows, however, that a more refined model with 9 scenarios allows a further reduction in energy consumption. Our technique scales better to finer-grained models than the state-based technique.

We also performed some experiments for processing elements with different numbers of the frequency points. By increasing the number of frequency points, we provide frequencies with higher resolutions. To assess the gain of dynamic switching, our technique is also used to find a static solution in which each processing element runs at a fixed frequency. Applying our technique to a parametric SADF in which one parameter (scale factor) is specified per processing element across all scenarios determines a static solution. Figure 6.7 depicts the results when our technique, the static approach and the technique from [89] are used to devise a DVFS controller for the WLAN application. Increasing the number of frequency points increases the number of states for the application which is why the technique from [89] is not capable of finding any solutions for the cases with more than two frequency points for the WLAN application; the analysis times are shown in Figure 6.7(a). Energy consumption values are shown in Figure 6.7(b); the values are normalized using the largest value. Our technique assures 10% to 41% lower energy consumption compared to the state-space based technique when the number of frequency points increases from 2 to 10. Increasing the number of frequency points provides more refined frequencies to save more energy. Figure 6.7(b) also shows that both DVFS techniques provide less energy consumption compared to the static technique. The analysis time of our technique rises by increasing the number of frequency points because this increases the number of parameter points to be verified in our algorithm. However, the analysis time of our technique is not too high to make the analysis infeasible. The static approach based on our technique is fast because the number of parameters in the parametric SADF is limited to only the number of processing elements. Our algorithm scales reasonably well because typically only a limited number of parameters are involved in critical timing cycles.

6.5.2 Results of Online Approach

In this subsection, we evaluate the effect of the online DVFS approach when it is combined with an off-line DVFS approach. We have used the initial frequencies generated using our off-line DVFS technique (see Section 6.3) and a static frequency assignment approach. The static frequencies are derived using our off-line DVFS technique when fixed frequencies are used for all processing elements across all scenarios (i.e., a single parameter representing the operating frequency of a processing element in all scenarios). We also simulated the effect of the online DVFS approach, explained in Section 6.4, on the total energy consumption. Table 6.3 contains the results for the MP3 decoder and the MPEG4 decoder applications. These two applications are used in our experiment since the distribution functions for the actual execution times of the actors in both models are known

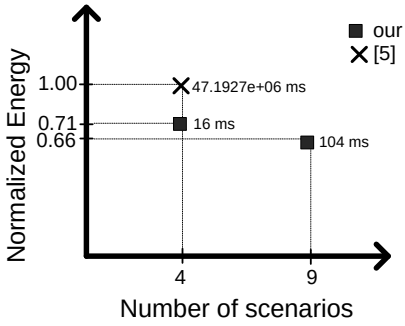
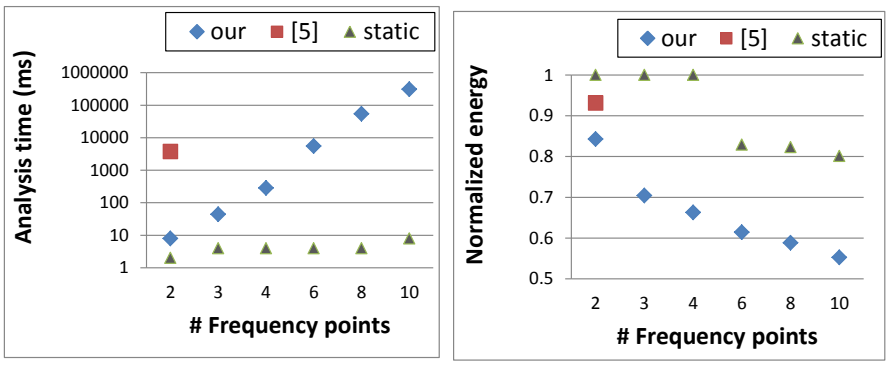


Figure 6.6: Impact of scenario clustering on energy consumption and analysis time.



(a) Analysis time. (b) Normalized energy.

Figure 6.7: Results of WLAN for different number of frequency points.

Table 6.3: The normalized energy consumption values for our/static with online/offline DVFS approach.

MP3 decoder		
	Our	**Static**
offline	0.868	1.000
offline+online	0.854	0.998

MPEG4 decoder		
	Our	**Static**
offline	0.596	1.000
offline+online	0.225	0.390

in the literature [32, 74]. We have used the existing distribution functions to generate the actual execution time of each actor in the model in our run-time simulator. The obtained energy consumption numbers are normalized using the highest energy consumption values (i.e., energy consumption of the static design-time DVFS technique which is an off-line approach). The results are the average of running each experiment 50 times with different seeds for generating actual actor execution times according to the related distribution functions.

As expected, using our design-time DVFS setting technique is beneficial on both benchmark applications. The online approach obtains more gain in case of the MPEG4 decoder application when compared to the MP3 decoder application. The actual and worst-case execution time values in case of MP3 decoder are fairly close. Hence, a design-time DVFS approach can find the suitable solution for the MP3 decoder application. For the MPEG4 decoder, the actual and worst-case actor execution times show larger differences, which is why adding online DVFS leads to larger gains.

6.6 Summary

A technique is developed to synthesize a DVFS controller for SADFs to reduce the energy consumption while meeting a throughput requirement. The SADF model is extended to a parametric model in order to capture the processor frequencies of the platform to which the application is mapped. The proposed technique uses a symbolic version of a Max-Plus automaton graph analysis to identify the critical timing cycles. Initially, the application is set to the lowest possible energy mode. Then, the critical cycles that are violating the timing requirement are repetitively resolved by refining the processor frequencies. Our analysis is faster than the state of the art technique for dataflow graphs; the experiments show that our technique furthermore constructs more compact DVFS controllers with lower energy consumption. Moreover, an online DVFS approach is considered to further lower the operating frequencies by exploiting the possible timing slack identified at run-time. The online approach obtains substantial energy saving when the actual and worst-case actor execution times show large differences.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

Dynamic behavior is prominent in modern streaming applications. These applications usually operate in different modes, which are called scenarios, throughout their run-time. SADF models can be used to capture the dynamic behavior of streaming applications. An SADF uses a scenario graph, which is an SDF graph, to model the behavior of an application operating in a specific scenario. Considering dynamism in the implementation of streaming applications can help to efficiently utilize the processing and memory resources. In this thesis, we have developed some modeling and analysis techniques that are in line with the existing successful strategies in developments of embedded systems (i.e., platform-based design, model-based design and predictable design). We have assumed that the suitable hardware platforms composed of multiple processing elements and memory units are given. Since the focus of this thesis is streaming applications, we have considered that the input dynamic applications are given with SADF graphs.

We have used one of the state of the art dataflow design tools (i.e., the SDF^3 toolkit) as our target design flow platform. The mapping of an application to an MPSoC platform is generated using the SDF^3 toolkit. In Chapter 3, we have developed a multi-processor scheduler that considers the presence of off-chip memories in the system. The proposed scheduler makes a trade-off between executing actors in a code-driven or data-driven manner to either minimize the latency or maximize the throughput of the application. The scheduler also considers prefetching when choosing a suitable execution order. The scheduler is evaluated using some realistic and synthetic graphs; our results show significant improvement over some well-known heuristics (i.e., HEFT and G-CPU).

The binding and scheduling decisions must be encoded in the model to enable the analysis of performance properties (e.g, throughput) or resource requirements (e.g., buffer sizes) under resource constraints (e.g., [9, 10, 54, 69, 86]). The only generally applicable technique to model schedules in an SADF graph is to convert its scenario graphs, represented by SDF graphs, to the equivalent HSDF

graphs. This may lead to an exponential increase in the graph size and/or to sub-optimal analysis results (e.g., overestimation in buffer sizes). In Chapter 4, we have proposed techniques to model two types of static-order schedules, i.e., periodic schedules and periodic single appearance schedules, directly in an S(A)DF graph. Experimentally, we show that our techniques produce more compact graphs compared to the technique that relies on a conversion to an HSDF. This results in reduced analysis times for performance properties and tighter resource requirements.

The timing property of a model may be altered during the different steps in a design flow; for example, Dynamic Voltage and Frequency Scaling (DVFS) scales the execution times. The SADF model can be parameterized by associating the execution time of each actor in the model with a linear function of some parameters. In Chapter 5, we have developed a throughput calculation technique for such parameterized SADF graphs. Our technique identifies a throughput function for a parameterized SADF; evaluating this function for a specific parameter point reveals the throughput value of the SADF model for that parameter point.

The SADF model can be parameterized to accommodate frequencies of the processing elements in an MPSoC platform. Applying our parametric throughput analysis to such a parameterized SADF identifies the critical timing cycle of the system. The extracted critical cycle reveals which processing elements are limiting the throughput; in Chapter 6, this information is utilized to assign suitable frequencies to each scenario of an SADF graph running on an MPSoC. The outcome of our DVFS technique is a design that minimizes the energy consumption while guaranteeing a certain throughput requirement. Moreover, the resulting DVFS settings are combined with an online DVFS approach to further exploit the timing slack produced at run-time.

All in all, this thesis expands the existing design flows for streaming applications modeled with SADF graphs. The modeling and analysis techniques proposed in this thesis are implemented within the SDF^3 toolkit that is freely available from `http://www.es.ele.tue.nl/sdf3`.

7.2 Recommendations for Future Research

This thesis provides initial modeling and analysis approaches towards achieving a design flow for dynamic streaming applications. However, there are still some opportunities in integration and optimization of the proposed techniques that invite for future research:

- **Guided partitioning, binding and buffer sizing:** The throughput function identified using the parametric throughput analysis presented in Chapter 5 reveals how actors of a model contribute to the critical timing cycles. Often, allocating more buffers to a channel is a way to break such critical cycles. Moreover, binding actors involved in the same critical cycle to different processing elements can also alter the critical cycle. These are some initial handles to tackle the partitioning, binding and buffer sizing problems.
- **Scalable scheduling:** The hybrid prefetch-aware scheduling technique pro-

posed in Chapter 3 considers prefetching of only one memory object during the execution of a single actor. This assumption is made in our approach to limit the size of the ILP formulation. An approach that can relax this issue can further utilize the prefetching efficiency.

- **Resource allocation:** The allocation of a processing element type (i.e, a micro-processor or DSP or specialized hardware etc.) to actors of a model can be modeled using a parameterized SADF. Verifying combinations of all different processing elements often demands large analysis time and this makes such resource allocation infeasible in practice. The results obtained from applying our parametric throughput analysis, presented in Chapter 5, to the parameterized model can be used to facilitate the mentioned resource allocation. The resource allocation can be guided by considering the effect of using a processing element type on the critical time cycles. This way, a better allocation may be performed with shorter analysis time.

- **Actor migration:** Actors running on a processing element may need to be migrated to another processing element for different reasons such as load-balancing, thermal management, reliability improvement, etc. The timing overhead of migrating an actor is usually high and it may deteriorate the achievable throughput. However, the timing slack accumulated at run-time of an application may allow an actor migration in some intervals without affecting the throughput. The migration process can be modeled as a scenario reconfiguration. The FSM of the application should be updated to accommodate the migration scenario. A design-time approach can study how many iterations are required to build enough time slack for an actor migration. Further research in this area can assess the possibility of a real-time actor migration for streaming applications.

- **Time-constrained DVFS using game theory:** A game theory-based technique to synthesize a controller for SADFs is presented in [86]. The authors of [86] optimize throughput by exploring different possible schedules. The controller resulting from [86] is optimized for a single design metric (i.e., only throughput). The same game-based approach can be adapted to reduce the energy consumption. Extending the game theory-based technique of [86] to consider two design metrics is a relevant, but challenging problem which requires further research.

- **Simultaneous design:** Different steps are involved in a design-flow such as partition, binding, scheduling, buffer sizing and other optimizations; the order with which these steps are applied affects the quality of the final design. Sometimes it is desirable to perform multiple steps simultaneously for a better design; however this can increase the complexity of the problem. For example, performing the DVFS optimization suggested in Chapter 6 at the same time that schedules are being formed can give a chance to find a better solution. Simultaneous design can be an interesting yet challenging topic for the future research.

Bibliography

[1] A. Alimonda, S. Carta, A. Acquaviva, A. Pisano, and L. Benini. A feedback-based approach to DVFS in data-flow applications. *IEEE Transactions on CAD of Integrated Circuits and Systems*, 28(11):1691–1704, 2009.

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

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

[4] 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.

[5] M. Barr. Embedded systems glossary, 2007. `http://www.barrgroup.com/embedded-systems/glossary`.

[6] M. Bekooij, R. Hoes, O. Moreira, P. Poplavko, M. Pastrnak, B. Mesman, J. 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.

[7] M. Benazouz, O. Marchetti, A. Munier-Kordon, and T. Michel. A new method for minimizing buffer sizes for cyclo-static dataflow graphs. In *8th IEEE Workshop on Embedded Systems for Real-Time Multimedia, ESTIMedia 10, Proceedings*, pages 11–20. IEEE, 2010.

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

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

[10] A. Bonfietti, M. Lombardi, M. Milano, and L. Benini. Throughput constraint for synchronous data flow graphs. In *6th International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, CPAIOR 09, Proceedings*, pages 26–40. Springer-Verlag, 2009.

[11] A. W. Brekling, M. R. Hansen, and J. Madsen. Models and formal verification of multiprocessor system-on-chips. *Journal of Logic and Algebraic Programming*, 77(1-2):1–19, 2008.

[12] T. Burd and R. Brodersen. Energy efficient CMOS microprocessor design. In *28th Hawaii International Conference on System Sciences, HICSS 95, Proceedings*, pages 288–297. IEEE, 1995.

[13] T. Burd, T. Pering, A. Stratakos, and R. Brodersen. A dynamic voltage scaled microprocessor system. *IEEE Journal of Solid-State Circuits*, 35:1571–1580, 2000.

[14] A. P. Chandrakasan, S. Sheng, and R. W. Brodersen. Low power CMOS digital design. *Journal of Solid State Circuits*, 27(4):473–484, April 1992.

[15] H. Chang and W. Sung. Access-pattern-aware on-chip memory allocation for simd processors. *IEEE Transactions on CAD of Integrated Circuits and Systems*, 28:158–163, 2009.

[16] F. Chen, T. W. O'Neil, and E. H.-M. Sha. Optimizing overall loop schedules using prefetching and partitioning. *IEEE Transactions on Parallel and Distributed Systems*, 11:604–614, 2000.

[17] J. Choi, H. Oh, S. Kim, and S. Ha. Executing synchronous dataflow graphs on a SPM-based multicore architecture. In *49th Design Automation Conference, DAC 12, Proceedings*, pages 664–671. ACM, 2012.

[18] P. Choudhury, P. P. Chakrabarti, and R. Kumar. Online dynamic voltage scaling using task graph mapping analysis for multiprocessors. In *20th International Conference on VLSI Design, VLSID 07, Proceedings*, pages 89–94. IEEE, 2007.

[19] M. Damavandpeyma, S. Stuijk, T. Basten, M. Geilen, and H. Corporaal. Schedule-extended synchronous dataflow graphs. *IEEE Transactions on CAD of Integrated Circuits and Systems*. to appear.

[20] M. Damavandpeyma, S. Stuijk, T. Basten, M. Geilen, and H. Corporaal. Hybrid code-data prefetch-aware multiprocessor task graph scheduling. In *14th Euromicro Conference On Digital System Design: Architectures, Methods and Tools, DSD 11, Proceedings*, pages 583 –590. IEEE, 2011.

[21] M. Damavandpeyma, S. Stuijk, T. Basten, M. Geilen, and H. Corporaal. Modeling static-order schedules in synchronous dataflow graphs. In *15th Conference on Design Automation and Test in Europe, DATE 12, Proceedings*, pages 775 –780. EDAA, 2012.

[22] M. Damavandpeyma, S. Stuijk, T. Basten, M. Geilen, and H. Corporaal. Throughput-constrained dvfs for scenario-aware dataflow graphs. In *19th Real-Time and Embedded Technology and Applications Symposium, RTAS 13, Proceedings*, pages 175–184. IEEE, 2013.

[23] M. Damavandpeyma, S. Stuijk, M. Geilen, T. Basten, and H. Corporaal. Parametric throughput analysis of scenario-aware dataflow graphs. In *30th International Conference of Computer Design, ICCD 12, Proceedings*, pages 219–226. IEEE, 2012.

[24] L. De-feng, P. Guo-teng, and X. Lun-guo. Multi-bank memory access scheduler and scalability. In *2nd International Conference on Computer Engineering and Technology, ICCET 10, Proceedings*, pages 723–727. IEEE, 2010.

[25] G. De Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, 1st edition, 1994.

[26] A. Ferrari and A. Sangiovanni-Vincentelli. System design: Traditional concepts and new paradigms. In *17th International Conference on Computer Design, ICCD 99, Proceedings*, pages 2–12. IEEE, 1999.

[27] M. Gallet, M. Jacquelin, and L. Marchal. Scheduling complex streaming applications on the cell processor. In *International Symposium on Parallel Distributed Processing, IPDPS 10, Proceedings*, pages 1–8. IEEE, 2010.

[28] M. Gallet, L. Marchal, and F. Vivien. Efficient scheduling of task graph collections on heterogeneous resources. In *International Symposium on Parallel Distributed Processing, IPDPS 09, Proceedings*, pages 1–11. IEEE, 2009.

[29] M. Geilen. Synchronous dataflow scenarios. *ACM Transactions on Embedded Computing Systems, TECS*, 10(2):16:1–16:31, Jan. 2011.

[30] M. 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.

[31] M. Geilen, J. Falk, C. Haubelt, T. Basten, B. Theelen, and S. Stuijk. Performance analysis of weakly-consistent scenario-aware dataflow graphs. Technical report, TU Eindhoven, 2011.

[32] M. Geilen and S. Stuijk. Worst-case performance analysis of synchronous dataflow scenarios. In *8th International Conference on Hardware-Software Codesign and System Synthesis, CODES+ISSS 10, Proceedings*, pages 125–134. ACM, 2010.

[33] A. Ghamarian, M. Geilen, T. Basten, and S. Stuijk. Parametric throughput analysis of synchronous data flow graphs. In *Conference on Design Automation and Test in Europe, DATE 08, Proceedings*, pages 116 –121. IEEE, 2008.

[34] A. Ghamarian, M. Geilen, S. Stuijk, T. Basten, A. Moonen, M. Bekooij, B. Theelen, and M. 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.

[35] S. V. Gheorghita, T. Basten, and H. Corporaal. Scenario selection and prediction for DVS-aware scheduling of multimedia applications. *J. Signal Process. Syst.*, 50(2):137–161, 2008.

[36] Z. Guz, E. Bolotin, I. Keidar, A. Kolodny, A. Mendelson, and U. Weiser. Many-core vs. many-thread machines: Stay away from the valley. *Computer Architecture Letters*, 8(1):25–28, 2009.

[37] E. C. Hall. *Journey To The Moon: The History Of The Apollo Guidance Computer*. American Institute of Aeronautics and Astronautics, 1996.

[38] A. Hansson. *A Composable and Predictable On-Chip Interconnect*. PhD thesis, TU Eindhoven, June 2009.

[39] K. Heloue, S. Onaissi, and F. Najm. Efficient block-based parameterized timing analysis covering all potentially critical paths. *IEEE Transactions on CAD of Integrated Circuits and Systems*, 31:472–484, 2012.

[40] R. Henia, A. Hamann, M. Jersak, R. Racu, K. Richter, and R. Ernst. System level performance analysis - the SymTA/S approach. *IEE Proccedings Computers and Digital Techniques*, 152(2):148–166, 2005.

[41] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 4th edition, 2007.

[42] W. Hongmei, S. Lei, Z. Tiejun, and W. Donghui. Dynamic management of scratchpad memory based on compiler driven approach. In *3rd IEEE International Conference on Computer Science and Information Technology, ICCSIT 10, Proceedings*, pages 668–672. IEEE, 2010.

[43] J. Hu and R. Marculescu. Energy- and performance-aware mapping for regular noc architectures. *IEEE Transactions on CAD of Integrated Circuits and Systems*, 24(4):551–562, 2005.

[44] ILOG CPLEX: High-performance software for mathematical programming and optimization, 1997-2008.

[45] International Data Corporation, IDC. `http://www.idc.com`.

[46] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the cell multiprocessor. *IBM Journal of Research and Development*, 49(4/5):589–604, July 2005.

[47] K. Keutzer, S. Malik, A. Newton, J. Rabaey, and A. Sangiovanni-Vincentelli. System-level design: Orthogonalization of concerns and platform-based design. *IEEE Transactions on CAD of Integrated Circuits and Systems*, 19(12):1523–1543, December 2000.

[48] W. Kim, M. Gupta, G.-Y. Wei, and D. Brooks. System level analysis of fast, per-core DVFS using on-chip switching regulators. In *14th International Symposium on High Performance Computer Architecture, HPCA 08, Proceedings*, pages 123 –134. IEEE, 2008.

[49] M.-Y. Ko, P. K. Murthy, and S. S. Bhattacharyya. Compact procedural implementation in DSP software synthesis through recursive graph decomposition. In *8th International Workshop on Software and Compilers for Embedded Processors, SCOPES 04, Proceedings*, pages 47–61. ACM, 2004.

[50] K. Kuchcinski. Constraint-driven scheduling and resource assignment. *ACM Transactions on Design Automation of Electronic Systems*, 8(3):355–383, July 2003.

[51] S. V. Kumar, C. V. Kashyap, and S. S. Sapatnekar. A framework for block-based timing sensitivity analysis. In *45th Design Automation Conference, DAC 08, Proceedings*, pages 688–693. ACM.

[52] S. Kwon, C. Lee, and S. Ha. Data-parallel code generation from synchronous dataflow specification of multimedia applications. In *5th IEEE Workshop on Embedded Systems for Real-Time Multimedia, ESTIMedia 07, Proceedings*, pages 91–96. IEEE, 2007.

[53] E. Lee and D. Messerschmitt. Synchronous data flow. *Proceeding of the IEEE*, 75(9):1235–1245, 1987.

[54] W. Liu, M. Yuan, X. He, Z. Gu, and X. Liu. Efficient SAT-based mapping and scheduling of homogeneous synchronous dataflow graphs for throughput optimization. In *29th Real-Time Systems Symposium, RTSS 08, Proceedings*, pages 492–504. IEEE, 2008.

[55] D. Martín-Sacristán, J. F. Monserrat, J. Cabrejas-Peñuelas, D. Calabuig, S. Garrigas, and N. Cardona. On the way towards fourth-generation mobile: 3GPP LTE and LTE-advanced. *EURASIP J. Wirel. Commun. Netw.*, 2009:4:1–4:10, Mar. 2009.

[56] M. Meijer, J. de Gyvez, and R. Otten. On-chip digital power supply control for system-on-chip applications. In *10th International Symposium on Low Power Electronics and Design, ISLPED 05, Proceedings*, pages 311 – 314. IEEE, 2005.

[57] A. Moonen, M. Bekooij, R. v. d. Berg, and J. v. Meerbergen. Practical and accurate throughput analysis with the cyclo static dataflow model. In *15th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, MASCOTS 07, Proceedings*, pages 238–245. IEEE, 2007.

[58] O. Moreira. *Temporal analysis and scheduling of hard real-time radios running on a multi-processor*. PhD thesis, TU Eindhoven, 2012.

[59] O. Moreira, T. Basten, M. Geilen, and S. Stuijk. Buffer sizing for rate-optimal single-rate data-flow scheduling revisited. *IEEE Transactions on Computers*, 59(2):188–201, 2010.

[60] P. K. Murthy, S. S. Bhattacharyya, and E. A. Lee. Joint minimization of code and data for synchronous dataflow programs. *Formal Methods in System Design*, 11(1):41–70, 1997.

[61] A. Nelson, O. Moreira, A. Molnos, S. Stuijk, B. T. Nguyen, and K. Goossens. Power minimisation for real-time dataflow applications. In *14th Euromicro Conference On Digital System Design: Architectures, Methods and Tools, DSD 11, Proceedings*, pages 117–124. IEEE, 2011.

[62] H. Oh and S. Ha. Fractional rate dataflow model for efficient code synthesis. *Journal of VLSI Signal Processing*, 37(1):41–51, May 2004.

[63] 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.

[64] 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.

[65] I. Sander and A. Jantsch. System modeling and transformational design refinement in forsyde. *IEEE Transactions on CAD of Integrated Circuits and Systems*, 23:17–32, 2004.

[66] D. Shin and J. Kim. Power-aware scheduling of conditional task graphs in real-time multiprocessor systems. In *8th International Symposium on Low Power Electronics and Design, ISLPED 03, Proceedings*, pages 408–413. IEEE, 2003.

[67] S. Sriram and S. Bhattacharyya. *Embedded Multiprocessors: Scheduling and Synchronization, Second Edition.* CRC Press, 2009.

[68] S. Stuijk. *Predictable Mapping of Streaming Applications on Multiprocessors.* PhD thesis, TU Eindhoven, 2007.

[69] S. Stuijk, T. Basten, M. 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.

[70] S. Stuijk, M. 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.

[71] S. Stuijk, M. 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.

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

[73] S. Stuijk, M. Geilen, B. D. Theelen, and T. Basten. Scenario-aware dataflow: Modeling, analysis and implementation of dynamic applications. In *International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation, IC-SAMOS 11, Proceedings*, pages 404–411. IEEE, 2011.

[74] B. Theelen, M. Geilen, T. Basten, J. Voeten, S. 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.

[75] B. Theelen, M. Geilen, S. Stuijk, S. Gheorghita, T. Basten, J. P. M. Voeten, and A. Ghamarian. Scenario-aware dataflow. Technical Report ESR-2008-08, TU Eindhoven, 2008.

[76] D. Thiele and R. Ernst. Optimizing performance analysis for synchronous dataflow graphs with shared resources. In *15th Conference on Design Automation and Test in Europe, DATE 12, Proceedings*, pages 635–640. EDAA, 2012.

[77] L. Thiele, S. Chakraborty, and M. Naedele. Real-time calculus for scheduling hard real-time systems. In *International Symposium on Circuits and Systems, ISCAS 00, Proceedings*, pages 101–104. IEEE, 2000.

[78] H. Topcuoglu, S. Hariri, and M. Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Transactions on Parallel and Distributed Systems*, 13:260–274, 2002.

[79] P. Vivet, E. Beigne, H. Lebreton, and N.-E. Zergainoh. On line power optimization of data flow multi-core architecture based on vdd-hopping for local DVFS. In *20th international conference on Integrated circuit and system design: power and timing modeling, optimization and simulation, PATMOS 10, Proceedings*, pages 94–104. Springer-Verlag, 2010.

[80] A.-P. Wang, J. Hahn, M. Roumi, and P. H. Chou. Buffer optimization and dispatching scheme for embedded systems with behavioral transparency. *ACM Transactions on Design Automation of Electronic Systems*, 17(4):41:1–41:26, 2012.

[81] 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.

[82] M. H. Wiggers, M. J. Bekooij, and G. J. Smit. Monotonicity and run-time scheduling. In *9th International Conference on Embedded Software, EMSOFT 09, Proceedings*, pages 177–186. ACM, 2009.

[83] H. H. Wu, C. ching Shen, N. Sane, W. Plishker, and S. S. Bhattacharyya. A model-based schedule representation for heterogeneous mapping of dataflow graphs. In *20th International Heterogeneity in Computing Workshop, HCW 11, Proceedings*, pages 66–77. IEEE, 2011.

[84] H. Yang and S. Ha. Pipelined data parallel task mapping/scheduling technique for MPSoC. In *12th Conference on Design Automation and Test in Europe, DATE 09, Proceedings*, pages 69–74. IEEE, 2009.

[85] Y. Yang. *Exploring Resource/Performance Trade-offs for Streaming Applications on Embedded Multiprocessors.* PhD thesis, TU Eindhoven, 2012.

[86] Y. Yang, M. Geilen, T. Basten, S. Stuijk, and H. Corporaal. Automated bottleneck-driven design-space exploration of media processing systems. In *13th Conference on Design Automation and Test in Europe, DATE 10, Proceedings*, pages 1041–1046. European Design and Automation Association, 2010.

[87] N. Young, R. Tarjan, and J. Orlin. Faster parametric shortest path and minimum-balance algorithms. *Networks*, 21(2):205–221, October 1991.

[88] L. Zhang, M. Qiu, W. C. Tseng, and E. H. Sha. Variable partitioning and scheduling for MPSoC with virtually shared scratch pad memory. *Journal of Signal Processing Systems*, 58:247–265, 2010.

[89] J. Zimmermann, O. Bringmann, and W. Rosenstiel. Analysis of multi-domain scenarios for optimized dynamic power management strategies. In *15th Conference on Design Automation and Test in Europe, DATE 12, Proceedings*, pages 862–865. EDAA, 2012.

Glossary

Acronyms and abbreviations

CCR	communication-to-computation ratio
DAG	directed acyclic graph
DMA	direct memory access
DSE	design space exploration
DSM	decision state modeling
DSP	digital signal processing
DVFS	dynamic voltage and frequency scaling
FPGA	field-programmable gate array
FSM	finite state machine
GCD	greatest common divisor
G-CPU	greedy CPU
GPS	global positioning system
HEFT	heterogeneous earliest finish time
HSDF	homogeneous synchronous dataflow
IDC	international data corporation
IDCT	inverse discrete cosine transform
ILP	integer linear programming
IP	intellectual property
IQ	inverse quantization
Lfp	list schedule uses forward priorities
LMA	left-most actor
Lrp	list schedule uses reverse priorities
LS	looped schedule
LTE	long term evolution
MC	motion compensation
MCM	maximum cycle mean
MCR	maximum cycle ratio
MoC	model-of-computation
MPAG	Max-Plus automaton graph
MPEG	motion pictures experts group
MPSoC	multi-processor system-on-chip

NRE	non-recurring engineering
PSOS	periodic static-order schedule
QCIF	quarter common intermediate format
RMA	right-most actor
SADF	scenario-aware dataflow
SAS	single appearance schedule
SASM	single appearance schedule modeling
SDF	synchronous dataflow
VLD	variable length decoding
VLSI	very large scale integration
WCET	worst-case execution time
WLAN	wireless LAN

Symbols and Notations

Chapter 2

a	actor
c	channel
$port$	port
A	set of actors
C	set of channels
$Ports$	set of ports
In	set of input ports
Out	set of output ports
$Rate(p)$	rate of port p
$InC(a)$	set of all channels connected to input ports of actor a
$OutC(a)$	set of all channels connected to output ports of actor a
γ	repetition vector
ω_i	an SDF state
$\omega_0(c)$	number of initial tokens on channel c
σ	an SDF execution
ϕ	a scenario of an SADF graph
Φ	set of all scenarios in an SADF graph
$Prgs$	the progress value of a scenario
FSM	the FSM of an SADF
q	an FSM state
Q	set of all states in an FSM
Q_0	set of all potential starting states in an FSM
T	set of all possible state transitions in an FSM
L	state-labeling function of an FSM
M	the characteristic max-plus matrix of an SDF
θ	token timestamp
gf	grouping factor

Chapter 3

pt	processing tile in a platform graph
pe	processing element of a processing tile
$[\texttt{mc of } pt_i]$	the code memory size of the processing tile pt_i (in bytes)
$[\texttt{md of } pt_i]$	the data memory size of the processing tile pt_i (in bytes)
H	the read/write latency of the remote memory (in cycles)
$\eta(a)$	the processing tile to which the actor a is mapped
$\kappa(a)$	the code size of the actor a (in bytes)
$\tau(a)$	the execution time of the actor a (in cycles)
$\psi(a)$	the function identifier of the actor a
$\epsilon(c)$	the communication delay of transferring a data object via channel c (in cycles)
$\vartheta(c)$	the size of a token (in bytes) getting transferred via c
χ_i	start time of the actor a_i
LAT	latency
π_{ij}	receives 1 when actor a_j scheduled immediately after actor a_i; otherwise, receives 0
IT	initialization time
WD	write data
RD	read data
RC	read code
II	iteration interval
CSL	code space limitation
DSL	data space limitation
CPA	code prefetched amount
DPA	data prefetched amount
TL	temporal limitation
DTL	data temporal limitation

Chapter 4

Ω	set of all decision states
Δ_j	set of the opponent actors of a decision state ω_j
μ	normalization factor

Chapter 5

p	parameter
P	set containing all parameters
I	an interval to which parameter $p_i \in P$ belongs
ν	a parameter point
e	a throughput expression

Chapter 6

SFS	scale factor set
$Period$	period or the inverse of the throughput
f	frequency
ρ	the number of parameters found in a critical cycle

Acknowledgments

I am glad to leave this thesis as a result of my 22 years of student life. This achievement would not have been possible without the support and guidance that I have received from many people in many ways. Here at the end of my thesis, I would like to express my great gratitude to all those people who have contributed to my study and have made it a joyful and an unforgettable experience for me.

First of all, I would like to thank my first promoter Henk Corporaal for providing me the research opportunity within the Electronics System group at TU Eindhoven. His broad experience has helped me in considering different angles and aspects of my work.

My most sincere gratitude goes to my second promoter Twan Basten. His expertise in the field of dataflow analysis always guided me in the right research direction to achieve the finest results in the lowest possible time. He has always supported me during my PhD study and I owe the success of accomplishing this thesis to all discussions and meetings that I had with him. Twan will remain my role-model in my future research career.

I would also like to thank my co-promoter and daily supervisor Sander Stuijk. I learned a lot from Sander during my weekly meetings with him. The brainstorming discussions with him always challenged and triggered my mind in developing and forming novel ideas and approaches in the dataflow field. The other person who had a great impact on my PhD work was Marc Geilen. His critical way of thinking, insight and knowledge have directed my work towards flawless outcomes.

The members of my thesis committee are gratefully acknowledged for reading the thesis, providing useful comments and being present in my defense session. It is my privilege to have Prof. Christian Haubelt, Prof. Marco Bekooij, Prof. Kees van Berkel and Dr. Andy Pimentel in my thesis committee. Also Ton Backx is thanked for being the chairman of the PhD committee.

It was always a great pleasure to relax on the sofa in our coffee corner and have friendly talks with people from different cultures and nationalities. I take this opportunity to thank the members of the Electronic Systems group who have made my stay on the ninth floor of the PT building a pleasant experience. I highly thank our former group leader Ralph Otten who carefully took the responsibilities of managing our group. I express my best thanks to Rehan Afzal, Benny Akesson, Francesco Comaschi, Rosilde Corvino, Erkan Diken, Shakith Fernando, Raymond Frijns, Davit Mirzoyan, Sebastian Moreno Londono, Majid Nabi, Alexandre Nery, Maurice Peemen, HamidReza Pourshaghaghi, Dongrui She, Firew Siyoum, Marcel Steine, Gert-Jan van den Braak, Yang Yang, and all other members of the Electronic Systems group for being so friendly, and organizing enjoyable meetings, social events, and coffee breaks. I would also like to express my warmest thanks to the secretaries of the group, Marja de Mol and Rian van Gaalen for being patient in assisting me in filling out different administrative forms and requests. I specially thank Rian for helping us in improving our Dutch language skills. I would also like to express my gratitude to all my Iranian friends in the Netherlands for all unforgettable moments, trips, feasts, and parties.

Last, but by no means least, I devote my heartfelt thanks to my mom, dad, and sister for always believing in me and encouraging me to pursue my dreams. My hard-working parents have sacrificed their lives for my sister and myself and provided unconditional love and care. I would not have made it this far without them. *I dedicate this thesis to them, with love and admiration.*

Morteza Damavandpeyma
October 2013

Curriculum Vitae

Morteza Damavandpeyma was born in Iran on March 28, 1984. In 2002, he graduated from highschool in mathematics and physics. He received his Bachelor and Master degrees in Computer Hardware Engineering from the University of Tehran, Iran in 2006 and 2008, respectively. He was also working at the research and development group of Kavoshcom Asia Co from 2006 till 2009; during this period, he mainly contributed to the design and development of a digital baseband receiver for the global positioning system (GPS).

Since September 2009, he has been a Ph.D. student within the Electronic Systems group at the Electrical Engineering Department of Technische Universiteit Eindhoven (TU/e) in Eindhoven, the Netherlands. His research was funded by STW (de Stichting voor de Technische Wetenschappen) within project NEST: Netherlands Streaming. His research has led amongst others to several publications and this thesis.

List of Publications

The following is the list of publications of Morteza Damavandpeyma in the field of real-time embedded systems and multi-processor design, when affiliated by Technische Universiteit Eindhoven. A part of these publications is covered in this thesis.

Publications Covered in the Thesis

- **M. Damavandpeyma**, S. Stuijk, T. Basten, M. Geilen and H. Corporaal. Schedule-Extended Synchronous Dataflow Graphs, in *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, TCAD*, IEEE, (to be published).

- **M. Damavandpeyma**, S. Stuijk, T. Basten, M. Geilen and H. Corporaal. Throughput-Constrained DVFS for Scenario-Aware Dataflow Graphs, in 19^{th} *Real-Time and Embedded Technology and Applications Symposium, RTAS '13, IEEE, Proceedings*, p.p. 175–184, IEEE, 2013.

- **M. Damavandpeyma**, S. Stuijk, M. Geilen T. Basten, and H. Corporaal. Parametric Throughput Analysis of Scenario-Aware Dataflow Graphs, in 30^{th} *International Conference on Computer Design, ICCD '12, IEEE, Proceedings*, p.p. 219–226, IEEE, 2012.

- **M. Damavandpeyma**, S. Stuijk, T. Basten, M. Geilen and H. Corporaal. Modeling Static-Order Schedules in Synchronous Dataflow Graphs, in 15^{th} *Conference on Design, Automation and Test in Europe, DATE '12, ACM, Proceedings*, p.p. 775–780, IEEE, 2012.

- **M. Damavandpeyma**, S. Stuijk, T. Basten, M. Geilen and H. Corporaal. Hybrid Code-Data Prefetch-Aware Multiprocessor Task Graph Scheduling, in 14^{th} *Euromicro Conference on Digital System Design, DSD '11, IEEE, Proceedings*, p.p. 583–590, IEEE, 2011.

Publications not Covered in the Thesis

- **M. Damavandpeyma**, S. Stuijk, T. Basten, M. Geilen and H. Corporaal. Thermal-Aware Scratchpad Memory Design and Allocation, in 28^{th} *International Conference on Computer Design, ICCD '10, IEEE, Proceedings*, p.p. 118–124, IEEE, 2010.