Adaptive Streaming Applications: Analysis and Implementation Models

Jiali Teddy Zhai

Adaptive Streaming Applications: Analysis and Implementation Models

PROEFSCHRIFT

ter verkrijging van
de graad van Doctor aan de Universiteit Leiden,
op gezag van Rector Magnificus prof.mr. C.J.J.M. Stolker,
volgens besluit van het College voor Promoties
te verdedigen op woensdag 13 mei 2015
klokke 10:00 uur

door

Jiali Teddy Zhai
geboren in 1982

Samenstelling promotiecommissie:

Promotor:	Prof. Dr. Ir. Ed F.A. Deprettere	Universiteit Leiden
Co-Promotor:	Dr. Todor P. Stefanov	Universiteit Leiden
Overige leden:	Prof. Dr.-Ing. Jügen Teich	Universität Erlangen-Nürnberg
	Dr. Ingo Sander	KTH Kungliga Tekniska högskolan
	Prof. Dr. Ir. Twan A.A. Basten	Technische Universiteit Eindhoven
	Prof. Dr. Joost N. Kok	Universiteit Leiden
	Prof. Dr. Farhad Arbab	Universiteit Leiden
	Prof. Dr. Harry A.G. Wijshoff	Universiteit Leiden

Adaptive Streaming Applications: Analysis and Implementation Models
Jiali Teddy Zhai. -
Dissertation Universiteit Leiden. - With ref. - With summary in Dutch.

Copyright © 2015 by Jiali Teddy Zhai. All rights reserved.

This dissertation was typeset using LaTeX in Linux.
Cover designed by Shanshan Yang

Printed in the Netherlands.

Contents

Table of Contents v

List of Figures ix

List of Tables xiii

1 Introduction 1
- 1.1 Embedded Streaming System Design 2
  - 1.1.1 Design Requirements 3
  - 1.1.2 Application Characterization 4
  - 1.1.3 Platform Implications 7
  - 1.1.4 Model-based Design Methodology 9
  - 1.1.5 Summary 14
- 1.2 Problem Statement 15
- 1.3 Research Contributions 17
- 1.4 Thesis Organization 19

2 Models-of-Computation (MoC) 21
- 2.1 Polyhedral Process Networks (PPN) 21
- 2.2 Actor-based Data Flow MoCs 29
  - 2.2.1 Synchronous Data Flow (SDF) 30
  - 2.2.2 Cyclo-Static Data Flow (CSDF) 32
- 2.3 Hard Real Time Scheduling of Acyclic (C)SDF Graphs 33

3 Automated Analysis Model Construction: Deriving CSDF from Equivalent PPN 39
- 3.1 The Algorithm 40
- 3.2 Experimental Results 47

4 **Exploiting Maximum Data-level Parallelism without Inter-processor Communication** **51**
- 4.1 Motivating Example 53
- 4.2 Related Work 57
- 4.3 Finding all Dependences in a PPN 58
- 4.4 Computing the Number of Communication-free Partitions 60
- 4.5 Communication-free Partitioning Algorithm 64
- 4.6 Experimental Results 69

5 **Exploiting Just-enough Parallelism in Hard Real-time Systems** **75**
- 5.1 Related Work 77
- 5.2 Unfolding of SDF Graphs 78
- 5.3 Problem Formulation 80
- 5.4 Period Scaling under Hard Real-time Scheduling 83
- 5.5 Bounding Solution Space 83
- 5.6 The Algorithm 87
- 5.7 Experimental Evaluation 92

6 **A New MoC for Modeling Adaptive Streaming Applications** **99**
- 6.1 Related Work 100
- 6.2 Model Definition 101
  - 6.2.1 Parameterized Polyhedral Process Networks 101
  - 6.2.2 Operational Semantics 104
- 6.3 Consistency 107
- 6.4 Experimental Results 111

7 **Hard Real-time Scheduling of Adaptive Streaming Applications** **115**
- 7.1 Related Work 117
- 7.2 Model Definition 118
  - 7.2.1 Mode-Aware Data Flow (MADF) 118
  - 7.2.2 Operational Semantics 123
  - 7.2.3 Mode Transition 125
- 7.3 Hard real-time Scheduling of MADF 130
- 7.4 Case Study 139

8 **Summary and Outlook** **145**

Bibliography **149**

Samenvatting **163**

List of Publications **169**

Index **171**

List of Abbreviations **173**

List of Figures

1.1 Three examples of embedded streaming applications. 2
1.2 Block diagram of a H.264 decoder (taken from [18]). Each task is represented by a rectangular block. 5
1.3 Processing part of an MPSoC platform for mobile devices from Nvidia (taken and simplified from [11]). 8
1.4 Template of a baseband processor (taken and simplified from [103]). All interfaces and peripheries are omitted. 8
1.5 X-chart: a general design process (adopted from [50]). 9
1.6 Comparison of dataflow MoCs for streaming applications (taken and extended from [112]). The MoCs underlined are proposed in this thesis. 10
1.7 Daedalus^{RT} design flow. The grey boxes highlight the contributions of this thesis. The dashed box and lines denote the parts that are currently not fully implemented. 13

2.1 A polyhedron. 23
2.2 The polyhedral representation of the execution of function `read_image` in Listing 1. 29
2.3 PPN corresponding to the SANLP in Listing 1. 29
2.4 An example of an image filter algorithm modeled using the SDF MoC. 30
2.5 A CSDF graph G_1 of a pacemaker application (taken from [99]). . . . 33
2.6 An example of a CSDF graph and its real-time task-set representation. Since the execution of the actors repeats indefinitely, the last execution of $A_{2,1}$,$A_{3,1}$,$A_{3,2}$, and $A_{3,3}$ in the figure is truncated and shown in black. 36

3.1 CSDF graph equivalent to the PPN shown in Figure 2.3. 41
3.2 Domains of process snk in Figure 2.3. 42

3.3 Suffix tree for the sequence of process variants S_{snk}. The tildes represent the omitted part of the tree. 46

4.1 An example of a PPN and its communication-free partitions. 54
4.2 Mapping of the PPN in Figure 4.1(a) onto 2 PEs achieving the maximum performance. 55
4.3 Performance results of mapping the initial PPN and the alternative PPN after communication-free partitioning. 55
4.4 The PPN in Figure 4.1(a) after communication-free partitioning and its mapping. 56
4.5 Domain of PPN process P_3 in Figure 4.1(a). The input port domain of IP_3 (surrounded by the solid triangle), output port domain of OP_3 (surrounded by the dotted triangle), and dependence relation R_3 (denoted by the arrows between dots). 59
4.6 Finding transitive dependences of the PPN. 61
4.7 The PPN in Figure 4.1(a) after communication-free partitioning. . . . 66
4.8 Performance results of mapping the MJPEG encoder onto (a) FPGA-based MPSoC platforms and onto (b) a desktop multi-core platform. 70
4.9 Performance results of mapping the FM radio application onto (a) FPGA-based MPSoC platforms and onto (b) a desktop multi-core platform. 71

5.1 (a) An example of an SDF graph and (b) its equivalent CSDF graph by unfolding actor A_3 by factor 3. 80
5.2 G_3: Optimal alternative graph of G_1 in Figure 5.1(a) with unfolding factors $f_2 = 2, f_3 = 4$ when scheduled on 2 PEs. 85
5.3 The list produced by the algorithm for G_1 in Figure 5.1(a) on 2 PEs with $\rho = 0.95$. 91
5.4 Period ratio (lower is better). 94
5.5 Running time of our algorithm. 95
5.6 An example of an individual. The first replica of A_1 is allocated on the jth PE and the $\hat{f}_1$th replica of A_1 does not exist. 96
5.7 The ratios of total execution time Ω_t and total code size Ω_Θ for GA and our algorithm. 98

6.1 Comparsion between a PPN and a P^3N. 102
6.2 Control process and evaluation function. 106
6.3 Consistent execution of process P_2 and P_3 w.r.t. edge E_3. 108
6.4 Which combinations (M,N) do ensure consistency of P^3N? 108
6.5 Two alternatives of Function `Check` in Figure 6.2(b). 111

6.6 P^3N of our experiment 112
6.7 Performance results of PPN and P^3N implementations 113

7.1 An example of MADF graph (G_1). 119
7.2 Two modes of the MADF graph in Figure 7.1. 123
7.3 Execution of two iterations of both modes SI^1 and SI^2 under self-timed scheduling. 125
7.4 An execution of G_1 in Figure 7.1 with two mode transitions under the ST transition protocol. *MCR1* at time t_{MCR1} denotes a transition request from mode SI^2 to SI^1, and *MCR2* at time t_{MCR2} denotes a transition request from mode SI^1 to SI^2. 127
7.5 An illustration of the Maximum-Overlap Offset (MOO) calculation. 128
7.6 The execution of G_1 with two mode transitions under Maximum-Overlap Offset (MOO) protocol. 129
7.7 Upper bounds of earliest starting times for transition from mode SI^2 to SI^1. 133
7.8 Earliest starting times for transition from mode SI^2 to SI^1 with the MOO protocol. 134
7.9 Allocation of all MADF actors in Figure 7.1 to 3 PEs. 136
7.10 Earliest starting times for transition SI^2 to SI^1 on 2 PEs shown in Figure 7.9. 137
7.11 MADF graph of Vocoder. 141
7.12 Allocation of dataflow actors of Vocoder to 4 PEs. The control edges are omitted to avoid cluttering. 143

List of Tables

1.1 Processing requirements for video decoding (taken from [19]). 3
1.2 Processing requirements for image processing (taken from [43]). All applications are assumed to operate at 24 FpsFrame per second. 3

2.1 Mathematical notations. 22
2.2 Polyhedral notations. 23
2.3 Data flow notations. 31
2.4 Notations for HRT scheduling of CSDF MoCs. 34

3.1 Additional notations used in Chapter 3 besides the ones introduced in Chapter 2. 39
3.2 Consumption/production sequences for actor snk in Figure 3.1. . . . 47
3.3 Characteristics of benchmarks and running times to derive their corresponding CSDF graphs. 48
3.4 Execution times of the phases in the Daedalus^{RT} flow for three streaming applications on a single MPSoC platform. 48

4.1 Execution time on benchmarks. 73

5.1 Additional notations used in Chapter 5 besides the ones introduced in Chapter 2. 81
5.2 Benchmark characteristics. 93
5.3 Parameters for the genetic algorithm. 97

7.1 Additional notations used in Chapter 7 besides the ones introduced in Chapter 2. 116
7.2 Mapping relation M_2 for actor A_2 in Figure 7.1. 120
7.3 Function MC_5 defined for actor A_5 in Figure 7.1. 120
7.4 Actor parameter for G_1 in Figure 7.1. 132
7.5 WCETs of all actors in Vocoder (in clk.). 142

7.6 Performance results of four modes of Vocoder in the steady-state. . . 143
7.7 Performance results for all mode transitions of Vocoder. 144

Chapter 1

Introduction

EMBEDDED systems are an essential part of our lives and exist in a wide variety. In 2010, more than 15 billion embedded systems were sold globally [5]. The market for embedded systems was $113 billion [17]. This market has exhibited steady growth at a compound annual growth rate of 7% for the past 5 years.

An embedded system [84] is an information processing system embedded into devices, products or other systems, for instance mechanical or electrical systems. Different from servers or desktop PCs, embedded systems are often application domain specific and perform certain specific functions tightly coupled to their environment. Such systems can be hidden inside small and simple entities such as digital watches and traffic lights. They can be also a part of large and complex systems, such as Mars Exploration Rover [9].

Embedded streaming systems are an important class of embedded systems, which are specifically designed to process *streaming applications*. A streaming application [59] is a software program that processes large volume of continuous data streams in short periods of time. Typically, the same operation is performed on large set of data items in the stream. Therefore, there is little control flow between processing different data items. Each data item has short life time and is discarded after being processed. This type of applications is ubiquitous in telecommunications, health-care, transportation, retail, science, security, emergency response, and finance. This thesis focuses on those streaming applications that are commonly used in embedded systems. Figure 1.1 shows three popular streaming applications widely used in our daily lives on mobile phones.

(a) Navigation. (b) Gaming. (c) Video decoding.

Figure 1.1: Three examples of embedded streaming applications.

1.1 Embedded Streaming System Design

Designing embedded streaming systems is definitely a complex process. It involves three main aspects illustrated in the Y-chart [67], namely target applications, underlying platforms, and used design methodologies. This thesis primarily deals with the aspect of design methodology by proposing several novel techniques and a highly automated design framework. The proposed design techniques are highly optimized towards the target applications and platforms with important design requirements in mind. Therefore, first presenting the desired requirements, target applications, and platforms helps understand better the context and contributions of this thesis.

We first show in Section 1.1.1 that the design requirements for embedded streaming systems in general are more strict than general-purpose computing systems. The design techniques proposed in this thesis focus on an important subset of requirements, namely high throughout and hard real-time guarantees. Fortunately, properties of streaming applications can be well exploited with appropriate design techniques, such that the design requirements are satisfied. In addition, application properties show that modern streaming applications exhibit adaptive behavior that has to be explicitly captured in the design methodology. This is one major topic of this thesis. Therefore, we characterize the target applications in Section 1.1.2. Afterwards, we discuss the state-of-the-art hardware platforms in Section 1.1.3 and aim at understanding the architecture capabilities. Ideally, the proper design techniques should exploit application properties in such a way that matches exactly the underlying architecture capabilities. This reinforces the contributions of this thesis. Based on the application properties and selected architecture, we given an overview of a widely acknowledged design methodology in Section 1.1.4. The techniques developed in this thesis significantly extend and strengthen this design methodology.

Application	Resolution	Frame rate	Uncompressed bit rate	Compressed bit rate
HD-DVD	1920x1080	25	607 Mbps	8-20 Mbps
HDTV	1280x720	25	607 Mbps	2-8 Mbps
DVD	720x576	25	121 Mbps	1-2 Mbps
Video conferencing	352x288	25	30 Mbps	128-1000 Kbps
Mobile video	176x144	15	9 Mbps	50-1000 Kbps

Table 1.1: Processing requirements for video decoding (taken from [19]).

Application	Resolution	Uncompressed bit rate	Compressed bit rate
Projection Electronic Cinema	1280x720	350 Mbps	17.5 Mbps
Production HDTV	1920x1080	995 Mbps	140 Mbps
Projection Digital Cinema	4096x2048	6040 Mbps	450 Mbps
Production Digital Cinema	4096x3112	11000 Gbps	2200 Mbps

Table 1.2: Processing requirements for image processing (taken from [43]). All applications are assumed to operate at 24 FpsFrame per second.

1.1.1 Design Requirements

Requirements referred in this section are non-functional ones, such as performance, timing predictability, thermal aspects [60], security [69], and reliability [128]. The functional requirements such as deadlock-free execution are implicit. This thesis addresses the requirements of high performance and timing predictability.

Embedded streaming systems are expected to have high performance. Sometimes high performance is used interchangeably with high throughput. System throughput is a performance metric which denotes the average number of output data produced by the system per time unit. In general, a system with high throughput is referred to be *fast*. A Digital Video Broadcasting-Handheld (DVB-H) receiver found in mobile devices is a typical embedded streaming system with a certain throughput requirement. Unable to satisfy the throughput requirement results in the videos in slow motion and greatly degrades the user experience. In the video processing domain, the requirements of processing power has also increased drastically as screen resolution increases. Table 1.1 shows processing requirements for different resolutions. The state-of-the-art mobile phones, such as Samsung S4 [12], already have screens with the HD-DVD resolution. From the $5th$ column in Table 1.1, it should be clear that designing embedded video streaming system that satisfies the HD-DVD resolution poses a huge challenge. Image processing applications also require high throughput. Table 1.2 shows the processing requirements of different image processing applications used for digital and electronic cinema. The extreme high data rates clearly exceed the processing capacity of conventional embedded streaming systems. For the wireless communication, the requirements have significantly evolved over generations. The 3G standard targets 2 Mbps multimedia service including voice, video, and wireless Internet access. In contrast, it has been proposed in the 4G standard to increase the bandwidth of 100 Mbps or even 1 Gbps.

Besides high performance requirements, many embedded streaming systems pose *hard real-time* (HRT) requirements. In a HRT system [33], each application in the system has a deadline to indicate the maximum time within which the application must complete its execution. Missing any deadline may cause catastrophic consequence on the system. As noted in [33], a HRT system does not necessarily need to have high throughput requirements. Instead, the timing predictability is the major concern of the HRT system. That is, e.g., if a video conferencing system is guaranteed to produce a decoded video within 1 hour, this system still can be called a HRT system. Of course, this guarantee may not be useful in practice because the latency of producing an output is beyond being acceptable. For a realistic embedded streaming system, HRT constraints often come together with high throughput requirements. For instance, a collision avoidance system in the automotive or avionics domain is such an example. Processing input frames must be completed within a tight deadline. Missing the deadline will lead to catastrophic consequence for the vehicles, for instance potential collision to obstacles. At the same time, it has been reported in [6] that these algorithms require approximately 170 million calculations for each frame update, with the expectation of being executed on up to 64 processors.

1.1.2 Application Characterization

Although the requirements presented in Section 1.1.1 seem strict, streaming applications often contain ample amount of parallelism which can be exploited to satisfy the requirements. Therefore, a characterization of the application properties is needed, which heavily influences and motivates the solutions proposed in this thesis. In this section, streaming applications from different domains are characterized in terms of availability of parallelism and its different forms, computation and data communication characteristics, and adaptive behavior. The selected application domains contain those that are commonly used in embedded system, including video processing, wireless communication, and image processing/computer vision. Below we start by defining different forms of parallelism.

The type of parallelism is often categorized into three forms as follows:

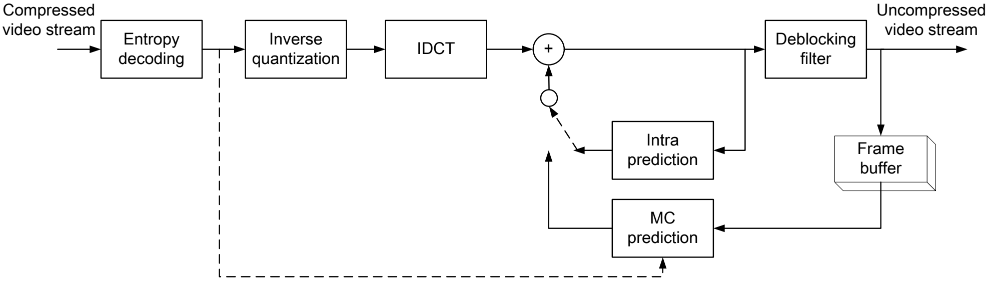


Figure 1.2: Block diagram of a H.264 decoder (taken from [18]). Each task is represented by a rectangular block.

1. Task-Level Parallelism (TLP): TLP refers to running different tasks of an application concurrently.
2. Data-Level Parallelism (DLP): DLP refers to running the same set of operations on multiple datum simultaneously.
3. Pipeline-Level Parallelism (PLP): PLP refers to running different iterations of a pair of producer and consumer tasks simultaneously.

In literature, TLP is often referred as thread level parallelism [58]. For instance, the block diagram of an H.264 decoder is depicted in Figure 1.2. Its computation can be partitioned into several tasks shown as blocks. Some of these tasks can run on different processors concurrently, thereby increasing the performance. DLP can be considered as a special case of instruction level parallelism [58], which was intensively studied in the past. The difference lies in the fact that DLP is explored at coarser level, e.g., at the processor level, whereas instruction level parallelism is exploited at finer level, e.g., using different functional units such as multiple Arithmetic and Logic Unit (ALU), floating point multipliers, *etc*. For instance in case of the H.264 video decoding, executing several video frames simultaneously on different PEs results in performance gain. PLP is an important form to exploit when parallelizing stateful computation (computation with cyclic dependencies) [54].

Video processing applications are in general good candidates for parallelization and demonstrate inherently adaptive behavior. For instance, a H.264 decoder contains major tasks, such as motion estimation, intra prediction, inverse discrete cosine transform, deblocking filter, and entropy coding. The H.264 decoder operates on data as set of Groups of Pictures (GoP). A GoP contains a set of frames. Several slices constitute a frame. Finally, a slice consists of several marcoblocks. Parallel scalability of H.264 video decoding is empirically studied in [85]. Large amount of DLP is shown to exist at different levels. The authors emphasize that DLP at different levels must be explored and especially at frame and marcoblock levels. In addition, the H.264 decoder also exhibits adaptive application behavior, namely three main types of slices/frames: I, P, and B types. For instance, a typical GoP consists of a I-P-B-B-P-B-B sequence of frames. On the one hand, processing an I-frame is independent from other frames. On the other hand, processing a P-frame depends on one or more previous frames, whereas processing B-frames depends on previous and future frames.

Software-Defined Radio (SDR) [88] applications also exhibit high parallelization opportunity and run-time adaptivity. For instance, the authors in [77] show that a 3G protocol, namely Wideband Code Division Multiple Access (WCDMA), demonstrates adaptive application behavior at different levels due to different operation modes and states. In the active mode, all computational tasks are active to process high rate traffic, whereas all tasks process at low rate in the control-hold mode. In the active mode, strict HRT requirements must be guaranteed to avoid buffer overflow, while the timing requirements are much more relaxed in the idle mode. The authors in [77] further characterize the computational workload of the tasks in the WCDMA protocol. Computationally intensive tasks, such as Branch Metric Calculation and Add Compare Select, contain enormous amount of DLP and TLP. This fact can be exploited to achieve an efficient parallel implementation. The authors in [130] study the computational workload of major 4G tasks. The tasks, such as Space Time Block Codes and Vertical Bell Laboratories Layered Space-time, contain abundant amount of DLP.

Computer vision is another important target application domain of embedded streaming systems. The applications in this domain are widely used in the fields of automotive, robotics, medicine, etc. Disparity Map [82] is such an example application that is used for adaptive cruise control on robotics or vehicles. It continuously processes a pair of images taken at slightly different positions. A disparity map is then computed in which depth information of all objects is represented. Since image processing kernels are often used, computer vision and image processing applications are categorized together in this thesis. In general, the applications in this domain contain large amount of DLP and TLP [121]. Typically, the same operations are performed repeatedly on all pixels in each image. At a higher level, there exists a few data dependencies between images in many applications. In this case, DLP at the image level can be also exploited. Next to DLP, different tasks of an application can execute normally in a feedforward pipeline fashion. Thus, there also exits a large amount of TLP and PLP to explore. In some applications, adaptive application behavior is an inherent part. For example, Feature Tracking [81] aims at extracting motion information from a set of consecutively captured images. During its execution at run-time, the features are first extracted. The number of extracted features and their width are expressed as parameters. The parameter values cannot be completely determined at compile-time and their values must be updated at run-time.

Finally, a collection of 65 real-life streaming applications is characterized in the StreamIT benchmark suit [116], to study their impact on language and compiler design. The applications are from different domains including video/audio processing, graphics rendering, DSP, and encryption. An important finding is that DLP should be considered as the first class citizen for performance optimization. In another important finding, the authors emphasize that cyclic data dependencies are uncommon in the application specifications. Around 90% of the studied benchmarks does not have cyclic data dependencies.

1.1.3 Platform Implications

Traditionally, the solution to achieve higher performance always involves the design of a system with higher frequency. However, as the technology node reaches below 100 nm, a single processor running at high frequency leads to extremely high power consumption [73]. Using Multi-Processor System-on-Chip (MPSoC) platforms partially addresses this problem by running processors at a lower frequency, which reduces power consumption. An MPSoC [132] is a very large scale integration system that incorporates most or all the components, including multiple programmable Processing Elements (PE) [1], peripheries, and memories, necessary for an application. It is widely acknowledged that MPSoC platforms are the best candidate to cope with various increasing requirements for embedded streaming systems. This thesis focuses on two important components, namely multiple PEs and the interconnection transferring data between them.

As the technology node further shrinks, chips with the same size of die is capable of accommodating more PEs. Together with the increasing performance requirements as motivated previously in Section 1.1.1, it can be expected that the number of PEs on a single chip will continue to increase. The processing part of an MPSoC platform for mobile devices is shown in Figure 1.3. Normally, the platform is equipped with multi-core CPUs which handle high-level applications, such as rendering Web pages and user interface functionalities. Next to the multi-core CPU, the multi-core GPU contains a set of PEs, which performs 2D/3D graphical processing. For instance, the Nvidia Tegra 4 [10] platform offers a quad-core CPU and a 72-core GPU. In addition to CPU and GPU, there are other programmable processors dedicated to certain class of functionalities. For example, a dedicated processor is often used to handle wireless communication protocols. To

---

[1] The term “PE” is used interchangeably with “core” or “processor” in this thesis.

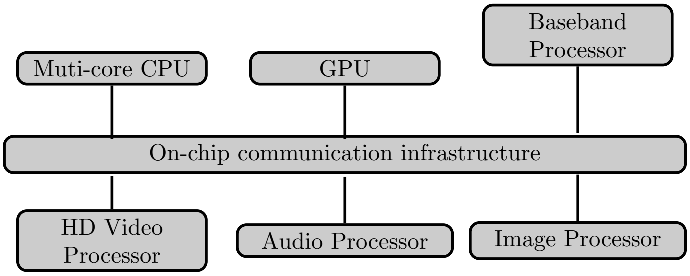


Figure 1.3: Processing part of an MPSoC platform for mobile devices from Nvidia (taken and simplified from [11]).

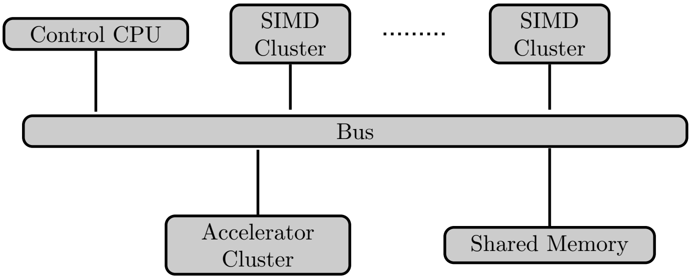


Figure 1.4: Template of a baseband processor (taken and simplified from [103]). All interfaces and peripheries are omitted.

be able to support multiple protocols, a programmable solution at the physical layer has emerged. For a baseband processor using the SDR technique, its template is illustrated in Figure 1.4. It consists of a control CPU for the processing protocol stack and hosting OS to orchestrate computation on other parts of the platform. For computationally intensive parts of applications, several Single Instruction Multiple Data (SIMD) clusters are used to support different algorithms in various wireless protocols. For instance, the Ardbeg [131] architecture has two SIMD clusters with one PE in each cluster. A PE is mainly a SIMD core with local memory. X-GOLD [103] is another instance of a baseband processor. It mainly differs from Ardbeg in the number of SIMD cores and size of local memory.

In addition to the PEs on an MPSoC platform, another important architectural element is the on-chip communication infrastructure. Network-on-Chip (NoC) [27] as the communication paradigm has emerged to alleviate the problem of platform scalability and its design has been one of the hottest research topics in the past decade. Æthereal [53] and Xpipes [28] are two prominent examples of NoC developed in academia. Æthereal provides bandwidth guarantees and thus it is more suitable

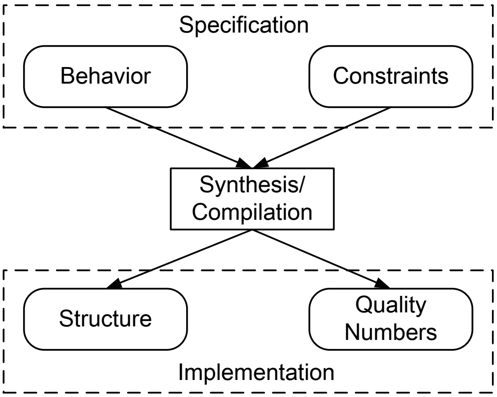


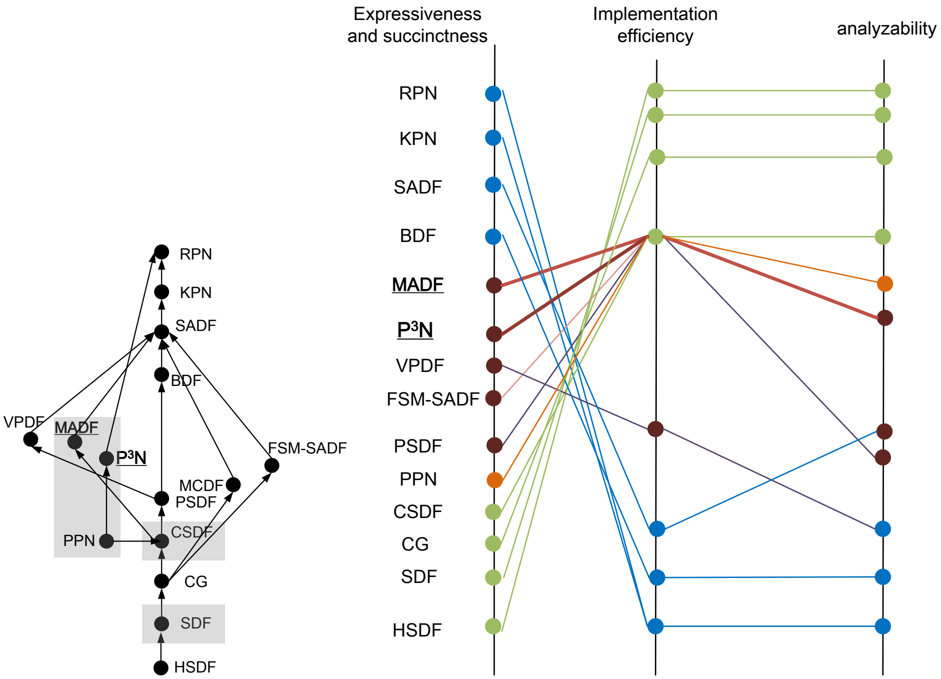
Figure 1.5: X-chart: a general design process (adopted from [50]).

for real-time systems due to bounded communication latency. Commercial NoC solutions [3] also have been integrated into the state-of-art MPSoCs for mobile phones.

1.1.4 Model-based Design Methodology

The high system requirements presented in Section 1.1.1 and platform complexity presented in Section 1.1.3 impose huge design challenges for designers to develop an efficient system manually. The traditional design process at a low-level of abstraction becomes very error-prone and time-consuming. It is widely recognized in the research community that rising the level of abstraction to Electronic System Level (ESL) [50] seems inevitable to increase the design productivity.

A complete design flow defined in [50] is shown in Figure 1.5. For the *specification* layer sitting on the top, an important component is called *behavioral model*. The behavioral model is specified either in certain programming language, such as C/C++/SystemC, CAL [38], StreamIT [117], Verilog/VHDL, or graphical representations, such as LabVIEW-G [20] and Simulink [8]. Different from general programming, a behavioral specification used for embedded system design normally complies with the underlying *Model of Computation* (MoC). A MoC [74] defines components and communication protocols that constraint the mechanism by which components can interact. A MoC is a formal model of how computation works. Consequently, adopting MoCs during the design process allows automated tools to reason about both functional and non-functional properties of an application. In the context of this thesis, only *concurrent* MoCs are considered because they are the natural way to express parallelism in streaming applications in an explicit way. Normally, a concurrent MoC describes an application by a directed graph



(a) Expressive hierarchy of MoCs. The MoCs considered in this thesis are highlighted by the boxes.

(b) Three aspects when comparing MoCs.

Figure 1.6: Comparison of dataflow MoCs for streaming applications (taken and extended from [112]). The MoCs underlined are proposed in this thesis.

where nodes are application tasks representing computation and the arcs represent communication. Consequently, MoCs greatly facilitate parallelizing compilers to perform aggressive optimizations. Therefore, both industrial and academic design flows extensively adopt different MoCs.

Figure 1.6 shows different MoCs widely used for modeling streaming applications. They differ in *expressiveness*, implementation efficiency, and compile-time *analyzability*[2]. Figure 1.6(a) shows the expressive hierarchy of different MoCs. The expressiveness and succinctness [112] of a MoC indicate which system can be modeled and how compact the models in these MoCs are. In most of cases, an arrow from MoC A to MoC B indicates that that a model in MoC A can be transformed to an input-output equivalent model in MoC B. In general, the MoCs with high expressiveness exhibit low compile-time analyzability. Similarly, the MoCs with

---

[2] Analyzability is referred as *decidability* in [55].

high expressiveness generally have lower implementation efficiency. The analyzability of a MoC [112] is determined by the availability of analysis and synthesis algorithms at compile-time and the run-time need for an algorithm on a graph with a given number of nodes and edges. The third aspect, implementation efficiency of a MoC [112] is decided by the complexity of the run-time scheduling algorithm problem and the (code) size of the resulting schedules. When comparing adaptive MoCs, we also consider the incurred performance overhead during run-time reconfiguration. As shown in Figure 1.6(b), Reactive Process Network (RPN) [46], Kahn Process Network (KPN) [64], Scenario-Aware Data Flow (SADF) [114], and Boolean Data Flow (BDF) [32] are Turing-complete MoCs, thereby being highly expressive. That is, this type of MoC is able to perform any computation that any other computer is capable of. However, these MoCs do not offer many possibilities of analysis at compile-time. At the bottom part of Figure 1.6(b), the MoCs, such as Synchronous Data Flow (SDF) [76], Cyclo-Static Data Flow (CSDF) [30], and Polyhedral Process Network (PPN) [125], exhibit high compile-time analyzability. They are discussed in detail in Chapter 2. For these MoCs, various powerful analysis and compilation/synthesis methods have been developed over the past twenty years, e.g., to compute throughput [52, 87], buffer sizes [110], efficient static schedules for software compilation [91, 107, 124], and hardware synthesis [63, 120]. However, these MoCs are restricted to static application behavior. Modern streaming applications with adaptive behavior as explained in Section 1.1.2 cannot be expressed using these MoCs. To model adaptive behavior while having certain degree of compile-time analyzability, different adaptive MoCs, such as Mode-controlled Data Flow (MCDF) [89], Finite State Machine (FSM)-based Scenario-Aware Data Flow (FSM-SADF) [47], Parameterized SDF (PSDF) [29], and Variable-rate Phased Data Flow (VPDF) [129], have been proposed. For these MoCs, functional properties of the adaptive MoCs can only be partially decided at compile-time, and run-time verification is thus needed. For SADF, it is even possible to statically analyze functional properties at compile-time.

To take advantage of different properties of MoCs, some design flows separate the *analysis* model from the *implementation* model. Here the implementation model is the one that is close to the final implementation to be executed on the real MPSoC platform, whereas the analysis model is primarily used for analyzing non-functional properties. In this thesis, the timing property is of particular interest. For instance in the current industrial practice, a disciplined version [70] of C is used as the implementation model to program embedded radio applications, including code generation for communication and/or synchronization. On the other hand, analysis of real-time guarantees, required buffer sizes, etc., is performed on the SDF MoC, which serves as the analysis model.

Next to the behavioral model, the specification layer of the design flow shown in Figure 1.5 may contain *platform constraints* that explicitly specify the platform model. As explained in Section 1.1.3, that is, e.g., the type and number of PEs, the memory type and capability, and the interconnection between PEs. In addition to the platform constraints, other constraints can be used as input to the design flow in this thesis, such as timing constraints. In particular, the timing constraints are the essential property of a real-time streaming system.

With the behavior model, namely MoCs, and constraints in place, they are transformed in a step, called *synthesis* or *compilation* (in case of software models). This step normally determines e.g., allocation of PEs and necessary buffers if not given before hand, spatial mapping[3] of application tasks on PEs, temporal scheduling of all tasks on a PE, etc. Obtaining an efficient solution for these problems is certainly very challenging. In most cases, all possible combinations of PE allocation and assignment of tasks to PEs constitute an enormous design space with different conflicting objectives. For example, maximum throughput should be achieved while resource usage needs to be minimized. To efficiently search the design space and find an optimum solution, various Design Space Exploration (DSE) approaches proposed in the literature try to find a solution that is called *Pareto-optimal* point in the design space if, e.g., higher throughput cannot be achieved with fewer PEs. Currently, existing DSE approaches search the design space using different algorithms, e.g., stepwise refinement in [51], heuristics in [109] and [111], evolutionary algorithms in [100, 115], branch-and-bound in [34], and constraint programming in [139]. The synthesis/compilation step outputs a *structure* model as shown in Figure 1.5. Here the structure model is (or closer than the behavioral model to) the final, executable implementation. It may be in the form of pin-accurate netlists or Transaction-Level Models (TLM). As an output next to the structure model, *quality numbers* represent non-functional properties, e.g,. throughput, end-to-end latency, etc.

An Incarnation: Daedalus^{RT} Design Flow

The Daedalus^{RT} [23] design flow is based on the initial Daedalus [96, 97] framework, which covers all three layers in Figure 1.5, namely system-level DSE, synthesis, and prototyping of MPSoCs. Daedalus^{RT} has been recently proposed, as the name suggests, to address HRT requirements (see Section 1.1.1), The research work of this thesis has been performed in the context of the Daedalus^{RT} design flow and an overview of Daedalus^{RT} is shown in Figure 1.7. The grey boxes highlight the contributions of this thesis, which are explained in details in Section 1.3.

---

[3] Task mapping is often also referred as task allocation in literature and both are used interchangeably in this thesis.

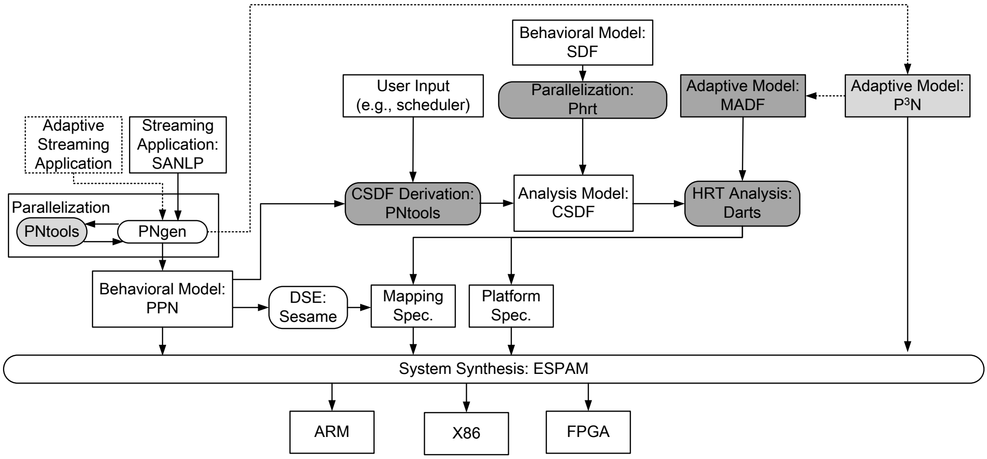


Figure 1.7: Daedalus^{RT} design flow. The grey boxes highlight the contributions of this thesis. The dashed box and lines denote the parts that are currently not fully implemented.

The input to Daedalus^{RT} is a streaming application specified as a sequential C code with restrictions, called Static Affine Nested Loop Program (SANLP) [125] (see Section 2.1). Many streaming applications are amenable to this restricted form [26]. Moreover, an early study [106] has shown that, out of 100,000 lines of loops, 53% of them can be converted to SANLPs. In the Parallelization step, a SANLP is automatically translated to its equivalent behavioral model, the PPN MoC using the PNgen compiler [125]. The resulting PPN exposes certain form of parallelism, specifically TLP of the initial SANLP. Currently, the PNgen compiler also extracts DLP from a SANLP in a particular way using a combination of transformations [86]. The formal definition of SANLP and the PPN MoC is detailed later in Section 2.1. Alternatively, application designers also have the flexibility to specify streaming applications as (C)SDF graphs directly. It is sometimes more convenient to do so using tools based on graphical interfaces. For adaptive streaming applications, they are specified as two new MoCs proposed in this thesis. Their details can be found in Chapter 6 and Chapter 7, respectively.

In the initial Daedalus framework, the second step, namely DSE, is realized using the Sesame [100] tool, which takes a PPN as input and generates a Pareto-optimal set of design points. A design point consists of a platform and mapping specifications. For HRT streaming systems, an analysis model, the CSDF MoC, is required. In Daedalus^{RT}, a PPN derived from a SANLP needs to be converted to its equivalent CSDF graph. Subsequently, the Darts tool replaces time-consuming DSE and performs the HRT analysis [22] on the resulting CSDF graph. The main advantage of the HRT analysis is the fast, yet accurate determination of the minimum number of PEs needed to schedule the CSDF graph and leveraging well-known HRT multiprocessor scheduling algorithms. The HRT analysis on the CSDF MoC is detailed in Section 2.3.

Finally in the third step, namely System Synthesis, the ESPAM [95, 96] tool takes a PPN with the platform and mapping specifications, and produces an executable implementation on various platforms. The platform consists of several tiles interconnected via certain communication infrastructure. On the FPGA-based platform, each tile consists of a PE in the form of the MicroBlaze [13] softcore from Xilinx with its local program and data memories. A communication memory resides in each tile and it is used as data storage for communication between application tasks mapped to different tiles. The interconnection between all tiles, the DDR off-chip memory, and peripheries is an AXI crossbar switch [2]. In principle, the crossbar switch can be replaced by e.g., the Æthereal [53] NoC, to provide guaranteed communication latency. For the PEs, ARM Cortex A9 [1] cores can be instantiated on the Xilinx Zynq [16] platform instead of the MicroBlaze softcore. In Daedalus, a static schedule [124] is used on each PE to temporally schedule all tasks allocated on the PE. Alternatively, a light-weight and multi-threaded OS, Xilkernel [14], is built on top of a PE to perform run-time scheduling. Later, support for the x86 platform has been added to the ESPAM backend [39]. The target is normally desktop multi-core platform, such as Intel i7-920 processor. For the x86 platform, application tasks implemented as threads can be dynamically scheduled by OS, such as Windows or Linux. In this case, OS either determines allocation and temporal schedule of all threads at run-time. Alternatively, the threads are statically bound to a PE by assigning core affinity. In the latter case, no run-time migration of threads is required, thereby reducing performance penalty. In Daedalus$^{\text{RT}}$, a RTOS, specifically FreeRTOS [7], is chosen to run on each PE. FreeRTOS implements fixed-priority scheduling and supports Xilinx FPGAs. The hardware and software architecture explained here is extensively used later throughout case studies and experiments.

1.1.5 Summary

Here, we summarize the key insights that can be drawn from the discussion in previous sections.

From the design requirements point of view, the following are the most significant requirements.

- Embedded streaming applications pose ever increasing throughput requirements.

- Embedded streaming systems require Hard Real Time (HRT) guarantees. Furthermore, it is not uncommon to have both HRT constraints and high throughput requirements at the same time.

From the application characteristics point of view:

- Data Level Parallelism (DLP) and Task Level Parallelism (TLP) are the most important forms of parallelism to exploit, which result in an efficient parallel implementation to achieve high throughput requirements.
- Embedded streaming applications commonly exhibit adaptive behavior in the form of parameter reconfigurations at run-time. This behavior should be explicitly captured in the application specification.

From the architectural perspective:

- An increasing number of Processing Elements (PE) on MPSoC platform is deployed to meet stringent performance requirements. The key question is thus how to utilize them efficiently.
- Network-on-Chip (NoC) emerges and is expected to become the standard communication infrastructure of an MPSoC platform in the near future. A corresponding design methodology is desired to program applications on NoC-based MPSoC platforms to manage communication latency.

From the design methodology perspective:

- Raising the abstraction level to ESL seems inevitable to cope with ever increasing complexity. To fully leverage the benefit of ESL, highly automated tools are needed.
- A central component of an ESL solution is the Model-of-Computation (MoC). Various MoCs, such as (C)SDF, PPN, SADF, PSDF, and VPDF, are extensively adopted to program and/or analyze embedded streaming applications.

1.2 Problem Statement

As motivated in Section 1.1.5, a de-facto solution to the problem of designing complex embedded streaming systems is the adoption of an ESL methodology and highly automated tools. In this thesis, we choose the Daedalus^{RT} design flow as a particular instance. We see several components missing in Daedalus^{RT} to address the requirements outlined in Section 1.1.5 and to efficiently exploit the proper application characteristics and emerging architectural features. Therefore, we address three main problems in this thesis as described below.

We first observe that the current MoC, namely the PPN MoC, used in the Daedalus^{RT} design flow works well as an implementation model. It is possible to efficiently generate code [95] automatically from the PPN MoC for task execution, communication, and synchronization. However, analysis on the PPN MoC, such as for timing guarantees, is rather difficult if not impossible. Both in Daedalus^{RT} and the current industrial practice [90], a more analyzable MoC, such as (C)SDF MoC, is adopted. So far, this analysis model is created manually. However, creating analysis model from an implementation model manually may introduce disparity between both types of models. It is thus hard to guarantee correctness of the analysis model. Based on the discussion above, we formulate the first problem addressed in this thesis: **derive automatically a CSDF graph as the analysis model from an equivalent PPN used as the implementation model**.

Generally, in the Synthesis step shown in Figure 1.5, the traditional DSE approaches like Sesame consider only different mapping and architectural alternatives. With respect to the behavior model, only a single application specification is considered during DSE. This single application specification is normally given by the application designer. Or, the PNgen compiler generates one instance of a PPN that exposes TLP. However, this application specification may not be the most appropriate one for the considered MPSoC platform. That is, the specification may not expose enough parallelism, particularly in the form of DLP, to satisfy the required performance. This is because application designers mainly focus on realizing certain application behavior, including the identification of the functionality of application tasks and the synchronization/communication between these tasks. Moreover, the computational capacity and communication cost of the MPSoC platform are often not taken into account when developing a parallel application specification. In particular, as mentioned in Section 1.1.3, the MPSoC platform is becoming more communication-centric with NoC as the interconnection. As a consequence, overwhelming communication between application tasks may cancel out the expected performance improvement when the application tasks are executed concurrently. Therefore, the second problem addressed in this thesis aims at effectively exploiting DLP in a streaming application. The second problem consists of two sub-problems. We formulate the first sub-problem in the context of Daedalus^{RT} as: **for an initial PPN, investigate an approach to derive an alternative PPN that contains only independent and load-balanced application tasks, if such an alternative PPN exists.**

On the other hand, if more parallelism is revealed than needed when selecting an alternative application specification, it will overload the underlying MPSoC platform.

The overwhelming parallelism leads to an inefficient task allocation. That is, the excessive number of tasks cannot be efficiently allocated and temporally scheduled on the available PEs. Moreover, the excessive number of tasks introduces significant memory overhead for both code and data. When a streaming application is initially modeled using the SDF MoC and requires to meet HRT constraints, we exploit DLP and TLP simultaneously by actor (i.e., tasks) unfolding and transform the initial SDF graph to its equivalent CSDF graph. Therefore, we formulate the second sub-problem in the context of Daedalus$^{\mathrm{RT}}$ as: **for an initial SDF graph, derive an alternative CSDF graph that exhibits just-enough parallelism to fully utilize the available PEs, such that HRT constraints are met**.

The third problem addressed in this thesis relates to adaptive application behavior as explained in Section 1.1.2. Such behavior is usually expressed by using parameters whose values need to be reconfigured and updated at run-time. We call such parameters dynamic parameters and their values are not known at design-time. Models such as (C)SDF or PPN used in the Daedalus$^{\mathrm{RT}}$ design flow have the limitation of allowing only static parameters. The values of the static parameters are fixed at design-time and they can not be changed at run-time. As a consequence, the adaptive behavior is not amenable to the models such as SDF/CSDF and PPN. Therefore, more expressive MoCs are needed. The MoCs such as BDF and KPN shown in Figure 1.6 provide capability of modeling adaptive application behavior. However, these general MoCs are not analyzable at design-time. Therefore, we are interested in an adaptive MoC which is able to capture adaptive/dynamic behavior in applications while allowing design-time analyzability to some extent. Furthermore, if an adaptive streaming application has HRT requirements, the existing methods lack the ability to efficiently reason about timing behavior based on the chosen adaptive MoC. Moreover, a feasible and efficient way of implementing such an adaptive MoC on MPSoC platforms has not been taken into consideration. Therefore, as the third problem, we **investigate new adaptive MoCs to model adaptive streaming applications and techniques to schedule those adaptive MoCs under HRT constraints**.

1.3 Research Contributions

To address the problems outlined in Section 1.2, this thesis provides several contributions highlighted using the grey boxes in Figure 1.7.

To address the first problem, we develop a step, called *CSDF Derivation*, in this thesis as shown in Figure 1.7. This step primarily contains an **algorithm**, published as a major part of [23], to derive the analysis MoC, i.e., the CSDF MoC, from the implementation model, i.e., the PPN MoC. We present such an algorithm in

Chapter 3. This algorithm is a key enabler of a highly automated design flow, namely Daedalus^{RT} [23], for designing embedded streaming systems with hard real-time constraints. The automated CSDF derivation avoids manual creation of analysis models, thereby greatly improving the productivity of designing such complex embedded streaming systems. Beyond the above-mentioned advantage, automated CSDF derivation can be applied together with other compilation frameworks in which CSDF is adopted as the intermediate representation, e.g., the compilation toolchain [21] for the $\sum$C language, the MAMPS [62] design flow, and the CompSoC [57] framework.

Our second contribution consists of the two *Parallelization* steps shown in Figure 1.7 addressing the second problem stated in Section 1.2. First, we propose in Chapter 4 a parallelization approach next to the PNgen compiler for the PPN MoC, called **communication free partitioning**, published in [138] and [137]. Our approach analytically determines the maximum amount of DLP in the form of a set of communication-free partitions from a given PPN specification. When mapping theses partitions onto different PEs, the communication between PEs is completely eliminated. This parallelization approach is thus highly relevant to emerging NoC-based MPSoC platforms as mentioned in Section 1.1.3, where communication latency may play a significant role on the total execution time of an application. Our approach also can be applied to applications with cyclic dependences, which are traditionally considered as performance bottleneck and hard to parallelize. Second, we propose in Chapter 5 a *Parallelization* step for the SDF MoC, published in [135], to exploit **just-enough parallelism** by task unfolding that fully utilizes the underlying MPSoC platforms, while meeting hard real-time constraints. More specifically, our solution determines simultaneously which SDF actors (i.e., tasks) to unfold by what factor, and the allocation of unfolded actors onto PEs. We show that the solution space of the problem is bounded and derive its upper bounds. We then propose an efficient algorithm to find a solution to the problem, while the obtained solution meets a pre-defined quality.

To address the third problem in Section 1.2, we introduce in Chapter 6 and Chapter 7 two new MoCs, **Parameterized Polyhedral Process Networks** (P^3N), published in [136], and **Mode-Aware Data Flow** (MADF), for modeling adaptive streaming applications. We further define the operational semantics of both MoCs, which allows flexible update of parameter values at run-time. In addition, we propose a consistency check approach for P^3N, which is applied at both, compile-time and run-time. Based on the P^3N semantics, we devise a compile-time approach to extract relations between parameters if they are dependent. This leads to a consistent parameterization of the P^3N MoC and moreover, it simplifies the run-time consistency check. The simplification reduces the run-time overhead. Subsequently, we **extend the capability of the hard real-time scheduling framework** used in Daedalus^{RT}

for CSDF to handle MADF. We propose a novel protocol that allows efficient mode transitions, i.e., parameter reconfiguration. As a result, the transition protocol enables us to show an efficient analysis technique to reason about guaranteed timing behavior, particularly during mode transitions.

All contributions mentioned above are implemented either in Daedalus or in Daedalus^{RT}. Furthermore, both Daedalus and Daedalus^{RT} are publicly available [4] for further research. A detailed user manual [24] including an installation guideline and step-by-step tutorial is also available for the benefit of the research community.

1.4 Thesis Organization

The remaining part of this thesis is organized in a self-contained way. That is, every chapter starts with more elaborated introduction and scope of work. More importantly, each chapter has its own related work.

In Chapter 2, we first introduce different MoCs considered in this thesis, particularly (C)SDF and PPN, to better understand our research contributions in later chapters.

In Chapter 3, we present the algorithm to derive the CSDF MoC from its equivalent PPN MoC. The benefit of the proposed algorithm is demonstrated in the context of the Daedalus^{RT} real-time extension.

In Chapter 4, we present the analytical approach to determine the number of communication-free partitions of a PPN. Subsequently, we present the procedure to transform the initial PPN to an alternative PPN that has only set of communication-free partitions, if possible.

In Chapter 5, we present the approach to simultaneously unfold an acyclic SDF graph to its functionally equivalent CSDF graph and allocate all unfolded actors onto PEs, such that HRT constraints are met.

In Chapter 6, we introduce a new adaptive MoC, called Parameterized Polyhedral Process Networks (P^{3}N) and its operational semantics. Subsequently, we show how consistency check can be performed for P^{3}N at compile-time and run-time.

In Chapter 7, we present the hard real-time scheduling approach for another adaptive MoC, which we propose and call Mode-Aware Data Flow (MADF). The approach contains a novel protocol to change scenarios. Based on the protocol, we derive an efficient analysis to reason about timing guarantees, not only within individual scenarios, but also during scenario transitions.

Finally, we conclude this thesis with a summary and some suggestions for future work.

Chapter 2

Models-of-Computation (MoC)

THIS chapter is dedicated to different Models of Computation (MoC) that serve as the application specification. In particular, we focus on a process-based MoC, namely Polyhedral Process Networks (PPN), in Section 2.1, and two actor-based MoCs, SDF and CSDF in Sections 2.2.1 and 2.2.2, respectively. The PPN MoC is used as the implementation model in Daedalus^{RT} and it is the input to the solutions proposed in Chapters 3 and 4. The SDF MoC is the input to the solution proposed in Chapter 5. The CSDF MoC is used to perform HRT analysis. In Section 2.3, an overview of the HRT analysis is given to better understand the solutions proposed in Chapters 5 and 7. Throughout this thesis, we use the set of mathematical notations listed in Table 2.1.

Both PPN and (C)SDF MoCs are specified as a graph consisting of vertices and edges. Normally, all vertices denote concurrently executing computation tasks. For (C)SDF, the vertices are called *actors*, whereas the vertices in a PPN are called *processes*. The edges denote FIFOs for data communication between actors/processes. It is possible to compute a safe FIFO size [110, 125] for each edge that guarantees the absence of deadlock in the graph.

2.1 Polyhedral Process Networks (PPN)

An important advantage of adopting the PPN MoC in Daedalus^{RT} is that it can be automatically derived from an input-output equivalent sequential specification with certain restrictions using the PNgen [125] compiler. Thus, the error-prone process of deriving a concurrent model manually can be avoided. Moreover, the ESPAM [96] tool is able to generate final parallel implementation for the PPN MoC in an automated way. Consequently, design productivity can be significantly improved. In the following sub-sections, we first explain the *polytope model*, which

Notation	Meaning
$\mathbb{N}$	the set of natural numbers excluding zero
$\mathbb{Q}$	the set of rational numbers
$\mathbb{Z}$	the set of integer numbers
$\check{x}$	lower bound (minimum) of values x
$\hat{x}$	upper bound (maximum) of values x
lcm	least common multiple
$\lceil x \rceil$	smallest integer that is greater than or equal to x
$\lfloor x \rfloor$	greatest integer that is smaller than or equal to x
$\lvert \mathcal{X} \rvert$	cardinality of a set $\mathcal{X}$
$\vec{x}$	vector x

Table 2.1: Mathematical notations.

we use as the formal representation of the PPN MoC and the P^3N MoC developed in Chapter 6. It is followed by an explanation of the sequential specification with restrictions in detail. Then, we introduce the PPN MoC based on the polytope model derived from the sequential specification.

Polytope Model

The *Polytope model* [42] is often used in the compiler domain to represent loop nests, which perfectly match the behavior of streaming applications. The polytope model allows powerful transformation techniques that are used to explore and exploit parallelism in Chapters 3 and 4. It also serves as the foundation of the analysis presented in Chapter 6. This section presents an overview of the polytope model to make this thesis self-contained. A more detailed treatment of the polytope model can be found in [26]. The mathematical background can be found in popular textbooks, such as [105]. Throughout this thesis, the notations related to the polytope mode are listed in Table 2.2.

We start with some fundamental definitions. Assuming a vector $\vec{y} \in \mathbb{R}^n$ and a constant α, $\mathcal{H} = \{\vec{x} \mid \vec{x} \cdot \vec{y}^T \geq \alpha\}$ is called a *closed half-space*. Then, we define a *polyhedron* as follows:

Definition 2.1.1 (Polyhedron). A polyhedron $\mathcal{D}$ is the intersection of a set of finitely many closed half-space, i.e.,

$$\mathcal{D} = \{\vec{x} \in \mathbb{Q}^d \mid A\vec{x} \geq \vec{c}\}, \tag{2.1}$$

where $A \in \mathbb{Z}^{m \times d}$ is a constant matrix and $c \in \mathbb{Z}^m$ is a constant vector.

$\mathcal{D}$	a polyhedron
$\mathcal{D}(\vec{p})$	a parametric polyhedron
$\bar{\mathcal{D}}$	a polytope
$\bar{\mathcal{D}}(\vec{p})$	a parametric polytope
$\lvert\bar{\mathcal{D}}\rvert$	cardinality of polytope
R	dependence relation
ran$R$	range of a dependence relation
dom$R$	domain of a dependence relation

Table 2.2: Polyhedral notations.

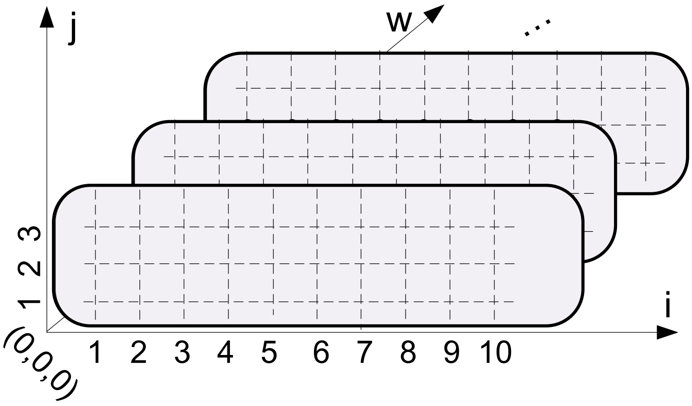


Figure 2.1: A polyhedron.

Definition 2.1.2 (Polytope)**.** A polytope $\bar{\mathcal{D}}$ is a bounded polyhedron.

Consider for instance a polyhedron defined as follows:

$$\mathcal{D} = \left\{ (w,i,j) \in \mathbb{Q}^3 \mid \begin{bmatrix} 0 & 1 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & -1 \\ 1 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} w \\ i \\ j \end{bmatrix} \geq \begin{bmatrix} 1 \\ -10 \\ 1 \\ -3 \\ 0 \end{bmatrix} \right\},$$

$$= \{(w,i,j) \in \mathbb{Q}^3 \mid w \geq 0 \wedge 1 \leq i \leq 10 \wedge 1 \leq j \leq 3\}. \tag{2.2}$$

The polyhedron is illustrated in Figure 2.1 using grey boxes. We can see that the polyhedron is unbounded along w-dimension. If we consider any w equal to a constant c, we obtain a polytope $\bar{\mathcal{D}}_w$ as:

$$\bar{\mathcal{D}} = \left\{ (w,i,j) \in \mathbb{Q}^3 \mid \begin{bmatrix} 0 & 1 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & -1 \\ 1 & 0 & 0 \\ -1 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} w \\ i \\ j \end{bmatrix} \geq \begin{bmatrix} 1 \\ -10 \\ 1 \\ -3 \\ c \\ -c \end{bmatrix} \right\},$$

$$= \{(w,i,j) \in \mathbb{Q}^3 \mid w = c \wedge 1 \leq i \leq 10 \wedge 1 \leq j \leq 3\}.$$

We can see that the initial $\mathcal{D}$ in Equation (2.2) is now bounded on the w dimension. More specifically, $\bar{\mathcal{D}}$ can be considered as a "plane" spread along i and j axes shown in Figure 2.1.

In Chapter 6, we use the concept of parametric polyhedron to represent adaptive streaming applications.

Definition 2.1.3 (Parametric Polyhedron)**.** A parametric polyhedron $\mathcal{D}(\vec{p})$ is a polyhedron $\mathcal{D}$ affinely depending on a parameter vector $\vec{p} \in \mathbb{Q}^n$, i.e.,

$$\mathcal{D}(\vec{p}) = \{\vec{x} \in \mathbb{Q}^d \mid A \cdot \vec{x} \geq B \cdot \vec{p} + \vec{b}\}, \tag{2.3}$$

where $\vec{p}$ is bounded by a polytope $\bar{\mathcal{D}}_{\vec{p}} = \{\vec{p} \in \mathbb{Q}^n \mid C \cdot \vec{p} \geq \vec{h}\}$. A, B, and C are constant integer matrices. $\vec{b}$ and $\vec{h}$ are constant vectors.

Similarly, we have the notion of parametric polytope, which is a bounded parametric polyhedron.

Consider two parameters m and n that are bounded by a polytope

$$\bar{\mathcal{D}}_{(m,n)} = \{(m,n) \in \mathbb{Q}^2 \mid 0 \leq m \leq 100 \wedge 0 \leq n \leq 100\}. \tag{2.4}$$

We can have a parametric polyhedron defined as follows:

$$\mathcal{D}(m,n) = \{(w,i,j) \in \mathbb{Q}^3 \mid w > 0 \wedge 1 \leq i \leq 2m \wedge 1 \leq j \leq n - 2i\}.$$

A parametric polytope can be

$$\bar{\mathcal{D}}_1(m,n) = \{(w,i,j) \in \mathbb{Q}^3 \mid w = 1 \wedge 1 \leq i \leq 2m \wedge 1 \leq j \leq n - 2i\}. \tag{2.5}$$

In this thesis, we are also interested in the number of integer points in a set $\bar{\mathcal{D}}(\vec{p}) \cap \mathbb{Z}^d$, called *cardinality* and denoted by $|\bar{\mathcal{D}}(\vec{p})|$. For a set $\bar{\mathcal{D}} \cap \mathbb{Z}^d$, its cardinality $|\bar{\mathcal{D}}|$ can be obtained as a constant, whereas $|\bar{\mathcal{D}}(\vec{p})|$ is expressed as a piecewise quasi-polynomial. A piecewise quasi-polynomial consists of one or more quasi-polynomials.

Definition 2.1.4 (Quasi-polynomial). A quasi-polynomial $q(x)$ in the integer variables x is a polynomial expression in greatest integer parts of affine expressions in the variables.

Definition 2.1.5 (Piecewise Quasi-polynomial). A piecewise quasi-polynomial $q(\vec{x})$, with $\vec{x} \in \mathbb{Z}^d$ consists of one or more quasi-polynomials. Each quasi-polynomial $q_i(\vec{x})$ is defined only for a disjoint piece $\bar{\mathcal{D}}_i(\vec{x})$ of a parametric polytope $\bar{\mathcal{D}}(\vec{x})$. Each $\bar{\mathcal{D}}_i(\vec{x})$ is also called a *chamber* C_i. For a given point $\vec{x} \in \bar{\mathcal{D}}(\vec{x})$, the piecewise quasi-polynomial evaluates to

$$q(\vec{x}) = \begin{cases} q_i(\vec{x}) & \text{if } x \in \bar{\mathcal{D}}_i(\vec{x}) \\ 0 & \text{otherwise.} \end{cases} \tag{2.6}$$

Consider the parametric polytope $\bar{\mathcal{D}}_1(m,n)$ in Equation (2.5) with parameters m and n bounded by the polytope in Equation (2.4). For the number of integer points in the set $\bar{\mathcal{D}}_1(m,n) \cap \cap \mathbb{Z}^3$, $|\bar{\mathcal{D}}_1(m,n)|$ can be obtained as a piecewise quasi-polynomial as follows:

$$\begin{cases} -2m - 4m^2 + 2mn & \text{if } (m,n) \in C1 \\ -\frac{1}{4}n + \frac{1}{4}n^2 - \frac{1}{2} \cdot \{0,1\}_n & \text{if } (m,n) \in C2 \end{cases}$$

where $\{0,1\}_n$ is called a periodic number with period 2. $C1$ and $C2$ are called chambers given as

$$C1 = \{(m,n) \in \mathbb{Z}^2 \mid 2 + 4m \le n \land 1 \le m \le 100 \land 0 \le n \le 100\},$$
$$C2 = \{(m,n) \in \mathbb{Z}^2 \mid n \le 1 + 4m \land 3 \le n \le 100 \land 0 \le m \le 100\}.$$

Often when we use the polytope mode to represent execution of a program, we need the definition of a lexicographic order.

Definition 2.1.6 (Lexicographic order). Given that two vectors $\vec{a}, \vec{b} \in \mathbb{Z}^n$ are elements of a polyhedron. $\vec{a} \prec \vec{b}$ denotes that $\vec{a}$ is lexicographically smaller than $\vec{b}$, if

$$\bigvee_{i=1}^{n} \left(a_i < b_i \land \bigwedge_{j=1}^{i-1} a_j = b_j\right)$$

For instance, given $\vec{a} = (w,i,j) = (0,1,3)$ and $\vec{b} = (w,i,j) = (0,2,1)$, we have $\vec{a} \prec \vec{b}$.

When using the polytope model to represent loop nests, we often need to deal with dependence relations to express data dependencies.

Definition 2.1.7 (Dependence Relation [122])**.** A dependence relation R, also called a basic polyhedral map, is defined as

$$R = \{\vec{x}_1 \to \vec{x}_2 \in \mathbb{Z}^{d_1} \times \mathbb{Z}^{d_2} \mid \vec{x}_1 \in \mathcal{D}_1 \wedge \vec{x}_2 \in \mathcal{D}_2 \wedge \vec{x}_2 = A\vec{x}_1 + \vec{c}\}, \tag{2.7}$$

where A is an integer matrix and $\vec{c}$ is a constant vector. The polyhedron $\mathcal{D}_1$ is the domain of dependence relation R, denoted by dom$R$. The polyhedron $\mathcal{D}_2$ is the range of dependence relation R, denoted by ran$R$.

For instance, we have a dependence relation

$$\begin{aligned} R =\{&(w1,i1,j1) \to (w2,i2,j2) \in \mathbb{Z}^3 \times \mathbb{Z}^3 \\ &\mid (w1,i1,j1) \in \mathcal{D}_1 \wedge (w2,i2,j2) \in \mathcal{D}_2 \wedge i2 = i1 - 1 \wedge j2 = j1 + 1 \wedge w1 = w2\}, \end{aligned}$$

where

$$\text{dom}R = \mathcal{D}_1 = \{(w1,i1,j1) \in \mathbb{Z}^3 \mid w1 \geq 0 \wedge 1 \leq i1 \leq 7 \wedge 0 \leq j1 \leq 7 - i1\}$$

and

$$\text{ran}R = \mathcal{D}_2 = \{(w2,i2,j2) \in \mathbb{Z}^3 \mid w2 \geq 0 \wedge 0 \leq i2 \leq 6 \wedge 1 \leq j2 \leq 8 - i2\}.$$

Static Affine Nested Loop Programs (SANLP)

The sequential application specifications considered in this thesis are in the form of *Static Affine Nested Loop Programs* (SANLP).

A SANLP consists of several primitive *functions*. A function is considered as a primitive in this thesis. This means that no explicit parallelization is performed within the function. In general, parallelism within functions can be explored at finer level, e.g., by vectorization [98]. A function serves mainly as the computational part of an application task. Note that there is no restriction on the structure within a function. That means that a function may contain an arbitrary structure of code.

However, restrictions do exist at the level of SANLP, in which functions are called and executed. We summarize the key restrictions of SANLPs as follows.

Definition 2.1.8 (Static Affine Nested Loop Program (SANLP) [41])**.** A static affine nested loop program contains a set of functions, each of which is enclosed by one or more loops and *if*-statements. The loops and *if*-statements have the following restrictions:

- loops have a constant step size;

- loop bounds are affine expressions of the enclosing loop iterators, static parameters, and constants. Static parameters are those whose value cannot change at run-time;
- *if*-statements have affine conditions in terms of the loop iterators, static parameters, and constants;
- index expressions of array references are affine constructs of the enclosing loop iterators, static parameters, and constants;
- the data flow between functions in the loop is explicit, which prohibits that two functions communicate through shared variables invisible at the SANLP level.

An example of a SANLP is shown in Listing 1. Although it is represented using the C syntax, in principle SANLP can be expressed in other forms, such as Matlab [66] or Fortran [101]. Four functions `read_image`, `fiter1`, `filter2`, and `write_image` only exchange data through indexed arrays `img` and `ref_img`. Executing the loop body once is called an *iteration*. For function `read_image`, the polyhedral representation of its execution is given in Equation (2.2) and illustrated in Figure 2.2. The black dots denote individual iterations. According to Definition 2.1.6, iteration $\vec{a} = (w, i, j) = (0, 1, 3)$ is executed before $\vec{b} = (w, i, j) = (0, 2, 1)$, denoted as $\vec{a} \prec \vec{b}$.

PPN

A Polyhedral Process Networks (PPN) [125] is defined as a graph $G = (\mathcal{P}, \mathcal{E})$, where $\mathcal{P}$ is the set of processes and $\mathcal{E}$ is the set of edges. The PPN MoC is a special

```
while(1){
  for (i = 1; i <= 10; i++){         // Width
    for (j = 1; j <= 3; j++){        // Height
      read_image(&img[i][j], &ref_img[i][j]);

      if (j <= 2)
        img[i][j] = filter1(img[i][j]);
      else
        img[i][j] = filter2(img[i][j]);

      write_image(img[i][j], ref_img[i][j]);
} } }
```

Listing 1: An example of a SANLP

case of the Kahn Process Networks (KPN) [64] MoC. That is, PPN processes are synchronized through FIFOs, i.e., any process is blocked when attempting to read from an empty FIFO or write to a full FIFO. In the definition of the KPN MoC, no restriction is imposed on the structure of the KPN processes. In contrast, a PPN process has a particular structure due to the fact that it is automatically derived from a SANLP using the PNgen [125] compiler.

Each function in a SANLP corresponds to a separate process in the derived PPN. If two functions access the same data array through their input/output arguments, they may thus have data dependencies, which is determined by Array Dataflow Analysis (ADA) [41].

The execution of a PPN process is specified using affine nested *for*-loops, called *domain*. Formally, a domain D is defined as a polyhedron following Definition 2.1.1, i.e., $D = \{\vec{I} \in \mathbb{Z}^d \mid A \cdot \vec{I} \geq \vec{b}\}$, where $A \in \mathbb{Z}^{m \times d}$, $\vec{b} \in \mathbb{Z}^d$, $\vec{I}$ is an *iteration vector*, and d indicates the nested-loop depth. At each iteration $\vec{I}$ during the execution of a PPN process P, namely $\vec{I} \in D_P$, P first reads data from input ports (*IP*) in the input port domain D_{IP} if $\vec{I} \in D_{IP}$. Then the process executes the process function (computation) and subsequently writes results to output ports (*OP*) in the output port domain D_{OP} if $\vec{I} \in D_{OP}$. The order of executing different iterations in a process domain is specified by a lexicographic order according to Definition 2.1.6 on page 25. The set of iterations, at which a PPN process writes data to the environment, are called *sink iterations*, denoted by D_{snk}. Furthermore, a dependence relation R_E in Definition 2.1.7 on page 26 is defined for each edge E in a PPN. For an edge E, R_E is specified as $R_E = \{\vec{I} \rightarrow \vec{J} \in \mathbb{Z}^{d1} \times \mathbb{Z}^{d2} \mid \vec{I} \in D_{IP} \wedge \vec{J} \in D_{OP} \wedge \vec{J} = B \cdot \vec{I} + \vec{c}\}$. It indicates that data produced at iteration $\vec{J} \in D_{OP}$ is consumed at iteration $\vec{I} \in D_{IP}$ if output port *OP* is connected to input port *IP* via edge E.

Consider the sequential C program given in Listing 1. The equivalent PPN that can be derived using the PNgen [125] compiler is shown in Figure 2.3. For the behavior of process snk, its process domain is given as

$$D_{snk} = \{(w, i, j) \in \mathbb{Z}^3 \mid w > 0 \wedge 1 \leq i \leq 10 \wedge 1 \leq j \leq 3\}. \tag{2.8}$$

Reading data tokens from input port IP_1 to initialize function argument `in1` of function `write_image` is represented as input port domain

$$D_{IP_1} = \{(w, i, j) \in \mathbb{Z}^3 \mid w > 0 \wedge 1 \leq i \leq 10 \wedge 1 \leq j \leq 2\}.$$

For edge E_5, the dependence relation R_{E_5} is expressed as

$$\begin{aligned} R_{E_5} = &\{(w1, i1, j1) \rightarrow (w2, i2, j2) \in \mathbb{Z}^3 \times \mathbb{Z}^3 \\ &\mid (w1, i1, j1) \in D_{IP_3} \wedge (w2, i2, j2) \in D_{OP_3} \wedge w1 = w2 \wedge i1 = i2 j1 = j2\}, \end{aligned}$$

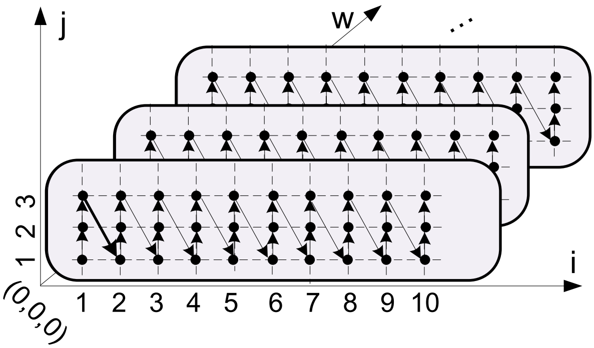
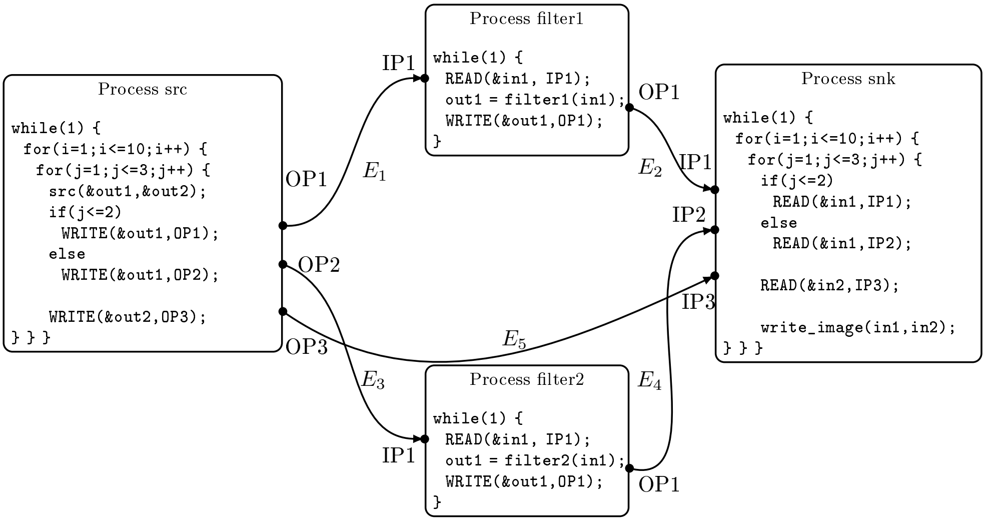


Figure 2.2: The polyhedral representation of the execution of function `read_image` in Listing 1.



```
while(1) {
  for(i=1;i<=10;i++) {
    for(j=1;j<=3;j++) {
      src(&out1,&out2);
      if(j<=2)
        WRITE(&out1,OP1);
      else
        WRITE(&out1,OP2);

      WRITE(&out2,OP3);
} } }
```

```
while(1) {
  READ(&in1, IP1);
  out1 = filter1(in1);
  WRITE(&out1,OP1);
}
```

```
while(1) {
  for(i=1;i<=10;i++) {
    for(j=1;j<=3;j++) {
      if(j<=2)
        READ(&in1,IP1);
      else
        READ(&in1,IP2);

      READ(&in2,IP3);

      write_image(in1,in2);
} } }
```

```
while(1) {
  READ(&in1, IP1);
  out1 = filter2(in1);
  WRITE(&out1,OP1);
}
```

Figure 2.3: PPN corresponding to the SANLP in Listing 1.

where $D_{OP_3} = D_{IP_3} = D_{snk}$.

2.2 Actor-based Data Flow MoCs

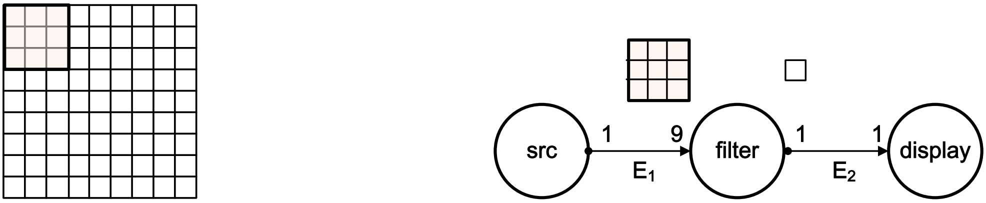
In this section, we give some important definitions concerning the SDF and CSDF MoCs. The related notations are listed in Table 2.3.

2.2.1 Synchronous Data Flow (SDF)

A Synchronous Data Flow (SDF) [75] graph G is defined as $G = (\mathcal{A}, \mathcal{E})$, where $\mathcal{A}$ is the set of actors and $\mathcal{E}$ is the set of edges. For each actor $A_i \in \mathcal{A}$, an execution is called *firing*. It produces/consumes a constant number of data *tokens* to/from edges, denoted by $prd \in \mathbb{N}^+$ and $cns \in \mathbb{N}^+$, respectively. As a special case, the MoC is called Homogeneous Synchronous Data Flow (HSDF) if $prd = cns = 1$ for all production/consumption rates and all actors. To be eligible to fire, each incoming edge E_j of an actor must contain at least cns_j tokens. In this thesis, we assume that auto-concurrent firing of actors are implicitly excluded. We also assume that all cns_j tokens are consumed at the beginning of a firing of an actor. At the end of the firing, all prd_k tokens are produced to each outgoing edge E_k. A token transfered through edges here refers to an atomic data object which can be either an integer or a complex data structure. Tokens are transfered in FIFO fashion. Let us consider for instance the image filter algorithm illustrated in Figure 2.4(a). Its corresponding SDF graph is shown in Figure 2.4(b). At the beginning of the firing, actor filter consumes $3 \times 3 = 9$ pixels from edge E_1 and produces 1 pixel to edge E_2 at the end of the firing.

One important advantage of the SDF MoC is that its functional properties, e.g., *consistency* and *deadlock-free*, can be verified at compile-time. Considering streaming applications which typically execute in a non-terminating fashion, both properties are important to ensure that a given SDF graph can execute indefinitely without causing unbounded token accumulation in FIFOs (buffer overflow), or deadlock. To verify consistency of an SDF graph, a balance equation [75] can be established as follows:

$$\Gamma_G \cdot \vec{q}_G = \vec{0}, \tag{2.9}$$



(a) filter operating on a 3 × 3 sliding window on the image from left to right and from top to bottom.

(b) An SDF graph G. FIFOs are not illustrated to avoid clutter.

Figure 2.4: An example of an image filter algorithm modeled using the SDF MoC.

A_i	actor
E_i	edge in data flow graph
prd	production rate
cns	consumption rate
PRD	production sequence
CNS	consumption sequence

Table 2.3: Data flow notations.

where Γ_G is called topology matrix and $\vec{q}_G$ is called *repetition vector*. Γ_G is defined as:

$$\Gamma_G = \begin{bmatrix} \Gamma_{1,1} & \cdots & \Gamma_{1,|\mathcal{A}|} \\ \vdots & \Gamma_{j,i} & \vdots \\ \Gamma_{|\mathcal{E}|,1} & \cdots & \Gamma_{|\mathcal{E}|,|\mathcal{A}|} \end{bmatrix} \tag{2.10}$$

with:

$$\Gamma_{j,i} = \begin{cases} prd_j & \text{if actor } A_i \text{ produces to edge } E_j \\ -cns_j & \text{if actor } A_i \text{ consumes from edge } E_j \\ 0 & \text{otherwise.} \end{cases} \tag{2.11}$$

In [75], it is shown that a connected SDF graph is consistent iff $\text{rank}(\Gamma_G) = |\mathcal{A}| - 1$, which ensures Γ_G has a 1-dimensional null space. That is, Equation (2.9) has a non-trivial solution for $\vec{q}_G$. To execute an SDF graph indefinitely with a periodic schedule without unbounded token accumulation, the consistency property is a necessary condition. Consider the SDF graph shown in Figure 2.4(b), its topology matrix Γ_G is given by

$$\Gamma_G = \begin{bmatrix} 1 & -9 & 0 \\ 0 & 1 & -1 \end{bmatrix}$$

Therefore, its repetition vector can be obtained as

$$\begin{aligned} \vec{q}_G &= [q_{src}, q_{filter}, q_{display}] \\ &= [9, 1, 1]. \end{aligned}$$

A consistent SDF graph may still deadlock due to insufficient amount of initial tokens. A SDF graph is said to be deadlocked if none of the actors is eligible to fire at certain point in time. To detect such a scenario, a periodic admissible schedule [75] can be constructed. If such a schedule does not exist, the SDF graph will deadlock during its execution. Finally, a consistent and deadlock-free SDF graph is said to be *live*. Only live SDF graphs are considered in this thesis.

2.2.2 Cyclo-Static Data Flow (CSDF)

A Cyclo-Static Data Flow (CSDF) [30] graph is similarly defined as $G = (\mathcal{A}, \mathcal{E})$, where $\mathcal{A}$ is the set of actors and $\mathcal{E}$ is the set of edges. CSDF generalizes the SDF MoC by introducing periodically changing token consumption and production rates, called *production/consumption sequence*, denoted by $PRD \in \mathbb{N}^{\phi}$ and $CNS \in \mathbb{N}^{\phi}$, respectively. The production/consumption sequences consist of ϕ *phases*. For the $x$the firing of an actor A_i, it consumes $CNS_j[((x-1) \mod \phi_i)+1]$ tokens from each incoming edge E_j and produces $PRD_k[((x-1) \mod \phi_i)+1]$ tokens to each outgoing edge E_k. PRD_k and CNS_j are defined as $PRD_k = [prd_1^k, \ldots, prd_{\phi}^k]$ and $CNS_j = [cns_1^j, \ldots, cns_{\phi}^j]$, respectively. The length of the production/consumption sequence may vary between CSDF actors. Note that auto-concurrent firing of CSDF actors are implicitly excluded as well.

Similar to the SDF MoC, the consistency of the CSDF MoC is also an important property. For a CSDF graph $G = (\mathcal{A}, \mathcal{E})$, the balance equation [30] is established as follows:

$$\Gamma_G \cdot \vec{r}_G = \vec{0}, \tag{2.12}$$

with

$$\Gamma_{j,i} = \begin{cases} \sum_{k=1}^{k=\phi_i} prd_k^j & \text{if actor } A_i \text{ produces to edge } E_j \\ -\sum_{k=1}^{k=\phi_i} cns_k^j & \text{if actor } A_i \text{ consumes from edge } E_j \\ 0 & \text{otherwise.} \end{cases} \tag{2.13}$$

Assuming $n = |\mathcal{A}|$, the repetition vector $\vec{q}_G = [q_1, \ldots, q_i, \ldots, q_n]$ is then given by

$$\vec{q}_G = Q \cdot \vec{r}_G \text{ with } Q = \mathbb{Z}^{n \times n} \text{ and } Q_{j,i} = \begin{cases} \phi_i & \text{if } j = i \\ 0 & \text{otherwise.} \end{cases} \tag{2.14}$$

ϕ_i is the length of consumption/production sequences of actor A_i. Again, only consistent and deadlock-free, namely live, CSDF graphs are considered in this thesis.

Consider the CSDF graph G_1 in Figure 2.5. The topology matrix of G_1 is given by

$$\Gamma_{G_1} = \begin{bmatrix} 1 & -40 & 0 & 0 \\ 0 & 1 & 0 & -1 \\ 0 & 0 & 1 & -66 \end{bmatrix},$$

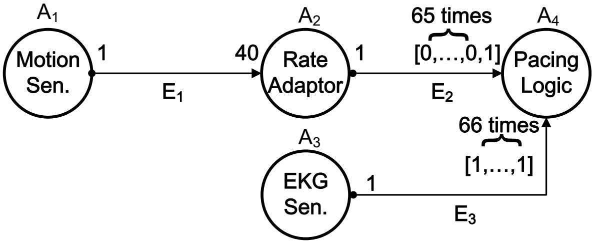


Figure 2.5: A CSDF graph G_1 of a pacemaker application (taken from [99]).

and Q in Equation (2.14) is given by

$$Q = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 66 \end{bmatrix}.$$

Thus, we can obtain

$$\begin{aligned} \vec{q}_G &= [q_1, q_2, q_3, q_4] \\ &= [40, 1, 66, 66]. \end{aligned}$$

2.3 Hard Real Time Scheduling of Acyclic (C)SDF Graphs

To find a schedule for CSDF graphs where certain performance constraints are guaranteed, periodic schedules are considered to be a common approach. We shall distinguish the periodic schedules considered here from the one defined in [30,75]. The periodic schedule considered in this thesis emphasizes on the fact that the schedule of each actor firing repeats in a strictly periodical way (see Definition 2.3.1). The authors in [22,31] have developed efficient techniques that can derive such Strictly Periodic Schedules (SPS) in polynomial time. Note that the periodic scheduling of the CSDF MoC can be also applied to the SDF MoC, since the CSDF MoC is the superset of the SDF MoC. In particular, the SPS framework for an acyclic CSDF graph developed in [22] is implemented in the Daedalus^{RT} design flow and thus it is considered in this thesis. To ease the discussion of the SPS concept, we use the notations listed in Table 2.4.

Definition 2.3.1 (Strictly Periodic Schedule (SPS)). A schedule of a CSDF graph $G = (\mathcal{A}, \mathcal{E})$ is said to be strictly periodic iff

$$\forall A_i \in \mathcal{A} \text{ and } x \in \mathbb{N}^+ \ : \ s_i(x) = S_i + (x-1)T_i, \tag{2.15}$$

where $s_i(x) \in \mathbb{N}^+$ denotes the xth release time of actor A_i, $S_i \in \mathbb{N}$ is the earliest starting time of A_i, and $T_i \in \mathbb{N}^+$ denotes the interval between two consecutive firings of A_i, called *period*.

Essentially, the actors of a CSDF graph under SPS are considered as a set of independent, real-time tasks with implicit deadlines [35]. Therefore, such a real-time task corresponding to a CSDF actor is associated with two parameters, namely period T and earliest starting time S, where the deadline of the task is equal to its period (i.e., implicit deadline).

The main advantage of SPS is that a variety of well-known HRT scheduling algorithms, such as Earliest Deadline First (EDF) [80] or Rate Monotonic (RM) [80], can be applied to temporally schedule CSDF actors allocated on a PE. Meanwhile, temporal isolation of different applications, i.e., different CSDF graphs, that share a single MPSoC platform can be achieved. Moreover, the required platform including the number of PEs and buffer sizes needed to schedule the CSDF graph can be determined in polynomial time.

Under SPS, a firing of a CSDF actor must finish before its deadline which is equal to its period. If the sink actor of a CSDF graph A_{snk} produces *prd* tokens per firing and has a period T_{snk}, the SPS thus guarantees a throughput $\frac{prd}{T_{snk}}$ for the CSDF graph. To compute the period of each actor, the following definition is needed first.

Definition 2.3.2 (Workload of an Actor)**.** The workload of a CSDF actor $A_i \in \mathcal{A}$ per graph iteration, denoted by W_i, is given by $W_i = q_i C_i$, where q_i is the repetition entry of A_i and C_i is the Worst Case Execution Time (WCET) of A_i.

Accordingly, the maximum workload per graph iteration, denoted by $\hat{W}_G$, is defined as $\hat{W}_G = \max_{A_i \in \mathcal{A}}(q_i C_i)$. The minimum period $\check{T}_i$ [22] of an actor A_i under SPS can be computed in linear time as

$$\check{T}_i = \frac{\mathrm{lcm}(\vec{q}_G)}{q_i} \left\lceil \frac{\hat{W}_G}{\mathrm{lcm}(\vec{q}_G)} \right\rceil, \tag{2.16}$$

C_i	Worst-case execution time of an actor A_i
T_i	guaranteed period of an actor A_i
H_i	iteration period of an actor A_i
u_i	utilization of an actor A_i
m	number of PEs

Table 2.4: Notations for HRT scheduling of CSDF MoCs.

where $\mathrm{lcm}(\vec{q}_G)$ is the least common multiple of all repetition entries $q_i \in \vec{q}_G$, and C_i is the WCET of firing a CSDF actor A_i. The minimum period of the sink actor for a CSDF graph determines the maximum throughout that this graph can achieve. To sustain a strictly periodic execution with the period derived by Equation (2.16), the earliest starting time $S_i \in \mathbb{N}$ [22] of an actor A_i can be obtained as

$$S_i = \begin{cases} 0 & \text{if } \mathrm{prec}(A_i) = \emptyset \\ \max_{A_k \in \mathrm{prec}(A_i)}(S_{k \to i}) & \text{otherwise,} \end{cases} \tag{2.17}$$

where $\mathrm{prec}(A_i)$ represents the set of predecessor actors of A_i and $S_{k \to i}$ is given by

$$\begin{aligned} S_{k \to i} = \min_{t \in [0, S_k + H]} \{ t : & \underset{[S_k, \max\{S_k, t\} + d)}{\mathrm{Prd}} (A_k, E_j) \geq \\ & \underset{[t, \max\{S_k, t\} + d]}{\mathrm{Cns}} (A_i, E_j),\ \forall d \in [0, H], d \in \mathbb{N} \}, \end{aligned} \tag{2.18}$$

where H is defined as an iteration period obtained by $H = q_i T_i$.
$\mathrm{Prd}_{[S_k, \max\{S_k, t\} + d)}(A_k, E_j)$ denotes the total number of tokens produced by actor A_k to edge E_j during the time interval $[S_k, \max\{S_k, t\} + d)$ and
$\mathrm{Cns}_{[t, \max\{S_k, t\} + d]}(A_i, E_j)$ denotes the total number of tokens consumed by actor A_i from edge E_j during the time interval $[t, \max\{S_k, t\} + d]$. In addition, edge E_j connects actors A_k and A_i.

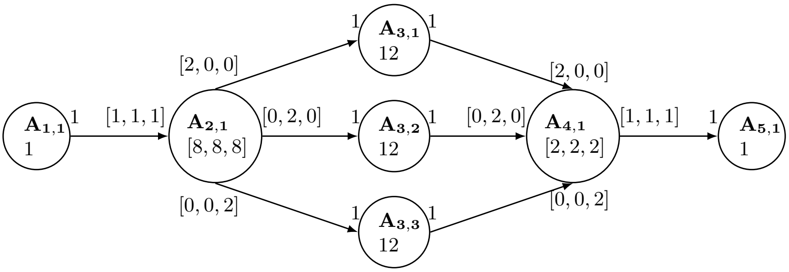
Let us consider the example of the acyclic CSDF graph G_2 in Figure 2.6(a). WCET C_i of each actor is given below the actor name A_i. We can first compute the repetition vector of G_2 in Figure 2.6(a) according to Equation (2.14) on page 32 as:

$$\begin{aligned} \vec{q}_{G_2} &= [q_{1,1}, q_{2,1}, q_{3,1}, q_{3,2}, q_{3,3}, q_{4,1}, q_{5,1}] \\ &= [3, 3, 2, 2, 2, 3, 3] \end{aligned}$$

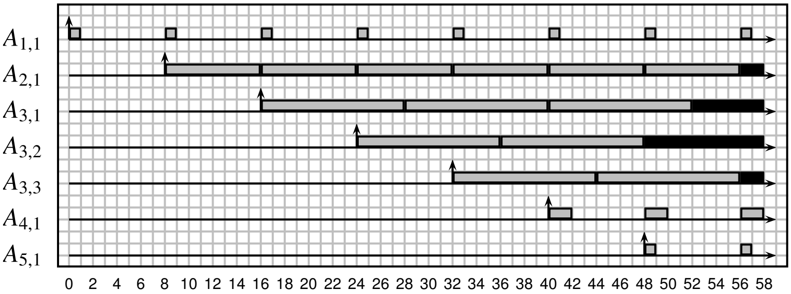
Under SPS, the period of each actor can be obtained using Equation (2.16) as:

$$\begin{aligned} \vec{\check{T}}_{G_2} &= [\check{T}_{1,1}, \check{T}_{2,1}, \check{T}_{3,1}, \check{T}_{3,2}, \check{T}_{3,3}, \check{T}_{4,1}, \check{T}_{5,1}] \\ &= [8, 8, 12, 12, 12, 8, 8] \end{aligned} \tag{2.19}$$

The periodic task-set representation of G_2 is illustrated in Figure 2.6(b). The x-axis represents time. The upper arrows indicate the earliest starting times of individual actors and the grey bars denote WCETs of actor firings. For the sake of discussion, Figure 2.6(b) only illustrates up to time unit 58 on the x-axis and the last firings of actors $A_{2,1}, A_{3,1}, A_{3,2}$ and $A_{3,3}$ are truncated. We can see in Figure 2.6(b) that, after the earliest starting time of each actor, the actor is scheduled in a strictly periodic



(a) A CSDF graph G_2. $A_{1,1}$ and $A_{5,1}$ are considered as the source and sink actors, respectively.



(b) Periodic task-set representation of G_2.

Figure 2.6: An example of a CSDF graph and its real-time task-set representation. Since the execution of the actors repeats indefinitely, the last execution of $A_{2,1}$,$A_{3,1}$,$A_{3,2}$, and $A_{3,3}$ in the figure is truncated and shown in black.

way. For instance, actor $A_{5,1}$ has the earliest starting time $S_{5,1} = 48$. After that, each firing of $A_{5,1}$ occurs every $\check{T}_{5,1} = 8$ time units. Given that $A_{5,1}$ has no outgoing edges and thus it is the sink actor of G_2. Therefore, the maximum throughput of G_2 is $\frac{1}{8}$.

Once periods and earliest starting times of all actors in an acyclic CSDF are derived, the next step is to determine the number of required PEs to schedule the actors and to guarantee that the deadlines (equal to derived periods) of actors are met. To this end, the SPS framework leverages extensively the results from the HRT scheduling theory. Here we only give a brief overview of the HRT scheduling theory that is relevant to this thesis. For the complete treatment of the HRT scheduling topic, please refer to [33]. First, the notion of *utilization* needs to be introduced. Let $G = (\mathcal{A}, \mathcal{E})$ be a CSDF graph, the period of a CSDF actor $A_i \in \mathcal{A}$ be T_i, and WCET of A_i be C_i. The utilization of A_i, denoted by u_i, can be computed as

$$u_i = \frac{C_i}{T_i}, \;\; \forall A_i \in \mathcal{A}. \tag{2.20}$$

For instance, using the EDF scheduling algorithm, a set of n actors is schedulable on a PE if the following equation is satisfied [80]:

$$\sum_{i=1}^{n} u_i \leq 1. \tag{2.21}$$

If migration of CSDF actors across PEs is allowed at run-time (global scheduling), the number of required PEs $M(G)$ for a CSDF graph G can be simply computed as

$$M(G) = \left\lceil \sum_{A_i \in \mathcal{A}} u_i \right\rceil. \tag{2.22}$$

In case that no migration of CSDF actors is allowed at run-time (partitioned scheduling), determining the number of required PEs is thus equivalent to the bin-packing problem and can be solved by either *exact* or *approximate* allocation algorithms. An example of an exact allocation algorithm is proposed in [83], which returns an optimal allocation of actors. One disadvantage of using an exact algorithm is its high computational complexity. Therefore, to have a trade-off between optimality of the allocation and computational complexity, an approximate allocation algorithm such as the First-Fit Decreasing (FFD) algorithm [61] can be considered. Let $M_{FFD}(G)$ denote the number of PEs needed for a CSDF graph G under FFD and $M_{OPT}(G)$ denote the number of PEs needed for G using an exact allocation algorithm. It is proven in [134] that the following inequality holds:

$$M_{FFD}(G) \leq \frac{11}{9} M_{OPT}(G) + 1. \tag{2.23}$$

Once allocation of a CSDF graph is determined, the schedule on each PE itself can be built either off-line for efficiency, or on-line for flexibility according to the system requirements.

Let us consider CSDF graph G_2 in Figure 2.6(a). Given $\vec{\check{T}}_{G_2}$ in Equation (2.19), we obtain

$$M(G_2) = \lceil \frac{1}{8} + \frac{8}{8} + \frac{12}{12} + \frac{12}{12} + \frac{12}{12} + \frac{2}{8} + \frac{1}{8} \rceil = 5. \tag{2.24}$$

That is, 5 PEs are required to schedule G_2 using the EDF algorithm and to achieve the maximum throughput of $\frac{1}{8}$.

Chapter 3

Automated Analysis Model Construction: Deriving CSDF from Equivalent PPN

Mohamed A. Bamakhrama, **Jiali Teddy Zhai**, Hristo Nikolov, Todor Stefanov, "A Methodology for Automated Design of Hard-Real-Time Embedded Streaming Systems", *In Proceedings of the Conference on Design, Automation and Test in Europe (DATE'12)*, pp. 941–946, Dresden, Germany, March 12-16, 2012.

In this chapter, we present an approach to convert a PPN to its input-output equivalent CSDF graph. As discussed previously, a wide range of powerful analysis techniques exist for the CSDF MoC, whereas it is easier to generate code for the PPN MoC. Considering the Daedalus^{RT} design flow shown in Figure 1.7 on page 13, deriving PPNs from SANLPs can be done in the PNgen compiler [125]. The code generation for the PPN MoC has been addressed in the ESPAM [96] tool.

Notation	Meaning
α	an input/output argument for a PPN function
$\mathcal{S}$	a sequence
Φ	a set of input/output ports associated with a process variant
v	a process variant

Table 3.1: Additional notations used in Chapter 3 besides the ones introduced in Chapter 2.

Hard real-time scheduling of acyclic CSDF graphs has been proposed in [22]. To have a fully automated design flow for designing hard real-time streaming systems, automated derivation of the CSDF MoC is the only missing step. From a high-level point of view, the contribution of this chapter enables to derive the equivalent CSDF graph from any SANLP.

It has been shown in [37] that a PPN without parameters is equivalent to a CSDF graph where the production/consumption sequences consist only of 0s and 1s. A '0' indicates that a token is not produced/consumed, whereas a '1' indicates that a token is produced/consumed. This chapter focues on the algorithm to derive the input-output equivalent CSDF graph from a PPN. Then, we demonstrate the applicability of the algorithm on a set of benchmarks in terms of time complexity. Finally, in the context of the Daedalus^{RT} design flow, we show that it is fast to derive CSDF graphs for three real-life streaming applications. Consequently, derivation of CSDF graphs enables to design multiple streaming applications on a single MPSoC platform in a short amount of time.

In addition to the notations introduced in Chapter 2, extra notations used in this chapter are summarized in Table 3.1.

3.1 The Algorithm

The procedure to derive a CSDF graph from its equivalent PPN is depicted in Algorithm 1. It consists of two main steps, namely 1) topology derivation and 2) consumption/production sequence derivation for input/output ports of each CSDF actor. Deriving the topology of the CSDF graph is straightforward. That is, the nodes, input/output ports, and edges in the CSDF graph have one-to-one correspondence to those in the PPN. Recall the SANLP described in Listing 1 on page 27 and its equivalent PPN shown in Figure 2.3. The derived CSDF graph is shown in Figure 3.1. It can be seen that the topology of the derived CSDF graph is the same as that of the PPN shown in Figure 2.3. Below, we focus on the second step which derives the consumption/production sequences for an input/output port of a CSDF actor.

The second step consists of three sub-steps. In the first sub-step (see line 3 in Algorithm 1), for each CSDF actor, we find the access pattern of the corresponding PPN process to its input/output ports. A more general MoC, Stream-based Functions [68], captures the regular access pattern of a function to its input/output ports using *function variant*. In this thesis, we introduce the notion of *process variant*, which captures the consumption/production behavior of the process.

Definition 3.1.1 (Process Variant). A process variant v of a PPN process is defined by a tuple (D_v, Φ), where D_v is the variant domain with $D_v \subseteq D_P$, and Φ is a set of

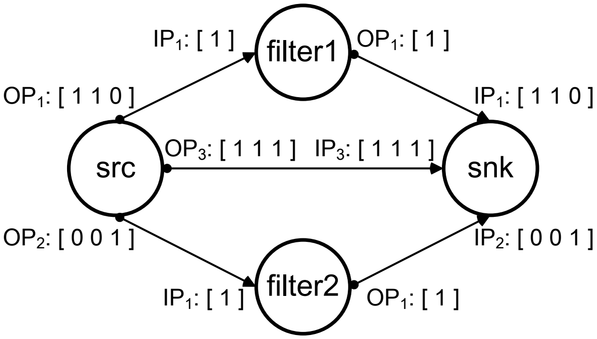


Figure 3.1: CSDF graph equivalent to the PPN shown in Figure 2.3.

```
Algorithm 1: Procedure to derive the CSDF MoC
Input: A PPN
Result: The equivalent CSDF graph
Derive the topology of the CSDF graph;
foreach CSDF actor in the CSDF graph do
    Derive process variants (see Definition 3.1.1) for its corresponding PPN
    process ;
    Derive a repetitive pattern of process variants ;
    foreach input/output port of the CSDF actor do
        foreach process variant in the derived pattern do
            Generate consumption/production sequence ;
```

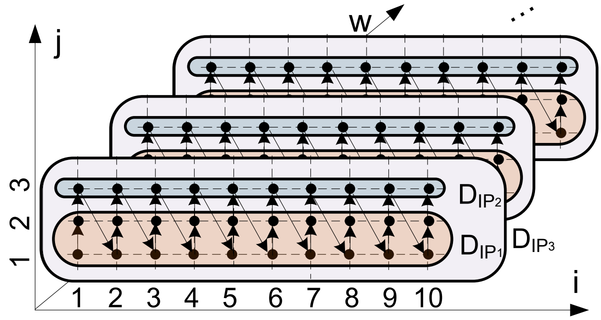
input/output ports.

For example, consider process snk shown in Figure 2.3 on page 29. One of the process variants is $(D_v, \{IP_1, IP_3\})$, where

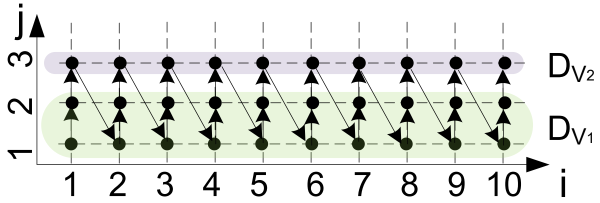
$$D_v = \{(w,i,j)\in\mathbb{Z}^3 \mid w \geq 0 \wedge 1 \leq i \leq 10 \wedge 1 \leq j \leq 2\}.$$

According to Definition 3.1.1, for all iterations in domain D_v during the execution of process snk, this process always reads data from input ports IP_1 and IP_3.

The infinite repetitive execution of a PPN process is initially represented by a polyhedron. (e.g., see D_{snk} in Equation (2.8)). Therefore, we project out dimension w which denotes the `while`-loop from all the domains because it is irrelevant for the subsequent steps. As a result, the execution of a PPN process is represented by a polytope. Algorithm 2 is devised to derive the process variants for each PPN process.



(a) Port domains



(b) Variant domains

Figure 3.2: Domains of process snk in Figure 2.3.

Standard integer set operations are applied to the process domains. The basic idea is that, each port domain bound to a process function argument is intersected with all other port domains. The intersected domain and the difference between two port domains are then added to the set of process variants. In this way, all process variants are iteratively derived.

Consider process snk in Figure 2.3 on page 29. Its process function `snk(in1, in2)` has two arguments represented as a set $\mathcal{R} = \{\texttt{in1}, \texttt{in2}\}$, which is the input to Algorithm 2. The port domains bound to `in1` are D_{IP_1} and D_{IP_2}, while the port domain bound to `in2` is D_{IP_3}. These port domains are illustrated in Figure 3.2(a), surrounded by bold lines. Following the procedure described in Algorithm 2, we start with projecting out dimension w in the port domains, which yields:

$$\begin{aligned}
\texttt{in1}: D'_{IP_1} &= \{(i,j) \in \mathbb{Z}^2 \mid 1 \leq i \leq 10 \wedge 1 \leq j \leq 2\},\\
D'_{IP_2} &= \{(i,j) \in \mathbb{Z}^2 \mid 1 \leq i \leq 10 \wedge j = 3\},\\
\texttt{in2}: D'_{IP_3} &= \{(i,j) \in \mathbb{Z}^2 \mid 1 \leq i \leq 10 \wedge 1 \leq j \leq 3\}.
\end{aligned}$$

Algorithm 2 produces the set of process variants $\mathcal{V} = \{v_1, v_2\}$, where

$$\begin{aligned} v_1 &= (D_{v_1}, \{IP_1, IP_3\}), \\ D_{v_1} &= D'_{IP_1} \cap D'_{IP_3} = \{(i,j) \in \mathbb{Z}^2 \mid 1 \le i \le 10 \wedge 1 \le j \le 2\}, \\ v_2 &= (D_{v_2}, \{IP_2, IP_3\}), \\ D_{v_2} &= D'_{IP_2} \cap D'_{IP_3} = \{(i,j) \in \mathbb{Z}^2 \mid 1 \le i \le 10 \wedge j = 3\}. \end{aligned}$$

Process variant domains D_{v_1} and D_{v_2} are also illustrated in Figure 3.2(b). Process snk reads data from input ports IP_1 and IP_3 in variant domain D_{v_1}, whereas it reads data from input ports IP_2 and IP_3 in variant domain D_{v_2}.

In the second sub-step, (see line 4 in Algorithm 1), we find a one-dimensional, repetitive pattern of the process variants derived in the first sub-step. To find the repetitive pattern, we first project out dimension w in the process domain D_P to obtain domain D'_P. For a PPN process P, we build a sequence $\mathcal{S}_{D'_P}$ of the iterations $I \in D'_P$ according to their lexicographic order (see Definition 2.1.6 on page 25) as follows:

$$\mathcal{S}_{D'_P} = [I_1, \ldots, I_i, I_j, \ldots, I_{|D'_P|}],$$

where

$$I_i \prec I_j, \;\; \forall 1 < i < j < |D'_P|.$$

Next, we replace each iteration in sequence $\mathcal{S}_{D'_P}$ with the process variant to which the iteration belongs. For a PPN process P, the sequence of process variants $\mathcal{S}_P$ is given by

$$\mathcal{S}_P = [v_1, \ldots, v_i, \ldots, v_{|D'_P|}] \text{ and } I_i \in v_i, \;\; \forall 1 < i < |D'_P|.$$

For example, for process snk in Figure 2.3 and D_{snk} given in Equation (2.8) on page 28, the process domain after projecting out the w dimension is given as

$$D'_{snk} = \{(i,j) \in \mathbb{Z}^2 \mid 1 \le i \le 10 \wedge 1 \le j \le 3\}. \tag{3.1}$$

The sequence of the iterations in process domain D'_{snk} is

$$\mathcal{S}_{D'_{snk}} = [(1,1), (1,2), (1,3), (2,1), \ldots, (10,3)].$$

The lexicographic order of iterations in the sequence is represented using the arrows in Figure 3.2(b). The corresponding sequence of the process variants of process snk is

$$\begin{aligned} \mathcal{S}_{snk} = [&v_1, v_1, v_2, v_1, v_1, v_2, v_1, v_1, v_2, v_1, v_1, v_2, v_1, v_1, v_2, \\ &v_1, v_1, v_2, v_1, v_1, v_2, v_1, v_1, v_2, v_1, v_1, v_2, v_1, v_1, v_2]. \end{aligned} \tag{3.2}$$

Algorithm 2: Procedure to derive process variants of a process

```
Input: $\mathcal{R}$: the set of process function arguments
Result: $\mathcal{V}$: a set of process variants
$\mathcal{V} \leftarrow \emptyset$;
foreach $\alpha \in \mathcal{R}$ do
    foreach $D_{port}$ bound to $\alpha$ do
        $y \leftarrow (D_{port}, \{port\})$;
        if $\mathcal{V} = \emptyset$ then
            $\mathcal{V} \leftarrow \{y\}$;
        else
            $\mathcal{X} \leftarrow \mathcal{V}$;
            foreach $V \in \mathcal{V}$ do
                $D_{intersect} \leftarrow v.D_{port} \cap y.D_{port}$;
                if $D_{intersect} \neq \emptyset$ then
                    $x_{intersect} \leftarrow (D_{intersect}, \{v.ports\} \cup \{y.ports\})$;
                    $x_{diff1} \leftarrow (v.D_{port} - y.D_{port}, \{v.ports\})$;
                    $x_{diff2} \leftarrow (y.D_{port} - v.D_{port}, \{y.ports\})$;
                    $\mathcal{X} \leftarrow \mathcal{X} \cup \{x_{intersect}\}$;
                    if $x_{diff1}.D_{port} \neq \emptyset$ then
                        $\mathcal{X} \leftarrow \mathcal{X} \cup \{x_{diff1}\}$;
                    if $x_{diff2}.D_{port} \neq \emptyset$ then
                        $\mathcal{X} \leftarrow \mathcal{X} \cup \{x_{diff2}\}$;
                else
                    $\mathcal{X} \leftarrow \mathcal{X} \cup \{y\}$;
            $\mathcal{V} \leftarrow \mathcal{X}$;
foreach $v \in \mathcal{V}$ do
    if $|IP \in v.ports| \neq |\alpha_{in} \in \mathcal{R}|$ then
        $\mathcal{V} \leftarrow \mathcal{V} \setminus v$;
```

Essentially, the length of the derived sequence is equal to the cardinality of process domain D'_p of a PPN process P, i.e., $|D'_p|$. Since $|D'_p|$ can be very large, the derived sequence might be very long. Thus, we express the sequence using the shortest repetitive pattern that covers the whole sequence. This shortest repetitive pattern can be found efficiently using a data structure called *suffix tree* [119]. In a suffix tree, the *root* node is defined as the node with only outgoing edges and the *leaf* nodes are defined as the nodes with only incoming edges. The remaining nodes are called *internal*. A suffix tree has the following properties:

- A suffix tree for a sequence $\mathcal{S}$ of characters can be built in $O(|\mathcal{S}|)$ time [119].
- Each edge in the suffix tree is labeled with a non-empty subsequence starting from character $\mathcal{S}[i]$ to character $\mathcal{S}[j]$, where $1 \leq i \leq j \leq |\mathcal{S}|$.
- No two edges out of a node in the tree can have labels beginning with the same character. That is, the starting character of the label is different for all outgoing edges of a node in the suffix tree.
- A subsequence obtained by concatenating all subsequences found on the path from the root node to any internal node i occurs k times in the whole sequence, where k is the number of leaf nodes that node i has.
- The suffix tree is padded with a terminal symbol \$.

Once a suffix tree is constructed for the sequence of process variants $\mathcal{S}_P$ according to the algorithm presented in [119], our problem can be formulated as: search the tree for the shortest repetitive pattern that covers the whole sequence $\mathcal{S}_P$, i.e., a subsequence of $\mathcal{S}_P$. Our problem can be solved based on finding the longest repeated substring in a given string, which can be solved in linear time. In our problem, we first pre-process the constructed suffix-tree to count the number of leaf nodes for each internal node. Among all outgoing edges of the root node, only the branch that has the same starting process variant as $\mathcal{S}_P$ is selected to explore. Then, a Breadth First Search (BFS) procedure is used, because shorter subsequences found at the levels closer to the root node are more likely to be the solution of our problem. For every path starting from the root to any internal node, the BFS procedure concatenates the labels on the edges. This concatenation results in a subsequence $\mathcal{S}_{sub}$ which occurs k times in the original sequence $\mathcal{S}_P$. Finally, we select the subsequence $\mathcal{S}_{sub}$ with the largest occurrence k that satisfies

$$|\mathcal{S}_{sub}| \times k = |\mathcal{S}_P|. \tag{3.3}$$

Similar to the longest repeated substring problem, our problem also has the linear time complexity.

Recall that the sequence of process variants $\mathcal{S}_{snk}$ for process snk is given in Equation (3.2). The corresponding suffix tree is constructed and illustrated in Figure 3.3. The shadow node denotes the root node and the solid nodes denote the leaf nodes. The others are internal nodes. It can be seen that every edge is labeled

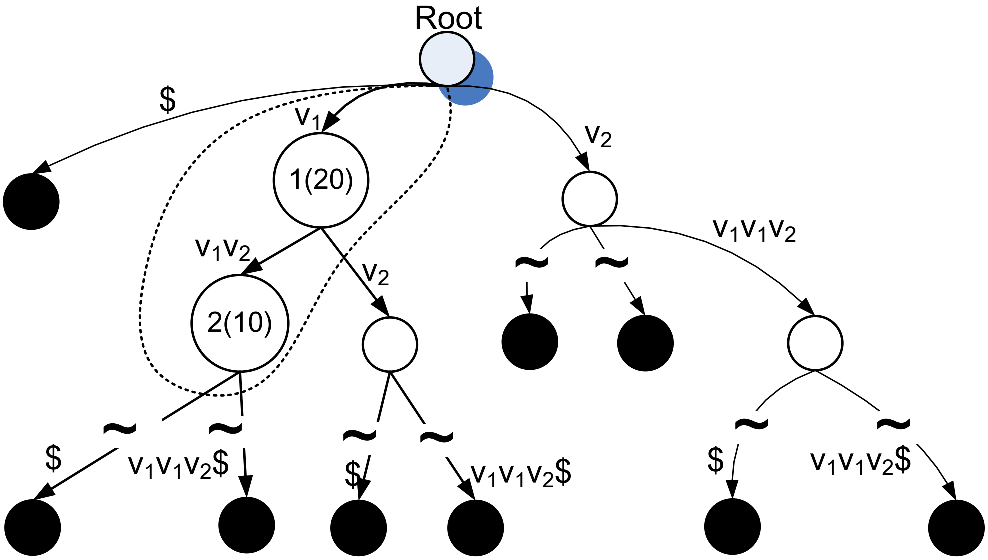


Figure 3.3: Suffix tree for the sequence of process variants $\mathcal{S}_{snk}$. The tildes represent the omitted part of the tree.

with a subsequence of process variants that occurs in the whole sequence $\mathcal{S}_{snk}$. In the pre-processing, computing for instance the number of leaf nodes for node 1 results in 20 (shown in the bracket in node 1). It means, the subsequence v_1 occurs in $\mathcal{S}_{snk}$ 20 times. In the beginning of BFS, only the edge connecting the root node to node 1 is selected to explore, because process variant v_1 labeled on the edge is the same as the first process variant in sequence $\mathcal{S}_{snk}$. In the next step, node 2 is selected and v_1 is concatenated with v_1v_2 labeled on the edge connecting node 1 and node 2. It yields

$$\begin{aligned} \mathcal{S}_{sub} &= [v_1, v_1, v_2]. \\ |\mathcal{S}_{sub}| \times 10 &= 30, \\ |\mathcal{S}_{snk}| &= 30. \end{aligned}$$

$\mathcal{S}_{sub}$ is shown in Figure 3.3 surrounded by a dashed line. At this step, the procedure terminates because the criteria, namely Equation (3.3), is satisfied.

In the last sub-step (see lines 5-7 in Algorithm 1), a consumption/production sequence is generated for each port of a CSDF actor. This is done by building a table in which each row corresponds to an input/output port, and each column corresponds to a process variant in the repetitive pattern derived in the second sub-step. If the input/output port is in the set of ports of the process variant, then its entry in the table is 1. Otherwise, its entry is 0. Each row in the resulting table represents a consumption/production sequence for the corresponding input/output port.

		Repetitive pattern		
		v_1	v_1	v_2
Input/output ports	IP_1	1	1	0
	IP_2	0	0	1
	IP_3	1	1	1

Table 3.2: Consumption/production sequences for actor snk in Figure 3.1.

Considering process snk, the consumption/production sequences of CSDF actor snk are generated as shown in Table 3.2. It can be seen that the consumption/production rates sequences for the ports are the same as the ones shown in Figure 3.1.

3.2 Experimental Results

We present in this section the experimental results of automated deriving CSDF graphs for some real-life applications specified as SANLPs (see Definition 2.1.8 on page 26). The main focus here is to demonstrate the applicability of our approach in terms of time-complexity. Then, we further demonstrate the application of automated CSDF derivation in the context of our Daedalus$^{\mathrm{RT}}$ framework for designing hard real-time streaming systems.

We took two applications, Filterbank and FMRadio, with original C code available from the StreamIT benchmark suit [54] and several reasonably complex[1] benchmarks from Polybench [101]. The characteristics of all benchmarks are shown in columns 2-4 in Table 3.3. For example, the Filterbank benchmark contains 367 lines of code in the SANLP. This excludes the code for each function in the SANLP. We believe that it is reasonably complex to express high-level behavior for most of real-life applications. Different benchmarks also vary in complexity of the access pattern to data arrays. For example, the ADI benchmark has very complex access pattern. Complex access pattern potentially increases the length of derived production/consumption sequences for input/output ports of CSDF actors. Our algorithm presented in this chapter was coded in C++ and integrated in PNtools as part of Daedalus$^{\mathrm{RT}}$ shown in Figure 1.7 on page 13. All experiments were conducted on an Intel Core 2 Duo T9600 CPU running at 2.80 GHz with 4GB memory in Linux Kubuntu 10.4.

The last column in Table 3.3 shows the running time needed to derive the corresponding CSDF graph for each benchmark. We can see that our algorithm is able to derive CSDF graphs in short amount of time. Note that the running time

---

[1] Other benchmarks either have simpler and less data dependencies or less number of tasks than the ones we selected.

Table 3.3: Characteristics of benchmarks and running times to derive their corresponding CSDF graphs.

Benchmarks	No. of actors	No. of channels	Lines of code (in SANLP)	Running time (sec.)
Filterbank	69	89	367	1.60
FMRadio	28	39	195	0.66
ADI[1]	28	167	209	7.26
FDTD[2]	17	71	144	0.89
Gauss[3]	11	26	75	7.82
Gram-Schmidt	9	20	48	1.85
Regularity detector	8	11	54	2.86

[1] ADI: Alternating direction implicit solver
[2] FDTD: 2D finite difference time domain kernel
[3] Gauss: 2D gauss blur filter for image processing

Table 3.4: Execution times of the phases in the Daedalus^{RT} flow for three streaming applications on a single MPSoC platform.

Phase	Time	Automation (Yes/No)
Parallelization	0.48 sec.	Yes
WCET analysis	1 day	No
Deriving the CSDF graphs	**5 sec.**	**Yes**
Deriving the platform/mapping	0.03 sec.	Yes
System synthesis	2.16 sec.	Yes
Total	$\sim$ 1 day	-
Total (excl. WCET analysis)	$\sim$ 7.67 sec.	-

here includes the time starting from SANLPs to CSDF graphs. In addition, the time to derive the implementation model, i.e., PPNs, using the PNgen compiler is also included. In this way, we can see clearly the benefits of starting from a SANLP and resulting in its equivalent CSDF graph. As mentioned previously, our approach can be readily integrated into, e.g., the $\sum$C toolchain [21], to greatly speedup application development process.

In the second experiment, we took three streaming applications specified in SANLPs, an edge-detection filter (Sobel) from the image processing domain, the Motion JPEG (M-JPEG) video encoder from the video processing domain, and the M-JPEG video decoder. Using the Daedalus^{RT} framework, we generated a functional implementation that can be synthesized in Xilinx Platform Studio 13.2 targeting

Xilinx Virtex-6 FPGA ML605 evaluation kit [15]. Table 3.4 shows the running time of each design phase in the Daedalus$^{\mathrm{RT}}$ design flow. We observe that, if the CSDF graphs of the three applications were derived manually by hand, it would take several days. Instead, using the solution presented in this chapter, deriving these three CSDF graphs takes 5 seconds. Thus, the automated CSDF derivation is one of the key enablers for a fully automated and fast design flow.

Chapter 4

Exploiting Maximum Data-level Parallelism without Inter-processor Communication

Jiali Teddy Zhai, Hristo Nikolov, and Todor Stefanov, "Mapping of Streaming Applications considering Alternative Application Specifications", in ACM Transactions on Embedded Computing Systems (TECS), vol. 12, Issue 1s, Article 34, March 2013.
Jiali Teddy Zhai, Hristo Nikolov, Todor Stefanov, "Mapping Streaming Applications considering Alternative Application Specifications (Extended Abstract)", *In Proceedings of the 10th IEEE Symposium on Embedded System for Real-Time Multimedia (ESTIMedia'12)*, Tampere, Finland, October 11-12, 2012.

As explained in Section 1.1.4, during the synthesis step of a model-based design methodology, all possible combinations of Processing Element (PE) allocation and assignment of application tasks to PEs constitute an enormous design space. To efficiently search the design space and find an optimum mapping solution, existing DSE approaches search the design space using different algorithms, e.g., stepwise refinement in [51], heuristics in [109] and [111], evolutionary algorithms in [100, 115], branch-and-bound in [34], and constraint programming in [139]. These DSE approaches consider only a single application specification given by application designers.

As mentioned in Section 1.2, an application specification given by application designers often does not take into account the underlying computation and communication capability of an MPSoC. Indeed, the authors in [71] showed that, for a set of representative streaming benchmarks, the theoretical speedup of mapping the initial parallel application specifications, given by the application designer, can only reach up to a limited number.

The discussion above indicates that alternative application specifications may be needed for efficient mapping of an application. In this thesis, we consider an alternative application specification as a different description of the same application behavior using the same MoC. For the same application behavior, there exists a large number of alternative specifications. Among them, the considered application specification should be the one that best matches the underlying MPSoC platform. Ideally, the best application specification, if it exists, to be mapped onto n PEs is the one that consists of n independent and load-balanced tasks. Then, without complex DSE, mapping these n tasks onto n PEs will always result in n times speedup. In this case, all PEs are equally loaded and 100% utilized without the need to synchronize and communicate data with each other.

In this chapter, we study the problem of whether an alternative PPN exists for an initial PPN, which consists of only independent and load-balanced processes. Specifically, we divide the problem into two stages. In the first stage, we analytically identify independent execution of PPN processes, called *communication-free partitions*. If they exist, the initial PPN is automatically transformed to a set of communication-free partitions, i.e., an alternative PPN. In the second stage, given n PEs, the application mapping problem is considered as grouping the set of obtained communication-free partitions to balance the application workloads across all PEs, such that the resulting performance (i.e., throughput) is maximized. To achieve the load-balancing, any existing DSE algorithm can be leveraged. As a result, mapping an application using this alternative PPN leads to better performance than mapping the initial PPN.

Scope of Work

In this chapter, we consider streaming applications which can be modeled using the PPN MoC. We assume that there are only one source and sink processes respectively and they are orders of magnitude faster than the remaining processes that perform computation. The source and sink processes represent environment and are thus not partitioned. Furthermore, the achievable performance of a PPN is not constrained by the buffer size required for each communication edge. It is possible to compute a buffer size for each PPN edge using the PNgen compiler such that larger buffer sizes do not increase the performance. We statically allocate a FIFO buffer for each PPN edge on target platforms. The target platforms considered in this chapter are homogeneous MPSoCs consisting of programmable PEs interconnected via any type of communication infrastructure. After communication-free partitioning, we assume that one partition completely fits onto one PE, in terms of program and data memory usage.

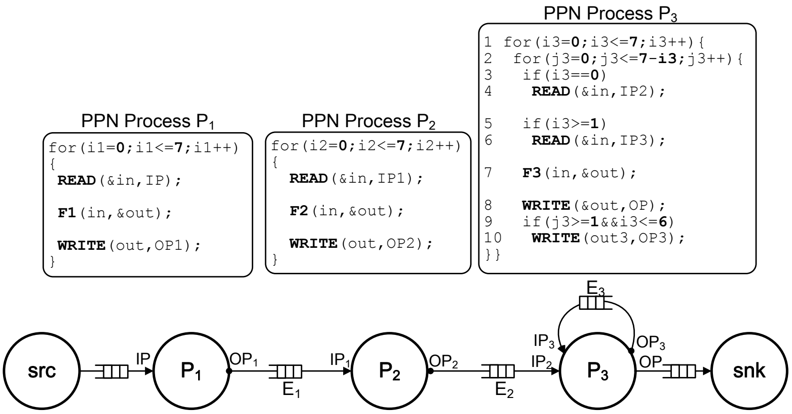
4.1 Motivating Example

To demonstrate the importance and usefulness of considering alternative application specifications, let us consider an example application modeled using the PPN MoC shown in Figure 4.1(a). This example is used throughout this chapter as a running example. The PPN represents a common topology of a parallel application specification and consists of three PPN processes P_1, P_2, and P_3 communicating data via FIFO edges. Note that P_3 has cyclic data dependences through edge E_3. The behavior of each PPN process is given as C code above the corresponding process. Besides the PPN processes expressing the application behavior, the processes src and snk represent the environment which provides input data and collects results. Their execution is expressed by two domains, namely $D_{src} = D_{IP}$ and $D_{snk} = D_{OP} = D_3$. For instance, D_{snk} is given as

$$D_{snk} = \{(i3, j3) \in \mathbb{Z}^2 \mid 0 \leq i3 \leq 7 \wedge 0 \leq j3 \leq 7 - i3\}. \tag{4.1}$$

Suppose that processes src and snk are much faster than the PPN processes and the PPN is to be mapped onto the platform shown in Figure 4.2. The workloads of functions F1, F2, and F3 in Figure 4.1(a) on the PEs are 6, 100, and 30 time units, respectively. Communication latency via the interconnection structure is assumed to be 5 time units and communication latency through local memory is considered as negligible. Naturally, the maximum performance of mapping the initial PPN can be achieved if each PPN process is mapped onto a separate PE, namely 3 PEs in this example. In case that more than 3 PEs are available, the existing DSE approaches are incapable of exploring the mapping possibilities that utilize all PEs. Thus, further performance improvements of the system are not explored. In fact, considering only the initial PPN shown in Figure 4.1(a), only 2 PEs are required to achieve the maximum performance if we perform DSE to obtain pareto-optimal mappings of processes. That is, processes P_1 and P_2 are pipelined and mapped onto PE1, while process P_3 is mapped onto PE2 as shown in Figure 4.2. Figure 4.3 shows the achieved speedup of pareto-optimal mappings of the initial PPN (denoted as *Initial*).

However, more parallelism is exposed and higher performance can be achieved, if the initial PPN is transformed to a set of communication-free partitions. A communication-free partition corresponds to a subset execution of PPN processes to produce an output of the PPN, without the need to communicate data with other partitions. To illustrate communication-free partitions, the execution of each PPN process in Figure 4.1(a) is visualized in Figure 4.1(b). The dots represent individual iterations of the PPN processes. For example, one iteration of P_3 comprises one execution of its loop body (lines 3 - 10 of P_3 in Figure 4.1(a)). The arrows between iterations denote data dependences. For this example, the initial PPN can be transformed to 8 communication-free partitions denoted as *Parti. 0 - 7* in Figure 4.1(b)



PPN Process P_1

```
for(i1=0;i1<=7;i1++)
{
 READ(&in,IP);

 F1(in,&out);

 WRITE(out,OP1);
}
```

PPN Process P_2

```
for(i2=0;i2<=7;i2++)
{
 READ(&in,IP1);

 F2(in,&out);

 WRITE(out,OP2);
}
```

PPN Process P_3

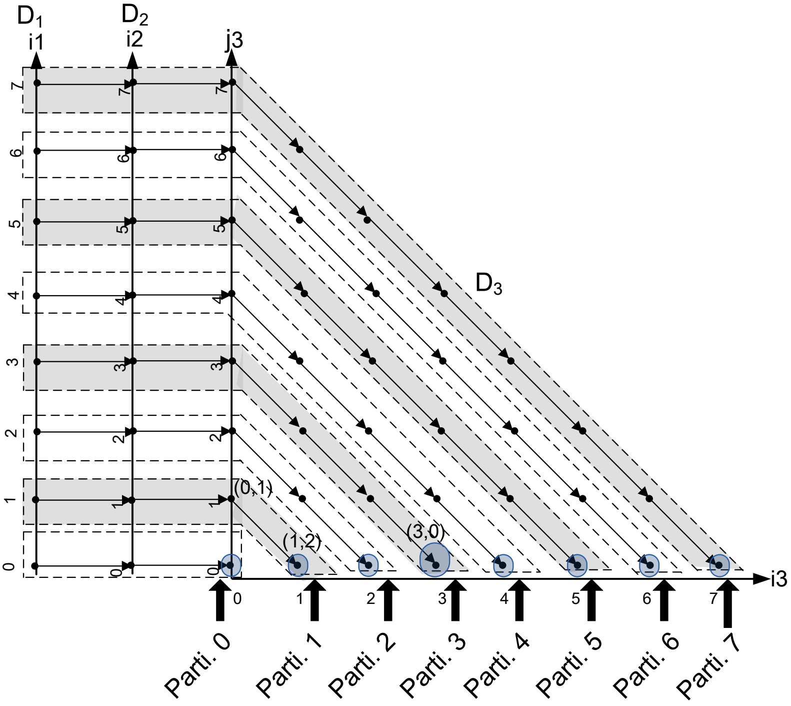
```
 for(i3=0;i3<=7;i3++){
  for(j3=0;j3<=7-i3;j3++){
   if(i3==0)
    READ(&in,IP2);

   if(i3>=1)
    READ(&in,IP3);

   F3(in,&out);

   WRITE(&out,OP);
   if(j3>=1&&i3<=6)
    WRITE(out3,OP3);
 }}
```

(a) The PPN and behavior of each process shown using C code.



(b) Process domains of all PPN processes and 8 communication-free partitions.

Figure 4.1: An example of a PPN and its communication-free partitions.

(each partition is surrounded by a dashed box). One can see in Figure 4.1(b) that no arrows (data dependences) exist across the partitions. Each partition contains a subset

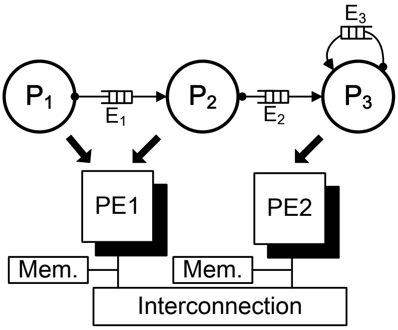


Figure 4.2: Mapping of the PPN in Figure 4.1(a) onto 2 PEs achieving the maximum performance.

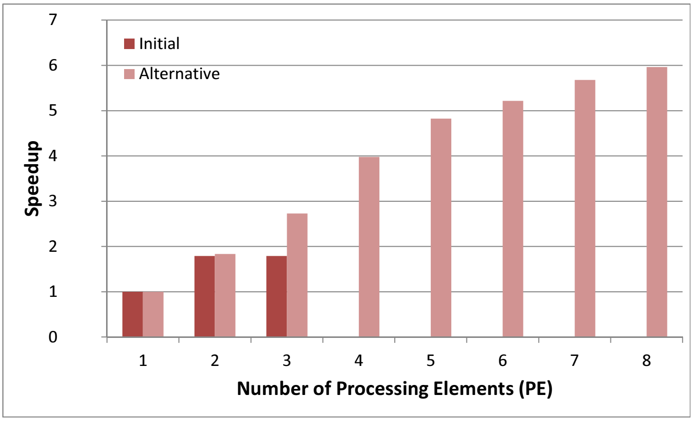
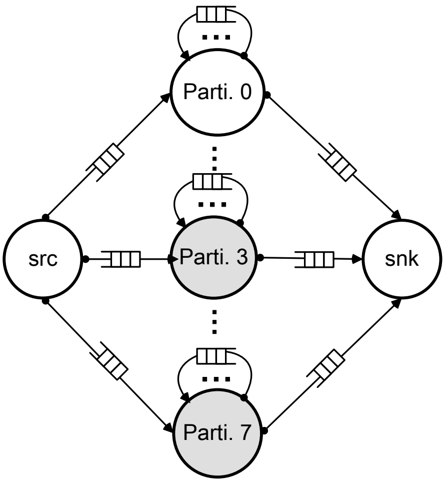
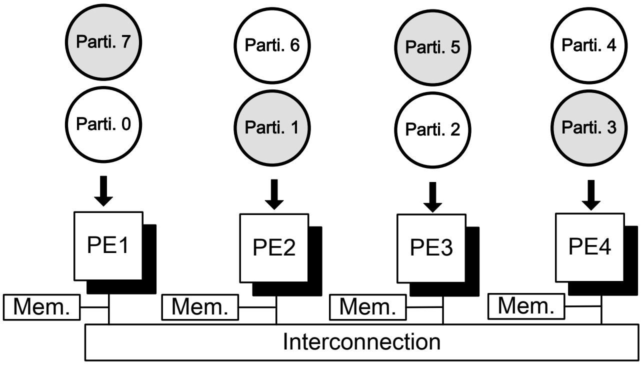


Figure 4.3: Performance results of mapping the initial PPN and the alternative PPN after communication-free partitioning.

execution of PPN processes P_1, P_2, and P_3 in Figure 4.1(a). After communication-free partitioning, the initial PPN in Figure 4.1(a) is transformed to the alternative PPN shown in Figure 4.4(a). The only communication between PEs occurs when input data is demultiplexed from process src to all partitions and output produced by the partitions is multiplexed to process snk. For example, this can be seen with the help of Figure 4.1(b). In the initial PPN, process src sends the input data to P_1 at its iterations from (0) to (7) due to a dependence relation (see Definition 2.1.7 on page 26). In the alternative PPN, with the same dependence relation, process src sends the input data at iteration (0) of P_1 to partition *Parti. 0*, the input data at iteration (1) of P_1 to partition *Parti. 1*, and so on. Analogously, in the alternative PPN, process snk collects the output data produced at iteration (0,0) of P_3 from



(a) The alternative PPN.



(b) Mapping of the alternative PPN onto 4 PEs (the data source and sink as well as all edges connected to both of them are omitted for succinctness).

Figure 4.4: The PPN in Figure 4.1(a) after communication-free partitioning and its mapping.

partition *Parti. 0*, the output data produced at iterations $(0,1)$ and $(1,2)$ of P_3 from partition *Parti. 1*, and so on. With a given dependence relation in the initial PPN, the correct demultiplexing and multiplexing in the alternative PPN from the data source to all partitions and from all partitions to the data sink are automatically generated by our approach. Except the communication between the partitions and the data source/sink, mapping the obtained partitions onto PEs will only result in local communication whose cost can be neglected on any platform. For instance, in case of 4 PEs available, mapping the derived alternative PPN in Figure 4.4(a) is shown in Figure 4.4(b).

Figure 4.3 also shows the achieved speedup of pareto-optimal mappings of the alternative PPN in Figure 4.4(a) (denoted as *Alternative*). Compared to mapping the initial PPN, mapping the alternative PPN constantly leads to a better performance. Moreover, the alternative PPN allows us to utilize up to 8 PEs, thereby achieving even higher speedup, which is not possible by considering only the initial PPN. Figure 4.3 shows that, for the alternative PPN, a linear speedup is observed up to 5 PEs. This is because the grouping of the 8 communication-free partitions can balance the workloads across up to 5 PEs. For instance, 4 groups with 2 partitions each shown in Figure 4.4(b) have the same workload, i.e., the total number of iterations (dots) in all such 4 groups is equal. The speedup of mapping the derived alternative PPN onto 6 to 8 PEs saturates due to unbalanced workloads. From this motivating example, we can see the necessity and usefulness of considering alternative application specifications, particularly the one containing only communication-free and load-balanced partitions.

4.2 Related Work

An alternative application specification modeled as a SDF graph is considered in [133]. To exploit better parallelism in the SDF graph, all actors in the initial SDF graph are converted to their equivalent Homogeneous SDF actors (all production/consumption rates equal to 1). The conversion may lead to an exponential increase in the size of the graph. Therefore, the authors propose a heuristic based on an evolutionary algorithm to find a mapping and a schedule for the resulting Homogeneous SDF graph. Compared to [133], we consider a more expressive MoC than SDF, i.e., the PPN MoC. Also, instead of completely unfolding all PPN processes (equal to unfolding actors in [133]), we operate on a compact representation which avoids the explosion in the size of the graph. Moreover, this compact representation also allows us to analytically determine the maximum amount of DLP, i.e., the maximal number of communication-free partitions.

Similar to [133], SDF is also used as the underlying MoC in [54]. Each SDF actor is furthermore restricted to have only one input and one output port. Based on this assumption, *stateless* actors (the actors without cyclic dependences) in the SDF graph are first fused into compound actors. Then, those compound actors are duplicated by inserting *splitters* and *joiners* to distribute data and collect results. Conceptually, this method also aims at extracting DLP without communication between the compound actors. Compared to [54], the PPN MoC considered in this chapter is more general with an arbitrary number of input and output ports of PPN processes. The problem addressed in this chapter is thus more difficult as simple fusion-duplication is not applicable to PPN processes. Also, *stateful* actors (see for instance process P_3 in Figure 4.1(a)) cannot be fused and duplicated in [54]. Instead, software pipelining techniques are applied to the stateful actors. Software pipelining brings performance improvement assuming that communication latency between different PEs, on which different pipeline stages are assigned, could be completely overlapped by computation. However, we believe that the communication latency may not be hidden and completely overlapped by computation, especially considering emerging MPSoC platforms interconnected via NoCs as motivated in Section 1.1.3. In contrast, our approach tries to extract maximum DLP even for the PPN processes with cyclic data dependences while completely avoiding communication between PEs. This parallelization strategy may fit better future MPSoC platforms with increasingly larger communication latency.

The PPN MoC is used in [86]. The authors suggest that a perfect alternative application specification can be achieved by first partitioning PPN processes and then merging some PPN processes into a compound one. However, a procedure of partitioning and merging PPN processes is not discussed. In this chapter, we propose a systematic procedure to partition and merge PPN processes in a PPN.

In [78], affine partitioning is used in the Brook language to map streaming applications. Similar to the affine partitioning, our communication-free partitioning also aims at obtaining coarse-grained PPN processes. In contrast, our partitioning strategy is able to completely eliminate communication, which might not be possible in some cases using affine partitioning.

4.3 Finding all Dependences in a PPN

For streaming applications, input data is read from the data source (i.e., environment), subsequently processed by PPN processes at their iterations during the execution, and finally written to the data sink. Recall that a PPN produces output to the environment, represented as a sink process whose domain is denoted as D_{snk}. The output produced at an iteration $\vec{I} \in D_{snk}$ directly or indirectly depends on several iterations of PPN processes. If two dependent iterations mapped onto different PEs, inter-PE communication will take place. To find out communication-free partitions in a PPN, we need to solve the problem of finding all “direct” and “indirect” data dependences in a PPN.

The direct dependences result immediately from the dependence relations as defined in Definition 2.1.7 on page 26. For example, Figure 4.5 illustrates the process domain D_3 of process P_3 in Figure 4.1(a). Dependence relation $R_3 = \{(1,2) \rightarrow (0,3)\}$ in Figure 4.5 (the bold arrow) expresses a direct dependence. In contrast, iteration

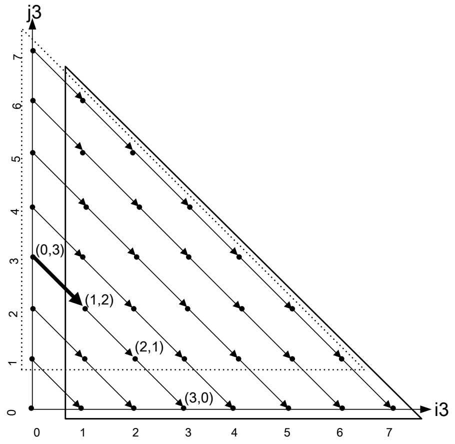


Figure 4.5: Domain of PPN process P_3 in Figure 4.1(a). The input port domain of IP_3 (surrounded by the solid triangle), output port domain of OP_3 (surrounded by the dotted triangle), and dependence relation R_3 (denoted by the arrows between dots).

(2,1) indirectly depends on iteration (0,3) through iteration (1,2). In this chapter, we formulate the problem of finding all direct and indirect data dependences by computing *transitive closure* [65, 102], denoted by R^+, of affine dependence relation R. It is formally defined as:

$$\vec{I} \to \vec{J} \in R^+ \Leftrightarrow (\vec{I} \to \vec{J}) \in R \vee \exists \vec{K} \text{ s.t. } (\vec{I} \to \vec{K}) \in R \wedge (\vec{K} \to \vec{J}) \in R^+. \tag{4.2}$$

From Equation (4.2), we can see that "direct" and "indirect" dependences are uniformly expressed as transitive closure of dependence relations. Thus, we use the term *transitive dependences* to denote both types of dependences. Note that transitive closure of a set of affine relations is not an affine form in general. An under-approximated and closed affine form is computed in [65]. In contrast, we consider an affine over-approximation in case of non-affine closed form. First, the over-approximation guarantees that a valid schedule always can be found for each communication-free partition, but at the cost of potentially fewer communication-free partitions. Second, existing powerful code generation methods [25] for affine dependence relations still can be leveraged.

Now, finding all transitive dependences in a PPN is translated to computing transitive closure of all dependence relations. Therefore, we first take a union R_{deps} of all dependence relations in a PPN as:

$$R_{deps} = \bigcup_{\forall E_i \in \mathcal{E}} R_i,$$

where $\mathcal{E}$ is the set of edges in the PPN. Subsequently, we can compute the transitive closure of the union R_{deps}. In this chapter, we use the *isl* [123] library to compute the transitive closure of affine dependence relations in a potentially over-approximated closed form. For the PPN in Figure 4.1(a), computing the union of all dependence relations yields:

$$R_{deps} = R_1 \cup R_2 \cup R_3.$$

Then, by computing the transitive closure of R_{deps}, we obtain:

$$R^+_{deps} = R_{deps} \cup R^+_{13} \cup R^+_{23} \cup R^+_{33},$$

where R^+_{13}, R^+_{23}, and R^+_{33} are transitive dependence relations, represented as follows:

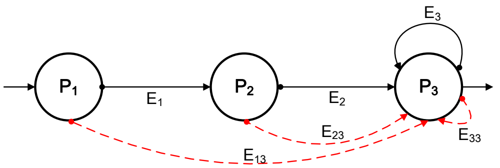
$$\begin{aligned} R^+_{13} &= \{(i3, j3) \rightarrow (i1) \mid 0 \le i3 \le i1 \wedge i1 \le 7 \wedge i1 = i3 + j3\}, && (4.3a)\\ R^+_{23} &= \{(i3, j3) \rightarrow (i2) \mid 0 \le i3 \le i2 \wedge i2 \le 7 \wedge i2 = i3 + j3\}, && (4.3b)\\ R^+_{33} &= \{(i3, j3) \rightarrow (i3', j3') \mid 1 \le i3 \le 7 \wedge 0 \le j3 \le 7 - i3 \wedge 0 \le i3' \le 6 \\ &\quad \wedge 0 \le i3' \le i3 + j3 - 1 \wedge j3' = i3 + j3 - i3'\}. && (4.3c) \end{aligned}$$

After computing the transitive closure of all dependence relations in the PPN in Figure 4.1(a), 3 extra edges E_{13}, E_{23}, and E_{33} corresponding to the transitive dependence relations are added in the PPN as shown in Figure 4.6(a). For the execution of the PPN (domains of PPN processes P_1, P_2, and P_3) shown in Figure 4.1(b), a set of transitive dependences is illustrated as dashed arrows in Figure 4.6(b). For instance, $R^+_{33} = \{(3, 0) \rightarrow (0, 3)\}$, shown as the bold and dashed arrow, indicates that iteration $(3, 0)$ of PPN process P_3 transitively depends on iteration $(0, 3)$ of itself.

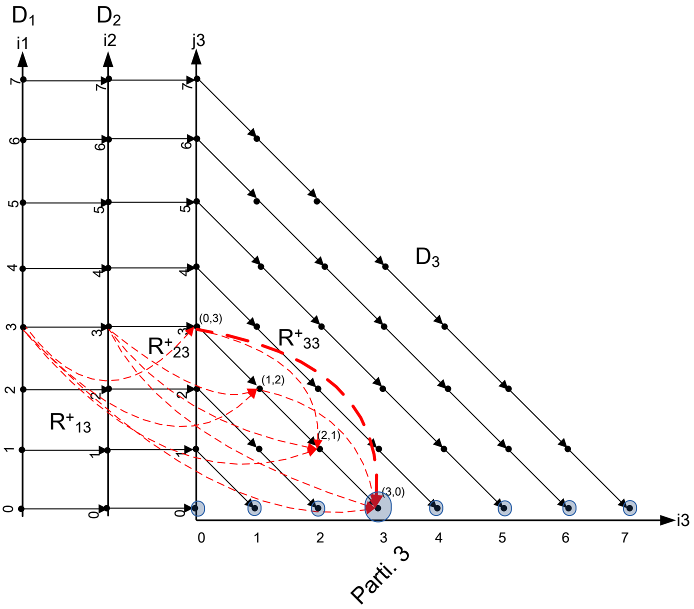
4.4 Computing the Number of Communication-free Partitions

As explained in Section 4.3, we derive all dependent iterations that generate an output at any iteration $\vec{I} \in D_{snk}$. Based on this information, in this section, we compute the number of communication-free partitions that can be derived from a given PPN.

Essentially, we need to find a set of iterations in domain D_{snk} that are independent from each other. Each of these iterations identifies a distinct communication-free partition (see the dashed boxes in Figure 4.1(b)). Consider the PPN in Figure 4.1(a) and its execution illustrated in Figure 4.1(b). As explained in Section 4.1, $D_{snk} = D_3$ (see the triangular part in Figure 4.1(b) denoted as D_3). Our goal is to find the 8 iterations marked by circles in Figure 4.1(b). It can be seen that they are independent



(a) Transitive dependences of the PPN in Figure 4.1(a).



(b) The set of transitive dependences for communication-free partition *Parti. 3* in Figure 4.1(b).

Figure 4.6: Finding transitive dependences of the PPN.

of each other and they identify the 8 communication-free partitions. Therefore, the number of these iterations determines the number of communication-free partitions.

In general, to find the set of iterations mentioned above, we first state the following lemma:

Lemma 4.4.1. *For a PPN, any transitive dependence relation*

$$R_i^+ = \{\vec{I} \rightarrow \vec{J} \mid \vec{I} \in D_{IP} \wedge \vec{J} \in D_{OP}\}$$

is a total and surjective affine relation, which maps iterations $\vec{I}$ in input port domain D_{IP} to iterations $\vec{J}$ in output port domain D_{OP}.

Proof. Totality of a transitive dependence relation R_E^+ holds because of the following property of the PPN MoC. For streaming applications operating on infinite input streams, we are only interested in consistent and deadlock-free PPNs[1]. Therefore, if an iteration in an input domain ($\vec{I} \in D_{IP}$) is unmapped, it means that the PPN will deadlock at this iteration during execution of the PPN. At the same time, R_E^+ is also surjective, because several iterations in an input domain can be mapped to the same iteration in an output domain. This can be seen from the definition of the transitive closure of affine relations in Equation (4.2). If there exists $\vec{I} \to \vec{K} \in R$ and $\vec{K} \to \vec{J} \in R$, both iterations $\vec{I}$ and $\vec{K}$ are mapped to iteration $\vec{J}$ in R^+. ■

For instance, transitive relation R_{23}^+ in Figure 4.6(b) denotes that iterations $(0,3)$, $(1,2)$, $(2,1)$, and $(3,0)$ of P_3 are mapped to iteration (3) of P_2. That is, R_{23}^+ maps the iterations $\vec{I} \in D_{IP_2} \cup D_{IP_3}$ to the iterations $\vec{J} \in D_{OP_2}$ shown in Figure 4.1(a).

In addition to Lemma 4.4.1, we introduce a definition, called *independent sink domain*, denoted by D_{snk}^{ind}.

Definition 4.4.1 (Independent Sink Domain). The independent sink domain D_{snk}^{ind} for a PPN is a subset of the process domain of the sink process D_{snk}, namely $D_{snk}^{ind} \subseteq D_{snk}$. The following condition holds for any two iterations $(\vec{I},\vec{J}) \in D_{snk}^{ind}$, where $\vec{I} \neq \vec{J}$:

$$\neg\exists(\vec{I},\vec{J}) \in D_{snk}^{ind} \; : \; \vec{I} \to \vec{J} \in R^+. \tag{4.4}$$

D_{snk}^{ind} is given by

$$D_{\mathrm{snk}}^{\mathrm{ind}} =$$

$$\{\vec{I} \in \mathbb{Z}^d \mid \exists R^+ : \vec{I} \to \vec{J} \in R^+ \wedge \vec{I} \in D_{snk} \wedge \vec{J} \in D_{snk} \wedge \vec{I} \in (\mathrm{dom}R^+ - \mathrm{ran}R^+)\} \tag{4.5a}$$

$$\bigcup$$

$$\{\vec{I} \in \mathbb{Z}^d \mid \forall R^+ : \vec{I} \to \vec{J} \in R^+ \wedge \vec{I} \in D_{snk} \wedge \vec{J} \notin D_{snk} \wedge \vec{I} \in \mathrm{dom}R^+\}, \tag{4.5b}$$

where $\mathrm{dom}R^+$ is the domain of transitive relation R^+ and $\mathrm{ran}R^+$ is the range of transitive relation R^+.

The condition in Equation (4.4) states that the iterations in D_{snk}^{ind} are not transitively dependent on each other.

Based on Lemma 4.4.1 and Definition 4.4.1, we can have the following theorem.

---

[1] The PPNs with these two properties are called *live*.

Theorem 4.4.1. *For any PPN, the number of communication-free partitions is equal to* $|D_{snk}^{ind}|$.

Proof. For an iteration $\vec{I} \in D_{snk}$, it satisfies one of two mutually exclusive conditions. That is, the iteration either transitively depends on other iterations $\vec{J} \in D_{snk}$, or does not transitively depend on any iteration $\vec{J} \in D_{snk}$. The former condition is stated as $\vec{I} \rightarrow \vec{J} \in R^{+} \wedge \vec{J} \in D_{snk}$ in Equation (4.5a), whereas the latter condition is expressed as $\vec{I} \rightarrow \vec{J} \in R^{+} \wedge \vec{J} \notin D_{snk}$ (Equation (4.5b)). For the former condition, the surjective property of a transitive dependence R^{+} stated in Lemma 4.4.1 indicates that multiple iterations $\vec{I} \in \mathrm{dom} R^{+} \subset D_{snk}$ may depend on the same $\vec{J} \in \mathrm{ran} R^{+} \subset D_{snk}$. We thus need to find out distinct iterations $\vec{I} \in \mathrm{dom} R^{+}$, which are not mapped from any other iterations $\vec{I} \in D_{snk}$. It is essentially equivalent to computing the lexicographically maximal iteration $\vec{I}$ if $\vec{I} \rightarrow \vec{J} \in R^{+}$. Such iterations $\vec{I}$ can be found by $\mathrm{dom} R^{+} - \mathrm{ran} R^{+}$. On the other hand, if an iteration $\vec{I} \in D_{snk}$ does not transitively depend on any other iteration $\vec{J} \in D_{snk}$, where $\vec{I} \neq \vec{J}$, all these iterations are independent. This means all such iterations can definitely find independent communication-free partitions. Finally all those iterations in domain $D_{snk}^{ind} \subseteq D_{snk}$ can be computed by taking the union as given in Equations (4.5a) and (4.5b). ■

Consider the PPN in Figure 4.1(a). The sink iterations are described by the domain of process snk, D_{snk} as given in Equation (4.1). Upon computing transitive closure R_{deps}^{+} of all dependence relations presented in Section 4.3, there are three transitive dependence relations on D_{snk}, namely R_{13}^{+}, R_{23}^{+}, and R_{33}^{+}. Among them, R_{33}^{+} satisfies the condition $\vec{I} \rightarrow \vec{J} \in R^{+} \wedge \vec{I} \in D_{snk} \wedge \vec{J} \in D_{snk}$ as stated in Equation (4.5a). The domain and range of R_{33}^{+} thus are:

$$\begin{aligned} \mathrm{dom} R_{33}^{+} &= \{(i3, j3) \in \mathbb{Z}^2 \mid 1 \leq i3 \leq 7 \wedge 0 \leq j3 \leq 7 - i3\}, \\ \mathrm{ran} R_{33}^{+} &= \{(i3, j3) \in \mathbb{Z}^2 \mid 0 \leq i3 \leq 7 \wedge 1 \leq j3 \leq 7 - i3\}. \end{aligned}$$

Then, $\mathrm{dom} R_{33}^{+} - \mathrm{ran} R_{33}^{+}$ in accordance with Equation (4.5a) yields:

$$D_{snk}^{ind1} = \{(i3, j3) \in \mathbb{Z}^2 \mid 1 \leq i3 \leq 7 \wedge j3 = 0\}. \tag{4.6}$$

Furthermore, we compute those iterations that satisfy the second condition in Equation (4.5b) , namely they do not depend on any other iterations in domain D_{snk}. That is:

$$D_{snk}^{ind2} = \{(i3, j3) \mid i3 = 0 \wedge j3 = 0\}. \tag{4.7}$$

Finally, D_{snk}^{ind} can be computed by taking the union of D_{snk}^{ind1} obtained in Equation (4.6) and D_{snk}^{ind2} obtained in Equation (4.7):

$$\begin{aligned} D_{snk}^{ind} &= D_{snk}^{ind1} \cup D_{snk}^{ind2} \\ &= \{(i3, i3) \in \mathbb{Z}^2 \mid 0 \leq i3 \leq 7 \wedge j3 = 0\}. \end{aligned} \tag{4.8}$$

In general, D_{snk}^{ind} computed in accordance with Equation (4.5) is a union of domains represented by polytopes. Then, computing the number of communication-free partitions is equal to counting the number of integer points in the union of polytopes, denoted by $|D_{snk}^{ind}|$. The counting problem can be efficiently solved in polynomial time using the *barvinok* [127] library. Finally, for the PPN shown in Figure 4.1(a) and D_{snk}^{ind} obtained in Equation (4.8), counting the number of integer points in D_{snk}^{ind} yields $|D_{snk}^{ind}| = 8$. This confirms the same number of communication-free partitions, namely 8 as shown in Figure 4.1(b). Also, D_{snk}^{ind} corresponds to the iterations marked by circles show in both Figures 4.1(b) and 4.6(b).

4.5 Communication-free Partitioning Algorithm

If the number of communication-free partitions computed in Section 4.4 is greater than 1, we can transform the initial PPN to a set of communication-free partitions. We first show an example of constructing one of the communication-free partitions for the PPN in Figure 4.1(a). Subsequently, we present the general partitioning algorithm.

An Illustrative Example

Consider the PPN in Figure 4.1(a) and its execution illustrated in Figure 4.1(b). Let us for example construct communication-free partition *Parti. 3* in Figure 4.1(b). In the partitioning algorithm for this example, our goal is to partition the domains of the PPN processes and obtain all iterations surrounded by the dashed box for *Parti. 3*. These iterations are transitively dependent on the iteration that identifies *Parti. 3*. In this case, *Parti. 3* is identified by iteration $(i3, j3) = (3, 0) \in D_{snk}^{ind}$ of process P_3 as computed in Equation (4.8). All transitive dependence relations R_{33}^{+}, R_{23}^{+}, and R_{13}^{+} to iteration $(3, 0)$ are computed in Equations (4.3a) to (4.3c) and illustrated in Figure 4.6(b). In the first step of the partitioning algorithm for *Parti. 3*, we instantiate process P_{33} (see Figure 4.7) of PPN process P_3 through R_{33}^{+}. Process P_{33} performs the same computational function as the original PPN process P_3 does. The only difference is that process P_{33} only executes in a subdomain D_{33} of the original domain D_3. For *Parti. 3*, besides that iteration $(3, 0)$ belongs to domain D_{33}

of process P_{33}, P_{33} contains also iterations $(2,1)$, $(1,2)$, and $(0,3)$ of P_3, on which iteration $(3,0)$ depends, as shown in Figure 4.6(b). These iterations can be derived by "substituting" iteration $(3,0)$ in R^+_{33} (see Equation (4.3c)), denoted as $R^+_{33}((3,0))$:

$$\begin{aligned} R^+_{33}((3,0)) &= \{(i3',j3') \mid (3,0) \rightarrow (i3',j3') \in R^+_{33}\} \\ &= \{(i3',j3') \mid 0 \le i3' \le 2 \wedge j3' = 3 - i3'\}. \end{aligned} \tag{4.9}$$

Then, domain D_{33} for *Parti. 3* can be obtained by taking a union of iteration $(3,0)$ with the ones computed in Equation (4.9):

$$\begin{aligned} D_{33} &= (3,0) \cup R^+_{33}((3,0)) \\ &= \{(i3',j3') \mid 0 \le i3' \le 3 \wedge j3' = 3 - i3'\}. \end{aligned} \tag{4.10}$$

Second, a process P_{23} (see Figure 4.7) of PPN process P_2 is instantiated due to R^+_{23} for *Parti. 3*. Domain D_{23} contains iteration (3) of P_2 as shown Figure 4.6(b). It can be derived by "substituting" domain D_{33}, obtained in Equation (4.10), in R^+_{23} (see Equation (4.3b)), denoted as $R^+_{23}(D_{33})$:

$$\begin{aligned} D_{22} = R^+_{23}(D_{33}) = {} & \{(j2) \mid (i3,j3) \rightarrow (j2) \in R^+_{23} \wedge (i3,j3) \in D_{33}\} \\ = {} & \{(i2) \in \mathbb{Z} \mid i2 = 3\}. \end{aligned} \tag{4.11}$$

Finally, we need to instantiate a process P_{13} (see Figure 4.7) with domain D_{13} due to R^+_{13}. Domain D_{11} corresponds to iteration (3) in domain D_{P1} as shown in Figure 4.6(b). Analogous to obtaining domain D_{23}, domain D_{13} can be obtained by "substituting" domain D_{33} in R^+_{13} (see Equation (4.3a)):

$$D_{13} = R^+_{13}(D_{33}) = \{(i1) \in \mathbb{Z} \mid i1 = 3\}.$$

Once all processes for *Parti 3* are instantiated, next we instantiate edges for the new processes. Basically, if an edge in the initial PPN is incident with the new process, a new edge is instantiated. For *Parti. 3*, edge E_3 in the initial PPN is incident with the new process P_{33}. Then, a new edge E_{33} is instantiated with the associated input port domain $D_{IP_{33}}$, output port domain $D_{OP_{33}}$, and dependence relation R_{33}:

$$\begin{aligned} D_{IP_{33}} = {} & D_{IP_3} \cap D_{33} \\ = {} & \{(i3,j3) \mid 1 \le i3 \le 3 \wedge j3 = 3 - i3\}, \\ D_{OP_{33}} = {} & D_{OP_3} \cap D_{33} \\ = {} & \{(i3',j3') \mid 0 \le i3' \le 2 \wedge j3' = 3 - i3'\}, \\ R_{33} = {} & \{(i3,j3) \rightarrow (i3',j3') \mid (i3,j3) \in D_{IP_{33}} \wedge (i3',j3') \in D_{OP_{33}} \\ & \wedge i3' = i3 - 1 \wedge j3' = j3 + 1\}. \end{aligned} \tag{4.12}$$

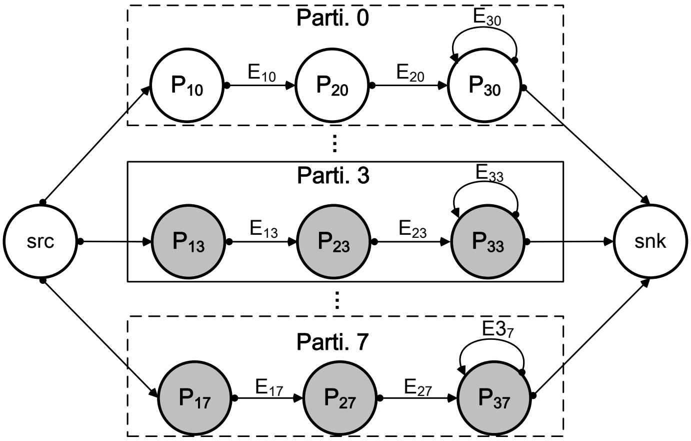


Figure 4.7: The PPN in Figure 4.1(a) after communication-free partitioning.

Two other edges E_{13}, E_{23} can be instantiated in a similar way, due to edges E_1, E_2 in the initial PPN. In this way, communication-free partition *Parti. 3* shown in Figure 4.1(b) is constructed and illustrated by the solid box in Figure 4.7. In the next step, we merge all process instances P_{33}, P_{23}, and P_{13} into a single compound process *Parti. 3* as shown in Figure 4.4(a). We generate a static schedule, similar to the one proposed in [124], that executes all dependent iterations of the new processes as close as possible.

General Partitioning Algorithm

In general, to instantiate new processes and edges, we devise Algorithm 3. The input to Algorithm 3 is a PPN with all transitive dependences ($\mathcal{E}^+$) computed in Section 4.3 and $D_{snk}^{ind} \subseteq D_{snk}$ obtained in Theorem 4.4.1. Every sink iteration $\vec{K} \in D_{snk}^{ind}$ is used to identify a distinct communication-free partition. The output of Algorithm 3 is $|D_{snk}^{ind}|$ communication-free partitions. The core part of the algorithm is presented below.

Algorithm 3 starts partitioning a PPN from the sink process, namely partitioning P_{snk} into $|D_{snk}^{ind}|$ number of processes P_{snk_inst}. For each iteration $\vec{K} \in D_{snk}^{ind}$ of the sink process, we instantiate a new process P_{snk_inst}. The loop iterates over all PPN processes to instantiate all processes in all communication-free partitions. Basically, for a particular partition, we construct the domain for each new process through all transitive dependence relations R_E^+ on iteration $\vec{K}$. First, we construct domain $D_{P_{snk\_inst}}$. If this iteration $\vec{K}$ transitively depends on other iterations in domain D_{snk},

Algorithm 3: Communication-free partitioning procedure

```
Input: A $PPN=(\mathcal{P},\mathcal{E})$, $\mathcal{E}^+$, and $D_{snk}^{ind}$ obtained in Theorem 4.4.1.
Result: A $PPN'=(\mathcal{P}',\mathcal{E}')$.
$\mathcal{P}' \leftarrow \emptyset$, $\mathcal{E}' \leftarrow \emptyset$ ;
Get sink process $P_{snk}$, $D_{snk\_inst} \leftarrow \emptyset$ ;
foreach $\vec{K} \in D_{snk}^{ind}$ do
    $P_{snk\_inst} \leftarrow P_{\mathrm{snk}}$ ;
    foreach Edge $E^+ \in \mathcal{E}^+$ incident with $P_{snk}$ do
        Get $R_E^+$ associated with edge $E^+$ ;
        if $\vec{K} \notin \mathrm{dom}R_E^+$ then
            continue;
        if $\mathrm{ran}R_E^+ \subseteq D_{snk}$ then /* $\vec{K}$ depends on other iterations in $D_{snk}$ */
            $D_{P_{snk\_inst}} \leftarrow D_{P_{snk\_inst}} \cup \vec{K} \cup R_E^+(\vec{K})$ ;
        else /* $\vec{K}$ depends on another process $P$ */
            $D_{P_{snk\_inst}} \leftarrow D_{P_{snk\_inst}} \cup \vec{K}$ ;
            Get process $P \in \mathcal{P}$ incident with edge $E^+$;
            $P_{inst} \leftarrow P$ ;
            $D_{P_{inst}} \leftarrow R_E^+(D_{snk\_inst})$ ;
            $\mathcal{P}' \leftarrow \mathcal{P}' \cup P_{inst}$ ;
        $\mathcal{P}' \leftarrow \mathcal{P}' \cup P_{snk\_inst}$ ;
foreach $P_{inst} \in \mathcal{P}'$ do
    $\mathcal{E}_{inst} \leftarrow instantiateChannels(P_{inst}, \mathcal{E})$ ;
    $\mathcal{E}' \leftarrow \mathcal{E}' \cup \mathcal{E}_{inst}$;
```

then domain $D_{P_{snk\_inst}}$ contains also all iterations in D_{snk} that iteration $\vec{K}$ depends on. All such iterations can be computed by *slicing* a transitive dependence R^+ using iteration $\vec{K}$, denoted as $R^+(\vec{K})$. It is formally defined as:

$$R^+(\vec{K}) = \{\vec{J} \mid \vec{I} \rightarrow \vec{J} \in R^+ \wedge \vec{I} = \vec{K}\},$$

where $\vec{K}$ is a constant vector (see an example in Equation (4.9)). Therefore, in this case, we can obtain $D_{P_{snk\_inst}}$ in Algorithm 3. In contrast, if the iteration $\vec{K}$ does not depend on any other iteration in D_{snk}, then $D_{P_{snk\_inst}}$ is simply equal to $\vec{K}$. Also,

Algorithm 4: Procedure *instantiateChannels*

```
Input: A process instance $P_{inst}$ and a set edges $\mathcal{E}$.
Result: A set of edges $\mathcal{E}_{inst}$ incident with process instance $P_{inst}$.
Get $D_{P_{inst}}$ of $P_{inst}$;
foreach Channel $E \in \mathcal{E}$ incident with $P_{inst}$ do
    Get $D_{IP}$ and $D_{OP}$ associated with edge $E$;
    $E_{inst} \leftarrow E$ ;
    if $D_{IP} \cap D_{P_{inst}} \neq \emptyset$ and $D_{OP} \cap D_{P_{inst}} \neq \emptyset$ then /* a self-edge */
        $D_{IP\_inst} \leftarrow D_{inst} \cap D_{IP}$ , $\mathrm{dom} R_{E_{inst}} \leftarrow D_{IP\_inst}$;
        $D_{OP\_inst} \leftarrow D_{inst} \cap D_{OP}$, $\mathrm{ran} R_{E_{inst}} \leftarrow D_{OP\_inst}$ ;
        $\mathcal{E}_{inst} \leftarrow \mathcal{E}_{inst} \cup E_{inst}$ ;
    else if $D_{IP} \cap D_{P_{inst}} \neq \emptyset$ and $D_{OP} \cap D_{P_{inst}} = \emptyset$ then /* an incoming edge */
        $D_{IP\_inst} \leftarrow D_{inst} \cap D_{IP}$, $\mathrm{dom} R_{E_{inst}} \leftarrow D_{IP\_inst}$ ;
        $\mathcal{E}_{inst} \leftarrow \mathcal{E}_{inst} \cup E_{inst}$ ;
    else if $D_{IP} \cap D_{P_{inst}} = \emptyset$ and $D_{OP} \cap D_{P_{inst}} \neq \emptyset$ then /* an outgoing edge */
        $D_{OP\_inst} \leftarrow D_{inst} \cap D_{OP}$, $\mathrm{ran} R_{E_{inst}} \leftarrow D_{OP\_inst}$ ;
```

in this case, $\vec{K}$ transitively depends on another PPN process P through transitive dependence relation R_E^+, where $P \neq P_{snk}$. Therefore, we need to instantiate a process instance P_{inst} for process P. Domain $D_{P_{inst}}$ can be computed by *applying* domain $D_{P_{snk\_inst}}$ to dependence relation R_E^+, denoted as $R_E^+(D_{P_{snk\_inst}})$. $R_E^+(D_{P_{snk\_inst}})$ is given as:

$$R_E^+(D_{P_{snk\_inst}}) = \{\vec{J} \mid \vec{I} \to \vec{J} \in R_E^+ \wedge \vec{I} \in D_{P_{snk\_inst}}\}.$$

An example of the applying operation can be seen in Equation (4.11). Finally, all process instances in the same communication-free partitions can be instantiated.

Once all processes for each communication-free partition are instantiated, as the next step, we need to instantiate edges for all processes in Algorithm 3. The procedure of instantiating all edges for a process instance is depicted in Algorithm 4. As input, it takes a process P_{inst} with constructed domain D_{inst} and all edges $\mathcal{E}$ in the initial PPN. The algorithm outputs a set of edges $\mathcal{E}_{inst}$ incident with process instance P_{inst}. In Algorithm 4, if both the input port and output port of a edge E are incident with P_{inst}, a new self-edge E_{inst} is instantiated with the corresponding input and output port domains. An example of instantiating self-edge E_{33} for new process

P_{33} can be seen in Equation (4.12). If only the input port or output port of an edge E is incident with P_{inst}, it represents a dependence relation from/to another process instance in the same communication-free partition. In other words, it is either an incoming or outgoing edge of process instance P_{inst}. In this case, we instantiate only one edge with its corresponding input and output port domains. Therefore, using Algorithm 4, we can instantiate all edges incident with a new process P_{inst}.

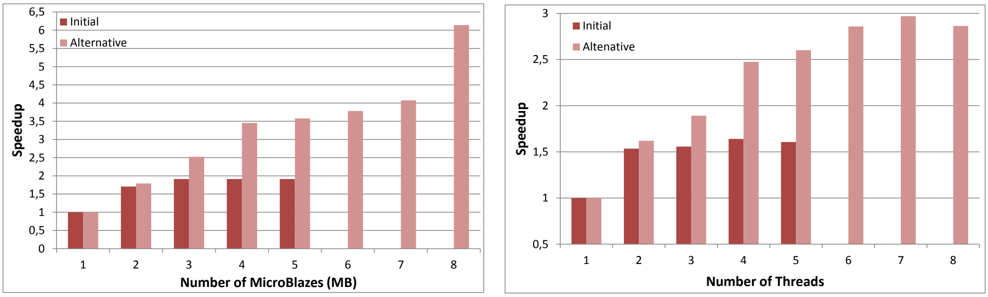
4.6 Experimental Results

In this section, we present the performance results obtained by applying our approach explained previously and prototyping two real-life streaming applications on two different platforms. Then, we present a set of experiments to evaluate the time complexity of our approach.

We selected two different platforms, a Xilinx ML605 board equipped with a Virtex 6 FPGA (referred as FPGA platform hereinafter) and a desktop multi-core platform containing an Intel i7-920 processor running at 2.66GHz with 4 cores and 4GB system memory (referred as desktop platform hereinafter). For the FPGA platform, the generated MPSoCs consist of up to 8 MicroBlaze (MB) soft-cores interconnected via Xilinx' Fast Simplex Link FIFOs. All MBs run at 100Mhz with their own 64KB program memory and 64KB data memory. On the desktop platform, a main thread was used to measure the performance and to spawn up to 8 threads, due to hyper-threading. The inter-core data communication cost on the desktop platform is much higher than that on the FPGA platform. Therefore, the performance gain introduced using our approach was evaluated on the platforms with different computation/communication characteristics. We implemented the partitioning algorithm presented in Section 4.5 in PNtool as part of the Daedalus^{RT} design flow shown in Figure 1.7 on page 13. We conducted all experiments using the ESPAM [96] tool, the Xilinx Platform Studio 13.2, and Microsoft Visual Studio 2008. All generated programs were compiled using compilers `mb-g++`4.6.2 and `g++`4.52 on the selected platforms respectively, with optimization level `-O2`.

Case Studies

We considered two real-life applications modeled using the PPN MoC, namely a Motion-JPEG (MJPEG) encoder used in [34] and the FM radio application taken from the StreamIT benchmark suite [54]. The MJPEG encoder encodes frames of size 128 × 128 pixels. For the FM radio application, we took the provided sequential C implementation to generate the initial PPN with the following parameters: decimation rate 4, tap size 64, and 10 equalization bands. To optimally balance the workloads across a particular number of PEs, we exhaustively mapped all possible



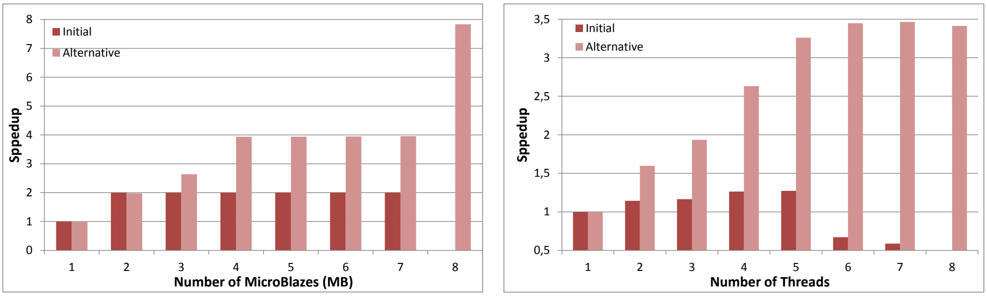
(a) (b)

Figure 4.8: Performance results of mapping the MJPEG encoder onto (a) FPGA-based MPSoC platforms and onto (b) a desktop multi-core platform.

groupings of the obtained communication-free partitions on both platforms. As a reference, we also implemented the initial PPNs of both applications on the selected platforms by performing maximal load-balancing and optimal pipelining, such that the best possible mapping was found for a given number of MBs or threads. The metric used to evaluate the performance results is the relative speedup compared to the 1-MB or 1-thread system implementation.

The performance results of mapping the MJPEG encoder are plotted in Figure 4.8(a) for the FPGA platform and in Figure 4.8(b) for the desktop platform. As expected, the implementation on the desktop platform results in less speedup than the one obtained on the FPGA platform for the same number of MBs or threads in use. This is because of the shared memory architecture and very costly inter-thread communication on the desktop platform. Also, the initial PPN mapped onto the desktop platform using 1 thread is already highly optimized by the compiler. For the mapping of the initial PPN (denoted as *Initial*), the initial PPN does not have enough processes to utilize more than 5 MBs or threads. It can be seen that up to 1.91X speedup for the FPGA platforms and 1.64X speedup for the desktop platform are achieved. The main reason is that the workloads of processes in the initial PPN are not well-balanced, as the Discrete Cosine Transform (DCT) dominates the total execution time of the MJPEG encoder. Although all PPN processes are fully pipelined, the speedup is limited by the longest pipeline stage, the DCT process. For the desktop platform, the pipelining leads to less benefits compared to the FPGA platform because the communication between threads mapped onto different cores cannot be completely overlapped by computation.

Compared to the mapping of the initial PPN for the MJPEG encoder, our approach (denoted as *Alternative* in Figure 4.8(a) and 4.8(b)) leads to better per-



(a) (b)

Figure 4.9: Performance results of mapping the FM radio application onto (a) FPGA-based MPSoC platforms and onto (b) a desktop multi-core platform.

formance. Our approach outperforms the mapping of the initial PPN by 5% to 87.05% on 2 to 5 MBs. As shown in Figure 4.8(a) for the FPGA platform, the speedup increases linearly for the mapping of the alternative PPN onto 1 to 4 MBs (3.45X speedup on 4 MBs). In case of 5 to 7 MBs, the speedup increases only slightly (3.6X to 4.09X speedup on 5 to 7 MBs). We found that unbalanced workloads and the single data sink become bottlenecks for these cases. As the number of MBs increases, a slightly unbalanced grouping of communication-free partitions has large impact on the performance. As a consequence, the single data sink is constantly blocking on the group of partitions with the heaviest workload. Of course, modern architectures may have multiple I/O ports, namely multiple data sinks. For instance, the authors in [54] observe 18.4% performance improvement on the 16-core RAW architecture with 16 data sinks compared to the one with the single data sink. In the best case, our approach results in 6.14X speedup on 8 MBs, when the grouping of the obtained partitions balances the workload across 8 MBs. For the results on the desktop platform shown in Figure 4.8(b), the mapping of the alternative PPN outperforms the mapping of the initial PPN by 5.5% to 61.97% using 2 to 5 threads. Moreover, the effect of unbalanced grouping of communication-free partitions is amortized by the higher communication cost compared to the FPGA platform. In the best case, 2.97X speedup is achieved using 7 threads. When 8 threads are used, the main thread, mentioned earlier, introduces extra overhead. Therefore, the 8-thread implementation performs 3.68% worse than the 7-thread implementation.

For the FM radio application, the workloads of PPN processes in the initial PPN are overall not balanced. The low pass and high pass filters in the equalizer dominate the total execution time of the application. Moreover, the communication between PPN processes is performed at more fine-grained level compared to the MJPEG encoder, i.e., at each iteration, one audio sample is flowed through all PPN processes instead of one macroblock as in the MJPEG encoder. The obtained speedup of mapping the initial PPN (denoted as *Initial*) is plotted in Figure 4.9(a) for the FPGA platform and in Figure 4.9(b) for the desktop platform. In the best case on the FPGA platform, by pipelining all processes in the initial PPN and offloading the high pass filter (or low pass filter) in the equalizer to a separate MB, $1.99X$ speedup is achieved on 2 MBs. On the desktop platform shown in Figure 4.9(b), the best mapping of the initial PPN is found using 5 threads occupying 4 cores, i.e., $1.27X$ speedup. In case of 6 and 7 threads, the implementation slows down compared to the 1-thread implementation. The fine-grained communication and the little workloads of some threads (e.g., the Demodulation and the Amplify processes in the Equalizer) fully expose the communication/synchronization overhead which dominates the total execution time.

After communication-free partitioning, the alternative PPN of the FM radio application exhibits ample data-level parallelism. Also, the fine-grained communication between MBs or threads in the initial PPN is completely eliminated, except the communication from the data source and to the data sink. For the results on the FPGA platform shown in Figure 4.9(a) (denoted as *Alternative*), the obtained speedup by mapping the alternative PPN outperforms mapping the initial PPN by 32.05% to 97.74% on 3 to 7 MBs. Compared to the 4-MB implementation, the mapping of the alternative PPN onto 5 to 7 MBs does not result in further improvements. This is because, as the number of MBs increases, the workloads of the obtained communication-free partitions cannot be evenly distributed. This fact combined with the relatively cheaper inter-MB communication on the FPGA platform, shows that our communication-free partitioning does not bring too much benefits on 5 to 7 MBs. Once the workload is balanced, $7.83X$ speedup is achieved on 8 MBs. On the desktop platform, our approach (denoted as *Alternative* in Figure 4.9(b)) outperforms the mapping of the initial PPN by 39.79% to 489.27% using 2 to 7 threads. In the best case, speedup $3.46X$ is observed using 7 threads. The 8-thread implementation performs 1.51% worse compared to the 7-thread one due to the overhead introduced by the main thread similar to the MJPEG case study.

Time Complexity of our Approach

To quantify the time complexity of our approach, we conducted experiments on a set of real-life benchmarks from Polybench [101]. Other benchmarks are less complex than the benchmarks listed in Table 4.1 in terms of their characteristics. The characteristics of each benchmark are given in columns 2 to 4 in Table 4.1. The benchmarks differ in the of number of PPN processes (denoted by $|\mathcal{P}|$) and edges (denoted by $|\mathcal{E}|$) in the initial PPNs, as well as dimensions of data arrays accessed in

Table 4.1: Execution time on benchmarks.

Benchmark	$\vert\mathcal{P}\vert$	$\vert\mathcal{E}\vert$	Array dimensions	Execution time (sec.)
ADI[1]	12	67	3	2.644
Gram-schmidt	8	19	2	0.924
FDTD[2]	9	27	2	0.604
Correlation	12	20	2	0.076
Reg-detect[3]	8	11	3	0.068
Dynprog[4]	8	12	3	0.064
Gauss[5]	11	18	2	0.044
Covariance	8	11	2	0.032

[1] ADI: Alternating direction implicit solver

[2] FDTD: 2D finite difference time domain kernel

[3] Reg-detect: Regularity detection

[4] Dynprog: Dynamic programming (2D)

[5] Gauss: 2D gauss blur filter for image processing

PPN processes. For instance, the ADI solver in Table 4.1 operates on 3 dimensional data arrays. In practice, it can be seen that, from the last column in Table 4.1, our approach takes less than 3 seconds to derive all communication-free partitions for the considered benchmarks. This shows that our approach is very fast even for relatively large PPNs such as the PPN of the ADI benchmark.

Chapter 5

Exploiting Just-enough Parallelism in Hard Real-time Systems

Jiali Teddy Zhai, Mohamed A. Bamakhrama, Todor Stefanov, "Exploiting Just-enough Parallelism when Mapping Streaming Applications in Hard Real-time Systems", *In the Proceedings of the 50th IEEE/ACM Design Automation Conference (DAC'13)*, pp. 170:1–170:8, Austin, TX, USA, June 2 - 6, 2013.

As we have seen in Chapter 4, the initial application specification, often in the form of a graph, may be transformed to an alternative one that exposes more parallelism while preserving the same application behavior. To this end, task unfolding is an effective technique to generate such alternative graphs. Basically, task unfolding replicates the functionality of a task by a certain number of times, referred as *unfolding factor*. Then, replicas of tasks concurrently process different data, thereby exploring also data-level parallelism next to the task-level parallelism. For data flow MoCs, such as SDF, the unfolding has been extensively applied in [40, 54, 56, 71].

In the context of Daedalus^{RT}, unfolding individual actors in an initial SDF graph by different unfolding factors results in a large number of possible alternative CSDF graphs. To transform the initial SDF graph to an alternative CSDF one by unfolding, the main problem is to determine a proper unfolding factor for each task. This problem is challenging because platform constraints must be considered during unfolding. The platform constraints can be the number of available PEs and temporal scheduling of actors on the PEs. In Chapter 4 and other literature [40, 133], an unfolding factor[1] is determined for each task in such a way that the obtained

---

[1] Our communication-free partitioning presented in Chapter 4 on PPN processes can be considered

alternative graph exposes the maximum DLP without considering the platform constraints. However, unfolding a task too many times reveals more parallelism than the processing capability of the execution platform. The overwhelming parallelism leads to an inefficient mapping of replicas of tasks. That is, the excessive number of replicas cannot be efficiently allocated and temporally scheduled on the available PEs. Moreover, the excessive number of replicas introduces significant memory overhead for both code and data. On the other hand in [54, 71, 113], the authors assume that the unfolding factor of a task cannot exceed the number of available PEs on the execution platform. This assumption, however, restricts the amount of revealed parallelism because a proper unfolding factor is not necessarily less than or equal to the number of available PEs. As a consequence, the aforementioned assumption might lead to under-utilized PEs. From the discussion above, we can see that exploiting excessive or insufficient parallelism may result in sub-optimal system utilization and performance. Therefore, in this chapter, we address the problem of determining a proper unfolding factor of each SDF actor in a given initial SDF graph, such that the obtained alternative CSDF graph exposes *just-enough* parallelism to fully utilize the available PEs. This is achieved by considering the platform constraints when determining the unfolding factors.

Scope of Work

In this chapter, we assume that a given SDF graph is acyclic. Note that this assumption is not directly related to our approach in this chapter. Rather, it is merely a restriction of the adopted hard real-time scheduling framework (see Section 2.3 on page 33). Such assumption covers a large set of applications as it has been empirically shown in [116] that around 90% of streaming applications can be modeled as acyclic SDF graphs. Once a cycle exists in an SDF graph, one can always fuse all actors in the cycle into a single stateful actor. A stateful actor is the one whose next execution depends on the current execution. As a consequence, our approach does not unfold stateful actors. Furthermore, the data source and sink actors, which represent the external environment, are not unfolded. The target platform assumed in this work is a homogeneous programmable MPSoC with distributed memory. The interconnection structure between PEs must provide guaranteed communication latency, e.g., Æthereal network-on-chip [53].

---

as a special case of unfolding.

5.1 Related Work

The approach in [113] is closely related to our work, although the considered problem is relaxed, i.e., without considering timing constraints, compared to our problem. A genetic algorithm based heuristic is proposed to determine the unfolding factor of an actor and allocation of all replicas. The unfolding factor of an actor cannot exceed the number of PEs, which might result in sub-optimal solutions as we show later in Section 5.5. Moreover, we show in the experiments that our approach outperforms significantly the genetic algorithm based heuristic in terms of running time.

In [71], an Integer Linear Programming (ILP) formulation gives exact solutions to minimize makespan on any PE while simultaneously unfolding actors in an SDF graph and allocating them to PEs. In the ILP formulation, an unfolding factor of an actor cannot exceed the number of available PEs. This assumption might lead to sub-optimal system performance as discussed previously. Moreover, it has been shown in [40] that the ILP formulation is even intractable for benchmarks with medium graph size. For instance, it takes around 70 hours to solve the ILP formulation for the FFT benchmark with 26 actors on 4 PEs (see Table 2 in [40]). In practice, real-life applications have been shown to contain up to 2868 actors [116]. Therefore, it is clear that the ILP-based approach suffers from severe scalability issues. In contrast, our proposed algorithm solves the combined problem within a reasonable amount of time as demonstrated later in Section 5.7.

To address the scalability issue of [71], the authors in [40] propose to decompose the actor unfolding and allocation problem into two problems and solve them separately. The separation of the two problems often leads to inferior performance, as both problems are strongly related. In contrast, our proposed algorithm is capable of solving the two problems simultaneously. Moreover, our algorithm takes into account timing constraints, while the work in [40] does not.

In the context of synthesizing an SDF graph using dedicated hardware, the authors in [56] also determine which actors to unfold and by what factor. The addressed problem is easier than ours because there is no need to consider allocation of actors after unfolding in case of hardware synthesis.

In [72], a synchronous programming model is used for the application specification under hard real-time scheduling. The term “synchronous” in this context refers to the fact that a master thread can *fork* a job into several parallel execution segments and they *join* upon completion. These parallel execution segments are, to some extent, similar to unfolded actors in our case. There is also no need to consider allocation of parallel segments at compile-time because migration at run-time is allowed targeting MPSoC platforms with shared memory. In contrast, we solve the problem of allocating actors at compile-time. Recall that we consider MPSoC platforms with distributed memory. On such platforms, migration of actors at run-time introduces non-negligible overhead.

5.2 Unfolding of SDF Graphs

Before the problem formulation, we first present an unfolding algorithm for SDF graphs. This will help better understand the problems stated in Section 5.3.

The unfolding operation on an SDF graph used in this thesis is conceptually similar to the one used in [40, 54, 56, 71], in which two special constructs *splitter* and *joiner* are employed for the unfolded actors. Given a vector $\vec{f} \in \mathbb{N}^n$ of unfolding factors, where f_i denotes the unfolding factor for actor A_i, the unfolding operation replaces A_i by f_i replicas of itself. Then, instead of inserting a splitter and joiner before and after the f_i replicas of A_i, we transform the initial SDF graph to a functionally equivalent CSDF graph. To ensure the functional equivalence, the production and consumption rates of an SDF actor are modified accordingly to the production and consumption sequences in the resulting CSDF graph. This modification results in a different repetition vector of the obtained CSDF graph to ensure its consistency.

The algorithm for performing the unfolding of actors in SDF graphs is given in Algorithm 5. The algorithm accepts as inputs an SDF graph G and a vector $\vec{f}$ of unfolding factors. The algorithm produces as an output a CSDF graph G', where $A_{i,f}$ denotes the fth replica of A_i with repetition $q_{i,f}$ given by

$$q_{i,f} = \frac{q_i \cdot \mathrm{lcm}(\vec{f})}{f_i}, \tag{5.1}$$

where q_i is the repetition of actor A_i in the initial SDF graph and $\mathrm{lcm}(\vec{f})$ denotes the least common multiple of $f_i \in \vec{f}$. It follows that the repetition vector of G', denoted by $\vec{q'} \in \mathbb{N}^{n'}$ where $n' = \sum_{A_i \in \mathcal{A}} f_i$, is given by $\vec{q'} = [q_{1,1}, \cdots, q_{1,f_1}, \cdots, q_{n,f_n}]^T$ and $n = |\mathcal{A}|$. After obtaining $\vec{q'}$ using Equation 5.1, production/consumption sequences of each CSDF actor are generated accordingly.

Let us consider an SDF graph G_1 is shown in Figure 5.1(a). The actors A_1 and A_5 are the data source and sink actors, respectively. G_1 has five actors and a repetition vector $\vec{q} = [1, 1, 2, 1, 1]^T$. The WCET of each actor is shown below its name, e.g., $C_3 = 12$ for actor A_3. Suppose that a vector of unfolding factors is given as $\vec{f} = [1, 1, 3, 1, 1]$ for G_1 in Figure 5.1(a). Algorithm 5 outputs a CSDF graph G_2 shown in Figure 5.1(b) with three replicas $A_{3,1}$, $A_{3,2}$ and $A_{3,3}$ for actor A_3 in G_1. The

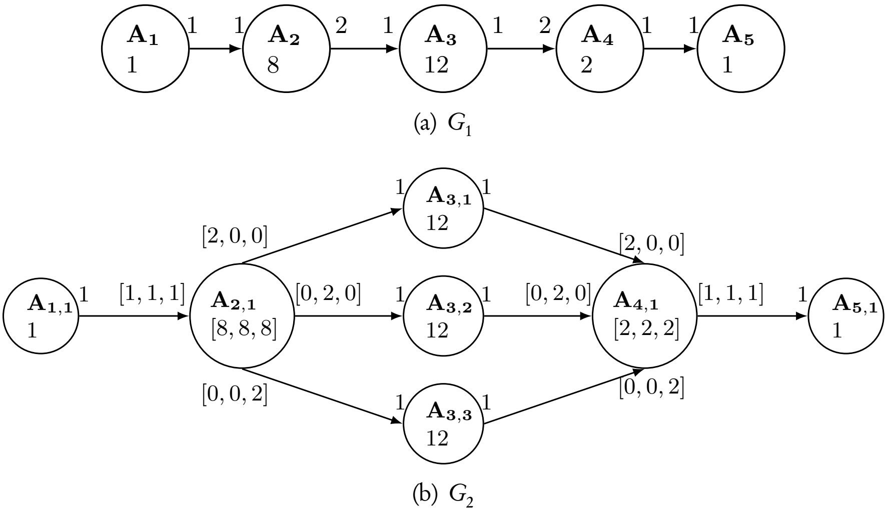
Algorithm 5: Unfolding an SDF graph.

```
Input: An SDF graph $G = (\mathcal{A}, \mathcal{E})$ and a vector $\vec{f}$ of unfolding factors.
Result: The equivalent CSDF graph $G' = \{\mathcal{A}', \mathcal{E}'\}$
$\mathcal{A}' = \emptyset, \mathcal{E}' = \emptyset$ ;
foreach $A_i \in \mathcal{A}$ do
    Add $f_i \in \vec{f}$ replicas of $A_i$ to $\mathcal{A}'$ ;
    Set repetition entry $q_{i,ii} = \frac{q_i \cdot \mathrm{lcm}(\vec{f})}{f_i}, \forall ii \in [1, f_i]$ ;
foreach $E \in \mathcal{E}$ do
    Get source actor $A_i$ and sink actor $A_j$ of edge $E$ ;
    Get production rate $prd(E)$ and consumption rate $cns(E)$ ;
    $lcm\_pc = \mathrm{lcm}(prd(E), cns(E))$ ;
    if $f_j$ is dividable by $f_i$ then $OP = f_j / f_i; IP = 1$;
    else if $f_i$ is dividable by $f_j$ then $IP = f_i / f_j; OP = 1$;
    else $IP = f_i / f_j; OP = 1$;
    for $ii = 1$ to $f_i$ do
        Add $OP$ output ports to $A_{i,ii}$;
        for $k = 1$ to $OP$ do
            Initialize a production sequence $\mathcal{P}_{i,ii}$ of length $q_{i,ii}$ to 0;
            $\mathcal{P}_{i,ii}[p] = prd(E), \forall p \in [(k-1)\frac{lcm\_pc}{prd(E)} + 1, k\frac{lcm\_pc}{prd(E)}]$ ;
            if $f_j$ is dividable by $f_i$ then $jj = (ii-1)OP + k$ ;
            else if $f_i$ is dividable by $f_j$ then $jj = ii / IP$ ;
            else $jj = k$ ;
            Initialize a consumption sequence $\mathcal{C}_{j,jj}$ of length $q_{j,jj}$ to 0;
            $\mathcal{C}_{j,jj}[c] = cns(E), \forall c \in [(ii-1)\frac{lcm\_pc}{cns(E)} + 1, ii\frac{lcm\_pc}{cns(E)}]$ ;
            Create a new channel $E'$ connecting replica $A_{i,ii}$ to replica $A_{j,jj}$ ;
            Add channel $E'$ to $\mathcal{E}'$ ;
Compact the production and consumption sequences of each actor in $\mathcal{A}'$;
```

unfolding results in a repetition vector of G_2 as:

$$\begin{aligned}\vec{q}'_{G_2} &= [q_{1,1}, q_{2,1}, q_{3,1}, q_{3,2}, q_{3,3}, q_{4,1}, q_{5,1}]^T \\ &= [3, 3, 2, 2, 2, 3, 3]^T\end{aligned}$$

For example, SDF actor A_4 executes only once ($q_4 = 1$) in G_1 per graph iteration,



(a) G_1

(b) G_2

Figure 5.1: (a) An example of an SDF graph and (b) its equivalent CSDF graph by unfolding actor A_3 by factor 3.

while executing three times ($q_{4,1} = 3$) in G_2 per graph iteration. Three consumption sequences of actor $A_{4,1}$ in G_2 behave similar to a joiner, with which $A_{4,1}$ collects data tokens from the three replicas $A_{3,1}$, $A_{3,2}$ and $A_{3,3}$. Analogous to a splitter, actor $A_{2,1}$ with three production sequences distributes tokens to the three replicas.

5.3 Problem Formulation

First of all, recall the notations used for (C)SDF MoCs in Table 2.3 on page 31 and the notations used for HRT scheduling of (C)SDF MoCs in Table 2.4. In addition, we introduce some extra notations in Table 5.1 used in this chapter to facilitate the following discussion. Let T_i be the actual period of actor $A_i \in \mathcal{A}$ of a CSDF graph $G = (\mathcal{A}, \mathcal{E})$. T_i can be obtained as $T_i = c\check{T}_i$, where $\check{T}_i$ is computed in Equation (2.16) on page 34 and c is called *scaling factor* (see Section 5.4). Now, we formally define our problem as follows:

Problem 5.3.1. Given an SDF graph G, where the actors are scheduled as strictly periodic tasks, and m available PEs. Suppose that each actor A_i in G is to be unfolded by an unfolding factor $f_i \in \mathbb{N}^+$. Find, for each actor A_i, the minimum value of f_i and the allocation of each replica $A_{i,d}$, where $1 \leq d \leq f_i$, such that the period of the sink actor T_{snk} (see definition of T_{snk} on page 34) in the unfolded graph is minimized.

If Problem 5.3.1 is considered as primal, its dual problem can be stated as follows:

Problem 5.3.2. Given an SDF graph G, where the actors are scheduled as strictly periodic tasks, and m available PEs. Suppose that each actor A_i in G is to be unfolded by an unfolding factor f_i. Find, for each actor A_i, the minimum value of f_i and the allocation of each replica $A_{i,d}$, where $1 \leq d \leq f_i$, such that the total utilization

$$U_{G'} = \sum_{A_{i,d} \in \mathcal{A}'} \frac{C_{i,d}}{T_{i,d}}$$

of the unfolded graph G' on m PEs is maximized, where $T_{i,d}$ is the actual period of replica $A_{i,d}$.

It can be seen that Problems 5.3.1 and 5.3.2 are not trivial. In general, for a given SDF graph, the number of possible alternative graphs that can be generated using unfolding grows exponentially as the number of actors increases. Furthermore, for each alternative graph, we have to perform allocation of unfolded actors which is by itself an NP-hard problem.

Lemma 5.3.1. *Problems 5.3.1 and 5.3.2 are equivalent.*

Proof. The lemma is proven by showing that a solution to Problem 5.3.1 is also a solution to Problem 5.3.2 (case I) and vice versa (case II).
Case I:

Let G' be the unfolded graph of G. Suppose that $\vec{f}$ is the solution to Problem 5.3.1. This means that T_{snk} is minimized. The period $T_{i,f}$ of a replica $A_{i,f}$ in G' based on Equation (2.16) on page 34 can be written as

$$T_{i,f} = c \cdot \check{T}_{i,f} = \frac{c \cdot \text{lcm}(\vec{q'})}{q_{i,f}} \left\lceil \frac{\hat{W}_{G'}}{\text{lcm}(\vec{q'})} \right\rceil, \tag{5.2}$$

Notation	Meaning
c	scaling factor for periods of all actors in a (C)SDF graph and $c \in \mathbb{Z}^+$
f_i	unfolding factor for actor A_i
Ω	ratio
ρ	quality factor $\rho \in (0,1]$
θ_i	code size of a (C)SDF actor A_i
Θ	total code size of a (C)SDF graph, $\Theta = \sum_{A_i \in \mathcal{A}} \theta_i$

Table 5.1: Additional notations used in Chapter 5 besides the ones introduced in Chapter 2.

where c is the scaling factor. Thus, for the sink actor, it is

$$T_{\text{snk}} \cdot q_{\text{snk}} = c \cdot \text{lcm}(\vec{q'}) \left\lceil \frac{\hat{W}_{G'}}{\text{lcm}(\vec{q'})} \right\rceil, \tag{5.3}$$

where $c \cdot \text{lcm}(\vec{q'}) \lceil \hat{W}_{G'} / \text{lcm}(\vec{q'}) \rceil$ is constant. Therefore, from Equations 5.2 and 5.3, it holds

$$T_{\text{snk}} \cdot q_{\text{snk}} = T_{i,f} \cdot q_{i,f}, \forall A_{i,f}. \tag{5.4}$$

Subsequently, Equation 5.4 can be re-written as

$$T_{i,f} = \frac{T_{\text{snk}} \cdot q_{\text{snk}}}{q_{i,f}}. \tag{5.5}$$

Due to the devised unfolding algorithm (see Algorithm 5), we have $q_{i,1} = q_{i,d}$ (see Equation (5.1)), where $1 \leq d \leq f$. Therefore, $\beta_i = q_{\text{snk}}/q_{i,f}$ is constant. It then follows

$$T_{i,f} = \beta_i T_{\text{snk}} \tag{5.6}$$

It follows from Equation 5.6 that when T_{snk} is minimized, then $T_{i,f}$ is minimized. Recall that the maximum total utilization $\hat{U}_{G'}$ is given by

$$\hat{U}_{G'} = \sum_{A_{i,f} \in \mathcal{A}} \frac{C_{i,f}}{T_{i,f}} \tag{5.7}$$

Since $T_{i,f}$ is minimized for all the actors, it follows that $\hat{U}_{G'}$ is maximized. Therefore, $\vec{f}$ is also the solution to Problem 5.3.2.

Case II:

Suppose that $\vec{f}$ is the solution to Problem 5.3.2. This means that $\hat{U}_{G'}$ is maximized. Using Equation 5.4, we can replace each $T_{i,f}$ in Equation 5.7 by $\frac{T_{\text{snk}} \cdot q_{\text{snk}}}{q_{i,f}}$, which results in:

$$\hat{U}_{G'} = \frac{C_{1,1} \cdot q_{1,1}}{T_{\text{snk}} \cdot q_{\text{snk}}} + \cdots + \frac{C_{2,1} \cdot q_{2,1}}{T_{\text{snk}} \cdot q_{\text{snk}}} + \cdots + \frac{C_{\text{snk}}}{T_{\text{snk}}} \tag{5.8}$$

The WCET $C_{i,f}$ and repetition $q_{i,f}$ of each replica is constant. Therefore, Equation 5.8 can be re-written as:

$$\hat{U}_{G'} = \frac{\alpha_{1,1}}{T_{\text{snk}}} + \cdots + \frac{\alpha_{2,1}}{T_{\text{snk}}} + \cdots + \frac{\alpha_{\text{snk}}}{T_{\text{snk}}} \tag{5.9}$$

where $\alpha_{i,f} = C_{i,f} \cdot q_{i,f}/q_{\text{snk}}$. Since $\hat{U}_{G'}$ is maximized, it follows that T_{snk} is minimized. Therefore, $\vec{f}$ is also a solution to Problem 5.3.1. ■

5.4 Period Scaling under Hard Real-time Scheduling

As given in Equation (2.16) on page 34, a (C)SDF graph under SPS can achieve the minimum period (inverse of maximum throughput) $\check{T}_i$ for each actor A_i. This maximum performance also requires the largest number of PEs. If the maximum performance is not necessary, the actually desired period T_i of A_i can be obtained by scaling up $\check{T}_i$ by a scaling factor $c \in \mathbb{Z}^+$ as $T_i = c\check{T}_i$. Using a scaling factor c, we can have a trade-off between processing resources and guaranteed performance as shown in the following proposition:

Proposition 5.4.1. *Let G be a CSDF graph that is schedulable using a scheduling algorithm SA and an allocation algorithm AA on $\check{m}$ PEs, when the minimum period of each actor A_i is equal to $\check{T}_i$. G is schedulable using the same SA and AA on $\lceil \frac{\check{m}}{c} \rceil$ PEs, when the period of each actor A_i is scaled by c.*

Proof. Let U_{SA} be the utilization bound of a scheduling algorithm SA. If G is schedulable on $\check{m}$ PEs using SA and any AA, then this means that the total utilization of the actors on each PE j, where $1 \leq j \leq \check{m}$, is $U_{\text{PE}_j} \in (0, U_{SA}]$. If we scale the periods of the actors in G by c, then this means that $U_{\text{PE}_j} \in (0, \frac{U_{SA}}{c}]$. Therefore, it is possible to combine the actors in every c PEs into 1 PE. Hence, the number of PEs needed after scaling the periods is $\lceil \frac{\check{m}}{c} \rceil$. ■

Considering G_2 in Figure 5.1(b), we have computed in Equation (2.24) on page 37 that it can be scheduled on 5 PEs while achieving $\vec{\check{T}}_{G_2}$. Therefore, it can be scheduled on $\lceil \frac{5}{2} \rceil = 3$ PEs achieving a period $T_{5,1} = 2 \times \check{T}_{5,1} = 16$, i.e., throughput $\frac{1}{16}$ by scaling all minimum periods by $c = 2$.

Now, suppose that AA is an approximate allocation algorithm with an approximation ratio R_{AA}. Then, we can have the following proposition:

Proposition 5.4.2. *Let G be a CSDF graph that is schedulable using a scheduling algorithm SA and any exact allocation algorithm on $\check{m}$ PEs, when the period of each actor A_i is equal to $c\check{T}_i$. G is schedulable using SA and any approximate allocation algorithm AA, with approximation ratio R_{AA}, on $\check{m}$ PEs, when the period of each actor A_i is equal to $cR_{AA}\check{T}_i$.*

5.5 Bounding Solution Space

In order to solve Problems 5.3.1 and 5.3.2 defined in Section 5.3, it is first necessary to show that the solution space of the problems is bounded, i.e., the values of the unfolding factors f_i must be bounded by finite integers. Bounding the solution space ensures that the algorithm devised in Section 5.6 terminates. Now, we define the *upper bound* on unfolding factors as follows:

Definition 5.5.1 (Upper bounds of Unfolding Factors)**.** Let G be an SDF graph, where the actors in G are under SPS shown in Section 2.3, and assume that the number of PEs is unlimited. Suppose that every actor A_i in G is to be unfolded by a factor f_i resulting in a CSDF graph G', for which $\check{T}_{i,f}$ is the minimum period of each replica $A_{i,f}$ and $C_{i,f} = C_i$ is its WCET. The *upper bound* on f_i, denoted by $\hat{f}_i$, is the minimum value which results in utilization

$$\frac{C_{i,f}}{\check{T}_{i,f}} = 1.0$$

for each replica $A_{i,f}$ in G'.

In other words, unfolding an SDF graph G by a vector of unfolding factors $\vec{\hat{f}} = [\hat{f}_1, \cdots, \hat{f}_n]$ results in a graph G' with utilization $U_{G'} = n'$, where n' is the number of actors in the unfolded graph. Hence, unfolding any actor A_i by an unfolding factor $f_i^* > \hat{f}_i$ cannot result in any increase in the total utilization of the unfolded graph. Moreover, the unfolded graph achieves the maximum achievable throughput since the sink actor fully utilizes the PE on which it executes. Therefore, $\vec{\hat{f}}$ bounds the solution space that has an impact on the total utilization of the unfolded graph.

Determining the upper bound $\vec{\hat{f}}$, however, is not trivial. One common assumption, e.g., in [54] and [71], is to set $\vec{\hat{f}} = [m, m, \cdots, m]$, where m is the number of PEs. In this section, we show, using an example, that this assumption may limit the solution space. As a consequence, the limited solution space might not contain the optimal solution to Problems 5.3.1 and 5.3.2.

Let us consider G_1 in Figure 5.1(a) and suppose that 2 PEs are available. The optimal alternative graph of G_1 is G_3, shown in Figure 5.2, when the vector of unfolding factors is $\vec{f} = [1, 2, 4, 1, 1]$. First, the repetition vector of G_3 can be computed according to Equation (5.1) as

$$\begin{aligned}\vec{q}_{G_3} &= [q_{1,1}, q_{2,1}, q_{2,2}, q_{3,1}, q_{3,2}, q_{3,3}, q_{3,4}, q_{4,1}, q_{5,1}] \\ &= [4, 2, 2, 2, 2, 2, 2, 4, 4].\end{aligned}$$

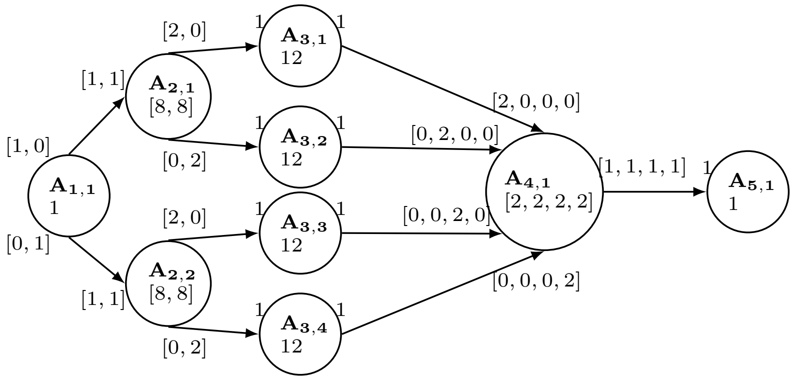


Figure 5.2: G_3: Optimal alternative graph of G_1 in Figure 5.1(a) with unfolding factors $f_2 = 2, f_3 = 4$ when scheduled on 2 PEs.

It follows that $\hat{W}_{G_3} = q_{3,1} \times C_{3,1} = 2 \times 12$ and $\mathrm{lcm}(\vec{q}_{G_3}) = 4$. Solving Equation (2.16) on page 34 yields the minimum period of the sink actor $A_{5,1}$ as

$$\check{T}_{snk} = \frac{4}{4} \cdot \lceil \frac{24}{4} \rceil = 6.$$

To achieve $\check{T}_{\mathrm{snk}} = 6$, it requires 6 PEs. Then, we can scale all periods of the actors in G_3 by $c = 3$, which yields a period $T_{\mathrm{snk}} = 3\check{T}_{\mathrm{snk}} = 18$. According to Proposition 5.4.1, the graph G_3 is schedulable on $\lceil \frac{6}{3} \rceil = 2$ PEs. After scaling the periods of all actors, the total utilization U_{G_3} of G_3 on 2 PEs is $U_{G_3} = 2.0$. Since Lemma 5.3.1 states that the maximum utilization corresponds to the minimum period that a CSDF graph can achieve, no shorter period can be achieved for G_3. Thus, G_3 is the optimal alternative graph of G_1 for 2 PEs with an unfolding factor $f_3 = 4$, which is greater than the number of PEs available. Therefore, this example shows that the optimal solution is beyond $\hat{\vec{f}} = [2,2,2,2,2]$, which defines the solution space if we set $\hat{\vec{f}} = [m,m,m,m,m]$. Hence, we conclude that the upper bound on an unfolding factor is not necessarily equal to the number of PEs.

Now, we derive the upper bound on the unfolding factor for each actor in the initial SDF graph by stating the following theorem:

Theorem 5.5.1. *Given an SDF graph G under SPS, suppose that each actor A_i is to be unfolded by a factor f_i. The upper bound on f_i according to Definition 5.5.1 can be computed as follows:*

$$\hat{f}_i = \frac{\mathrm{lcm}\{x_1, x_2, \cdots, x_n\}}{x_i}, \tag{5.10}$$

where

$$x_i = \frac{\mathrm{lcm}\{W_1, W_2, \cdots, W_n\}}{W_i}. \tag{5.11}$$

W_i is the workload of actor A_i given in Definition 2.3.2 on page 34.

Proof. Suppose that G' is the CSDF graph obtained by unfolding each actor A_i in the initial SDF graph G by $\hat{f}_i$. From Definition 5.5.1, it follows that every replica $A_{i,f}$ in G' has $\check{T}_{i,f} = C_{i,f} = C_i$. Therefore, we can re-write Equation 2.16 on page 34 as:

$$C_i = \frac{\mathrm{lcm}(\vec{q'})}{q_{i,f}} \left\lceil \frac{\hat{W}_{G'}}{\mathrm{lcm}(\vec{q'})} \right\rceil \tag{5.12}$$

where $q_{i,f}$ is the repetition of $A_{i,f}$ in G'. Equation 5.12 can be re-written as:

$$q_{i,f} C_i = \mathrm{lcm}(\vec{q'}) \left\lceil \frac{\hat{W}_{G'}}{\mathrm{lcm}(\vec{q'})} \right\rceil \tag{5.13}$$

Since $\mathrm{lcm}(\vec{q'}) \lceil \hat{W}_{G'} / \mathrm{lcm}(\vec{q'}) \rceil$ is constant, then we re-write Equation 5.13 as:

$$q_{1,1} C_1 = q_{1,2} C_1 = \ldots = q_{1,f_1} C_1 = \ldots = q_{n,f_n} C_n \tag{5.14}$$

Now, we can write $q_{i,f} = x_i \cdot q_i$, where q_i is the repetition of A_i in the initial SDF graph and x_i is an integer factor. That is:

$$x_1 q_1 C_1 = x_2 q_2 C_2 = \cdots = x_n q_n C_n \tag{5.15}$$

Equation 5.15 can be re-written as:

$$x_1 W_1 = x_2 W_2 = \cdots = x_n W_n \tag{5.16}$$

where W_i is the workload of actor A_i according to Definition 2.3.2 on page 34. The minimum solution to Equation 5.16 is:

$$x_i = \frac{\mathrm{lcm}\{W_1, W_2, \cdots, W_n\}}{W_i} \tag{5.17}$$

Since $q_{i,f} = x_i q_i$ and the graph is unfolded by $\vec{\hat{f}}$, we can substitute this in Equation 5.1 to get:

$$x_i q_i = \frac{q_i \,\mathrm{lcm}(\vec{\hat{f}})}{\hat{f}_i} \tag{5.18}$$

which can be re-written as:

$$x_i \hat{f}_i = \mathrm{lcm}(\vec{\hat{f}}) \tag{5.19}$$

Since $\mathrm{lcm}(\vec{\hat{f}})$ is constant, Equation 5.19 can be re-written as:

$$x_1 \hat{f}_1 = x_2 \hat{f}_2 = \cdots = x_n \hat{f}_n \tag{5.20}$$

The minimum solution to Equation 5.20 is:

$$\hat{f}_i = \frac{\mathrm{lcm}\{x_1, x_2, \cdots, x_n\}}{x_i} \tag{5.21}$$

■

Now, we give an example on how to compute $\vec{\hat{f}}$. For G_1 in Figure 5.1(a), we first compute $\mathrm{lcm}\{W_1, W_2, W_3, W_4, W_5\} = 24$. Then, $\vec{x}$ containing the values of x_i is given by

$$\begin{aligned} \vec{x} &= [x_1, x_2, x_3, x_4, x_5] \\ &= [\frac{24}{1}, \frac{24}{8}, \frac{24}{24}, \frac{24}{2}, \frac{24}{1}] \\ &= [24, 3, 1, 12, 24]. \end{aligned}$$

Finally we obtain $\mathrm{lcm}(\vec{x}) = 24$, and

$$\begin{aligned} \vec{\hat{f}} &= [f_1, f_2, f_3, f_4, f_5] \\ &= [\frac{24}{24}, \frac{24}{3}, \frac{24}{12}, \frac{24}{12}, \frac{24}{24}] \\ &= [1, 8, 24, 2, 1]. \end{aligned}$$

5.6 The Algorithm

Based on the upper bounds on unfolding factors, we devise, in this section, an efficient algorithm to solve Problems 5.3.1 and 5.3.2 under a given number of PEs.

The algorithm accepts as an input the following:

1. the initial SDF graph G;
2. the number of available PEs m;

3. the vector containing the upper bounds on the unfolding factors $\vec{\hat{f}}$ computed using Equation 5.10;

4. a pre-specified quality factor $\rho \in (0,1]$, which is used to terminate the algorithm. $\rho = 1$ indicates the highest quality that m PEs must be fully utilized when allocating the resulting graph to the m PEs.

The outputs of the algorithm are:

1. a vector of unfolding factors that is the solution to Problems 5.3.1 and 5.3.2;

2. the allocation of the unfolded SDF graph on m PEs;

3. the minimum achievable period of the sink actor in the unfolded SDF graph on m PEs which is the objective of Problem 5.3.1;

4. the maximum utilization of the unfolded SDF graph on m PEs which is the objective of Problem 5.3.2.

The algorithm builds, incrementally during its execution, a list of nodes in which each node represents a possible vector of unfolding factors $\vec{f}$. Initially, the list contains only a single node which corresponds to the given initial SDF graph with a vector of unfolding factors $\vec{f} = \vec{1}$. Then, we compute the minimum period of the sink actor T_{snk} in the initial SDF graph G, when G is allocated on m PEs, and its total utilization U_G. Both values initialize a tuple $(T_{\text{best}}, U_{\text{best}})$ which holds the period and total utilization of the current best solution. During the execution of the algorithm, new nodes are created and added to the list, where a node represents an alternative CSDF graph G' of the initial graph G with a vector $\vec{f}$ of unfolding factors. Each entry $f_i \in \vec{f}$ ranges from 1 up to $\hat{f}_i$ derived in Equation (5.10).

A newly created node inherits from its previous node a copy of the unfolding factors vector $\vec{f}_{\text{prev}}$ used by the previous node to generate the unfolded graph G'_{prev}. After that, we search in G'_{prev} for the bottleneck actor, denoted by $A_{b,f}$, which is the one with the maximum workload $\hat{W}_G$ as defined in Definition 2.3.2 on page 34. If multiple actors have the same maximum workload, then the one with the smallest code size is selected. Next, we increment by one the entry f_b in the inherited unfolding factors vector $\vec{f}_{\text{prev}}$, thereby, obtaining $\vec{f}_{\text{curr}}$. Then, we unfold the initial graph G by the factors in $\vec{f}_{\text{curr}}$ which results in a CSDF graph G'_{curr}. The next step is to evaluate the unfolded graph G'_{curr} when it is allocated on m PEs. The procedure for evaluating G'_{curr} is explained in details later. The result of the evaluation procedure is the minimum period of the sink actor T_{snk} in G'_{curr}, when G'_{curr} is allocated on m

PEs, and the total utilization of the graph U_{curr}. If the obtained U_{curr} is higher than U_{best} corresponding to the current best solution (i.e., T_{snk} smaller than T_{best}), then T_{best} and U_{best} are updated with T_{snk} and U_{curr}, respectively. Otherwise, T_{best} and U_{best} remain unchanged.

The creation of new nodes is terminated when one of the following conditions is met:

1. The total utilization $U_{G'}$ of the CSDF graph at the current node satisfies $U_{G'} \geq \rho m$, where $\rho \in (0,1]$ is the quality factor given as an input to the algorithm.
2. The unfolding factor f_i of an actor A_i exceeds either its upper bound $\hat{f}_i$ if A_i is stateless, or 1 if A_i is stateful or a data source/sink actor. Recall that stateful actors together with the data source and sink actors cannot be unfolded.

After the creation of new nodes is terminated, we select the first node in the list that has a minimum sink period and a total graph utilization equal to T_{best} and U_{best}, respectively. The selected node contains the solution to Problems 5.3.1 and 5.3.2.

Evaluating Unfolded Graphs

As explained previously, at each node, the initial SDF graph G is unfolded to produce a CSDF graph $G' = (\mathcal{A}', \mathcal{E}')$. Then, we compute two values for G', i.e.,

1. the minimum sink actor period T_{snk} when G' is allocated on m PEs;
2. its total utilization $U_{G'}$.

In this section, we explain in detail how these two values are computed. Recall from Section 5.4 that T_{snk} can be scaled by a scaling factor c given by $T_{\text{snk}} = c\check{T}_{\text{snk}}$, and $U_{G'}$ can be computed as follows:

$$U_{G'} = \sum_{A_{i,f} \in \mathcal{A}'} \frac{C_{i,f}}{c \cdot \check{T}_{i,f}}. \tag{5.22}$$

Recall also that the objective of Problem 5.3.2 is to maximize the utilization. Therefore, we need to find a value of scaling factor c, such that all actors $A_{i,f} \in G'$ are schedulable on m PEs and $U_{G'}$ is maximized. To do so, we first bound the search range for c by deriving its lower and upper bounds. Using any allocation algorithm, we have from Proposition 5.4.1 a lower bound on c, denoted by $\check{c}$, as follows:

$$\check{c} = \left\lceil \frac{1}{m} \sum_{A_{i,f} \in \mathcal{A}'} \frac{C_{i,f}}{\check{T}_{i,f}} \right\rceil. \tag{5.23}$$

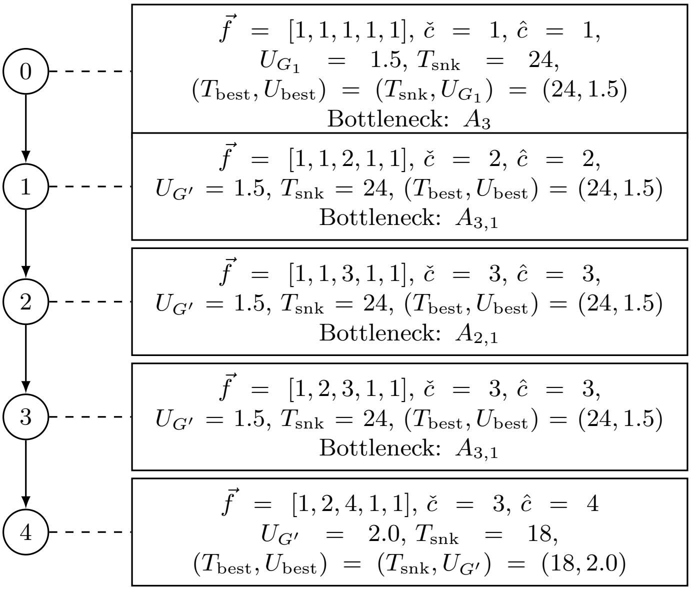
Algorithm 6: The procedure for evaluating an unfolded graph.

```
Input: A CSDF graph $G'$, number of available PEs $m$, and the period and
       total utilization corresponding to the current best solution $T_{\text{best}}$ and
       $U_{\text{best}}$.
Result: $alloc$ which is an $m$-partition describing the allocation of the actors in
        $G'$ onto $m$ PEs
Compute $\check{c}$ using Equation 5.23 and $\hat{c}$ using Equation 5.24 ;
for $s = \check{c}$ to $\hat{c}$ do
    Compute the period $T_{i,f}$ of each actor $A_{i,f}$ as $T_{i,f} = c\check{T}_{i,f}$ ;
    if $T_{snk} \geq T_{best}$ then
        return $\emptyset$ ;
    Compute the utilization $U_{G'}$ using Equation 5.22;
    Find an $m'$-partition of the actors in $G'$, denoted by $alloc$, using the FFD
    algorithm and assuming the EDF scheduling algorithm;
    if $m' \leq m$ then
        $U_{\text{best}} = U_{G'}$, $T_{\text{best}} = T_{\text{snk}}$;
        return $alloc$ ;
```

That is, for any AA, the scaling factor c cannot be less than $\check{c}$. From Proposition 5.4.2, we compute, using the approximation ratio of the First-Fit Decreasing (FFD) allocation algorithm given in Equation (2.23) on page 37, the upper bound on the scaling factor c, denoted by $\hat{c}$, as follows:

$$\hat{c} = \left\lceil \frac{11}{9m} \sum_{A_{i,f} \in \mathcal{A}'} \frac{C_{i,f}}{\check{T}_{i,f}} \right\rceil + 1. \tag{5.24}$$

Once the lower and upper bounds of c are found using Equation (5.23) and Equation (5.24), respectively, we perform a linear search to seek the smallest c, such that CSDF graph G' is schedulable on m PEs. Specifically, we check if an m-partition of all actors in G' exists, assuming the EDF scheduling algorithm and the FFD allocation algorithm explained in Section 2.3. The complete procedure for evaluating the unfolded graphs is depicted in Algorithm 6. If the period resulting from a given scaling factor c is greater than T_{best}, then Algorithm 6 terminates immediately to speed-up the search.



$\vec{f} = [1,1,1,1,1]$, $\check{c} = 1$, $\hat{c} = 1$, $U_{G_1} = 1.5$, $T_{\text{snk}} = 24$, $(T_{\text{best}}, U_{\text{best}}) = (T_{\text{snk}}, U_{G_1}) = (24, 1.5)$
Bottleneck: A_3

$\vec{f} = [1,1,2,1,1]$, $\check{c} = 2$, $\hat{c} = 2$, $U_{G'} = 1.5$, $T_{\text{snk}} = 24$, $(T_{\text{best}}, U_{\text{best}}) = (24, 1.5)$
Bottleneck: $A_{3,1}$

$\vec{f} = [1,1,3,1,1]$, $\check{c} = 3$, $\hat{c} = 3$, $U_{G'} = 1.5$, $T_{\text{snk}} = 24$, $(T_{\text{best}}, U_{\text{best}}) = (24, 1.5)$
Bottleneck: $A_{2,1}$

$\vec{f} = [1,2,3,1,1]$, $\check{c} = 3$, $\hat{c} = 3$, $U_{G'} = 1.5$, $T_{\text{snk}} = 24$, $(T_{\text{best}}, U_{\text{best}}) = (24, 1.5)$
Bottleneck: $A_{3,1}$

$\vec{f} = [1,2,4,1,1]$, $\check{c} = 3$, $\hat{c} = 4$, $U_{G'} = 2.0$, $T_{\text{snk}} = 18$, $(T_{\text{best}}, U_{\text{best}}) = (T_{\text{snk}}, U_{G'}) = (18, 2.0)$

Figure 5.3: The list produced by the algorithm for G_1 in Figure 5.1(a) on 2 PEs with $\rho = 0.95$.

Example

Now, we illustrate our algorithm using graph G_1 in Figure 5.1(a) and schedule the resulting graph G' on 2 PEs (i.e., $m = 2$) with the EDF algorithm. Suppose that $\rho = 0.95$, i.e., the algorithm terminates when $U_{G'} \geq 0.95 \times 2 = 1.9$. The whole list produced by the algorithm is illustrated in Figure 5.3. The numbers inside the nodes correspond to the sequence in which the nodes are created. The algorithm starts with the initial G_1 in node 0 and computes the scaling factors $\check{c}$ and $\hat{c}$ which result in $U_{G_1} = 1.5$ and period $T_{\text{snk}} = 24$. At this point, U_{best} is initialized to 1.5 and T_{best} to 24. Node 1 inherits from node 0 a vector of unfolding factors equal to $[1,1,1,1,1]$. After that, we search in $G'_{\text{prev}} = G_1$ for the bottleneck actor which is A_3. Next, we increment f_3 in the inherited vector of unfolding factors at node 1 resulting in $\vec{f} = [1,1,2,1,1]$. Then, G' is generated and Algorithm 6 is invoked. Since U_{best} cannot be improved, the algorithm continues by creating node 2. At node 2, a new bottleneck actor $A_{2,1}$ is introduced. Therefore, at node 3, the unfolding factor f_2 is incremented by 1. Then, the algorithm continues to node 4, at which one termination criterion is met, namely $U_{G'} \geq 1.9$. As a result, $\vec{f} = [1,2,4,1,1]$ is the solution with $T_{\text{best}} = 18$ and $U_{\text{best}} = 2.0$.

5.7 Experimental Evaluation

In this section, we present the results of evaluating our algorithm presented in Section 5.6 using a set of real-life streaming applications. We evaluate the algorithm by performing two experiments. In the first experiment, we run our algorithm on the applications and report the following:

1. the performance gain resulting from mapping the SDF graph unfolded using the unfolding factors obtained from our algorithm, compared to mapping the initial SDF graph without unfolding;
2. the total time needed to execute our algorithm.

In the second experiment, we compare our proposed algorithm with one of the state-of-the-art search meta-heuristics because Problems 5.3.1 and 5.3.2 in general can be readily formulated and solved by these meta-heuristics, such as genetic algorithms, simulated annealing, etc. However, meta-heuristics normally require parameter tuning to achieve a good solution. In this work, we select a particular meta-heuristic, namely Genetic Algorithms (GA) for two reasons:

1. they have been applied by several researchers to solve similar problems (e.g., [113]);
2. several researchers have reported the optimal parameter settings for GA in the context of our problem (e.g., [118]).

In particular, we compare our proposed algorithm with the one based on the NSGA-II genetic algorithm [36]. Specifically, we compare two metrics:

1. the total execution time needed by each algorithm to find a solution;
2. the total code size of the returned solution.

We conducted all experiments on 11 real-life streaming applications from the StreamIt benchmarks suite [54]. The exact characteristics of the benchmarks are outlined in Table 5.2. Overall, the number of actors in the benchmarks varies from 8 to 120. The WCET of each actor was profiled on the RAW achitecture. The benchmarks include two applications with stateful actors, namely MPEG2 and Vocoder. Both our algorithm and the meta-heuristic were developed in the Phrt tool as part of Daedalus^{RT} shown in Figure 1.7 on page 13. For the NSGA-II GA, we used the implementation from the DEAP [44] framework. All experiments were performed on an Intel Core 2 Duo T9600 CPU running at 2.80 GHz with Linux Kubuntu 10.4.

Table 5.2: Benchmark characteristics.

Benchmark	Num. of Actors	Num. of Edges	Has Stateful Actors?
DCT	8	7	No
FFT	17	16	No
Filterbank	85	99	No
TDE	29	28	No
DES	53	60	No
Serpent	120	128	No
Bitonic	40	46	No
MPEG2	23	26	Yes
Vocoder	114	147	Yes
FMRadio	43	53	No
Channel	55	70	No

Evaluating the Proposed Algorithm

First, we present the performance gain resulting from mapping the unfolded SDF graph, compared to mapping the initial SDF graph without unfolding. We do so by running the algorithm on the benchmarks and mapping each application on a number of PEs that varies from 2 up to 128 PEs. We evaluate the trade-off between the performance gain and total execution time by setting different quality factors $\rho \in \{0.8, 0.85, 0.9, 0.95\}$. To measure the performance gain, we compute, for each benchmark, the ratio between the sink actor period resulting from mapping the unfolded SDF graph, and the period resulting from mapping the initial SDF. This ratio is denoted by Ω and is given by

$$\Omega = \frac{T_{\text{snk}} \text{ of } G'}{T_{\text{snk}} \text{ of } G},$$

where G' is the unfolded graph, and G is the initial SDF graph. A lower value of Ω indicates a shorter sink actor period in the unfolded graph, and therefore, a higher throughput. In Figure 5.4, each vertical line shows the variations in Ω for all the benchmarks. The marker at the middle of each vertical line represents the Geometric Mean (GM) of Ω, while the upper and lower ends of the line represent the maximum and minimum values of Ω, respectively. It can be seen that mapping the unfolded SDF graphs of the benchmarks achieves significant performance improvement compared to mapping the initial SDF graphs of the benchmarks. As the number of PEs increases, the unfolded SDF graphs utilize the PEs much better than the initial SDF

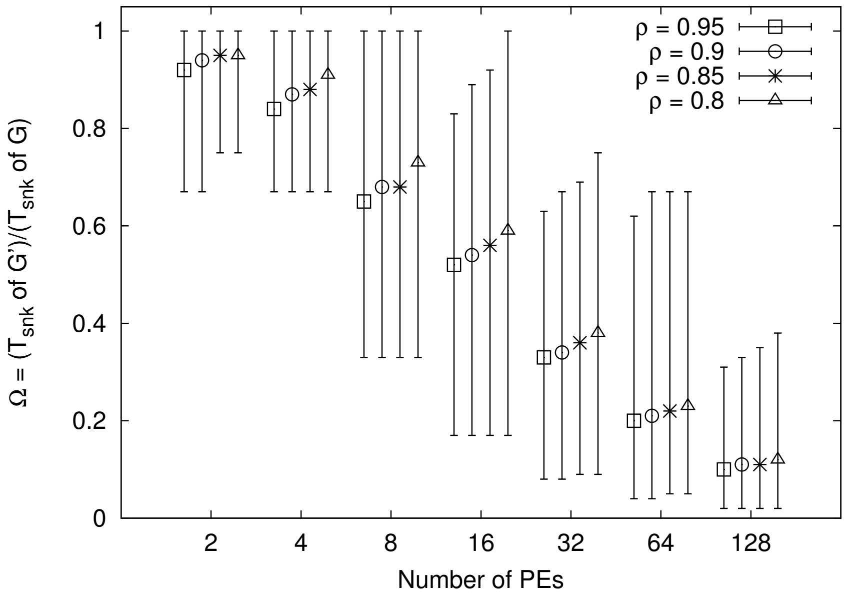


Figure 5.4: Period ratio (lower is better).

graphs. For example, on 64 and 128 PEs, mapping the unfolded SDF graphs with quality factor $\rho = 0.95$ achieves a GM of Ω equal to 0.2 and 0.1, respectively. The DCT benchmark benefits significantly from the algorithm and achieves a GM of Ω equal to 0.021 and 0.042 on 128 and 64 PEs, respectively. Even when a small number of PEs is available, the unfolded SDF graphs still achieve, with quality factor $\rho = 0.95$, a GM of Ω equal to 0.92 and 0.85 on 2 and 4 PEs, respectively.

During the experiment, we also find that the unfolding factor of an actor, obtained using our algorithm, is not necessarily equal to the number of PEs. For example, the obtained unfolded SDF graph of the Vocoder benchmark, when mapped onto 8 PEs, requires the RectangularToPolar actor in the initial SDF graph to be unfolded by a factor of 20. This confirms our statement in Section 5.5. With our provable upper bound, our algorithm results in 4% period reduction for this benchmark compared to other approaches, in which the RectangularToPolar actor is only unfolded with factor 8.

We also evaluate the total execution time of our algorithm, denoted by t_{ours}, when it is invoked on the benchmarks. Figure 5.5 shows the total execution time of our algorithm in seconds for all the benchmarks. For all benchmarks, our algorithm takes a GM of 6.07 seconds for 128 PEs with utilization ratio $\rho = 0.95$. The Serpent benchmark (the largest graph size with 120 actors) takes the longest running time

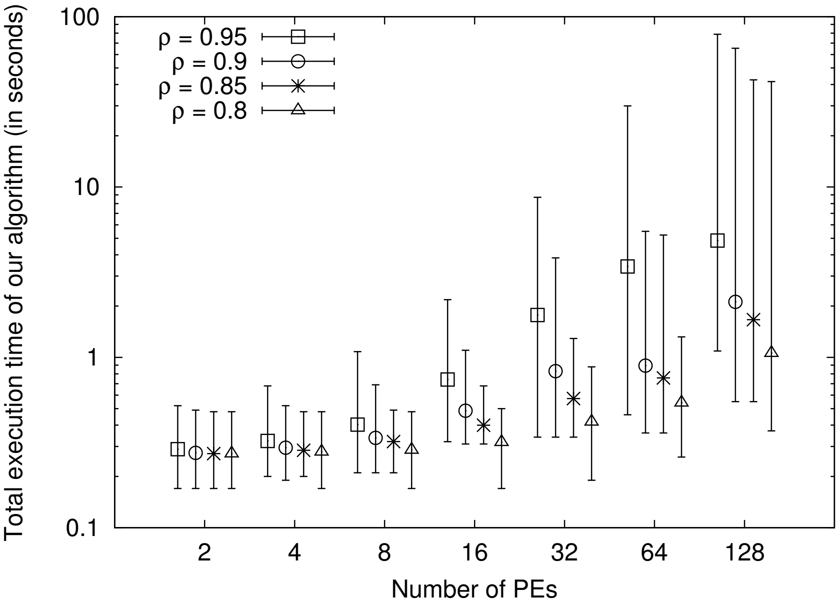


Figure 5.5: Running time of our algorithm.

(78.90 seconds), while the DCT benchmarks takes the shortest running time (1.09 seconds). As the quality factor ρ is decreased from 0.95 to 0.9, the GM of the running time drops to 2.49 seconds for 128 PEs. These results show clearly that our algorithm results, within a reasonable amount of time, in a large performance gain.

Comparison with Genetic Algorithm

To compare our algorithm with the GA-based heuristic, we perform the following steps. First, we run the GA to map each benchmark onto 64 PEs. It outputs an achievable period T and total utilization U_{GA}. Then, we run our algorithm to map the same benchmark onto 64 PEs with a termination criterion $U_{G'} \geq U_{\mathrm{GA}}$. This criterion ensures a fair comparison since our algorithm runs till it finds the same or better solution in terms of the sink actor period and total utilization compared to the best solution found by the GA-based heuristic. Then, we compare two metrics:

1. the total execution time of each algorithm;

2. the total code size Θ resulting from the unfolding factors returned by each

$A_{1,1}$	...	$A_{1,\hat{f}_1}$	...	$A_{n,1}$	...	$A_{n,\hat{f}_n}$
j	...	0	...	1	...	2

Figure 5.6: An example of an individual. The first replica of A_1 is allocated on the jth PE and the $\hat{f}_1$th replica of A_1 does not exist.

algorithm. The total code size is computed as

$$\Theta = \sum_{A_{i,f} \in \mathcal{A}'} \theta_{i,f}$$

where $\theta_{i,f}$ is the code size for actor $A_{i,f}$.

For the GA-based heuristic, each individual (also known as a chromosome) encodes a particular unfolding vector $\vec{f}$ of the initial SDF graph and the allocation of the replicas on m PEs. The structure of an individual is visualized in Figure 5.6. Basically, in an individual, each SDF actor A_i in the initial graph has $\hat{f}_i$ cells as derived in Equation 5.10, indicating that A_i may have up to $\hat{f}_i$ replicas. Each cell may have a value varying from 0 up to m. A value of 0 denotes that the replica does not exist, while a value of 1 up to m denotes the PE on which the replica is allocated. Then, we formulate Problem 5.3.1 as a multi-objective optimization problem with two objectives. The first objective is to minimize the sink actor period, and the second one is to minimize the total code size of the unfolded graph. During the search, we use the evaluation function shown in Algorithm 7. The GA outputs a set of Pareto points, for which we select the one with the shortest achievable period. In order to control the GA, we use the parameters reported in [118], because the target application domain and used platforms are similar to ours. The values of these parameters are given in Table 5.3.

Figure 5.7 shows two ratios. The first ratio (shown in white bars) is the total execution time ratio given by

$$\Omega_t = \frac{t_{\text{GA}}}{t_{\text{ours}}},$$

where t_{GA} is the total time needed by the GA, and t_{ours} is the total time needed by our algorithm. The second ratio in Figure 5.7 (shown in black bars) is the total code size ratio given by

$$\Omega_\Theta = \frac{\Theta_{\text{GA}}}{\Theta_{\text{ours}}},$$

Algorithm 7: Evaluation function in the GA-based meta-heuristic

```
Input: An individual to be evaluated
Result: An achievable period and total code size.
Check if the given individual is valid ;
if the individual is invalid then return $(-1, -1)$ ;
Build the vector of unfolding factors $\vec{f}$ from the individual ;
Generate the CSDF graph $G'$ by unfolding $G$ with $\vec{f}$ using Algorithm 5;
Compute the minimum achievable period $\check{T}_{i,f}$ of each actor $A_{i,f}$ using Equation (2.16) ;
Compute $\check{c}$ according to Equation 5.23 ;
$c = \check{c}$ ;
while true do
    Compute the period $T_{i,f}$ of each actor $A_{i,f}$ as $T_{i,f} = c\check{T}_{i,f}$ ;
    if $G'$ is schedulable on $m$ PEs then
        Compute total code size $\Theta = \sum_{A_{i,f} \in \mathcal{A}'} \theta_{i,f}$;
        Get the period $T_{\text{snk}}$ of the sink actor in $G'$ ;
        return $((T_{snk}, \Theta)$ ;
    else
        $c = c + 1$ ;
```

Table 5.3: Parameters for the genetic algorithm.

Parameter	Recommended value in [118]
Population size	80
Number of generations	300
Crossover rate	0.9
Mutation rate	0.05
Mating rate	0.1

where Θ_{GA} is the total code size of the solution obtained using the GA, and Θ_{ours} is the total code size of the solution obtained using our algorithm. Our algorithm is on average 104 times faster than the GA-based heuristic. For example, to unfold and map the FMRadio benchmark onto 64 PEs, our algorithm takes only 3 seconds, while the GA-based heuristic takes 2439 seconds. This means that our algorithm,

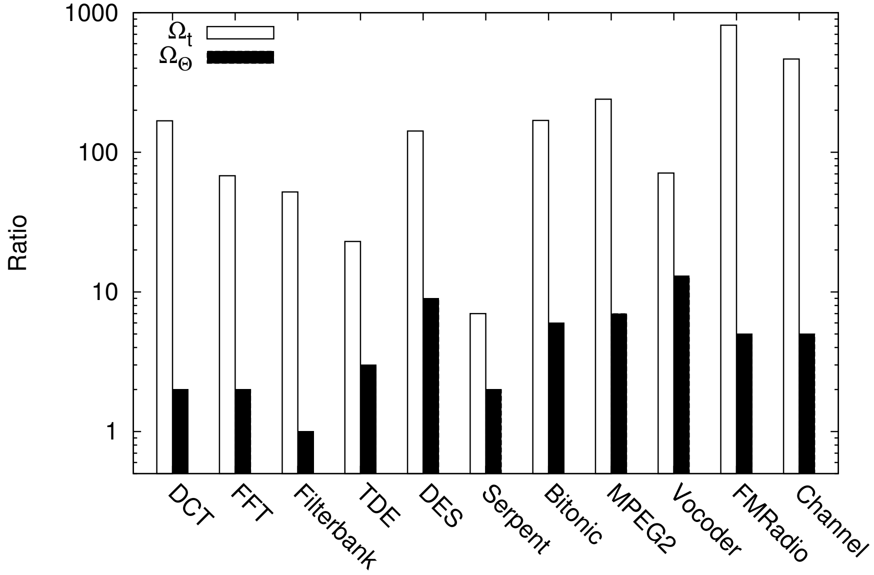


Figure 5.7: The ratios of total execution time Ω_t and total code size Ω_Θ for GA and our algorithm.

for the FMRadio benchmark, is 813 times faster. We also see from Figure 5.7 that our algorithm results in less total code size compared to the GA-based heuristic. These results show clearly that our algorithm outperforms the GA-based heuristic in terms of

1. the time needed to obtain the solution;
2. the total code size of the obtained solution.

Chapter 6

A New MoC for Modeling Adaptive Streaming Applications

Jiali Teddy Zhai, Hristo Nikolov, Todor Stefanov, “Modeling Adaptive Streaming Applications with Parameterized Polyhedral Process Networks”, *In the Proceedings of the 48th IEEE/ACM Design Automation Conference (DAC’11)*, pp. 116–121, San Diego, CA, USA, June 5-9, 2011.

THE popular parallel MoCs for streaming applications are compared in Figure 1.6 on page 10 in terms of expressiveness against compile-time analyzability. For example, models such as SDF, CSDF, and PPN are fairly popular due to their design-time analyzability. However, they have the limitation of allowing only static parameters, whose values are fixed at design-time and they can not be changed at run-time. As a consequence, adaptive streaming applications cannot be expressed using these MoCs.

In contrast, the general MoCs shown in Figure 1.6 include BDF, SADF, KPN, and RPN. They provide capability of modeling adaptive application behavior. However, these general models are not analyzable at design-time. Therefore, we are interested in a model which is able to capture adaptive/dynamic behavior in applications while allowing design-time analyzability to some extent. In this context, Parameterized SDF/CSDF (PSDF/PCSDF) and FSM-SADF models have been proposed as extensions of the SDF/CSDF models. However, scenario reconfiguration in FSM-SADF is limited to a set of pre-defined scenarios. For PSDF/PCSDF, a complex consistency check and computing schedules have to be performed at run-time.

To overcome these issues, in this chapter we introduce a parameterized extension of the PPN model, called Parameterized Polyhedral Process Networks (P^3N). P^3N

improves the expressiveness of PPN, allowing to model adaptive streaming applications. Compared to the aforementioned PSDF/PCSDF and FSM-SADF models, P^3N allows more flexible parameter reconfiguration and enables efficient techniques (less complex) for run-time consistency check by performing part of the consistency check at design-time. In the Daedalus^{RT} design flow, the P^3N MoC is used as the implementation model similar to the PPN MoC for static streaming applications.

6.1 Related Work

In [94], a mathematical model and semantics for reconfiguration of dataflow models are proposed. This approach analyzes where and how parameter values can be changed dynamically and consistently according to dependence relations between parameters. Our P^3N model provides similar semantics for reconfiguration. In particular, for P^3Ns, it is possible to extract dependence relations between dependent parameters at design-time, which is not discussed in [94].

In PSDF/PCSDF [29], separate *init* and *sub-init* graphs are proposed to reconfigure *body* graphs in a hierarchical manner. In the PSDF/PCSDF models, for every combination of parameter values, both computing a schedule and verifying consistency need to be resolved at run-time. In contrast, our P^3N model does not require computing schedules at run-time because all processes are self-scheduled based on the KPN semantics. Therefore, at run-time, only the consistency check has to be performed. The consistency check is furthermore facilitated by the efficient approach we have devised (and present further in this chapter) to extract relations between dependent parameters at design-time.

In SADF [114] and FSM-SADF [47], *detector* actors are introduced to parameterize the SDF model. All valid scenarios must be pre-defined at design-time. Each scenario consists of a set of valid parameter combination that determines a *scenario* of SADF. This guarantees the consistency of SADF in individual scenarios, therefore, no run-time consistency check is required. In a scenario, the SADF model behaves the same ways as the SDF model. Therefore, an SADF graph can be seen as a set of SDF graphs. In the initial FSM-SADF definition, all the production and consumption rates of the dataflow edges are constant within a graph iteration of a scenario. Recently in [49], an extension, called weak consistency, has been made to FSM-SADF. A weakly-consistent FSM-SADF graph allows scenario changes within a graph iteration of a scenario. For P^3N, no prior knowledge of valid parameter combinations is assumed, as the run-time consistency check (see Section 6.3) will guarantee consistency of the P^3N model. We consider that this flexibility is desired compared to FSM-SADF. Once a P^3N model is reconfigured, it behaves as a PPN model. Therefore, a P^3N can be seen as a set of PPNs. Production and consumption patterns in the P^{3}N model thus may still vary during the execution of a particular parameter configuration[1].

For all parameterized models discussed above, the performance penalty due to reconfiguration of parameters at run-time has never been evaluated when these models are executed on MPSoC platforms. This can be an important factor in determining the metric of implementation efficiency (see Figure 1.6(b)) while comparing two adaptive models. In contrast, in this chapter we study the performance penalty introduced by the run-time consistency check and the reconfiguration of our P^{3}Ns on real MPSoC implementations.

6.2 Model Definition

Consider the example of a P^{3}N given in Figure 6.1(c) and a non-parameterized PPN in Figure 6.1(a). Although the dataflow topology of the P^{3}N is the same as the PPN, processes P_2 and P_3 are parameterized by two parameters M and N which values are updated by the environment at run-time using process Ctrl and edges E_7, E_8, E_9. PPN process P_3 is shown in Figure 6.1(b) and P^{3}N Process P_3 is shown in Figure 6.1(d). We use this example throughout the chapter. Below, we formally define the P^{3}N model.

6.2.1 Parameterized Polyhedral Process Networks

Definition 6.2.1 (Parameterized Polyhedral Process Network)**.** A Parameterized Polyhedral Process Network (P^{3}N) is defined by a graph $G = (\mathcal{P}, P_{ctrl}, \mathcal{E})$, where

- $\mathcal{P} = \{P_1, ..., P_{|\mathcal{P}|}\}$ is a set of dataflow processes,
- P_{ctrl} is the control process,
- $\mathcal{E} = \{E_1, ..., E_{|\mathcal{E}|}\}$ is a set of edges, which are FIFOs.

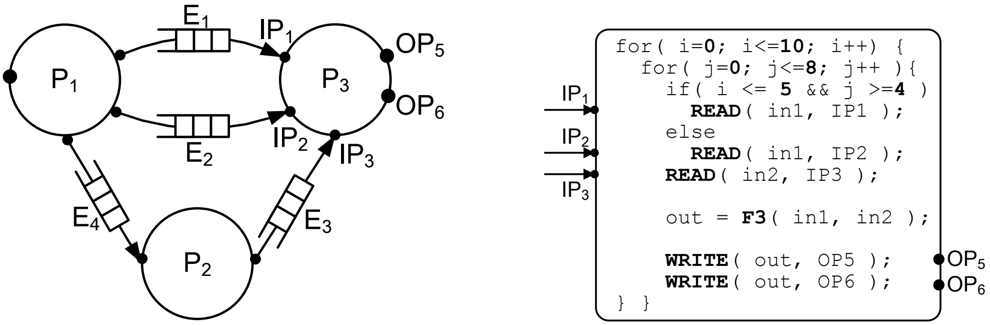
For the P^{3}N shown in Figure 6.1(c), $\mathcal{P} = \{P_1, P_2, P_3\}$ is the set of dataflow processes. Process Ctrl is the control process P_{ctrl}. $\mathcal{E} = \{E_1, E_2, E_3, E_4, E_7, E_8, E_9\}$ is the set of edges, which are FIFOs.

Definition 6.2.2 (Dataflow Process)**.** A dataflow process P is described by a tuple $(\mathcal{I}_P, \mathcal{O}_P, F_P, D_P)$, where

- $\mathcal{I}_P = \{IP_1, ..., IP_{|\mathcal{I}_P|}\}$ is a set of input ports,
- $\mathcal{O}_P = \{OP_1, ..., OP_{|\mathcal{O}_P|}\}$ is a set of output ports,

---

[1] This is called a *process cycle* in Definition 6.2.8 and it is equivalent to a scenario in the SADF model.

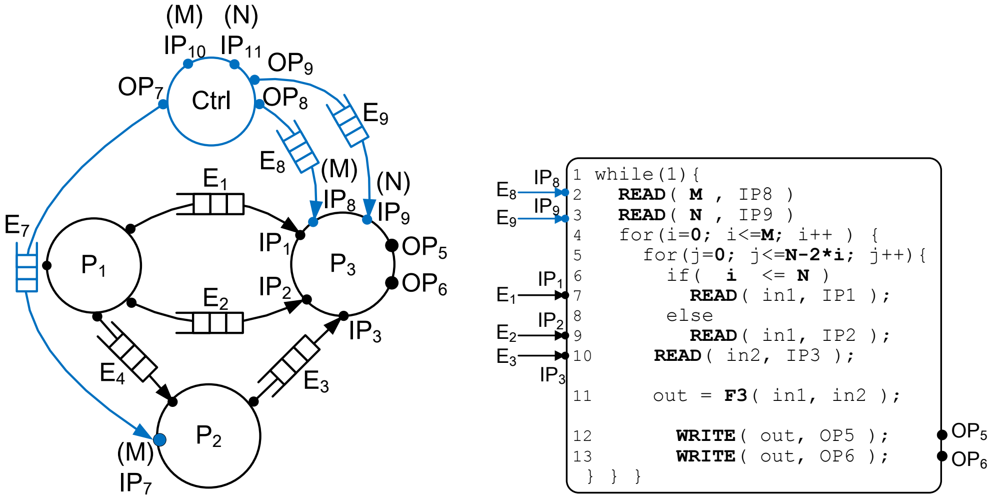


```
for( i=0; i<=10; i++) {
  for( j=0; j<=8; j++ ){
    if( i <= 5 && j >=4 )
      READ( in1, IP1 );
    else
      READ( in1, IP2 );
    READ( in2, IP3 );

    out = F3( in1, in2 );

    WRITE( out, OP5 );
    WRITE( out, OP6 );
} }
```

(a) A PPN. (b) PPN process P_3.



```
while(1){
  READ( M , IP8 )
  READ( N , IP9 )
  for(i=0; i<=M; i++ ) {
    for(j=0; j<=N-2*i; j++){
      if(  i   <= N )
        READ( in1, IP1 );
      else
        READ( in1, IP2 );
     READ( in2, IP3 );

     out = F3( in1, in2 );

       WRITE( out, OP5 );
       WRITE( out, OP6 );
 } } }
```

(c) A P^{3}N. (d) P^{3}N process P_3.

Figure 6.1: Comparsion between a PPN and a P^{3}N.

- F_P is the process function defined by a tuple $(M_P, ARG_{in}, ARG_{out})$, where ARG_{in} and ARG_{out} are sets of variables and $M_P : ARG_{in} \to ARG_{out}$ is a mapping relation,
- D_P is the process domain defined by a parametric polyhedron (see Definition 2.1.3 on page 24).

In Figure 6.1(d), dataflow process P_3 has input ports $I_{P_3} = \{IP_1, IP_2, IP_3, IP_8, IP_9\}$ and output ports $O_{P_3} = \{OP_5, OP_6\}$. Process function $F_3 = (\texttt{F3}, \{\texttt{in1}, \texttt{in2}\}, \texttt{out})$ maps variables `in1` and `in2` to variable `out` with process function `F3`. Assume that the range of parameters M and N is bounded by the polytope (see Definition 2.1.2

on page 23) $\bar{\mathcal{D}}_{(M,N)}$ as

$$\bar{\mathcal{D}}_{(M,N)} = \{(M,N) \in \mathbb{Z}^2 \mid 0 \leq M \leq 100 \wedge 0 \leq N \leq 100\},$$

then the process domain of P_3 is represented as a parametric polyhedron

$$D_{P_3}(M,N) = \{(w,i,j) \in \mathbb{Z}^3 \mid w > 0 \wedge 0 \leq i \leq M \wedge 0 \leq j \leq N - 2i\}.$$

Definition 6.2.3 (Input Port). An input port IP of process P is described by a tuple (V, D_{IP}), where

- V is a variable which:
    - binds the port to process function F_P if $V \in ARG_{in}$;
    - binds the port to process domain D_P or other port domains D_{IP}, D_{OP} if $V \in \vec{p}$, where $\vec{p}$ is the parameter vector defined in Definition 2.1.3 on page 24,
- D_{IP} is the input port domain defined by a parametric polyhedron, where $D_{IP} \subseteq D_P$.

Definition 6.2.4 (Output Port). An output port OP of process P is described by a tuple (V, D_{OP})

- V is a variable which binds the port to process function F_P if $V \in ARG_{out}$,
- D_{OP} is the output port domain defined by a parametric polyhedron, where $D_{OP} \subseteq D_P$.

In Figure 6.1(d), input port IP_1 of process P_3 is defined as $IP_1 = (\texttt{in1}, D_{IP_1})$, where

$$D_{IP_1}(M,N) = \{(w,i,j) \in \mathbb{Z}^3 \mid w > 0 \wedge 0 \leq i \leq M \wedge i \leq N \wedge 0 \leq j \leq N - 2i\}.$$

Similarly, output port OP_5 is defined as $OP_5 = (\texttt{out}, D_{OP_5})$, where OP_5 is bound to variable out and $D_{OP_5}(M,N) = D_3(M,N)$.

Definition 6.2.5 (Control Process). A control process P_{ctrl} is described by a tuple $(I_{ctrl}, F_{ctrl}, O_{ctrl}, D_{ctrl})$, where

- $I_{ctrl} = \{(\perp, p_1, D_{IP}), \ldots, (\perp, p_m, D_{IP})\}$ is a set of input ports.
- F_{ctrl} is the process function defined by a tuple $(Eval, \{\vec{p}, \vec{p}_{old}\}, \vec{p}_{new})$, where $\vec{p}$, $\vec{p}_{old}$ and $\vec{p}_{new}$ are parameter vectors. $Eval : (\vec{p}, \vec{p}_{old}) \rightarrow \vec{p}_{new}$ is the specific mapping relation discussed in Sections 6.2.2 and 6.3.

- $O_{ctrl} = (V, D_{OP}), ..., (V, D_{OP})$ is a set of output ports, where $V \in \vec{p}_{new}$.
- D_{ctrl} is the process domain, where $D_{ctrl} = D_{IP} = D_{OP} = \{w \in \mathbb{Z} \mid w > 0\}$.

Control process Ctrl of the P^{3}N shown in Figure 6.1(c), is given in Figure 6.2(a). Its structure and behavior are discussed in Section 6.2.2 in detail.

Definition 6.2.6 (Edge). An edge $E \in \mathcal{E}$ is defined by a tuple

$$\Big((P_i, OP_k), (P_j, IP_l)\Big)$$

where

- P_i is the process that writes data to edge E through output port OP_k.
- P_j is the process that reads data from edge E through input port IP_l.

In P^{3}Ns, the process domain and port domains are formally defined as parametric polyhedrons (see Definition 2.1.3 on page 24), which allows for mathmatical analysis and manipulation. The polyhedral representation of P^{3}N can be easily converted to sequential nested-loop programs [25] and vice versa [126]. Thus, for the sake of clarity, we present processes in the form of sequential programs in the examples of this chapter.

6.2.2 Operational Semantics

The processes in our P^{3}N MoC execute autonomously and communicate via FIFOs obeying the KPN semantics, which is similar to the PPN MoC. In this section, we formally define our additional, specific operational semantics of the P^{3}N MoC that makes it different from the PPN MoC.

Definition 6.2.7 (Process Iteration). A process iteration of process P is a point $(w, x_1, ..., x_d) \in D_P$, where the following operations are performed sequentially:

1. reading one token from each IP if $(w, x_1, ..., x_d) \in D_{IP}$.
2. executing process function F_P.
3. writing one token to each OP if $(w, x_1, ..., x_d) \in D_{OP}$.

In process P_3 shown in Figure 6.1(d), a process iteration (lines 6-13) consists of reading one token for variable `in1` from either input port IP_1 or IP_2, one token for variable `in2` from input port IP_3, executing process function `F3`, and writing one token for variable `out` to output ports OP_5 and OP_6.

Definition 6.2.8 (Process Cycle)**.** The ith process cycle $CYC_P(i,\ \vec{p_i}) \in D_P$ of a process P is a set of lexicographically ordered process iterations. It is expressed as a polytope

$$CYC_P(i,\vec{p_i}) = \{(w,x_1,...,x_d) \in \mathbb{Z}^{d+1} \mid A\cdot(w,x_1,...,x_d)^T \geq B\cdot\vec{p_i} + b \wedge w = i\},$$

where $i \in \mathbb{Z}^+$ and $\vec{p_i} \in \bar{\mathcal{D}}^P_{\vec{p}} \subseteq \bar{\mathcal{D}}_{\vec{p}}$.

Definition 6.2.9 (Process Execution)**.** Process execution EX_P is a sequence of process cycles denoted by

$$CYC_P(1,\vec{p}_1) \leftarrow CYC_P(2,\vec{p}_2) \leftarrow ... \leftarrow CYC_P(i,\vec{p}_i),$$

where $i \rightarrow \infty$ and $\vec{p_i} \in \bar{\mathcal{D}}^P_{\vec{p}}$.

Overall, every process in a P^3N executes on indefinite number of process cycles in accordance with Definition 6.2.9. For instance, $CYC_{P_3}(2,(7,8))$ denotes the second process cycle that corresponds to the execution of the nested *for*-loops (lines 4-13) when $(M,N) = (7,8)$ during the execution of process P_3 given in Figure 6.1(d).

In the P^3N model, parameters in dataflow processes can change values during the execution, i.e., $\vec{p}_i \neq \vec{p}_{i+1}$. Thus, it is necessary to define the operational semantics related to changing of parameter values. Similar to quiescent points in [94], we also define the points at which changing the value of $\vec{p}$ is permitted.

Definition 6.2.10 (Quiescent Point of a Dataflow Process)**.** A point

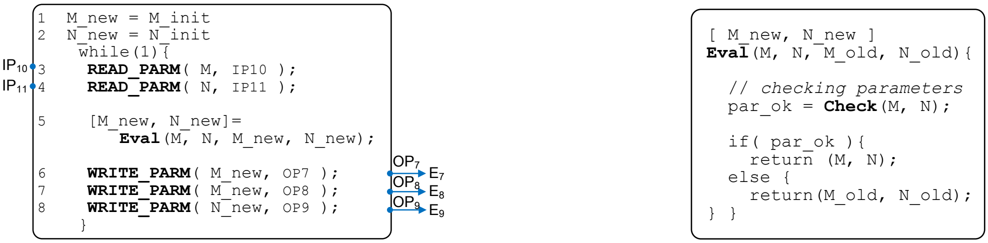
$$Q_P(i,\vec{p_i}) \in CYC_P(i,\vec{p_i})$$

of dataflow process P is a quiescent point if $CYC_P(i,\vec{p_i}) \in EX_P$ and it satisfies

$$\neg(\exists(w,x_1,...,x_d) \in CYC_P(i,\vec{p_i})\ :\ (w,x_1,...,x_d) \prec Q_P(i,\vec{p_i}))$$

According to Definition 6.2.10, dataflow processes can change parameter values at the first process iteration of any process cycle during the execution. For instance, process P_3 given in Figure 6.1(d) updates parameters (lines 2-3) before executing the nested *for*-loops in every process cycle. Generally, updating parameters at each quiescent point is initiated by reading from edges which are connected to the control process.

The control process plays an important role in the P^3N's operational semantics. It reads parameter values from the environment and propagates only valid parameter values to the dataflow processes. Valid parameter values lead to consistent execution of P^3Ns (see Section 6.3). The validity of the parameter values is evaluated by



```
1  M_new = M_init
2  N_new = N_init
    while(1){
3     READ_PARM( M, IP10 );
4     READ_PARM( N, IP11 );

5     [M_new, N_new]=
         Eval(M, N, M_new, N_new);

6     WRITE_PARM( M_new, OP7 );
7     WRITE_PARM( M_new, OP8 );
8     WRITE_PARM( N_new, OP9 );
    }
```

```
[ M_new, N_new ]
Eval(M, N, M_old, N_old){

  // checking parameters
  par_ok = Check(M, N);

  if( par_ok ){
    return (M, N);
  else {
    return(M_old, N_old);
} }
```

(a) Control process Ctrl in Figure 6.1(c).

(b) Function `Eval`.

Figure 6.2: Control process and evaluation function.

process function *Eval* defined in Definition 6.2.5. The control process sends the latest parameter combination that has been evaluated as valid, which means that P^3Ns always respond to changes of the environment as fast as possible. Also, the dataflow processes need to read the parameter values in the correct order. Therefore, to keep the same order of parameter values for all dataflow processes, the control process writes to the control edges, e.g., edges E_7, E_8 and E_9 in Figure 6.1(c), only when all control edges have at least one buffer space available. Here the control edges are implemented as non-bloking-write FIFOs. In case that any of these FIFOs is full, the incoming parameter combination is discarded and the control process continue to read the next parameter combination from the environment. Furthermore, the depth of the FIFOs of the control edges determines how many process cycles of the dataflow processes are allowed to overlap.

Let us consider the P^3N shown in Figure 6.1(c). The behavior of the control process is given in Figure 6.2(a). Process Ctrl starts with at least one valid parameter combination (lines 1-2) and then reads parameters from the environment (lines 3-4) repetitively. For every incoming parameter combination, the process function `Eval` (line 5) checks whether the combination of parameter values is valid. The implementation of function `Eval` is given in Figure 6.2(b). In Section 6.3, we present details about the implementation of function `Check`. If the combination is valid, then function `Eval` returns the current parameter values (`M, N`). Otherwise, the last valid parameters combination (propagated through `M_new, N_new` in this example) is returned. After the evaluation of the parameter combination, process Ctrl writes the parameter values to output ports (lines 6-8) when all edges E_7, E_8, and E_9 have at least one buffer space available.

6.3 Consistency

As defined in Section 6.2, P^{3}Ns operate on input streams with infinite length. Thus, the P^{3}Ns, we are interested in, must be able to execute without deadlocks and only using FIFOs with finite capacity. This kind of P^{3}Ns is considered to be *consistent*. In this section, we first define the consistency condition of the P^{3}N model and then present an approach to preserve the consistent execution of P^{3}Ns at run-time.

Definition 6.3.1 (Consistency of a P^{3}N). A P^{3}N is consistent if

$$\forall E = ((P_i, OP_k), (P_j, IP_l))$$

and $k \to \infty$, it satisfies

$$|D_{OP_k}^{CYC}| = |D_{IP_l}^{CYC}|,$$

where

$$D_{OP}^{CYC} = CYC_{P_i}(c, \vec{p_s}) \cap D_{OP_k},$$

$$D_{IP}^{CYC} = CYC_{P_j}(c, \vec{p_t}) \cap D_{IP_l},$$

$CYC_{P_i}(c, \vec{p_s}) \in EX_{P_i}$, and $CYC_{P_j}(c, \vec{p_t}) \in EX_{P_j}$.

Consider edge E_3 connecting processes P_2 and P_3 of the P^{3}N given in Figure 6.1(c). The execution of processes P_2 and P_3 is illustrated in Fig. 6.3. The access of both processes to edge E_3 is depicted in Figure 6.4. Definition 6.3.1 requires that, for each corresponding process cycle of both processes $CYC_{P_2}(i, M_i)$ and $CYC_{P_3}(i, M_i, N_i)$, the number of tokens $|D_{OP_3}^{CYC}(M)|$ produced by process P_2 to edge E_3 must be equal to the number of tokens $|D_{IP_3}^{CYC}(M, N)|$ consumed by process P_3 from edge E_3.

It is not trivial to preserve the consistent execution of a P^{3}N as defined in Definition 6.3.1. First of all, at each quiescent point Q_P during the execution of a process, the incoming parameter values $\vec{p_s}$ and $\vec{p_t}$ are unknown at design-time, which may result in different $|D_{OP_k}^{CYC}|$ and $|D_{IP_l}^{CYC}|$ at run-time for any edge E connecting dataflow processes. Therefore, whether a P^{3}N can be executed consistently with a given parameter combination, has to be checked at run-time. Secondly, computing $|D_{OP_k}^{CYC}|$ and $|D_{IP_l}^{CYC}|$ is challenging as well. Below, we demonstrate the difficulties associated with checking the consistency using edge E_3 given in Figure 6.4 as an example. One question that naturally arises is which combinations of (M, N) ensure the consistency condition as defined by Definition 6.3.1. For instance, if $(M, N) = (7, 8)$, P_2 produces 25 tokens to E_3 and P_3 consumes 25 tokens from the same edge after one corresponding process cycle of both processes. It can be verified that P_2

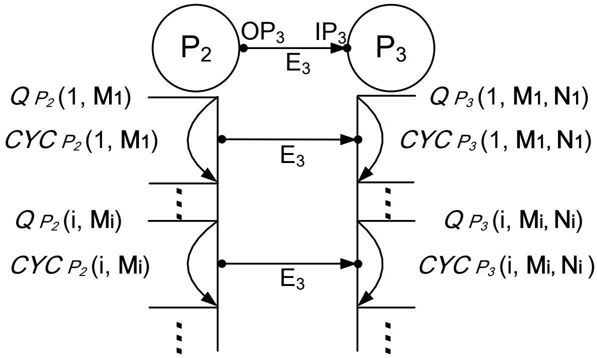
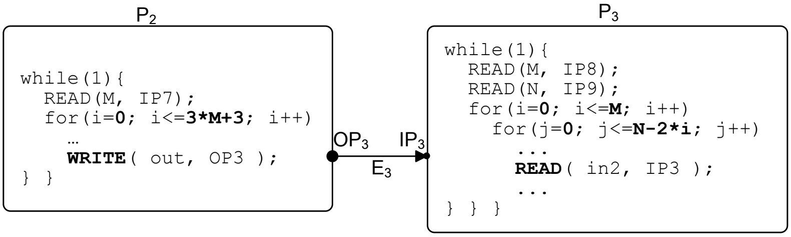


Figure 6.3: Consistent execution of process P_2 and P_3 w.r.t. edge E_3.



```
while(1){
  READ(M, IP7);
  for(i=0; i<=3*M+3; i++)
    ...
    WRITE( out, OP3 );
} }
```

```
while(1){
  READ(M, IP8);
  READ(N, IP9);
  for(i=0; i<=M; i++)
    for(j=0; j<=N-2*i; j++)
      ...
      READ( in2, IP3 );
      ...
} } }
```

Figure 6.4: Which combinations (M,N) do ensure consistency of P^{3}N?

produces 13 tokens to E_3 while P_3 requires 20 tokens from it if $(M,N) = (3,7)$ in a corresponding process cycle. Thereby, in order to complete one execution cycle of P_3 in this case, it will read data from E_3 which will be produced during the next execution cycle of P_2. Evidently this leads to an incorrect execution of the P^{3}N. From this example, we can clearly see that the incoming values of (M,N) must satisfy certain relation to ensure the consistent execution of the P^{3}N.

Although the consistency of a P^{3}N has to be checked at run-time, still some analysis can be done at design-time. First, from Definition 6.3.1, we can see that both D_{OP}^{CYC} and D_{IP}^{CYC} are parametric polytopes. We can check the condition $|D_{OP}^{CYC}| = |D_{IP}^{CYC}|$ by comparing the number of integer points in both parametric polytopes D_{OP}^{CYC} and D_{IP}^{CYC}. This is thus equivalent to computing cardinality of both D_{OP}^{CYC} and D_{IP}^{CYC}. In this work, we use the *Barvinok* library [127] to compute cardinality of a parametric polytope. The *Barvinok* library can solve the problem in polynomial time. In general, the number of integer points inside a parametric polytope is defined as a list of (quasi-)polynomials (see Definition 2.1.4 on page 25). A quasi-polynomial is a polynomial with periodic numbers as coefficients. For instance, considering input port IP_3 shown in Figure 6.4, $D_{IP_3}^{CYC}$ is given as

$$D_{IP_3}^{CYC}(M,N) = \{(i,j) \in \mathbb{Z}^2 \mid 0 \leq i \leq M \wedge 0 \leq j \leq N - 2i\}.$$

The number of tokens $|D_{IP_3}^{CYC}(M,N)|$ read by function `READ(in2, IP3)` in one process cycle is represented as the list of polynomials found by the *Barvinok* library:

$$\begin{cases} 1 + N + N \cdot M - M^2 & \text{if } (M,N) \in C1 \\ 1 + \frac{3}{4}N + \frac{1}{4}N^2 + \frac{1}{4}N - \frac{1}{4} \cdot \{0,1\}_n & \text{if } (M,N) \in C2 \end{cases} \tag{6.1}$$

where $C1$ and $C2$ are called *chambers* (see Definition 2.1.4 on page 25) given as

$$C1 = \{(M,N) \in \mathbb{Z}^2 \mid M \leq N \wedge 2M \geq 1 + N\},$$
$$C2 = \{(M,N) \in \mathbb{Z}^2 \mid 2M \leq N\}.$$

In addition, the second polynomial is a quasi-polynomial, in which $\{0,1\}_n$ is a periodic coefficient with period 2. For instance, function `READ(in2, IP3)` reads $1 + \frac{3}{4} \times 7 + \frac{1}{4} \times 7^2 + \frac{1}{4} \times 7 - \frac{1}{4} \times 1 = 20$ tokens in one process cycle if $(M,N) = (3,7) \in C2$. Below, we present the approach we have devised to extract all parameter combinations that satisfy the consistency condition defined in Definition 6.3.1. Algorithm 8 summarizes the analysis we performed at design-time. Recall that the condition $|D_{OP}^{CYC}| = |D_{IP}^{CYC}|$ must be satisfied for a consistent execution of a P^{3}N. Thus, for each edge connecting dataflow processes, we first compute $|D_{OP}^{CYC}|$ and $|D_{IP}^{CYC}|$. Two lists of (quasi-)polynomials are obtained. If a P^{3}N can execute consistently with a certain parameter combination, individual (quasi-)polynomials in both lists must be equal. We check the equivalence by subtracting the (quasi-)polynomials from both lists symbolically. The symbolic subtraction can result in zero, a non-zero constant, or (quasi-)polynomial. If the result is zero, the consistency is always preserved for all parameters within the range of chamber C_{res}. At run-time, these parameters are propagated immediately to destination dataflow processes. If a non-zero constant is obtained, all parameters within the range of chamber C_{res} are discarded at run-time, because these parameter values would break the consistency condition of the resulting P^{3}N. In the third case, the result is a (quasi-)polynomial in which only some parameter combinations within the range of chamber C_{res} are valid for the consistency condition. We provide two alternatives to extract all valid parameter combinations within this range by solving the resulting equation $q_{res}(\vec{p_{jt}}) = 0$. In the first alternative, the equation can be solved at design-time against all possible parameter combinations. A table, which contains all solutions, i.e., all valid parameter combinations, is generated and stored in function *Check*. At run-time, the control process only propagates those incoming parameter combinations

Algorithm 8: Generation of polynomials for function *Check*

```
   Input: A P^3N
   Result: A list of (quasi-)polynomials
1  foreach edge E corresponding ($OP_{P_O}$, $IP_{P_I}$) do
2      Compute $|D_{OP}^{CYC}|$ and $|D_{IP}^{CYC}|$ using the Barvinok library;
3      foreach (quasi-)polynomial $q_{OP}(\vec{p_j})$ in $|D_{OP}^{CYC}|$ do
4          Get chamber $C$ ;
5          foreach (quasi-)polynomial $q_{IP}(\vec{p_t})$ in $|D_{IP}^{CYC}|$ do
6              Get chamber $C'$ ;
7              Compute $q_{res}(\vec{p_{jt}}) = q_{OP}(\vec{p_j}) - q_{IP}(\vec{p_t})$;
8              Compute chamber $C_{res} = C \cup C'$ ;
9              if $q_{res}(\vec{p_{jt}}) = 0$ then
10                 Consistency is preserved for chamber $C_{res}$;
11             else if $q_{res}(\vec{p_{jt}})$ is a non-zero constant then
12                 Eliminate chamber $C_{res}$;
13             else
14                 Store (quasi-)polynomial $q_{res}(\vec{p_{jt}})$ with $C_{res}$ ;
```

that match an entry in the table. In the second alternative, function *Check* evaluates $q_{res}(\vec{p_{jt}})$ against zero with incoming parameter values at run-time.

Let us consider the example shown in Figure 6.4 again. We apply Algorithm 8 to extract the valid parameter combinations. Besides $|D_{IP_3}^{CYC}(M,N)|$ as given in Equation (6.1), $|D_{OP_3}^{CYC}(M)| = 3M + 4$ is obtained. Subtraction of the (quasi-)polynomials in $|D_{OP_2}^{CYC}(M)|$ and $|D_{IP_2}^{CYC}(M,N)|$ yields two $q_{res}(M,N)$:

$$\begin{cases} (1+N+N\cdot M-M^2)-(3M+4)=0 & \text{if } (M,N)\in C1 \\ (1+\frac{3}{4}N+\frac{1}{4}N^2+\frac{1}{4}N-\frac{1}{4}\cdot\{0,1\}_n)-(3M+4)=0 & \text{if } (M,N)\in C2 \end{cases} \qquad (6.2)$$

where chambers $C1$ and $C2$ are equal to the chambers in Equation (6.1). Clearly this corresponds to the third case in Algorithm 8 (see line 14). The structure of the two alternatives of function `Check` is given in Figures 6.5(a) and 6.5(b). The solutions to Equation (6.2) stored in table `tab` is shown in Figure 6.5(a), whereas evaluating Equation (6.2) directly against zero at run-time is depicted in Figure 6.5(b). In this example, if the range of the parameters is $0 \leq M,N \leq 100$, then there are only 10 valid parameter combinations. In addition, if $0 \leq M,N \leq 1000$, the valid

```
[ par_ok ]
Check ( M , N ) {
  tab =  [ (4, 6),
           (7, 8),
           (15, 12),
            …  ];

  // found M , N in tab
  if ( found )
    return par_ok = true;
  else
    return par_ok = false;
}
```

(a) First alternative.

```
[ par_ok ]
Check ( M , N ) {
  // chamber C1
  if ( M <= N && 2M >= N+1 )
    res = -M^2 + N *M - 3M + N -3;

  // chamber C2
  if ( 2M <= N  )
    res = N^2/4 + 3N/4 + (N%2)/2 - 3M -3;

  if ( res == 0)
    return(true);
  else
    return(false);
}
```

(b) Second alternative.

Figure 6.5: Two alternatives of Function `Check` in Figure 6.2(b).

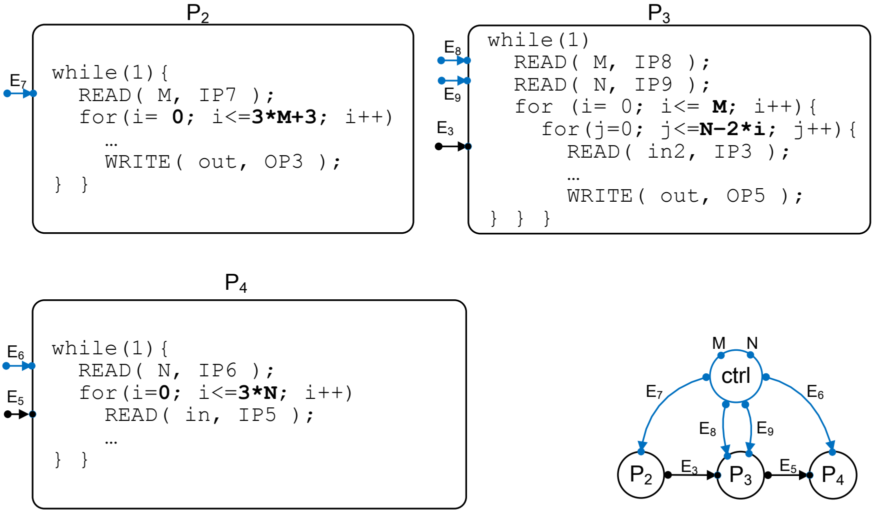
number of parameter combinations are 34, and if $0 \leq M, N \leq 10000$, the number of combinations increases to 114.

6.4 Experimental Results

In order to evaluate the run-time overhead introduced by the reconfiguration of our P^{3}N model, in this section, we present the results we have obtained by mapping a P^{3}N onto a Xilinx Virtex 6 FPGA platform. We have selected a synthetic P^{3}N with complex quasi-polynomials in order to quantify the performance penalty caused by evaluating complex quasi-polynomials at run-time. In order to measure the run-time reconfiguration overhead, we have also implemented the reference PPNs. These PPNs contain only the dataflow processing of the corresponding P^{3}N. The experiments have been conducted using the ESPAM tool and the Xilinx Platform Studio (XPS) 13.2 tool. The generated MPSoCs consist of several MicroBlaze (MB) soft-core processors connected using Xilinx' Fast Simplex Link (FSL) FIFOs. To avoid additional execution overhead, in these experiments, every process has been mapped onto a separate MB processor.

The P^{3}N we consider is depicted in Fig. 6.6. It is formed by the processes in Figure 6.4 and one additional process P_4. Figure 6.6 also shows the representation of processes P_3 and P_4 in order to show the domains $D^{CYC}_{OP_5}(M,N)$ and $D^{CYC}_{IP_5}(N)$ of ports OP_5 and IP_5, connected to edge E_5. Consequently, applying Algorithm 8 yields the following two polynomials for edge E_5:

$$\begin{cases} (1+N+N\cdot M-M^2)-(3N+1)=0 & \text{if } (M,N)\in C1 \\ (1+\frac{3}{4}N+\frac{1}{4}N^2+\frac{1}{4}N-\frac{1}{4}\cdot\{0,1\}_n)-(3N+1)=0 & \text{if } (M,N)\in C2 \end{cases} \tag{6.3}$$



```
while(1){
  READ( M, IP7 );
  for(i= 0; i<=3*M+3; i++)
    …
    WRITE( out, OP3 );
} }
```

```
while(1)
  READ( M, IP8 );
  READ( N, IP9 );
  for (i= 0; i<= M; i++){
    for(j=0; j<=N-2*i; j++){
      READ( in2, IP3 );
      …
      WRITE( out, OP5 );
} } }
```

```
while(1){
  READ( N, IP6 );
  for(i=0; i<=3*N; i++)
    READ( in, IP5 );
    …
} }
```

Figure 6.6: $\mathrm{P}^3\mathrm{N}$ of our experiment

where

$$C1 = \{(M,N) \in \mathbb{Z}^2 \mid M \le N \wedge 2M \ge 1 + N\},$$
$$C2 = \{(M,N) \in \mathbb{Z}^2 \mid 2M \le N\}.$$

For edge E_3, the dependence relation of parameters M and N is already given in Equation (6.2). In a first implementation alternative, we solved Equations 6.2 and 6.3 at design-time and stored all possible parameter values that have been found in a table into function *Check* of control process ctrl. In a second implementation alternative, the polynomials in Equations 6.2 and 6.3 have been evaluated directly in function *Check* at run-time. Furthermore, we have configured five different workloads of the dataflow processes by gradually increasing the execution latency of processes P_2, P_3, and P_4. We have run the MPSoC implementations on an FPGA board for 10 different valid parameter combinations, i.e., process ctrl reconfigures the dataflow processes 10 times within parameter range $0 \le M, N \le 100$.

For the $\mathrm{P}^3\mathrm{N}$s which evaluate the polynomials at run-time (the third bar of each configuration), we have made the following observations. First, configurations 1 and 2 show a relatively large overhead. This is because these configurations correspond to the situation where the execution latency of processes P_2, P_3, and P_4 is very small. That is, the dataflow processes are very light-weight, therefore, they are mostly blocked on reading from the control edges in order to update values of parameters M and N. In this way, configurations 1 and 2 give a good indication about the time needed to evaluate the polynomials. Second, if we increase the execution latency

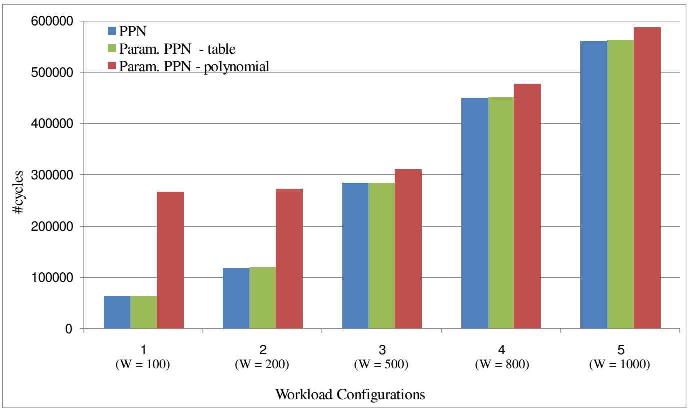


Figure 6.7: Performance results of PPN and P^3N implementations

of the dataflow processes, then the introduced overhead is significantly reduced, see configurations 3, 4, and 5 in Figure 6.7. In these three configurations, the overhead is only 9%, 5%, and 4%, respectively. In addition, we have observed that the absolute values of the overhead (in clk cycles) stay constant. This is because in these three configurations, the dataflow completely overlaps with the evaluation of the polynomials. We have found that the difference with the reference PPN is caused by i) the time for the first evaluation of the polynomials at the beginning of the P^3N execution, i.e., in the beginning no overlap is possible, and ii) the time to read the parameter values from the control edges, i.e., such reading is not present in the reference PPNs. This is an important observation because it shows that the run-time reconfiguration of the P^3N model can be very efficient. Moreover, in most real-life streaming applications, a process execution latency is large enough to cancel out the overhead caused by the evaluation of the polynomials. For example, a discrete cosine transform (used in JPEG encoders) implemented on a MB processor requires a couple of thousand of clk cycles. Therefore, we conclude that the introduced run-time overhead is reasonable considering the more expressive power that the P^3N model provides than other models.

Chapter 7

Hard Real-time Scheduling of Adaptive Streaming Applications

THE initial scheduling framework in the Daedalus^{RT} design flow considers the CSDF MoC as the analysis model and the PPN MoC as the implementation model. For adaptive streaming applications, we have proposed P^{3}N as the implementation model in Chapter 6. However, an analysis MoC for adaptive streaming applications is still missing in Daedalus^{RT}. More importantly, we need proper operational semantics for such a MoC that potentially allows adaptive execution of the MoC and easy HRT analysis. In this chapter, we propose a new analysis MoC in the Daedalus^{RT} design flow that models adaptive streaming applications.

There already exist some adaptive MoCs in literature [89, 114, 129]. Unfortunately, each of them has certain drawback that does not fulfill our needs. For instance, we would like to explicitly have the notion of *mode* in an adaptive MoC. A mode of an adaptive MoC is essentially a static MoC, e.g., the CSDF MoC, when the values of all dynamic parameters are fixed. As a result, the existing HRT analysis developed in Daedalus^{RT} for the CSDF MoC can be reused. For the adaptive MoCs shown in Figure 1.6 on page 10, parameterized CSDF and VPDF MoCs are thus excluded from our consideration because they do not have the notion of mode. At the same time, the expressiveness of MCDF is too restricted.

Furthermore, support for the HRT scheduling and the associated analysis is limited in the existing MoCs, especially during mode transitions. In particular, we wish to have a composable analysis for mode transitions. That is, the analysis of any mode transition is independent from the mode transitions occurred in past. This composable analysis will significantly reduce the complexity of the analysis, as the complexity merely depends on the number of allowed transitions. This is crucial for applications with a large number of modes and possible transitions. As a by-product

Notation	Meaning
c	a computation
Δ	transition delay
L	iteration latency
p	a dynamic parameter
Π	a set of parameter vectors defined in Definition 7.2.1
ψ	parameters used for actors defined in Definitions 7.2.4 and 7.2.5
x	Maximum-Overlap Offset (MOO)

Table 7.1: Additional notations used in Chapter 7 besides the ones introduced in Chapter 2.

of this composable analysis, the implementation efficiency of such a HRT system to support adaptive behavior will be much higher. No complex calculation is needed at run-time, as most of parameters (see Section 7.3) can be computed at compile-time.

Based on the discussion above, we develop a new MoC, Mode-Aware Data Flow (MADF), in this chapter that has the advantages of SADF and VPDF. Inspired by SADF, we characterize the adaptive application behavior with individual modes[1] (see Definition 7.2.7) and transitions (see Definition 7.2.11) between them. Similar to VPDF, the length of production/consumption sequences for an actor varies from one mode to another. The length is only fixed when the mode is known. Based on the clear distinction between modes and transitions, we define operational semantics, in particular a novel transition protocol, to avoid timing interference between modes and transitions. As a result, our HRT analysis is simpler than the state-of-the-art timing analysis [47]. To ease discussion, we use additional notations listed in Table 7.1 besides the ones introduced in Chapter 2.

Scope of Work

We assume that an adaptive streaming application does not have cyclic data dependences. The considered MPSoC platforms in this chapter are homogeneous, i.e., they may contain multiple, but the same type of programmable PEs with distributed memories. Moreover, the platform must be predictable, which means timing guarantees are provided on the response time of hardware components and OS schedulers. The precision-timed (PRET) [79] platform is such an example. On the software side, we assume partitioned scheduling algorithms, i.e, no migration of actors between PEs is allowed. The considered scheduling algorithms on each PE include Fixed-Priority Pre-emptive Scheduling (FPPS) algorithms, such as RM [80], or dynamic

---

[1] "Scenario" for SADF is equivalent to "mode" in our case.

scheduling algorithms, such as EDF [80].

7.1 Related Work

For FSM-SADF [47], the authors proposed an approach to compute worst-case performance among all mode transitions, assuming the self-timed transition protocol (explained later in Section 7.2.3). Although it is an exact analysis, the approach has inherently exponential time-complexity. Moreover, the approach leads to timing interference between modes upon mode transitions, which makes this approach not applicable for our problem. In contrast, our approach does not introduce interference between modes due to the novel transition protocol proposed in Section 7.2.3. The timing behavior of individual modes and during mode transitions can be analyzed independently. In addition, our approach considers allocation of actors on PEs, which by itself is a harder problem than the one in [47]. In [48], the authors proposed to model scenario transitions in a single FSM. Delays due to scenario reconfiguration are given and explicitly modeled in the FSM. The problem addressed in this chapter is different as we aim at deriving such a delay.

In [45], the author proposes to use a linear model to capture worst-case transition delay and period during scenario transitions of FSM-SADF. Our Maximum-Overlap Offset (MOO, see Section 7.2.3) transition protocol is conceptually very similar to the linear model. However, we obtain the linear model in a different way, specifically simplified for the adopted hard real-time scheduling framework. For instance, finding a reference schedule is not necessary in our case, but being crucial in the tightness of the analysis proposed in [45]. Moreover, our approach solves the problem of changing graph structure during mode transitions, which was not studied in [45].

For VPDF [129], the analysis has been limited to computing buffer sizes under throughput constraints so far. The execution of a VPDF graph on MPSoC platforms under HRT constraints has not been studied. In particular, the allocation of actors and how to switch from one mode to another one are not discussed. Moreover, delay due to mode transitions has not been investigated. Our approach, on the other hand, takes these important factors into account. Therefore, our analysis results are directly reflected in a real implementation.

Mode-controlled data flow (MCDF) [89] is another adaptive MoC whose properties can be partly analyzed at compile-time. The MCDF MoC primarily focuses on SDR applications, where different sub-graphs need to be active in different modes. This is achieved by using *switch* and *select* actors. The author implicitly assumes self-timed scheduling during mode transitions. Based on this assumption, a worst-case timing analysis is developed. Similar to the case of SADF, use of the self-timed scheduling introduces timing interference between modes. As a consequence, the analysis must take into account the sequence of mode transitions of interest. Although the author provides an upper bound of timing behavior for a parameterized sequence of mode transitions, the accuracy is still unknown. In contrast, our approach results in a timing analysis of mode transitions that is independent from already occurred transitions. Moreover, the analysis results are directly reflected in the final implementation. In this sense, our analysis is exact in the timing behavior of mode transitions.

In [93], an analysis is proposed to reason about worst-case response time of a task graph in case of mode change. However, the task graph has very limited expressiveness and is not able to model adaptive application behavior. In this chapter, we define a more expressive MoC that is amenable to adaptive application behavior.

In [104, 108], the authors focus on timing analysis for mode changes of real-time tasks. The starting times of new mode tasks need to be delayed to avoid overloading PEs. The algorithms to compute the starting times were provided. Both works are related to ours because actors allocated on the same PE may also overload the PE after mode transitions. In this case, the starting times of actors in the new mode need to be delayed. In [104, 108], it was assumed that tasks are independent. The proposed algorithms are thus not applicable to the adaptive MoCs, since the starting times of actors in the adaptive MoCs depend on each other due to data dependencies. Moreover, the algorithms in [104, 108] involve high computational complexity because fixed-point equations must be solved at every step in the algorithms.

7.2 Model Definition

7.2.1 Mode-Aware Data Flow (MADF)

Definition 7.2.1 (Mode-Aware Data Flow (MADF)). A Mode-Aware Data Flow (MADF) is a multi-graph defined by a tuple $(\mathcal{A}, A_c, \mathcal{E}, \Pi)$, where

- $\mathcal{A} = \{A_1, \ldots, A_{|\mathcal{A}|}\}$ is a set of dataflow actors;
- A_c is the control actor to determine modes and their transitions;
- $\mathcal{E}$ is the set of edges for data/parameter transfer;
- $\Pi = \{\vec{p}_1, \ldots, \vec{p}_{|\mathcal{A}|}\}$ is the set of parameter vectors, where each $\vec{p}_i \in \Pi$ is associated with a dataflow actor A_i.

Throughout this section, we use graph G_1 shown in Figure 7.1 as the running example to illustrate the definition of MADF and the hard real-time scheduling analysis related to MADF. For G_1, $\mathcal{A} = \{A_1, A_2, A_3, A_4, A_5\}$ is the set of dataflow

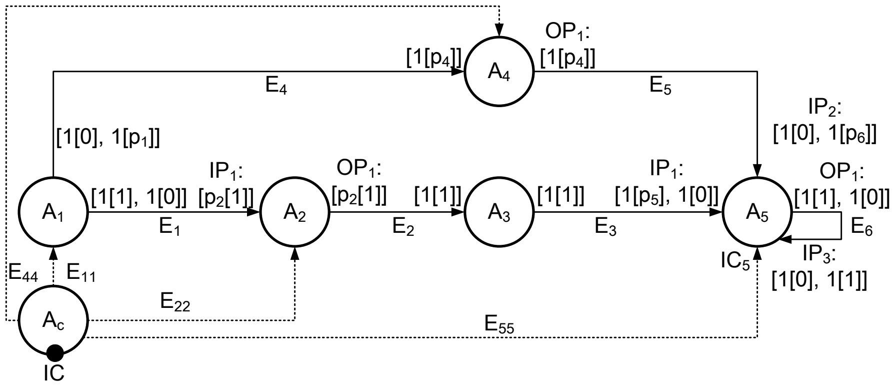


Figure 7.1: An example of MADF graph (G_1).

actors. A_c is the control actor. $\mathcal{E} = \{E_1, E_2, E_3, E_4, E_5, E_6, E_{11}, E_{22}, E_{44}, E_{55}\}$ is the set of edges. For actor A_5, $\vec{p}_5 = [p_5, p_6]$ is the parameter vector. The input port IP_1 of actor A_5 has a consumption sequence $[1[p_5], 1[0]]$, which can be interpreted as $[p_5, 0]$.

Definition 7.2.2 (Dataflow Actor). A dataflow actor A_i is described by a tuple $(\mathcal{I}_i,\ IC_i,\ \mathcal{O}_i, \mathcal{C}_i, M_i)$, where

- $\mathcal{I}_i = \{IP_1, \ldots, IP_{|\mathcal{I}_i|}\}$ is the set of data input ports of actor A_i;
- IC_i is the control input port that reads parameter vector $\vec{p}_i$ for actor A_i;
- $\mathcal{O}_i = \{OP_1, \ldots, OP_{|\mathcal{O}_i|}\}$ is the set of data output ports of actor A_i;
- $\mathcal{C}_i = \{c_1, \ldots, c_{|\mathcal{C}|}\}$ is the set of computations. When actor A_i fires, it performs a computation $c_k \in \mathcal{C}_i$;
- $M_i : \vec{p}_i \rightarrow \{\phi, \bar{C}_i\}$ is a mapping relation, where $\vec{p}_i \in \Pi$, $\phi \in \mathbb{N}^+$, and $\bar{C}_i \subseteq \mathcal{C}_i$ is a sequence of computations $[\bar{C}_i(1), \ldots, \bar{C}_i(k), \ldots, \bar{C}_i(\phi)]$ with $\bar{C}_i(k) \in \mathcal{C}_i, 1 \leq k \leq \phi$.

Actor A_2 in Figure 7.1 has a set of one input port $\mathcal{I}_2 = \{IP_1\}$, a set of one output port $\mathcal{O}_2 = \{OP_1\}$ as well as a control input port IC_2. A set of computations $\mathcal{C}_2 = \{c_1, c_2, c_3\}$ is associated with A_2. The mapping relation M_2 is given in Table 7.2. It can be interpreted as follows: If $p_2 = 2$, actor A_2 repetitively performs computations according to sequence $\bar{C}_2 = [c_1, c_2]$ every time when firing A_2. When $p_2 = 1$, firing A_2 performs computation c_3.

Table 7.2: Mapping relation M_2 for actor A_2 in Figure 7.1.

$\vec{p}_2 = [p_2]$	ϕ	$\bar{C}_2$
2	2	$[c_1, c_2]$
1	1	$[c_3]$

Table 7.3: Function MC_5 defined for actor A_5 in Figure 7.1.

$\mathcal{S}$	$\mathbb{N}^2$
SI^1	$[2,0]$
SI^2	$[1,1]$

Definition 7.2.3 (Control Actor)**.** The control actor A_c is described by a tuple $(IC, \mathcal{O}_c, \mathcal{S}, \mathcal{M}_c)$, where

- $\mathcal{S} = \{SI^1, \ldots, SI^{|\mathcal{S}|}\}$ is a set of mode identifiers, each of which specifies a unique mode;
- IC is the control input port which is connected to the external environment. Mode identifiers are read through the control input port from the environment;
- $\mathcal{O}_c = \{OC_1, \ldots, OC_{|\mathcal{A}|}\}$ is a set of control output ports. Parameter vector $\vec{p}_i$ is sent through $OC_i \in \mathcal{O}_c$ to actor A_i;
- $\mathcal{M}_c = \{MC_1, \ldots, MC_{|\mathcal{A}|}\}$ is a set of functions defined for each actor $A_i \in \mathcal{A}$. For each $MC_i \in \mathcal{M}_c$, $MC_i : \mathcal{S} \rightarrow \mathbb{N}^{|\vec{p}_i|}$ is a function that takes a mode identifier and outputs a vector of non-negative integer values.

For G_1 in Figure 7.1, we have two mode identifiers $\mathcal{S} = \{SI^1, SI^2\}$. At run-time, control actor A_c reads these mode identifiers through control port IC (black dot in Figure 7.1). For actor A_5, $MC_5 \in \mathcal{M}_c$ is given in Table 7.3. As explained previously, the parameter vector for actor A_5 is $\vec{p}_5 = [p_5, p_6]$. Therefore, MC_5 takes a mode identifier and outputs a 2-dimensional vector as shown in the second column in Table 7.3. For instance, mode SI^1 results in a non-negative integer vector $[2,0]$.

To further define production/consumption sequences with variable length, we use the notation $n[m]$ for a sequence of n elements with integer value m, i.e.,

$$n[m] = [\overbrace{m, \ldots, m}^{n \text{ times}}]$$

Definition 7.2.4 (Input Port)**.** An input port *IP* of an actor is described by a tuple (CNS, M_{IP}), where

- $CNS = [\phi_1[cns_1], \ldots, \phi_K[cns_K]]$ is the consumption sequence with ϕ phases, where $\phi = \sum_{i=1}^{K} \phi_i$ is determined by the mapping relation M in Definition 7.2.2, and $cns_1, \ldots, cns_K \in \mathbb{N}$;
- $M_{IP} : \vec{p}_i \rightarrow \psi_{IP}$ is a mapping relation, where $\vec{p}_i \in \Pi$ and

$$\psi_{IP} = \{\phi_1, \ldots, \phi_K, cns_1, \ldots, cns_K\}. \tag{7.1}$$

Definition 7.2.5 (Output Port)**.** An output port *OP* of an actor is described by a tuple (PRD, M_{OP}), where

- $PRD = [\phi_1[prd_1], \ldots, \phi_K[prd_K]]$ is the production sequence with ϕ phases, where $\phi = \sum_{i=1}^{K} \phi_i$ is determined by the mapping relation M in Definition 7.2.2, and $prd_1, \ldots, prd_K \in \mathbb{N}$.
- $M_{OP} : \vec{p}_i \rightarrow \psi_{OP}$ is mapping relation, where $\vec{p}_i \in \Pi$ and

$$\psi_{OP} = \{\phi_1, \ldots, \phi_K, prd_1, \ldots, prd_K\}. \tag{7.2}$$

The consumption/production sequence defined here is a generalization of that for the CSDF MoC (see Section 2.2.2 on page 32). We can see that a CSDF actor has a constant ϕ phases in its consumption/production sequences, whereas the length of the phase of an MADF actor is parameterized by $\phi = \sum_{i=1}^{K} \phi_i$. In addition, the mapping relation M_{IP}/M_{OP} must be provided by the application designer. Consider the two input ports IP_1 and IP_2 of actor A_5 in Figure 7.1. The mapping relations M_{IP_1} and M_{IP_2} are represented as follows:

$$M_{IP_1} : \vec{p}_5 = [p_5, p_6] \rightarrow \psi_{IP_1} = \{\phi_1, \phi_2, cns_1, cns_2\} = \{1, 1, p_5, 0\}, \tag{7.3}$$
$$M_{IP_2} : \vec{p}_5 = [p_5, p_6] \rightarrow \psi_{IP_2} = \{\phi_1, \phi_2, cns_1, cns_2\} = \{1, 1, 0, p_6\}. \tag{7.4}$$

It can be seen that parameter p_5 is mapped to cns_1 of IP_1, parameter p_6 is mapped to cns_2 of IP_2, and ϕ_1 and ϕ_2 both are constant equal to 1. Therefore, the consumption sequence of IP_1 is $CNS = [1[p_5], 1[0]]$ and the consumption sequence of IP_2 is $CNS = [1[0], 1[p_6]]$. Similarly considering output port OP_1 of actor A_4, its mapping relation M_{OP_1} is given as:

$$M_{OP_1} : \vec{p}_4 = [p_4] \rightarrow \psi_{OP_1} = \{\phi_1, prd_1\} = \{1, p_4\}. \tag{7.5}$$

In this case, parameter p_4 is mapped to prd_1 and $\phi_1 = 1$. Therefore, production sequence $PRD = [1[p_4]]$ is obtained for OP_1 of A_4.

Definition 7.2.6 (Edge)**.** An edge $E \in \mathcal{E}$ is defined by a tuple

$$\Big((A_i, OP), (A_j, IP)\Big),$$

where

- actor A_i produces a parameterized number of tokens to edge E through output port OP;
- actor A_j consumes a parameterized number of tokens from E through input port IP.

Considering edge E_5 in Figure 7.1, it connects output port OP_1 of actor A_4 to input port IP_2 of actor A_5.

Definition 7.2.7 (Mode of MADF)**.** A mode SI^i of MADF is a live CSDF [30] graph, denoted as G^i, obtained by setting values of Π in Definition 7.2.1 as follows:

$$\forall k \in \Pi \; : \; \vec{p}_k = MC_k(SI^i), \tag{7.6}$$

where function MC_k is given in Definition 7.2.3.

Definition 7.2.8 (Mode of MADF Actor)**.** An actor A_k in mode SI^i, denoted by A_k^i, is a CSDF [30] actor obtained from A_k as follows:

$$\vec{p}_k = MC_k(SI^i). \tag{7.7}$$

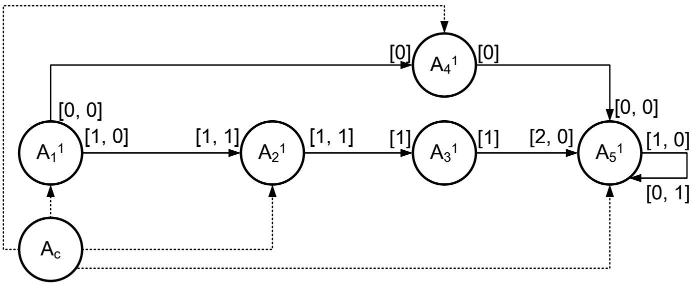
Figure 7.2(a) shows the CSDF graph in mode SI^1 and Figure 7.2(b) shows the CSDF graph in mode SI^2. Consider function MC_5 for actor A_5 in Table 7.3 with parameter vector $\vec{p}_5 = [p_5, p_6]$. For instance, mode SI^1 results in $\vec{p}_5 = [p_5, p_6] = [2, 0]$, where parameter values $p_5 = 2$ and $p_6 = 0$. Consequently, according to mapping relations M_{IP_1} and M_{IP_2} given in Equations 7.3 and 7.4, $cns_1 = p_5 = 2$ can be obtained for input port IP_1 and $cns_2 = p_6 = 0$ for IP_2. This determines actor A_5^1 shown in Figure 7.2(a) for mode SI^1.

Definition 7.2.9 (Inactive Actor)**.** An MADF actor A_i^k is inactive in mode SI^k if the following conditions hold:

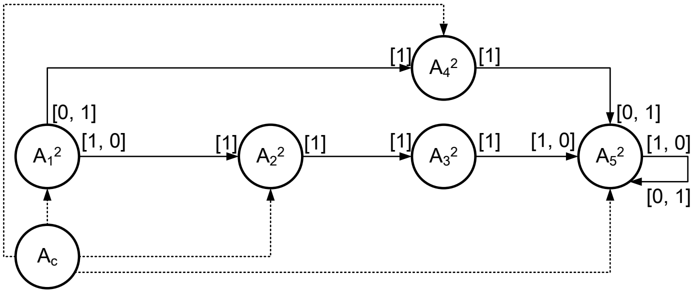
1. $\forall IP \in \mathcal{I}_i \; : \; CNS = [0, \ldots, 0]$;
2. $\forall OP \in \mathcal{O}_i \; : \; PRD = [0, \ldots, 0]$.

Otherwise, A_i^k is called *active* in mode SI^k.

For actor A_4^1 shown in Figure 7.2(a), it has consumption and production sequence $[0]$. Therefore, actor A_4 is said to be inactive in mode SI^1.



(a) CSDF graph G_1^1 of mode SI^1.



(b) CSDF graph G_1^2 of mode SI^2.

Figure 7.2: Two modes of the MADF graph in Figure 7.1.

7.2.2 Operational Semantics

During execution of a MADF graph, it can be either in a steady-state or mode transition.

Definition 7.2.10 (Steady-state)**.** A MADF graph is in a steady-state of a mode SI^i, if it satisfies Equation (7.6) with the same SI^i for all its actors.

Definition 7.2.11 (Mode Transition)**.** A MADF graph is in a mode transition from mode SI^o to SI^l, where $o \neq l$, if some actors have SI^o for Equation (7.7) and the remaining active actors have SI^l for Equation (7.7).

In the steady-state of a MADF graph, all active actors execute in the same mode. As defined previously in Definition 7.2.7 and shown in Figure 7.2(a) and Figure 7.2(b), the steady-state of the MADF graph has the same operational semantics as a CSDF [30] graph. We use $\langle A_i^k, x\rangle$ to denote the xth firing of actor A_i in mode SI^k. At $\langle A_i^k, x\rangle$, it executes computation

$$\bar{C}_i\big((x-1) \mod \phi + 1\big),$$

where $\bar{C}_i$ is given in Definition 7.2.2. The number of tokens consumed and produced are specified according to Definitions 7.2.4 and 7.2.5, respectively. For instance, the xth firing of A_i^k produces the following number of tokens through an output port OP:

$$PRD\big((x-1) \mod \phi + 1\big).$$

In each mode SI^k, the MADF graph is a live CSDF graph and thus has the notion of graph iterations with a non-trivial repetition vector $\vec{q}^k \in \mathbb{N}^{|A|}$ resulting from Equation (2.14) on page 32. Next, we further define mode iterations.

Definition 7.2.12 (Mode Iteration)**.** One iteration It^k of a MADF graph in mode SI^k consists of one firing of control actor A_c and $q_i^k \in \vec{q}^k$ firings of each MADF actor A_i^k.

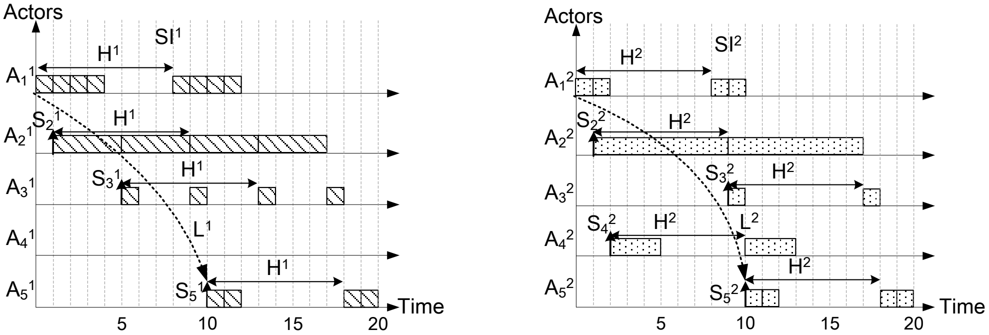
Consider the two modes shown in Figure 7.2(a) and Figure 7.2(b). Repetition vectors $\vec{q}^1$ and $\vec{q}^2$ are:

$$\begin{aligned} \vec{q}^1 &= [4,2,2,0,2], \\ \vec{q}^2 &= [2,1,1,1,2]. \end{aligned} \tag{7.8}$$

For any mode of a MADF graph, i.e., a live CSDF graph, under *any* valid schedule, it has (eventually) periodic execution in time. This holds for CSDF graphs under self-timed schedule [110], K-periodic schedule [31], and SPS [22]. The length of the periodic execution, called *iteration period*, determines the minimum time interval to complete one graph iteration (*cf.* Definition 7.2.12). The iteration period, denoted by H^k, is equal for any actor in the same mode SI^k. During a periodic execution, the starting time of each actor A_i^k, denoted by S_i^k, indicates the time distance between the start of source actor A_{src}^k and the start of actor A_i^k in the same iteration period. Based on the notion of starting times, we define *iteration latency* L^k of a MADF graph in mode SI^k as follows:

$$L^k = S_{\text{snk}}^k - S_{\text{src}}^k, \tag{7.9}$$

where S_{snk}^k and S_{src}^k are the earliest starting times of the sink and source actors, respectively. Figure 7.3 illustrates the execution of both modes SI^1 and SI^2 given in Figure 7.2 under the self-timed schedule. A rectangle denotes WCET of an actor firing. The WCETs of all actors in both modes are given in the third row of Table 7.4 on page 132. Now, it can be seen in Figure 7.3 that iteration period $H^1 = H^2 = 8$. Based on the starting time of each actor, we obtain iteration latencies $L^1 = S_5^1 - S_1^1 = 10 - 0 = 10$ and $L^2 = S_5^2 - S_1^2 = 10 - 0 = 10$ as shown in Figure 7.3.



(a) Mode SI^1 in Figure 7.2(a).

(b) Mode SI^2 in Figure 7.2(b).

Figure 7.3: Execution of two iterations of both modes SI^1 and SI^2 under self-timed scheduling.

7.2.3 Mode Transition

While the operational semantics of a MADF graph in steady-state are the same as that of a CSDF [30] graph, the transition of MADF graph from one mode to another is the crucial part that makes it fundamentally different from CSDF. The protocol for mode transitions has strong impact on the compile-time analyzability and implementation efficiency. In this section, we propose a novel and efficient protocol of mode transitions for MADF graphs.

During execution of a MADF graph, mode transitions may be triggered at run-time by receiving a Mode Change Request (MCR) from the external environment. We first assume that a MCR can be only accepted in the steady-state of a MADF graph, not in an ongoing mode transition. This means that any MCR occurred during an ongoing mode transition will be ignored. Consider a mode transition from SI^o to SI^l. The transition is accomplished by the control actor reading mode identifier SI^l from its control input port (see the black dot in Figure 7.1) and writing parameter values of $\vec{p}_i$ to the control output port connected to each dataflow actor A_i^l according to function MC_i given in Definition 7.2.3. Then, A_i^l reads new parameter values $\vec{p}_i$ from its control input port and sets the sequence of computations according to mapping relation M_i in Definition 7.2.2. The production and consumption sequences are obtained in accordance with M_{IP} and M_{OP} in Definition 7.2.4 and Definition 7.2.5, respectively. Similar to the P^{3}N MoC, we further define that mode transitions are only allowed at quiescent points [94].

Definition 7.2.13 (Quiescent Point of MADF)**.** For a transition from mode SI^o to SI^l, a quiescent point of a MADF actor A_i is a firing $\langle A_i^l, x\rangle$ in a mode iteration It^l

that satisfies

$$\neg\exists\langle A_i^l, y\rangle \in It^l \ : \ y < x. \tag{7.10}$$

Figure 7.4 shows an execution of G_1 in Figure 7.1 with two mode transitions. For instance, the MCR at time $t_{\text{MCR1}} = 1$ denotes a transition request from mode SI^2 to SI^1. The mode transition of actor A_1 is only allowed at the quiescent point (time 2 in Figure 7.4) right before the first firing in mode iteration It^1 of mode SI^1.

Definition 7.2.13 defines mode transitions of MADF graphs as a partially ordered actor firings. However, it does not specify at which time instance a mode transition actually starts. Therefore, below, we focus on the transition protocol that defines the points in time for occurrences of mode transitions. To quantify the transition protocol, we introduce a metric, called *transition delay*, to measure the responsiveness of a protocol to a MCR.

Definition 7.2.14 (Transition Delay). For a MCR at time t_{MCR} calling for a mode transition from mode SI^o to SI^l, the transition delay $\Delta^{o\to l}$ of a MADF graph is defined as

$$\Delta^{o\to l} = \sigma_{\text{snk}}^{o\to l} - t_{\text{MCR}}, \tag{7.11}$$

where $\sigma_{\text{snk}}^{o\to l}$ is the earliest starting time of the sink actor in the new mode SI^l.

In Figure 7.4, we can compute the transition delay for *MCR1* occurred at time $t_{\text{MCR1}} = 1$ as $\Delta^{2\to 1} = 18 - 1 = 17$.

Self-timed (ST) Transition Protocol

In the existing adaptive MoCs like FSM-SADF [47], a protocol, referred here as *Self-Timed* (ST) transition protocol, is adopted. The ST protocol specifies that actors are scheduled in the self-timed manner not only in the steady-state, but also during a mode transition. For FSM-SADF upon a MCR, a firing of a FSM-SADF actor in the new mode can start immediately after the firing of the actor completes the old mode iteration. The only possible delay is introduced due to availability of input data. One reason behind the ST protocol is that the ST schedule for a (C)SDF graph (steady-state of FSM-SADF[2]) leads to its highest achievable throughput. However, the ST protocol generally introduces interference of one mode execution with another one. The time needed to complete mode transitions also fluctuate as the transition delay of an ongoing transition depends on the transitions occurred in the past. We consider this as an undesired effect because mode transitions using the ST protocol become potentially slow and unpredictable. Another consequence of the

---

[2] The steady-state of SADF is defined similarly to that of MADF. The only difference is that a scenario of FSM-SADF is a SDF graph, whereas a mode of MADF is a CSDF graph.

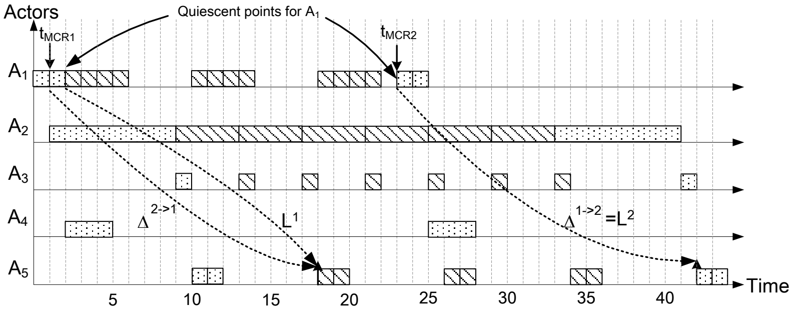


Figure 7.4: An execution of G_1 in Figure 7.1 with two mode transitions under the ST transition protocol. *MCR1* at time t_{MCR1} denotes a transition request from mode SI^2 to SI^1, and *MCR2* at time t_{MCR2} denotes a transition request from mode SI^1 to SI^2.

incurred interference between modes using the ST transition protocol is the high time complexity of analyzing transition delays, because transition delays cannot be analyzed independently for each mode transition. The analysis proposed in [47] uses an approach based on state-space exploration, which has the exponential time complexity.

Consider G_1 in Figure 7.1 and an execution of G_1 with the two mode transitions illustrated in Figure 7.4. The execution is assumed under the ST schedule for both steady-state and mode transitions of G_1. After *MCR1* at time t_{MCR1}, the transition from mode SI^2 to SI^1 introduces interference to execution of the new mode SI^1 from execution of the old mode SI^2. The interference increases the iteration latency of the new mode SI^1 to $L^1 = S_5^1 - S_1^1 = 18 - 2 = 16$ from initially 10 as shown in Figure 7.3(a) when G_1 is only executed in the steady-state of mode SI^1. Even worse, the interference is further propagated to the second mode transition after *MCR2* at time t_{MCR2}. In this case, the iteration latency $L^2 = S_5^2 - S_1^2 = 42 - 23 = 19$ is increased from initially 10 as shown in Figure 7.3(b) when G_1 is only executed in the steady-state of mode SI^2. This example thus clearly shows the problem of the ST protocol. That is, it introduces interference between the old and new modes due to mode transitions, thereby increasing the iteration latency of the new mode in the steady-state after the transition. Furthermore, the increase of iteration latency also potentially increases transition delays as it will be shown in the next section.

Maximum-Overlap Offset (MOO) Transition Protocol

To address the problem of the ST transition protocol explained above, we introduce in this chapter our new transition protocol, called *Maximum-Overlap Offset* (MOO).

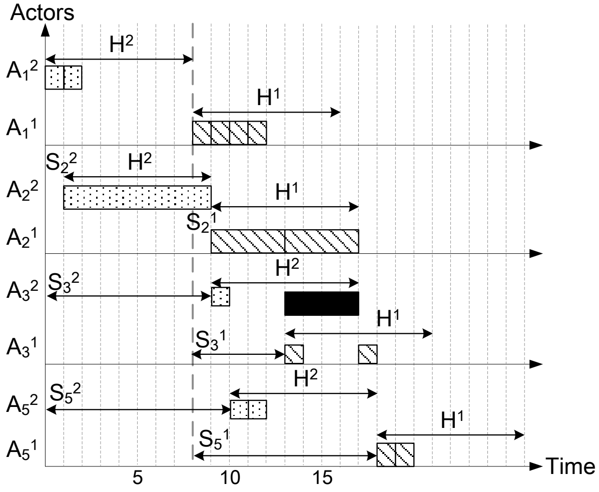


Figure 7.5: An illustration of the Maximum-Overlap Offset (MOO) calculation.

Definition 7.2.15 (Maximum-Overlap Offset (MOO)). For a MADF graph and a transition from mode SI^o to SI^l, Maximum-Overlap Offset (MOO), denoted by x, is defined as

$$x = \begin{cases} \max_{A_i \in \mathcal{A}^o \cap \mathcal{A}^l}(S_i^o - S_i^l) & \text{if } \max_{A_i \in \mathcal{A}^o \cap \mathcal{A}^l}(S_i^o - S_i^l) > 0 \\ 0 & \text{otherwise,} \end{cases} \tag{7.12}$$

where $\mathcal{A}^o \cap \mathcal{A}^l$ is set of actors active in both modes SI^o and SI^l.

Basically, we first assume that the new mode SI^l starts immediately after the source actor A^o_{src} of the old mode SI^o completes its last iteration It^o. All actors A_i^l of the new mode execute according to the earliest starting times S_i^l and iteration period H^l in the steady-state. Under this assumption, if the execution of the new mode overlaps with the execution of the old mode in terms of iteration periods H^o and H^l, we then need to offset the starting time of the new mode by the maximum overlap among all actors. In this way, the execution of the new mode will have the same iteration latency as that of the new mode in the steady-state, i.e., no interference between the execution of both old and new modes.

Consider *MCR1* at time t_{MCR1} shown in Figure 7.4. Obtaining MOO x is illustrated in Figure 7.5. We first assume that the new mode SI^1 starts at the time when the source actor A_1^2 completes the last iteration at time 8 (see bold, dashed line in Figure 7.5). Actors A_i^1 in the new mode start as if they executed in the steady-state of mode SI^1. Then, we can see that, for actor A_3, the execution of A_3^1 in

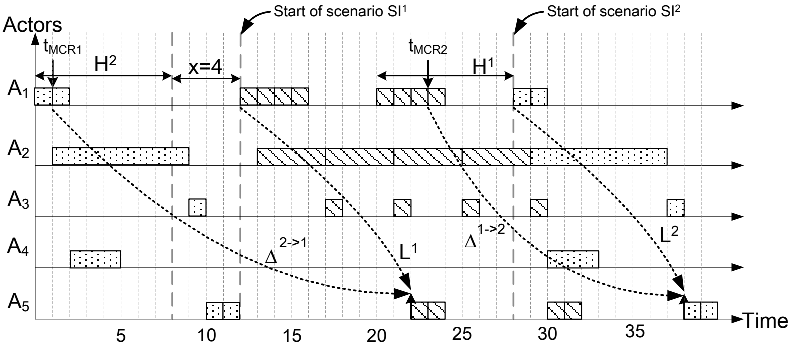


Figure 7.6: The execution of G_1 with two mode transitions under Maximum-Overlap Offset (MOO) protocol.

the new mode SI^1 according to S_3^1 in Figure 7.3(a) overlaps 4 time units (solid bar in Figure 7.5) with the execution of A_3^2 in the old mode SI^2 in terms of iteration periods H^2 and H^1. This is also the maximum overlap between the execution of actors in modes SI^2 and SI^1. According to Definition 7.2.15, x can be obtained through the following equations:

$$\begin{aligned}
S_1^2 - S_1^1 &= 0 - 0,\\
S_2^2 - S_2^1 &= 1 - 1 = 0,\\
S_3^2 - S_3^1 &= 9 - 5 = 4,\\
S_5^2 - S_5^1 &= 10 - 10 = 0.
\end{aligned}$$

Therefore, it results in an offset $x = \max(0,0,4,0) = 4$ to the start of mode SI^1 and is shown in Figure 7.6. The starting time of the new mode SI^1, namely the source actor A_1^1, must be first delayed to the time when A_2^1 completes the iteration period H^2 in the last iteration, namely time 8 as shown as the first bold line in Figure 7.6. In addition, the MOO $x = 4$ must be further added to the starting time of A_1^1 (the second bold line in Figure 7.6). Figure 7.6 also shows another transition from mode SI^1 to SI^2 with a MCR occurred at time $t_{\text{MCR2}} = 23$. The starting time of the source actor A_1^2 in the new mode SI^2 must be first delayed to the time 28 (the thrid bold line in Figure 7.6), namely the time when A_1^1 completes the last iteration in the old mode

SI^1. To calculate the MOO x for this transition, the following equations hold:

$$\begin{aligned}
S_1^1 - S_1^2 &= 0 - 0, \\
S_2^1 - S_2^2 &= 1 - 1 = 0, \\
S_3^1 - S_3^2 &= 5 - 9 = -4, \\
S_5^1 - S_5^2 &= 10 - 10 = 0.
\end{aligned}$$

Thus, the equations above result in $x = \max(0, 0, -4, 0) = 0$. For this transition, the new mode SI^2 starts at time 28 as shown in Figure 7.6.

The MOO protocol offers several advantages over the ST protocol. Essentially, the MOO protocol retains the iteration latency of the MADF graph in the new mode the same as the initial value, thereby avoiding the interference between the old and new modes. For instance, after *MCR1* and *MCR2* in Figure 7.6, mode SI^1 and SI^2 still have the initial iteration latency $L^1 = 10$ and $L^2 = 10$ as shown in Figure 7.3. Therefore, efficiently computing the starting time of MADF actors in the new mode becomes feasible and it plays an important role in deriving a hard-real time schedule for the MADF actors. As a result, analysis of the worst-case transition delay is much simpler (see Theorem 7.3.1) than that of the ST protocol, because the transition delay does not depend on the order of the transitions occurred previously.

Concerning the transition delay, it may be the case that the MOO protocol results in initially longer transition delay than the ST protocol does due to the offset given in Definition 7.2.15. For *MCR1* occurred at time t_{MCR1}, the transition delay of the MOO protocol is $\Delta^{2\to1} = 22 - 1 = 21$ as shown in Figure 7.6, whereas the transition delay of the ST protocol is equal to $\Delta^{2\to1} = 18 - 1 = 17$ as shown in Figure 7.4. On the other hand, let us consider the same transition request *MCR2* occurred at time $t_{\text{MCR2}} = 23$ shown in Figures 7.4 and Figure 7.6. For *MCR2*, the ST protocol results in transition delay $\Delta^{1\to2} = 42 - 23 = 19$ as shown in Figure 7.4. In contrast, the transition delay for the MOO protocol is $\Delta^{1\to2} = 38 - 23 = 16$ as shown in Figure 7.6. The MOO protocol could provide shorter transition delay than the ST protocol, thereby faster responsiveness to a mode transition.

7.3 Hard real-time Scheduling of MADF

Based on the proposed MOO protocol for mode transitions, in this section, we propose a HRT scheduling framework for MADF. We further show an analysis technique for mode transitions in MADF to reason about transition delays, such that timing constraints can be guaranteed. The HRT scheduling framework for acyclic MADF graphs is an extension of the SPS [22] framework initially developed for acyclic CSDF graphs.

As explained in Section 2.3, the key concept of the SPS framework is to derive a periodic taskset representation for a CSDF graph. Since the steady-state of a mode can be considered as a CSDF graph according to Definitions 7.2.7 and 7.2.10, it is thus straightforward to represent the steady-state of a MADF graph as a periodic taskset and schedule the resulting taskset using any well-known HRT scheduling algorithm. Using the SPS framework, we can derive the two main parameters for each MADF actor in mode SI^k, namely the period (T_i^k in Equation (2.16) on page 34) and the earliest starting time (S_i^k in Equation (2.17) on page 35). Under SPS, the iteration period in mode SI^k is obtained as $H^k = q_i^k T_i^k$, $\exists A_i^k \in \mathcal{A}$. Below, we focus on determining the earliest starting time of each actor in the new mode upon a transition. From the earliest starting time, we can reason about the transition delay to quantify the responsiveness of a transition.

Upon a MCR, a MADF graph can safely switch to the new mode if all of its actors have completed their last iteration in the old mode. In this case, the firings of MADF actors in the new mode do not overlap with the firings of actors in the old mode. This is called synchronous protocol [104] in real-time systems with mode change. One of its advantages is the simplicity, i.e., the synchronous protocol does not require any schedulability test at both compile-time and run-time. However, other protocols lead to earlier starting times than the synchronous protocol. Therefore, the synchronous protocol sets an upper bound on the earliest starting time for each MADF actor in the new mode.

Lemma 7.3.1. *For a MADF graph G under SPS and a MCR from mode SI^o to SI^l at time t_{MCR}, the earliest starting time of actor A_i^l, $\hat{\sigma}_i^{o\to l}$, is upper bounded by*

$$\hat{\sigma}_i^{o\to l} = F_{src}^o + S_{snk}^o + S_i^l, \tag{7.13}$$

where F_{src}^o indicates the time when the source actor A_{src}^o completes its last iteration It^o of the old mode SI^o and is given by

$$F_{src}^o = t_S^o + \left\lceil \frac{t_{MCR} - t_S^o}{H^o} \right\rceil H^o. \tag{7.14}$$

t_S^o is the starting time of mode SI^o and H^o is the iteration period of mode SI^o.

Proof. As explained previously for a transition from mode SI^o to SI^l, the upper bound of the earliest starting time for each actor A_i^l is computed in such a way that no firings of actors A_i^o and A_i^l occur simultaneously. This means, the start of an actor A_i^l must be later than all actors A_i^o have completed the last iteration It^o of the old

Table 7.4: Actor parameter for G_1 in Figure 7.1.

Mode	SI^1				SI^2				
Actor	A_1^1	A_2^1	A_3^1	A_5^1	A_1^2	A_2^2	A_3^2	A_4^2	A_5^2
WCET	1	4	1	1	1	8	1	3	1
period (T_i)	2	4	4	4	4	8	8	8	4
starting time (S_i)	0	2	6	14	0	4	12	8	20
utilization (u_i)	$\frac{1}{2}$	1	$\frac{1}{4}$	$\frac{1}{4}$	$\frac{1}{4}$	1	$\frac{1}{8}$	$\frac{3}{8}$	$\frac{1}{4}$

mode SI^o. Given that mode SI^o starts at time t_S^o, the completion time of all actors A_i^o in the last iteration It^o can be thus computed as

$$F_{\text{snk}}^o = t_S^o + \left\lfloor \frac{t_{MCR} - t_S^o}{H^o} \right\rfloor H^o + S_{\text{snk}}^o + H^o. \tag{7.15}$$

where F_{snk}^o is the time when the old mode SI^o completes the last iteration It^o. It is assumed that the sink actor A_{snk}^o is the last actor to complete the iteration, i.e., $\forall A_i^o \in \mathcal{A}, S_i^o \leq S_{\text{snk}}^o$. Given Equation (7.14), Equation (7.15) can be rewritten as

$$F_{\text{snk}}^o = t_S^o + \left\lceil \frac{t_{MCR} - t_S^o}{H^o} \right\rceil H^o + S_{\text{snk}}^o = F_{\text{src}}^o + S_{\text{snk}}^o.$$

Now, starting the source actor A_{src}^l at any time later than F_{snk}^o is valid without introducing simultaneous execution of actors A_i^o and A_i^l. Therefore, the earliest starting time of source actor A_{src}^l is $\hat{\sigma}_{\text{src}}^{o\to l} = F_{\text{snk}}^o$. For any actor $A_i^l \in \mathcal{A} \setminus A_{\text{src}}^l$, its earliest starting times must satisfy Equation (2.17) on page 35 imposed by the SPS framework. That is, the earliest starting starting time $\hat{\sigma}_i^{o\to l}$ of actor A_i^l can be obtained by adding S_i^l to $\hat{\sigma}_{\text{src}}^{o\to l}$. ■

Let us consider the actor parameters given in Table 7.4 for G_1 in Figure 7.1. The third row shows the WCET for each actor in modes SI^1 and SI^2. Based on WCETs, the period (fourth row in Table 7.4) and the earliest starting time (fifth row in Table 7.4) for each actor in the steady-state of both modes are obtained according to Equations 2.16 and 2.17, respectively. Given $\vec{q}^2$ in Equation (7.8), we can also compute iteration period $H^2 = q_1^2 T_1^2 = 2 \times 4 = 8$. Now consider the mode transition from mode SI^2 to SI^1 shown in Figure 7.7. Assume that the MCR occurs at time $t_{\text{MCR}} = 13$ and mode SI^2 starts at time $t_S^2 = 8$. The completion time of the last iteration It^2 is equal to the completion time of the sink actor A_5^2 computed as

$$F_{\text{snk}}^2 = t_S^2 + \left\lceil \frac{t_{\text{MCR}} - t_S^2}{H^2} \right\rceil H^2 + S_5^2 = 8 + \left\lceil \frac{13-8}{8} \right\rceil 8 + 20 = 36.$$

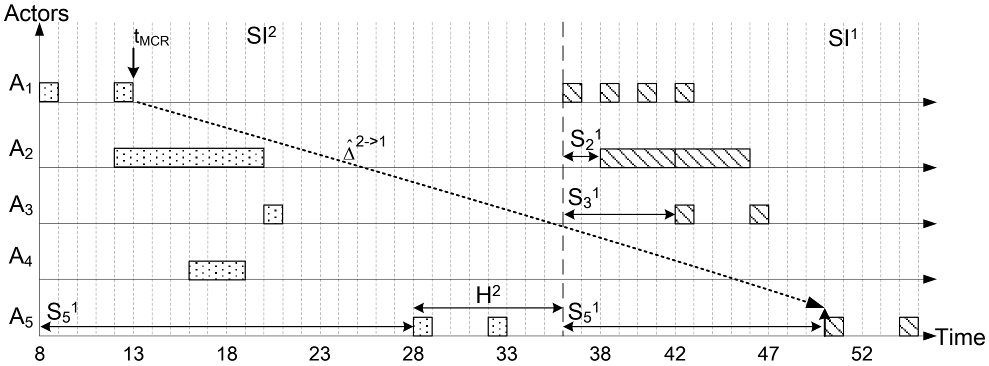


Figure 7.7: Upper bounds of earliest starting times for transition from mode SI^2 to SI^1.

In Figure 7.7, F^2_{snk} corresponds to the earliest starting time of the source actor A^1_1 (bold dashed line). Finally, we can compute the earliest starting time for each actor in the new mode SI^1 by adding S^1_i. Considering for instance the sink actor A^1_5 in the new mode with $S^1_5 = 14$, the upper bound of its earliest starting time can be obtained as

$$\hat{\sigma}^{2\to 1}_5 = F^2_{\text{src}} + S^2_5 + S^1_5 = F^2_{\text{snk}} + S^1_5 = 36 + 14 = 50.$$

We can thus compute the transition delay (*cf.* Definition 7.2.14) as

$$\hat{\Delta}^{2\to 1} = \hat{\sigma}^{2\to 1}_5 - t_{\text{MCR}} = 50 - 13 = 37.$$

Although the upper bound of the earliest starting times is easy to obtain for MADF actors in the new mode, it does not provide a responsive mode transition. Therefore, here we aim at deriving a lower bound of the earliest starting times with the proposed MOO protocol.

Lemma 7.3.2. *For a MADF graph under SPS and a MCR from mode SI^o to SI^l at time t_{MCR}, the earliest starting time of actor A^l_i using the MOO protocol is lower bounded by $\check{\sigma}^{o\to l}_i$ given as*

$$\check{\sigma}^{o\to l}_i = F^o_{src} + x + S^l_i, \tag{7.16}$$

where F^o_{src} is given in Equation (7.14) *and x is given in Equation* (7.12).

Proof. Under the MOO protocol, the start of actor A^l_i must be later than the time when A^o_i, if any, completes its last iteration in the old mode SI^o. We assume that the source actor A^l_{src} is the first actor to start in the new mode SI^l, i.e., $\forall A^l_i \in \mathcal{A}, S^l_i \geq S^l_{\text{src}}$. Thus, the starting time of the source actor A^l_{src} is at least equal to the completion

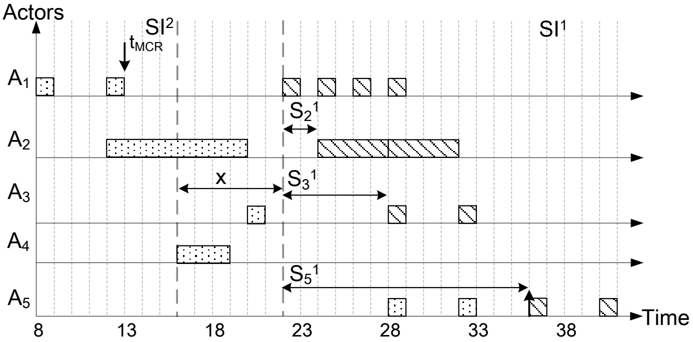


Figure 7.8: Earliest starting times for transition from mode SI^2 to SI^1 with the MOO protocol.

time of the last iteration of A^o_{src}, denoted by F^o_{src}. Given F^o_{src} in Equation (7.14), it thus holds $\check{\sigma}^{o\to l}_{\text{src}} \geq F^o_{\text{src}}$ Then, the offset x because of the MOO protocol given in Equation (7.12) must be taken into account. Consequently, the earliest starting time of A^l_{src} is lower bounded by $\check{\sigma}^{o\to l}_{\text{src}} = F^o_{\text{src}} + x$. For any actor $A^l_i \in \mathcal{A} \setminus A^l_{\text{src}}$, its earliest starting times must satisfy Equation (2.17) on page 35 imposed by the SPS framework. Hence, the earliest starting time $\check{\sigma}^{o\to l}_i$ of actor A^l_i can be obtained by adding S^l_i to $\check{\sigma}^{o\to l}_{\text{src}}$. ■

Let us consider again the transition from mode SI^2 to SI^1. With the MOO protocol, the mode transition is illustrated in Figure 7.8. Upon the MCR at time $t_{\text{MCR}} = 13$ and $t^2_S = 8$, source actor A^2_1 completes its last iteration It^2 in the old mode SI^2 at the time (*cf.* Equation (7.14)) given as

$$F^2_{\text{src}} = F^2_1 = t^2_S + \left\lceil \frac{t_{\text{MCR}} - t^2_S}{H^2} \right\rceil H^2 = 8 + \left\lceil \frac{13 - 8}{8} \right\rceil 8 = 16$$

This is the earliest possible time at which mode transition is allowed. For MOO, x can be computed according to Equation (7.12). Therefore, the following equations hold:

$$\begin{aligned} S^2_1 - S^1_1 &= 0 - 0, \\ S^2_2 - S^1_2 &= 4 - 2 = 2, \\ S^2_3 - S^1_3 &= 12 - 6 = 6, \\ S^2_5 - S^1_5 &= 20 - 14 = 6. \end{aligned}$$

It thus yields $x = \max(0, 2, 6, 6) = 6$, i.e., an offset $x = 6$ is added to F^2_{src}. It can be seen in Figure 7.8 that the source actor A^1_1 starts at time $F^2_{\mathrm{src}} + x = 16 + 6 = 22$. Finally, the earliest starting times of actors in mode SI^1 can be determined by adding S^1_i. Considering for instance A^1_5 in the new mode, the lower bound of its earliest starting time can be obtained as:

$$\check{\sigma}^{2\to 1}_5 = F^2_{\mathrm{src}} + x + S^1_5 = 16 + 6 + 14 = 36.$$

Now, the transition delay (*cf.* Definition 7.2.14) can be obtained as

$$\check{\Delta}^{2\to 1} = \check{\sigma}^{2\to 1}_5 - t_{\mathrm{MCR}} = 36 - 13 = 23.$$

Scheduling Analysis under a Fixed Allocation of Actors

During a mode transition of a MADF graph according to the MOO protocol, actors execute simultaneously in the old and new modes. The derived starting time in Lemma 7.3.2 for each actor is only the lower bound because the allocation of actors on PEs is not taken into account yet. That means, the derived starting times according to Lemma 7.3.2 can be only achieved during mode transitions when each actor is allocated to a separate PE. In a practical system where multiple actors are allocated to the same PE, the PE may be potentially overloaded during mode transitions. To avoid overloading PEs, the earliest starting times of actors may be further delayed.

Lemma 7.3.3. *For a MADF graph under SPS, a MCR from mode SI^o to SI^l, and an m-partition of all actors $\Psi = \{\Psi_1, \dots, \Psi_m\}$, where m is the number of PEs, the earliest starting time of an actor A^l_i without overloading the underlying PE is given by*

$$\sigma^{o\to l}_i = F^o_{src} + \delta^{o\to l} + S^l_i, \tag{7.17}$$

where F^o_{src} is computed by Equation (7.14) *and $\delta^{o\to l}$ is obtained as*

$$\delta^{o\to l} = \min_{t\in[x, S^o_{snk}]} \{t : U_j(k) \le UB,\ \forall k \in [t, S^o_{snk}] \wedge \forall \Psi_j \in \Psi\}. \tag{7.18}$$

UB denotes the utilization bound of the scheduling algorithm used to schedule actors on each PE. Ψ_j contains the set of actors allocated to PE$_j$. $U_j(k)$ is the total utilization of PE$_j$ at time k demanded by both mode SI^o and SI^l actors, and is given by

$$U_j(k) = \underbrace{\sum_{A^o_d \in \Psi_j} \left(u^o_d - h(k - S^o_d)\cdot u^o_d\right)}_{U^o_j(k)} + \underbrace{\sum_{A^l_d \in \Psi_j} \left(h(k - S^l_d - t)\cdot u^l_d\right)}_{U^l_j(k)}, \tag{7.19}$$

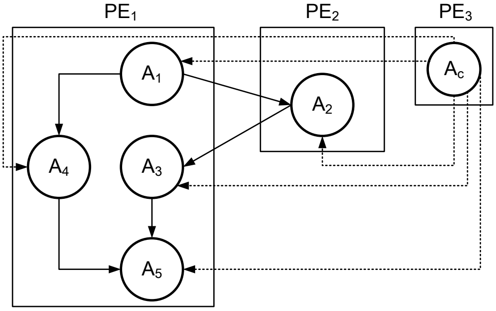


Figure 7.9: Allocation of all MADF actors in Figure 7.1 to 3 PEs.

$A_d^o \in \Psi_j$ *is an actor active in the old mode* SI^o *and allocated to* PE_j. $A_d^l \in \Psi_j$ *is an actor active in the new mode* SI^l *and allocated to* PE_j. $h(t)$ *is the Heaviside step function.*

Proof. Lemma 7.3.2 shows the lower bound of the earliest starting time for actor A_i^l in the new mode SI^l. However, starting A_i^l at time $\check{\sigma}_i^{o\to l}$ may overload PE j, i.e., the resulting total utilization of PE j, denoted by $U_j(\check{\sigma}_i^{o\to l})$, exceeds UB. Therefore, in this case, the earliest starting time $\sigma_i^{o\to l}$ must be delayed by $\delta^{o\to l}$ such that $U_j(\sigma_i^{o\to l}) \leq UB$ holds. From Equation (7.17) and Equation (7.16), we can see that $\delta^{o\to l}$ is lower bounded by x which corresponds to the MOO protocol. In addition, $\delta^{o\to l}$ is upper bounded by S_{snk}^o if we consider Equation (7.17) and Equation (7.13) on page 131.

$\delta^{o\to l}$ of interest is the minimum time t in the bounded interval $[x, S_{\mathrm{snk}}^o]$ that satisfies two conditions.

Condition 1: for each PE j, the total utilization cannot exceed UB at time t, i.e., $U_j(t) \leq UB$. The total utilization $U_j(t)$ in Equation (7.19) consists of two parts, namely $U_j^o(t)$ and $U_j^l(t)$. $U_j^o(t)$ denotes the PE capacity occupied by the actors in mode SI^o that are not completed yet. Additional PE capacity $U_j^l(t)$ is demanded by the already released actors in the new mode SI^l.

Condition 2: We need to check all time instants $k > t$ in the interval $[t, S_{\mathrm{snk}}^o]$, such that $U_j(k) \leq UB$, to guarantee that each PE_j is not overloaded during the mode transition. ■

Figure 7.9 shows all actors of G_1 in Figure 7.1 allocated to 3 PEs and let us assume that the actors allocated to each PE are scheduled using the EDF algorithm. The utilization bound of EDF is given as $UB = 1$ [80]. Given this allocation and the transition from mode SI^2 to SI^1 shown in Figure 7.8, the lower bound of the earliest

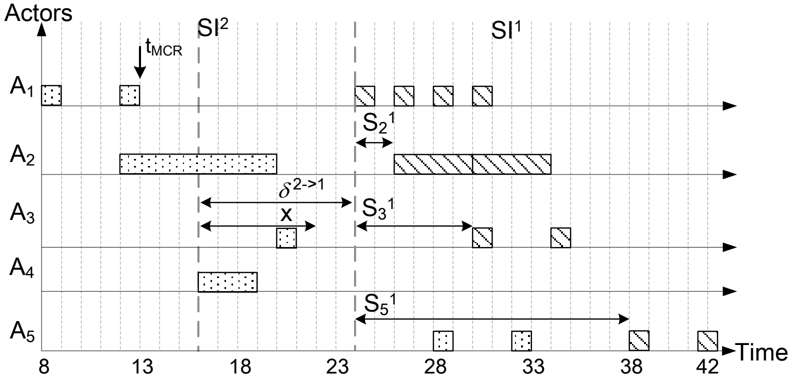


Figure 7.10: Earliest starting times for transition SI^2 to SI^1 on 2 PEs shown in Figure 7.9.

starting time $\check{\sigma}_1^{2\to1} = 22$ for actor A_1^1 cannot be achieved. At time 22, only actor A_1^2 has completed the last iteration It^2 on PE_1. Starting the new mode SI^1 at time 22 corresponds to $\delta^{2\to1} = x = 6$. The total utilization of PE_1 demanded by the actors in the old mode SI^2 at time 22, i.e., $U_1^2(6)$, can be computed as follows:

$$
\begin{aligned}
U_1^2(6) &= \sum_{A_d^2 \in \Psi_1} u_d^2 - h(6 - S_d^2) \cdot u_d^2, \; d \in \{1,3,4,5\} \\
&= u_1^2 - h(6) \cdot u_1^2 + u_3^2 - h(-6) \cdot u_3^2 + u_4^2 - h(-2) \cdot u_4^2 + u_5^2 - h(-14) \cdot u_5^2 \\
&= 0 + u_3^2 + u_4^2 + u_5^2 = \frac{1}{8} + \frac{3}{8} + \frac{1}{4} = \frac{3}{4}
\end{aligned}
$$

Enabling A_1^1 in the new mode SI^1 at time 22 would yield

$$
U_1(6) = U_1^2(6) + u_1^1 = \frac{3}{4} + \frac{1}{2} > UB = 1,
$$

thereby leading to being unschedulable on PE_1. In this case, the earliest times of all actors in mode SI^1 must be delayed by $\delta^{2\to1} = 8$ to time 24 as shown in Figure 7.10. At time 24, the total utilization demanded by mode SI^2 actors is

$$
\begin{aligned}
U_1^2(8) &= \sum_{A_d^2 \in \Psi_1} u_d^2 - h(8 - S_d^2) \cdot u_d^2, \; d \in \{1,3,4,5\} \\
&= u_1^2 - h(8) \cdot u_1^2 + u_3^2 - h(-4) \cdot u_3^2 + u_4^2 - h(0) \cdot u_4^2 + u_5^2 - h(-12) \cdot u_5^2 \\
&= 0 + u_3^2 + 0 + u_5^2 = \frac{1}{8} + \frac{1}{4} = \frac{3}{8}.
\end{aligned}
$$

Now, enabling A_1^1 in the new mode at time 24 results in the total utilization of PE_1 as

$$U_1(8) = U_1^2(8) + u_1^1 = \frac{3}{8} + \frac{1}{2} < 1.$$

Next, assuming that the new mode SI^1 starts at time 24, we need to check that the remaining actors in the new mode SI^1, namely A_3^1 and A_5^1, can start with S_3^1 and S_5^1 respectively without overloading PE_1. For instance, enabling A_3^1 at time 24 results in starting time $\sigma_3^{2\to1} = 24 + S_3^1 = 24 + 6 = 30$. At time 30, the total utilization of PE_1 can be obtained according to Equation (7.19) as follows:

$$\begin{aligned}
U_1^2(8+6) &= \sum_{A_d^2 \in \Psi_1} u_d^2 - h(14 - S_d^2)\cdot u_d^2,\ d \in \{1,3,4,5\}\\
&= u_1^2 - h(14)\cdot u_1^2 + u_3^2 - h(2)\cdot u_3^2 + u_4^2 - h(6)\cdot u_4^2 + u_5^2 - h(-6)\cdot u_5^2\\
&= 0 + 0 + 0 + u_5^2 = \frac{1}{4},\\
U_1^1(8+6) &= \sum_{A_d^1 \in \Psi_1} \Big(h(14 - S_d^1 - 8)\cdot u_d^1\Big),\ d \in \{1,3,5\}\\
&= h(6)u_1^1 + h(0)u_3^1 + h(-8)u_5^1\\
&= \frac{1}{2} + \frac{1}{4} = \frac{3}{4},\\
U_1(8+6) &= U_1^2(8+6) + U_1^1(8+6) = 1 = UB.
\end{aligned}$$

Hence, actors A_5^2, A_1^1, and A_3^1 are schedulable on PE_1 using EDF. Similarly, starting A_5^1 at time $\sigma_5^{2\to1} = 24 + S_5^1 = 38$ still keeps the resulting set of actors schedulable on PE_1.

Using Lemma 7.3.3, we can quantify the maximum and minimum transition delays for any transition from mode SI^o to SI^l.

Theorem 7.3.1. *For a MADF graph under SPS, a fixed allocation of all MADF actors $\Psi = \{\Psi_1, \ldots, \Psi_m\}$ to m PEs, and a MCR from mode SI^o to SI^l, the minimum transition delay is given by*

$$\Delta_{min}^{o\to l} = \delta^{o\to l} + S_{snk}^l \tag{7.20}$$

and the maximum transition delay is given by

$$\Delta_{max}^{o\to l} = \delta^{o\to l} + S_{snk}^l + H^o, \tag{7.21}$$

where $\delta^{o\to l}$ is computed by Lemma 7.3.3, S_{snk}^l is the starting time of the sink actor in the new mode SI^l, and H^o is the iteration period of the old mode SI^o.

Proof. For a MCR from mode SI^o to SI^l, the transition delay $\Delta^{o \to l}$ of a MADF graph is given in Definition 7.2.14 as $\Delta^{o \to l} = \sigma_{\text{snk}}^{o \to l} - t_{\text{MCR}}$, where the earliest starting time of the sink actor is calculated as $\sigma_{\text{snk}}^{o \to l} = F_{\text{src}}^{o} + \delta^{o \to l} + S_{\text{snk}}^{l}$ according to Lemma 7.3.3. Therefore, $\Delta^{o \to l}$ can be rewritten as $\Delta^{o \to l} = F_{\text{src}}^{o} + \delta^{o \to l} + S_{\text{snk}}^{l} - t_{\text{MCR}}$. Essentially, $\Delta^{o \to l}$ is composed of three parts. In the first part, the MOO transition protocol together with a fixed allocation of the MADF actors determine $\delta^{o \to l}$. The second part S_{snk}^{l} results from the SPS framework. These two parts thus can be determined at compile-time. The third part $F_{\text{src}}^{o} - t_{\text{MCR}}$ depends on when the MCR occurs, namely at t_{MCR}, which can only be determined at run-time. In the following, we distinguish two cases for t_{MCR}:

Case 1: Assume that the MCR occurs at the end of an iteration of the source actor in the old mode SI^o, i.e., $t_{\text{MCR}} = F_{\text{src}}^{o}$. Then, the source actor shall be only delayed by $\delta^{o \to l}$ to start in the new mode SI^l according to Lemma 7.3.3, thereby guaranteeing the fastest possible start of the new mode SI^l. As a consequence, it results in the minimum possible transition delay. Therefore, substituting $t_{\text{MCR}} = F_{\text{src}}^{o}$, we obtain

$$\Delta_{\min}^{o \to l} = F_{\text{src}}^{o} + \delta^{o \to l} + S_{\text{snk}}^{l} - F_{\text{src}}^{o} = \delta^{o \to l} + S_{\text{snk}}^{l}.$$

Case 2: Assume that the MCR occurs at the beginning of an iteration of the source actor in the old mode SI^o, i.e., $t_{\text{MCR}} = F_{\text{src}}^{o} - H^o$. Then, the source actor cannot start in the new mode before it completes the whole iteration in the old mode SI^o followed by the delay $\delta^{o \to l}$ according to Lemma 7.3.3. Therefore, the maximum transition delay is computed as follows:

$$\Delta_{\max}^{o \to l} = F_{\text{src}}^{o} + \delta^{o \to l} + S_{\text{snk}}^{l} - (F_{\text{src}}^{o} - H^o) = \delta^{o \to l} + S_{\text{snk}}^{l} + H^o.$$

■

It can be seen from Theorem 7.3.1 that the maximum and minimum transition delays solely depend on the allocation of MADF actors and the old and new modes in question, irrespective of the previously occurred transitions. The old and new modes determine H^o and S_{snk}^{l}, respectively, while the allocation of MADF actors determines the value of $\delta^{o \to l}$. Here, the offset x due to our MOO protocol is captured in $\delta^{o \to l}$ and can be considered as performance overhead if $x \neq 0$. The other parts, namely H^o and S_{snk}^{l}, in the maximum and minimum transition delays cannot be avoided as they will be present in any transition protocol.

7.4 Case Study

In this section, we present a case study of using the proposed MADF MoC and the developed HRT scheduling explained in Section 7.3. With the case study, we show that the MADF MoC is able to capture different application modes and the transitions between them. Then, the main focus of the case study is to analyze the transition delays and to demonstrate the effectiveness of the proposed MOO transition protocol.

We consider a real-life adaptive application from the StreamIT benchmark suit [54], called Vocoder, which implements a phase voice encoder and performs pitch transposition of recorded sounds from male to female. We modeled Vocoder with a MADF graph with 4 modes, which capture different workloads. The MADF graph of Vocoder is shown in Figure 7.11. Depending on the desired quality of audio encoding and various performance requirements, users may switch between four different modes of Vocoder at run-time. The four modes $\mathcal{S} = \{SI^8, SI^{16}, SI^{32}, SI^{64}\}$ specify different lengths of the Discrete Fourier Transform (DFT), denoted by $\mathsf{dl} \in \{8, 16, 32, 64\}$. Mode SI^8 ($\mathsf{dl} = 8$) requires the least amount of computation at the cost of the worst voice encoding quality among all DFT lengths. Mode SI^{64} ($\mathsf{dl} = 64$) produces the best quality of voice encoding among all modes, but is computationally intensive. The other two modes SI^{16} and SI^{32} explore the trade-off between the quality of the encoding and computational workload. A transition from one mode to any other one is possible, thereby resulting in totally 12 possible transitions. At run-time, reconfiguration of the parameter dl is triggered by the environment, e.g., the user in this case. Subsequently, control actor A_c propagates dl to the dataflow actors shown in Figure 7.11 through the dashed-lined edges.

We measured the WCETs of all dataflow actors in Figure 7.11 in the four modes on an ARM Cortex-A9 [1] PE. All dataflow actors were compiled using the compiler `arm-xilinx-eabi-gcc 4.7.2` with the vectorization option. The WCETs of all actors in all four modes are given in Table 7.5. It is worth to note that in mode SI^8, actors Spec2Env and male2female exhibit exceptionally high WCETs. It is because parameter dl represents the size of the inner-most loop in the computation of actors Spec2Env and male2female. Small dl (in this case $\mathsf{dl} = 8$) leads to the fact that the inner-most loop cannot be vectorized by the compiler. In the other modes from SI^{16} to SI^{64}, larger sizes of the inner-most loop (dl equal to 16, 32, and 64, respectively) lead to full vectorization of the computation of actors Spec2Env and male2female. Therefore, in these three modes, the WCETs of actors Spec2Env and male2female are even smaller than the ones in mode SI^8. The dataflow actors of Vocoder are allocated to 4 PEs as shown in Figure 7.12. This allocation guarantees that the shortest periods (maximum throughput) in the steady-states of all modes can be achieved.

Table 7.6 shows the performance results for the four modes in their steady-state under SPS. For instance, the second column at the first row in Table 7.6 indicates that it is guaranteed for sink actor WriteWave to produce 256 samples per 917,451

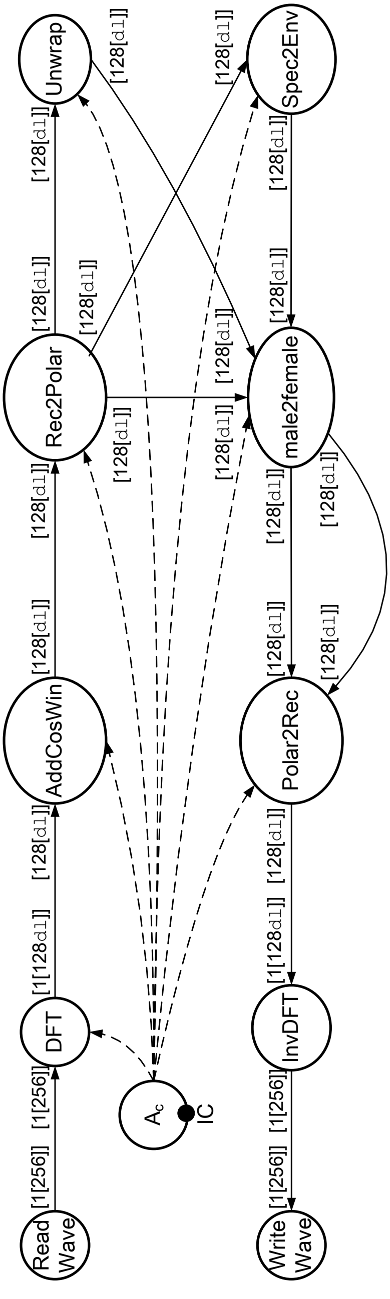


Figure 7.11: MADF graph of Vocoder.

Table 7.5: WCETs of all actors in Vocoder (in clk.).

Mode	ReadWave	DFT	AddCosWin	Rec2Polar	Unwrap	Spec2Env	male2female	Polar2Rec	InvDFT	WriteWave
SI^8	3704	16,775	16	90	359	7,168	1,093	3	236	3660
SI^{16}	3704	35,121	35	183	691	1,163	138	260	644	3660
SI^{32}	3704	71,337	75	366	1,393	1,392	210	507	988	3660
SI^{64}	3704	144,531	150	1,156	2,346	1,696	426	1,056	3,630	3660

clock cycles in mode SI^8. This is the “worst-case” performance among all four modes because the Spec2Env actor exhibits exceptionally high workload (*cf.* WCETs in Table 7.5 and Definition 2.3.2 on page 34) in mode SI^8. Consequently, actor Spec2Env becomes the “bottleneck” actor, so that mode SI^8 cannot be scheduled with

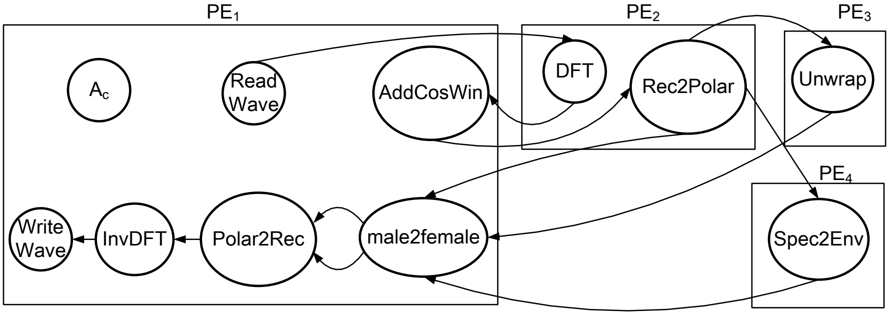


Figure 7.12: Allocation of dataflow actors of Vocoder to 4 PEs. The control edges are omitted to avoid cluttering.

Table 7.6: Performance results of four modes of Vocoder in the steady-state.

Mode	Period (T in clk.)	Total utilization (U)	Iteration latency (L)
SI^8	917,504	1.24	7,339,608
SI^{16}	148,864	2.36	1,191,436
SI^{32}	178,176	3.19	1,425,448
SI^{64}	300,288	3.4	2,402,550

higher throughput (shorter period). Nevertheless, all mode SI^8 actors as a whole require a total utilization of only 1.24 (see the third column in Table 7.6) which is the least among all modes. From Table 7.6, we can see that MADF together with the SPS framework brings another advantage of efficiently utilizing PE resources. For example, in case that Vocoder is switched to a mode with lower utilization, idle capacity of PEs can be efficiently utilized by admitting other applications at run-time without introducing interference to the currently running Vocoder.

Now, we focus on the performance results of the MOO protocol, namely transition delays, for all possible transitions between the four modes of Vocoder. Table 7.7 shows both the minimum and maximum transition delays in accordance with Theorem 7.3.1 for all transitions. We can see in the second column of Table 7.7 that, in the best case, the transition delays for 6 out of 12 transitions remain the same as the iteration latencies of the new modes. This can be seen as $x = 0$ shown in the fourth column. In these 6 transitions, the proposed MOO protocol does not introduce any extra delay. In the 6 remaining transitions, as expected, the MOO protocol introduces offset $x > 0$ to the transitions from an old mode with a longer iteration latency to a new mode with a shorter iteration latency. For instance, the largest x (in bold shown in Table 7.7) happens in case of a transition from mode SI^8 with the longest iteration latency (see the fourth column in Table 7.6) to mode SI^{16}

Table 7.7: Performance results for all mode transitions of Vocoder.

Transition (SI^o to SI^l)	$\Delta_{\min}^{o\to l}$ (in clk.)	$\Delta_{\max}^{o\to l}$ (in clk.)	x (in clk.)	$\delta^{o\to l}$ (in clk.)
$SI^8 \to SI^{64}$	3,636,815	4,554,266	1,234,264	1,234,264
$SI^8 \to SI^{32}$	2,903,988	3,821,439	1,478,540	1,478,540
$SI^8 \to SI^{16}$	2,728,479	3,645,930	**1,537,043**	1,537,043
$SI^{16} \to SI^{64}$	2,402,550	2,551,480	0	0
$SI^{16} \to SI^{32}$	1,425,448	1,574,378	0	0
$SI^{16} \to SI^{8}$	7,339,608	7,488,538	0	0
$SI^{32} \to SI^{64}$	2,402,550	2,580,731	0	0
$SI^{32} \to SI^{16}$	1,425,448	1,603,629	234,012	234,012
$SI^{32} \to SI^{8}$	7,339,608	7,517,789	0	0
$SI^{64} \to SI^{32}$	2,402,550	2,702,869	977,102	977,102
$SI^{64} \to SI^{16}$	2,402,550	2,702,869	1,211,114	1,211,114
$SI^{64} \to SI^{8}$	7,339,608	7,639,927	0	0

with the shortest iteration latency. To quantify x, we compute the percentage of x compared to both minimum and maximum transition delays as

$$\Omega_{\min} = \frac{x}{\Delta_{\min}^{o\to l}} \times 100\%, \quad \Omega_{\max} = \frac{x}{\Delta_{\max}^{o\to l}} \times 100\%.$$

$\Omega_{\min}$ varies from the worst-case 56% to the best case 16% with an average of 41%, whereas $\Omega_{\max}$ varies from the worst-case 44% to the best case 14% with an average of 33%. Therefore, the increase of the transition delays due to the MOO protocol is reasonable for this real-life application.

Next, we consider the effect of the actor allocation shown in Figure 7.12 on the earliest starting times of actors in the new mode upon a transition (*cf.* Lemma 7.3.3). In this particular example, we find out that no extra delay is incurred to any actor in all transitions due to the fixed actor allocation. This can be seen from the fourth and fifth columns in Table 7.7, where $\delta^{o\to l} = x$.

Chapter 8

Summary and Outlook

MODERN streaming applications exhibit increasingly adaptive behavior and required more complex computation. At the same time, high performance requirements and tight hard real-time guarantees must be satisfied. Driven by the advance of the semiconductor technology, MPSoC platforms will continue to have more computational capabilities to meet these demands. This situation asks for a correspondingly advanced design methodology to cope with the design complexity. Adoption of a model-based ESL design methodology seems to be an inevitable solution to increase the design productivity and handle the complexity. In this thesis, we have proposed several novel techniques to enhance a model-based ESL design methodology. In particular, we have adopted a specific instance of such design methodology, namely the Daedalus^{RT} design flow, for designing high performance, hard real-time, and adaptive streaming systems. Below, we provide a summary of this thesis.

In Chapter 3, we have presented an algorithm to derive a CSDF graph as the analysis model from an input-output equivalent PPN. Automated derivation of the CSDF MoC allows for the subsequent step of HRT analysis in Daedalus^{RT}. Therefore, it is a key enabler of the fully automated Daedalus^{RT} design flow. In addition, our approach can be considered as an enhancement to the PNgen [125] compiler, which derives an equivalent PPN from a SANLP. Now, all design flows that accept the CSDF MoC as application specification will benefit from our approach because it relieves designers from manual specification of the CSDF MoC, which in some case may not be trivial to do it manually.

In Chapter 4, we have shown that the mapping of streaming applications considering a single initial application specification cannot fully utilize the processing power of MPSoC platforms. Using the PPN MoC as the application specification, we have presented an analytical framework to determine the maximum DLP, i.e., the maximum number of communication-free partitions. Subsequently, we have proposed an approach to transform an initial PPN to a set of communication-free partitions, if it exists. The experimental results on FPGA-based MPSoCs and desktop multi-core platforms showed that our approach leads to significantly better performance than the approaches, in which alternative application specifications are not taken into account.

In Chapter 5, we have addressed the problem of exploiting just-enough parallelism when mapping a streaming application modeled using the SDF MoC in hard real-time systems. Exploiting just-enough parallelism is achieved by simultaneously unfolding and allocating the SDF actors onto an MPSoC platform, while considering the number of available PEs and hard real-time scheduling of actors on the PEs. We showed that the solution space to our problem is bounded and subsequently derived its upper bound. We devised an efficient algorithm to solve the problem and evaluated the algorithm on a set of real-life applications. The experiments showed that our algorithm results in a system specification with large performance gain. We also compared our algorithm with one of the state-of-the-art meta-heuristics, i.e., NSGA-II genetic algorithm, and showed that our algorithm is on average 100 times faster than the GA, while achieving the same quality of the solution.

In Chapter 6, we have introduced the Parameterized Polyhedral Process Network (P^3N) MoC that is able to capture adaptive/dynamic application behavior. Such behavior is usually expressed by parameters which values are updated at run-time. We have proposed a design-time approach to enable consistent execution of the P^3N MoC at run-time. The P^3N MoC is used as the implementation model for adaptive streaming applications in the Daedalus^{RT} design flow. Therefore, we have evaluated the possible run-time overhead caused by the parameterization of the P^3N model by designing and executing MPSoCs on an FPGA-based platform. The obtained results have shown that the parameterization we proposed is efficient in terms of the execution overhead introduced by the implementation of the process networks.

In Chapter 7, we have introduced the Mode-Aware Data Flow (MADF) MoC for adaptive streaming applications and its operational semantics. An adaptive streaming application is characterized by individual scenarios and scenario transitions. As an important part of the operational semantics, we proposed a novel protocol for scenario transitions. The main advantage of this transition protocol is that it does not introduce any timing interference between scenarios upon transitions. Based on the transition protocol, we have extended the initial HRT scheduling in Daedalus^{RT} for the MADF MoC. Furthermore, we have presented a HRT analysis on best-case and worst-case transition delays. Finally, we have conducted a case study using a real-life adaptive streaming application. The results have shown reasonable increase of transition delay due to our proposed transition protocol.

Although the techniques proposed in this thesis have significantly enhanced our model-based ESL design methodology, some interesting open research problems still exist that are highly related to the work presented in this thesis. Below, we outline some important ones.

Trade-off Exploration between Communication and Load-balancing

The communication-free partitioning developed in Chapter 4 only considers one special alternative application specification, i.e., no communication between PEs when all communication-free partitions are mapped on them. However, this comes at a cost: the workload on PEs may not be perfectly balanced any more. It is thus worthwhile considering other alternative application specifications as well, in which certain degree of communication between partitions is allowed while workload can be better distributed among partitions. The degree of communication between partitions depends on the computation and communication capabilities of target architectures. We can only have an optimum mapping when all these factors are taken into account in a design space. Clearly, even a larger design space will ask for more efficient DSE algorithms.

Equivalence between the Analysis MoC and the Implementation MoC for Adaptive Applications

It should not be too difficult to derive the P^3N MoC developed in Chapter 6 from a sequential specification based on the initial work in [92]. It is, however, important to show that, for any P^3N, we can find an input-output equivalent MADF graph. Only in this case, we can achieve a complete design flow for adaptive streaming applications as the Daedalus^{RT} design flow for the static streaming applications.

Optimization of Mode Transitions Considering Mapping

The HRT scheduling framework developed in Chapter 7 for the MADF MoC does not yet consider the effect of resource allocation and actor mapping. This immediately brings up an important problem: given a MADF graph and a fixed platform, e.g., m PEs, find a mapping of actors onto m PEs, such that transition delays are minimized while HRT constraints are met.

Bibliography

[1] ARM Cortex-A9 Processor. `http://www.arm.com/products/processors/cortex-a/cortex-a9.php` (Last access: December 2013).

[2] ARM Crossbar Switch. http://www.arm.com/products/system-ip/amba/amba-open-specifications.php.

[3] Arteris. `http://www.arteris.com/` (Last access: January 2014).

[4] Daedalus: System-level design for multi-processor system-on-chip. `http://daedalus.liacs.nl` (Last access: December 2013).

[5] Embedded system market - global industry analysis, size, share, growth, trends and forecast 2012 - 2018. `http://www.transparencymarketresearch.com/embedded-system.html` (Last access: December 2013).

[6] EU FP-7 parMERASA project. http://www.parmerasa.eu/.

[7] FreeRTOS. `http://www.freertos.org/` (Last access: November 2013).

[8] Mathworks Simulink. `http://www.mathworks.com/products/simulink/` (Last access: January 2014).

[9] NASA Mars Exploration Rovers. `http://marsrovers.jpl.nasa.gov/` (Last access: December 2013).

[10] NVIDIA Tegra 4. `http://www.nvidia.com/object/tegra-4-processor.html` (Last access: December 2013).

[11] NVIDIA: White paper - Tegra multi-processor architecture.

[12] Samsung Galaxy S4. `http://www.samsung.com/global/microsite/galaxys4/` (Last access: December 2013).

[13] Xilinx MicroBlaze soft processor core. `http://www.xilinx.com/tools/microblaze.htm` (Last access: December 2013).

[14] Xilinx OS and libraries document collection. `http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_2/oslib_rm.pdf` (Last access: December 2013).

[15] Xilinx Virtex-6 FPGA ML605 evaluation kit. `http://www.xilinx.com/products/boards-and-kits/EK-V6-ML605-G.htm` (Last access: December 2013).

[16] Xilinx Zynq-7000 SoC. `http://www.xilinx.com/products/silicon-devices/soc/zynq-7000` (Last access: November 2013).

[17] Embedded systems: Technologies and markets. `http://www.bccresearch.com`, January 2012.

[18] H.264 : Advanced video coding for generic audiovisual services. `http://www.itu.int/rec/T-REC-H.264-201304-I/en`, April 2013.

[19] M. Alvarez, E. Salamí, A. Ramirez, and M. Valero. A performance characterization of high definition digital video decoding using h.264/avc. In *Proceedings of the IEEE International Workload Characterization Symposium*, pages 24–33, 2005.

[20] H. A. Andrade and S. Kovner. Software synthesis from dataflow models for G and LabVIEW. In *Proc. Asilomar Conf. on Signals, Systems & Computers*, 1998.

[21] P. Aubry, P.-E. Beaucamps, F. Blanc, B. Bodin, S. Carpov, L. Cudennec, V. David, P. Dore, P. Dubrulle, B. D. de Dinechin, F. Galea, T. Goubier, M. Harrand, S. Jones, J.-D. Lesage, S. Louise, N. M. Chaisemartin, T. H. Nguyen, X. Raynaud, and R. Sirdey. Extended cyclostatic dataflow program compilation and execution for an integrated manycore processor. In *Proceedings of the International Conference on Computational Science (ICCS)*, pages 1624–1633, 2013.

[22] M. Bamakhrama and T. Stefanov. Hard-real-time scheduling of data-dependent tasks in embedded streaming applications. In *Proceedings of the ninth ACM international conference on Embedded software*, EMSOFT '11, pages 195–204, New York, NY, USA, 2011. ACM.

[23] M. Bamakhrama, J. T. Zhai, H. Nikolov, and T. Stefanov. A methodology for automated design of hard-real-time embedded streaming systems. In *Design, Automation & Test in Europe Conference & Exhibition*, DATE'12, pages 941–946, 2012.

[24] M. A. Bamakhrama and J. T. Zhai. Daedalus/Daedalus^{RT} user manual. `http://daedalus.liacs.nl/manual.pdf` (Last access: December 2013).

[25] C. Bastoul. Code generation in the polyhedral model is easier than you think. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, PACT '04, pages 7–16, Washington, DC, USA, 2004. IEEE Computer Society.

[26] C. Bastoul. *Improving Data Locality in Static Control Programs*. PhD thesis, University Paris 6, Pierre et Marie Curie, France, 2004.

[27] L. Benini and G. De Micheli. Networks on chips: A new SoC paradigm. *Computer*, 35(1):70–78, Jan. 2002.

[28] D. Bertozzi and L. Benini. Xpipes: a network-on-chip architecture for gigascale systems-on-chip. *Circuits and Systems Magazine, IEEE*, 4(2):18–31, Sept. 2004.

[29] B. Bhattacharya and S. Bhattacharyya. Parameterized dataflow modeling for DSP systems. *IEEE Trans. Signal Process.*, 2001.

[30] G. Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete. Cyclo-static data flow. *IEEE Trans. Signal Process.*, 44:397–408, 1996.

[31] B. Bodin, A. M. Kordon, and B. D. de Dinechin. Periodic schedules for cyclo-static dataflow. In *Proc. of 11th Int. IEEE Symposium on Embedded Systems for Real-Time Multimedia (ESTIMedia'13)*, pages 105–114, 2013.

[32] J. T. Buck. *Scheduling Dynamic Dataflow Graphs with Bounded Memory Using the Token Flow Model*. PhD thesis, EECS Department, University of California, Berkeley, 1993.

[33] G. C. Buttazzo. *Hard Real Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Springer, 3 edition, 2011.

[34] J. Cong, K. Gururaj, G. Han, and W. Jiang. Synthesis algorithm for application-specific homogeneous processor networks. *IEEE Trans. Very Large Scale Integr. Syst.*, 17(9):1318–1329, 2009.

[35] R. I. Davis and A. Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM Comput. Surv.*, 43(4):35:1–35:44, October 2011.

[36] K. Deb et al. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Trans. Evol. Comput.*, 6(2):182 –197, 2002.

[37] E. F. Deprettere, T. Stefanov, S. S. Bhattacharyya, and M. Sen. Affine nested loop programs and their binary parameterized dataflow graph counterparts. *2013 IEEE 24th International Conference on Application-Specific Systems, Architectures and Processors*, 0:186–190, 2006.

[38] J. Eker and J. Janneck. CAL language report. Technical report, University of California at Berkeley, 2003.

[39] T. Faragó. A framework for heterogeneous desktop parallel computing. Technical report, LIACS - Leiden University, 2008.

[40] S. M. Farhad, Y. Ko, B. Burgstaller, and B. Scholz. Orchestration by approximation: mapping stream programs onto multicore architectures. In *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, ASPLOS '11, pages 357–368, New York, NY, USA, 2011. ACM.

[41] P. Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20(1):23–53, 1991.

[42] P. Feautrier. Automatic parallelization in the polytope model. In *The Data Parallel Programming Model*, pages 79–103. Springer-Verlag, 1996.

[43] S. Foessel. Motion JPEG2000 and digital cinema. `http://www.jpeg.org/public/DCINEMA-JPEG2000.ppt` (Last access: December 2013).

[44] F.-A. Fortin, F.-M. D. Rainville, M.-A. Gardner, M. Parizeau, and C. Gagné. Deap: Evolutionary algorithms made easy. *Journal of Machine Learning Research*, 2171–2175(13), jul 2012.

[45] M. Geilen. Synchronous dataflow scenarios. *ACM Trans. Embed. Comput. Syst.*, 10(2):16:1–16:31, Jan. 2011.

[46] M. Geilen and T. Basten. Reactive process networks. In *Proceedings of the 4th ACM International Conference on Embedded Software*, EMSOFT '04, pages 137–146, New York, NY, USA, 2004. ACM.

[47] M. Geilen and S. Stuijk. Worst-case performance analysis of synchronous dataflow scenarios. In *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, CODES/ISSS '10, pages 125–134, New York, NY, USA, 2010. ACM.

[48] M. Geilen, S. Stuijk, and T. Basten. Predictable dynamic embedded data processing. In *International Conference on Embedded Computer Systems (SAMOS)*, pages 320–327, 2012.

[49] M. C. W. Geilen, F. J., C. Haubelt, T. Basten, B. Theelen, and S. Stuijk. Performance analysis of weakly-consistent scenario-aware dataflow graphs. In *In Proceedings of 48th Asilomar Conference on Signals, Systems & Computers*, 2014.

[50] A. Gerstlauer, C. Haubelt, A. Pimentel, T. Stefanov, D. Gajski, and J. Teich. Electronic system-level synthesis methodologies. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 28(10):1517–1530, Oct. 2009.

[51] A. Gerstlauer, J. Peng, D. Shin, D. Gajski, A. Nakamura, D. Araki, and Y. Nishihara. Specify-explore-refine (ser): from specification to implementation. In *Proceedings of the 45th annual Design Automation Conference*, DAC '08, pages 586–591, New York, NY, USA, 2008. ACM.

[52] A. H. Ghamarian, M. C. W. Geilen, S. Stuijk, T. Basten, B. D. Theelen, M. R. Mousavi, A. J. M. Moonen, and M. J. G. Bekooij. Throughput analysis of synchronous data flow graphs. In *ACSD '06: Proceedings of the Sixth International Conference on Application of Concurrency to System Design*, pages 25–36, Washington, DC, USA, 2006. IEEE Computer Society.

[53] K. Goossens, J. Dielissen, and A. Radulescu. Æthereal network on chip: Concepts, architectures, and implementations. *IEEE Des. Test*, 22(5):414–421, Sept. 2005.

[54] M. I. Gordon, W. Thies, and S. Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, ASPLOS-XII, pages 151–162, New York, NY, USA, 2006. ACM.

[55] S. Ha and H. Oh. Decidable signal processing dataflow graphs: Synchronous and cyclo-static dataflow graphs. In S. S. Bhattacharyya, E. F. Deprettere,

R. Leupers, and J. Takala, editors, *Handbook of Signal Processing Systems*, pages 851–874. Springer US, 2010.

[56] A. Hagiescu, W.-F. Wong, D. F. Bacon, and R. Rabbah. A computing origami: folding streams in FPGAs. In *Proceedings of the 46th Annual Design Automation Conference*, DAC '09, pages 282–287, New York, NY, USA, 2009. ACM.

[57] A. Hansson, K. Goossens, M. Bekooij, and J. Huisken. Compsoc: A template for composable and predictable multi-processor system on chips. *ACM Trans. Des. Autom. Electron. Syst.*, 14(1):2:1–2:24, Jan. 2009.

[58] J. L. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 5 edition, 2011.

[59] M. Hirzel, R. Soulé, S. Schneider, B. Gedik, and R. Grimm. A catalog of stream processing optimizations. Research Report RC25215, IBM, 2011.

[60] W. Huang, S. Ghosh, S. Velusamy, K. Sankaranarayanan, K. Skadron, and M. R. Stan. Hotspot: acompact thermal modeling methodology for early-stage vlsi design. *IEEE Trans. Very Large Scale Integr. Syst.*, 14(5):501–513, May 2006.

[61] D. S. Johnson. Approximation algorithms for combinatorial problems. In *Proceedings of the fifth annual ACM symposium on Theory of computing*, pages 38–49, 1973.

[62] R. Jordans, F. Siyoum, S. Stuijk, A. Kumar, and H. Corporaal. An Automated Flow to Map Throughput Constrained Applications to a MPSoC. In *Bringing Theory to Practice: Predictability and Performance in Embedded Systems*, volume 18 of *PPES*, pages 47–58, Dagstuhl, Germany, 2011.

[63] H. Jung and S. Ha. Hardware synthesis from coarse-grained dataflow specification for fast HW/SW cosynthesis. In *Proceedings of the 2Nd IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS '04, pages 24–29, New York, NY, USA, 2004. ACM.

[64] G. Kahn. The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, *Information Processing '74: Proceedings of the IFIP Congress*, pages 471–475. North-Holland, New York, NY, 1974.

[65] W. Kelly, W. Pugh, E. Rosser, and T. Shpeisman. Transitive closure of infinite graphs and its applications. *Int. J. Parallel Program.*, 24:579–598, December 1996.

[66] B. Kienhuis. MatParser: An array dataflow analysis compiler. Technical report, University of Canifornia, Berkeley, 2000.

[67] B. Kienhuis, E. Deprettere, K. Vissers, and P. van der Wolf. An approach for quantitative analysis of application-specific dataflow architectures. *2013 IEEE 24th International Conference on Application-Specific Systems, Architectures and Processors*, 0:338, 1997.

[68] B. Kienhuis and E. F. Deprettere. Modeling stream-based applications using the sbf model of computation. *J. VLSI Signal Process. Syst.*, 34(3):291–300, July 2003.

[69] D. Kleidermacher and M. Kleidermacher. *Embedded Systems Security: Practical Methods for Safe and Secure Software and Systems Development*. Newnes, 1 edition, 2012.

[70] P. Kourzanov, O. Moreira, and H. J. Sips. Disciplined multi-core programming in C. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, pages 346–354, 2010.

[71] M. Kudlur and S. Mahlke. Orchestrating the execution of stream programs on multicore platforms. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '08, pages 114–124, New York, NY, USA, 2008. ACM.

[72] K. Lakshmanan, S. Kato, and R. Rajkumar. Scheduling parallel real-time tasks on multi-core processors. In *Proceedings of the 31st IEEE Real-Time Systems Symposium*, RTSS'10, pages 259–268, 2010.

[73] D. Lammers. Intel cancels Tejas, moves to dual-core designs, May 2004.

[74] E. A. Lee. Embedded software — an agenda for research. Technical report, University of California at Berkeley, 1999.

[75] E. A. Lee and D. G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Trans. Comput.*, 36(1):24–35, 1987.

[76] E. A. Lee and D. G. Messerschmitt. Synchronous data flow. *Proc. IEEE*, 75(9):1235–1245, September 1987.

[77] H. Lee, Y. Lin, Y. Harel, M. Woh, S. Mahlke, T. Mudge, and K. Flautner. Software defined radio - a high performance embedded challenge. In *Proceedings of the First international conference on High Performance Embedded Architectures and Compilers*, HiPEAC'05, pages 6–26, Berlin, Heidelberg, 2005. Springer-Verlag.

[78] S.-w. Liao, Z. Du, G. Wu, and G.-Y. Lueh. Data and computation transformations for brook streaming applications on multiprocessors. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '06, pages 196–207, Washington, DC, USA, 2006. IEEE Computer Society.

[79] B. Lickly, I. Liu, S. Kim, H. D. Patel, S. A. Edwards, and E. A. Lee. Predictable programming on a precision timed architecture. In *Proceedings of the 2008 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, CASES '08, pages 137–146, New York, NY, USA, 2008. ACM.

[80] C. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, 1973.

[81] B. D. Lucas and T. Kanade. An iterative image registration technique with an application to stereo vision. In *Proceedings of the 7th international joint conference on Artificial intelligence - Volume 2*, IJCAI'81, pages 674–679, San Francisco, CA, USA, 1981. Morgan Kaufmann Publishers Inc.

[82] D. Marr and T. Poggio. Neurocomputing: foundations of research. chapter Cooperative computation of stereo disparity, pages 259–267. MIT Press, Cambridge, MA, USA, 1988.

[83] S. Martello and P. Toth. *Knapsack Problems: Algorithms and Computer Implementations*. John Wiley & Sons, 1 edition, 1990.

[84] P. Marwedel. *Embedded System Design*. Springer, 2 edition, 2006.

[85] C. Meenderinck, A. Azevedo, B. Juurlink, M. Alvarez Mesa, and A. Ramirez. Parallel scalability of video decoders. *J. Signal Process. Syst.*, 57(2):173–194, Nov. 2009.

[86] S. Meijer, H. Nikolov, and T. Stefanov. Combining process splitting and merging transformations for polyhedral process networks. In *Proc. of 8th Int. IEEE Workshop on Embedded Systems for Real-Time Multimedia (ESTIMedia'10)*, pages 97–106, 2010.

[87] S. Meijer, H. Nikolov, and T. Stefanov. Throughput modeling to evaluate process merging transformations in polyhedral process networks. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, pages 747–752, 2010.

[88] J. Mitola. *Software Radio*. John Wiley & Sons, Inc., 2003.

[89] O. Moreira. *Temporal Analysis and Scheduling of Hard Real-Time Radios on a Multi-Processor*. PhD thesis, Technische Universiteit Eindhoven, 2012.

[90] O. Moreira. Data flow modeling of radio applications. `http://www.es.ele.tue.nl/~sander/tutorials/hipeac-2013/files/moreira.pdf` (Last access: December 2013), 2013.

[91] P. K. Murthy and S. S. Bhattacharyya. *Memory Management for Synthesis of DSP Software*. CRC Press, 2006.

[92] D. Nadezhkin, H. Nikolov, and T. Stefanov. Automated generation of polyhedral process networks from affine nested-loop programs with dynamic loop bounds. *ACM Trans. Embed. Comput. Syst.*, 13(1s):28:1–28:24, Dec. 2013.

[93] M. Negrean, M. Neukirchner, S. Stein, S. Schliecker, and R. Ernst. Bounding mode change transition latencies for multi-mode real-time distributed applications. In *IEEE 16th Conference on Emerging Technologies & Factory Automation*, ETFA 2011, pages 1–10, Toulouse, France, 2011.

[94] S. Neuendorffer and E. Lee. Hierarchical reconfiguration of dataflow models. In *Proceedings of second ACM and IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE)*, pages 179–188, June 2004.

[95] H. Nikolov, T. Stefanov, and E. Deprettere. Multi-processor system design with ESPAM. In *CODES+ISSS '06: Proceedings of the 4th international conference on Hardware/software codesign and system synthesis*, pages 211–216, New York, NY, USA, 2006. ACM.

[96] H. Nikolov, T. Stefanov, and E. Deprettere. Systematic and automated multiprocessor system design, programming, and implementation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(3):542–555, March 2008.

[97] H. Nikolov, M. Thompson, T. Stefanov, A. Pimentel, S. Polstra, R. Bose, C. Zissulescu, and E. Deprettere. Daedalus: toward composable multimedia MP-SoC design. In *DAC '08: Proceedings of the 45th annual Design Automation Conference*, pages 574–579, New York, NY, USA, 2008. ACM.

[98] D. Nuzman, I. Rosen, and A. Zaks. Auto-vectorization of interleaved data for simd. In *Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '06, pages 132–143, New York, NY, USA, 2006. ACM.

[99] R. Pellizzoni, P. Meredith, M.-Y. Nam, M. Sun, M. Caccamo, and L. Sha. Handling mixed-criticality in soc-based real-time embedded systems. In *Proceedings of the seventh ACM international conference on Embedded software*, EMSOFT '09, pages 235–244, New York, NY, USA, 2009. ACM.

[100] A. D. Pimentel, C. Erbas, and S. Polstra. A systematic approach to exploring embedded system architectures at multiple abstraction levels. *IEEE Trans. Comput.*, 55:99–112, February 2006.

[101] L.-N. Pouchet. PolyBench: the polyhedral benchmark suite. `http://www.cs.ucla.edu/~pouchet/software/polybench/`.

[102] W. Pugh and E. Rosser. Iteration space slicing and its application to communication optimization. In *Proceedings of the 11th international conference on Supercomputing*, ICS '97, pages 221–228, New York, NY, USA, 1997. ACM.

[103] U. Ramacher, W. Raab, J. A. U. Hachmann, D. Langen, J. Berthold, R. Kramer, A. Schackow, C. Grassmann, M. Sauermann, P. Szreder, F. Capar, G. Obradovic, W. Xu, N. Brüls, K. Lee, E. Weber, R. Kuhn, and J. Harrington. Architecture and implementation of a software-defined radio baseband processor. In *Proceedings of IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 2193–2196, 2011.

[104] J. Real and A. Crespo. Mode change protocols for real-time systems: A survey and a new proposal. *Real-Time Syst.*, 26(2):161–197, Mar. 2004.

[105] A. Schrijver. *Theory of Linear and Integer Programming*. Wiley, 1998.

[106] Z. Shen, Z. Li, and P.-C. Yew. An empirical study on array subscripts and data dependencies. In *International Conference on Parallel Processing (ICPP)*, pages 145–152, 1989.

[107] S. Sriram and S. S. Bhattacharyya. *Embedded Multiprocessors: Scheduling and Synchronization*. CRC Press, 2 edition, February 2009.

[108] N. Stoimenov, S. Perathoner, and L. Thiele. Reliable mode changes in real-time systems with fixed priority or EDF scheduling. In *Proceedings of the*

Conference on Design, Automation and Test in Europe, DATE '09, pages 99–104, 3001 Leuven, Belgium, Belgium, 2009. European Design and Automation Association.

[109] S. Stuijk, T. Basten, M. C. W. Geilen, and H. Corporaal. Multiprocessor resource allocation for throughput-constrained synchronous dataflow graphs. In *DAC '07: Proceedings of the 44th annual Design Automation Conference*, pages 777–782, New York, NY, USA, 2007. ACM.

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

[111] S. Stuijk, M. Geilen, and T. Basten. A predictable multiprocessor design flow for streaming applications with dynamic behaviour. In *Proceedings of the 2010 13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools*, DSD '10, pages 548–555, Washington, DC, USA, 2010. IEEE Computer Society.

[112] S. Stuijk, M. Geilen, B. Theelen, and T. Basten. Scenario-aware dataflow: Modeling, analysis and implementation of dynamic applications. In *International Conference on Embedded Computer Systems (SAMOS), 2011*, pages 404–411, July 2011.

[113] A. Stulova, R. Leupers, and G. Ascheid. Throughput driven transformations of synchronous data flows for mapping to heterogeneous MPSoCs. In *SAMOS XII: International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation*, pages 144–151. IEEE, 2012.

[114] 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. *ACM/IEEE International Conference on Formal Methods and Models for Co-Design,*, 0:185–194, 2006.

[115] L. Thiele, I. Bacivarov, W. Haid, and K. Huang. Mapping applications to tiled multiprocessor embedded systems. In *Proc. 7th Intl Conference on Application of Concurrency to System Design (ACSD 2007)*, pages 29–40, Bratislava, Slovak Republic, 2007. IEEE Computer Society.

[116] W. Thies and S. Amarasinghe. An empirical characterization of stream programs and its implications for language and compiler design. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT '10, pages 365–376, New York, NY, USA, 2010. ACM.

[117] W. Thies, M. Karczmarek, and S. P. Amarasinghe. StreamIt: A language for streaming applications. In *Proceedings of the 11th International Conference on Compiler Construction*, CC ’02, pages 179–196, London, UK, UK, 2002. Springer-Verlag.

[118] M. Thompson. *Tools and techniques for efficient system-level design space exploration.* PhD thesis, University of Amsterdam, 2012.

[119] E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.

[120] S. van Haastregt and B. Kienhuis. Automated synthesis of streaming C applications to process networks in hardware. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE ’09, pages 890–893, 3001 Leuven, Belgium, Belgium, 2009. European Design and Automation Association.

[121] S. K. Venkata, I. Ahn, D. Jeon, A. Gupta, C. Louie, S. Garcia, S. Belongie, and M. B. Taylor. SD-VBS: The San Diego Vision Benchmark Suite. In *IEEE Interational Symposium on Workload Characterization*, 2009.

[122] S. Verdoolaege. *Handbook on signal processing systems*, chapter Polyhedral process networks. Springer, 2010.

[123] S. Verdoolaege. Isl: An integer set library for the polyhedral model. In *Proceedings of the Third International Congress Conference on Mathematical Software*, ICMS’10, pages 299–302, Berlin, Heidelberg, 2010. Springer-Verlag.

[124] S. Verdoolaege, F. Catthoor, M. Bruynooghe, and G. Janssens. Multi-dimensional incremental loop fusion for data locality. In *Proceedings 2003 Application-specific Systems, Architectures and Processors,*, pages 17–27. IEEE, 2003.

[125] S. Verdoolaege, H. Nikolov, and T. Stefanov. pn: a tool for improved derivation of process networks. *EURASIP J. Embedded Syst.*, 2007(1):19–19, 2007.

[126] S. Verdoolaege, H. Nikolov, and T. Stefanov. On demand parametric array dataflow analysis. In *3rd International Workshop on Polyhedral Compilation Techniques*, IMPACT’13, 2013.

[127] S. Verdoolaege, R. Seghir, K. Beyls, V. Loechner, and M. Bruynooghe. Counting integer points in parametric polytopes using barvinok’s rational functions. *Algorithmica*, 48(1):37–66, 2007.

[128] N. Wattanapongskorn and D. W. Coit. Fault-tolerant embedded system design and optimization considering reliability estimation uncertainty. *Reliability Engineering & System Safety*, 92(4):395 – 407, 2007.

[129] M. H. Wiggers, M. J. G. Bekooij, and G. J. M. Smit. Buffer capacity computation for throughput-constrained modal task graphs. *ACM Trans. Embed. Comput. Syst.*, 10:17:1–17:59, January 2011.

[130] M. Woh, Y. Lin, S. Seo, S. Mahlke, and T. Mudge. Analyzing the next generation software defined radio for future architectures. *J. Signal Process. Syst.*, 63(1):83–94, Apr. 2011.

[131] M. Woh, Y. Lin, S. Seo, S. Mahlke, T. Mudge, C. Chakrabarti, R. Bruce, D. Kershaw, A. Reid, M. Wilder, and K. Flautner. From SODA to scotch: The evolution of a wireless baseband processor. *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, 0:152–163, 2008.

[132] W. Wolf, A. A. Jerraya, and G. Martin. Multiprocessor system-on-chip (MPSoC) technology. *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, 27(10):1701–1713, Oct. 2008.

[133] H. Yang and S. Ha. Pipelined data parallel task mapping/scheduling technique for MPSoC. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '09, pages 69–74, 3001 Leuven, Belgium, Belgium, 2009. European Design and Automation Association.

[134] M. Yue. A simple proof of the inequality FFD (L) $\leq$11/9 OPT (L) $+$ 1, $\forall L$ for the FFD bin-packing algorithm. *Acta Mathematicae Applicatae Sinica*, 7:321–331, 1991.

[135] J. T. Zhai, M. A. Bamakhrama, and T. Stefanov. Exploiting just-enough parallelism when mapping streaming applications in hard real-time systems. In *Proceedings of the 50th Annual Design Automation Conference*, DAC '13, pages 170:1–170:8, New York, NY, USA, 2013. ACM.

[136] J. T. Zhai, H. Nikolov, and T. Stefanov. Modeling adaptive streaming applications with parameterized polyhedral process networks. In *Proceedings of the 48th Design Automation Conference*, DAC '11, pages 116–121, New York, NY, USA, 2011. ACM.

[137] J. T. Zhai, H. Nikolov, and T. Stefanov. Mapping streaming applications considering alternative application specifications (extended abstract). In *Proceedings of IEEE Symposium on Embedded Systems For Real-time Multimedia*, ESTIMedia'12, page 27, 2012.

[138] J. T. Zhai, H. Nikolov, and T. Stefanov. Mapping of streaming applications considering alternative application specifications. *ACM Trans. Embed. Comput. Syst.*, 12(1s):34:1–34:21, Mar. 2013.

[139] J. Zhu, I. Sander, and A. Jantsch. Constrained global scheduling of streaming applications on MPSoCs. In *Proceedings of the 2010 Asia and South Pacific Design Automation Conference*, ASPDAC '10, pages 223–228, Piscataway, NJ, USA, 2010. IEEE Press.

Samenvatting

Veel systemen bevatten een of meer ingebedde elektronische subsystemen die onontbeerlijk onmisbaar zijn voor een goed functioneren van het systeem als geheel. Een voorbeeld van een ingebed systeem is (de implementatie/realisatie) een 'functie' die een onbegrensde ingangsstroom van data (tokens) transformeert tot een onbegrensde uitgangsstroom van data (tokens). Denk aan audio en video coders en decoders, of aan netwerkknopen. Bij datastroom toepassingen van dit type kan als voorwaarde gesteld worden dat de tokens in de uitgangsstroom elkaar zo snel mogelijk opvolgen, dit wil zeggen dat de doorvoer maximaal is. Een andere, vaak opgelegde, voorwaarde kan zijn dat tokens in de uitgangsstroom strikt binnen een bepaalde tijd beschikbaar komen. Aan beide voorwaarden kan vaak slechts worden voldaan als de implementatie/realisatie van de onderliggende 'functie' zelf een netwerk van actoren is, dit wil zeggen een MPSoC (Multi-Processor System-on-Chip). Deze toepassingen kunnen bovendien adaptief zijn waardoor (her)configuratie van het ingebedde systeem mogelijk moet zijn om te kunnen schakelen tussen '*modes*'.

Dit proefschrift specificeert en beschrijft een extensie van het ontwerp raamwerk, Daedalus$^{\mathrm{RT}}$, dat bestaat uit een verzameling ontwerp/implementatie/realisatie modules waarmee datastroom toepassingen vrijwel geheel automatisch kunnen worden geimplementeerd/gerealiseerd als MPSoC systemen. Met de voorgestelde extensie wordt de verzameling van mogelijke datastroom toepassingen uitgebreid met dynamische toepassingen. Daedalus$^{\mathrm{RT}}$ maakt gebruik van twee datastroom modellen: een model voor de analyse, en een model voor de MPSoC implementatie. Met het analysemodel kan het gedrag in de tijd van het ingebedde systeem formeel worden bestudeerd. Met het implementatiemodel kan de efficiëntie van de code generatie voor de actoren, de communicatie tussen actoren, en de synchronisatie van actoren in de uiteindelijke realisatie nauwkeurig geschat worden. Het proefschrift breidt het ontwerp raamwerk Daedalus$^{\mathrm{RT}}$ uit met een analysemodel en een implementatie model voor adaptieve datastroomtoepassingen. Het voorgestelde analysemodel is een mode-bewust datastroommodel. Dit wil zeggen dat het model het tijdsgedrag van een aantal modes, en de mogelijke transities tussen modes adequaat weergeeft. Met de voorgestelde transitieprotocol kan een activeringsschema voor de actoren opgesteld worden dat voldoet aan hard real-time van de MPSoC realisatie/implementatie. Het implementatiemodel is een parametrisch hyper-polygon process netwerk model (P^3N) dat instant (her)configuratie van parameters verifieerbaar correct uitvoert. Metingen aan de door het raamwerk Daedalus^{RT} afgeleverde MPSoC P^3N implementatie/realisatie van adaptieve datastroom applicaties bevestigt dat de (her)configuratie van parameters een te verwaarlozen invloed heeft op de prestatie van de MPSoC realisatie.

Acknowledgments

First, I really appreciate that all professors in the defence committee carefully read this thesis and provide many valuable comments. In particular, I would like to thank Professor Twan Basten. He has spotted even small inconsistencies between images and text. His effort has greatly improved the quality of this thesis.

The idea of working towards this thesis started in 2007 when I was still at Institute of Hardware-Software-Co-Design under the guidance of Professor Jügen Teich and Dr. Frank Hannig. That was the first time I saw FPGA and hardware acceleration for high throughput imaging processing. With the courage of Professor Teich and Frank, I made the first step into this fantastic world of embedded system design. I can still remember Professor Teich and Frank's always-smiley face when we had discussions. I also learnt a lot from their research attitude. When we worked on my first international publication, Professor Teich and Frank spent one weekend right when I needed feedback urgently.

Finishing this thesis without help from colleagues would certainly be impossible. Since I started at Leiden Embedded Research Center, I have had luck to work closely with Sjoerd Meijer, Hristo Nikolov, Sven van Haastregt, Mohamed Bamakhrama. We have had countless discussions that truly broaden my horizon. Lately I have also worked with Jelena Spasic and Di Liu, which was great pleasure. In addition, I would not have had so much fun in the past 4 years in Leiden without colleague/friend/flatmate/wingman Emanuele Cannella. Both the discussion on work and sharing the life outside of the office has been precious memory to me. Among all students I have supervised, I would like to thank Frank van Smeden for his contribution to Daedalus^{RT}. Luckily we are finally colleagues at different place.

Outside of office hours, I shared most of my life with brothers and sisters at International Church of Leiden led by Pastor Andy and Helen. In particular, I have had so much fun with Emma, Dirk, Jerome, and Paulina. Not to mention the new little members, Andrew, Mark Jan, and Pippa. All of you bring so much joy to me. Thank you very much!

Finally I want to thank my beloved Shanshan, mam, and dad for their unconditional support. That has been the major source of encouragement to overcome all difficulties in my life. Although they do not exactly understand what I am working on, only one question they ask me from time to time “Teddy, when can you finish your PhD?”.

Jiali Teddy Zhai
December, 2013
Leiden, The Netherlands

Curriculum Vitae

Jiali Teddy Zhai was born on 16^{th} of October, 1982. In September 2009, he received Diplom Informatik (Master Degree in Computer Science) from Friedrich-Alexander-Universität Erlangen-Nürnberg, Germany. During his study, Teddy worked at Institute for Hardware-Software-Co-Design headed by Prof. Jürgen Teich with the focus on designing high-level synthesis tools targeting high-performance computing systems based on FPGA platforms. In October 2009, Teddy joined the Leiden Embedded Research Center (LERC) which is part of the Leiden Institute of Advanced Computer Science (LIACS) at Leiden University. He was appointed as a research and teaching assistant (Ph.D. student). He was involved in the NEtherlands STreaming (NEST) project in collaboration with NXP semiconductor, Philips Healthcare, etc. The research work culminated in the writing of this Ph.D. dissertation in 2013.

List of Publications

Journal Article

- **Jiali Teddy Zhai**, Hristo Nikolov, and Todor Stefanov, “Mapping of Streaming Applications considering Alternative Application Specifications”, in ACM Transactions on Embedded Computing Systems (TECS), vol. 12, Issue 1s, Article 34, March 2013.

Peer-Reviewed Conference Proceedings

- **Jiali Teddy Zhai**, Mohamed A. Bamakhrama, Todor Stefanov, “Exploiting Just-enough Parallelism when Mapping Streaming Applications in Hard Real-time Systems”, *In the Proceedings of the 50th IEEE/ACM Design Automation Conference (DAC’13)*, pp. 170:1–170:8, Austin, TX, USA, June 2 - 6, 2013. **WINNER of the 2013 HiPEAC Paper Award!**
- **Jiali Teddy Zhai**, Hristo Nikolov, Todor Stefanov, “Mapping Streaming Applications considering Alternative Application Specifications (Extended Abstract)”, *In Proceedings of the 10th IEEE Symposium on Embedded System for Real-Time Multimedia (ESTIMedia’12)*, Tampere, Finland, October 11-12, 2012.
- Mohamed A. Bamakhrama, **Jiali Teddy Zhai**, Hristo Nikolov, Todor Stefanov, “A Methodology for Automated Design of Hard-Real-Time Embedded Streaming Systems”, *In Proceedings of the Conference on Design, Automation and Test in Europe (DATE’12)*, pp. 941–946, Dresden, Germany, March 12-16, 2012.
- **Jiali Teddy Zhai**, Hristo Nikolov, Todor Stefanov, “Modeling Adaptive Streaming Applications with Parameterized Polyhedral Process Networks”, *In the Proceedings of the 48th IEEE/ACM Design Automation Conference (DAC’11)*, pp. 116–121, San Diego, CA, USA, June 5-9, 2011. **WINNER of the 2011 HiPEAC Paper Award!**

Peer-Reviewed Workshop Proceeding

- Mohamed A. Bamakhrama, **Jiali Teddy Zhai**, Todor Stefanov, "An Optimal Design Flow for Hard Real-Time Streaming Systems", *In the Proceedings of the 7th Junior Researcher Workshop on Real-Time Computing (JRWRTC'2013)*, in conjunction with the 21st International Conference on Real-Time and Network Systems (RTNS 2013), Sophia Antipolis, France, October 16-18, 2013.

Publications not Covered in this Thesis

- Di Liu, Jelena Spasic, **Jiali Teddy Zhai**, Gang Chen, and Todor Stefanov, "Resource Optimization for CSDF-modeled Streaming Applications with Latency Constraints", *In Proceedings of the Conference on Design, Automation and Test in Europe (DATE'14)*, Dresden, Germany, March 24-28, 2014.
- Mohamed A. Bamakhrama, **Jiali Teddy Zhai**, "Daedalus/Daedalus^{RT}User Manual", September 2012, the manual can be downloaded from `http://daedalus.liacs.nl/manual.pdf`
- Hritam Dutta, **Jiali Teddy Zhai**, Frank Hannig, Jürgen Teich, "Impact of Loop Tiling on the Controller Logic of Acceleration Engines", *In Proceedings of 20th IEEE International Conference on Application-specific Systems, Architectures, and Processors (ASAP)*, Boston, MA, USA, July 7-9, 2009.

Index

Daedalus^{RT} design flow, 12

Cyclo-Static Data Flow (CSDF), 32

Data-Level Parallelism (DLP), 5

Electronic System Level (ESL), 9

Hard Real Time (HRT), 4

implicit deadline, 34
iteration period (H), 35, 124

Kahn Process Network (KPN), 28

Maximum-Overlap Offset (MOO), 128
Mode-Aware Data Flow (MADF), 118
model-based design, 9
Multi-Processor System-on-Chip (MPSoC), 7

period (T), 34
Pipeline-Level Parallelism (PLP), 5
Polyhedral Process Networks (PPN), 27

repetition vector, 31, 32

Static Affine Nested Loop Programs (SANLP), 26
streaming applications, 1
Strictly Periodic Schedule (SPS), 33
Synchronous Data Flow (SDF), 30
system throughput, 3

Task-Level Parallelism (TLP), 5
Transition delay, 126

utilization, 36

workload (W), 34
Worst Case Execution Time (WCET), 35

List of Abbreviations

CPU	Central Processing Unit
CSDF	Cyclo-Static Data Flow
DLP	Data-Level Parallelism
DSP	Digital Signal Processing
EDF	Earliest Deadline First
ESL	Electronic System Level
FFD	First-Fit Decreasing
Gbps	Giga-bit per second
GPU	Graphic Processing Unit
HRT	Hard Real Time
Kbps	Kilo-bit per second
MADF	Mode-Aware Data Flow
Mbps	Mega-bit per second
MoC	Models of Computation
MPSoC	Multi-Processor System-on-Chip
NoC	Network-on-Chip
PE	Processing Element
PLP	Pipeline-Level Parallelism

RM	Rate Monotonic
SDR	Software-Defined Radio
SIMD	Single Instruction Multiple Data
TLP	Task-Level Parallelism
WCDMA	Wideband Code Division Multiple Access
WCET	Worst Case Execution Time