Performance Analysis and Implementation of Predictable Streaming Applications on Multiprocessor Systems-on-Chip

JUN ZHU

Doctoral Thesis
Royal Institute of Technology, Sweden 2010

TRITA-ICT/ECS AVH 09:02
ISSN 1653-6363
ISRN KTH/ICT/ECS/AVH-09/02-SE
ISBN 978-91-7415-306-4

KTH School of Information and
Communication Technology
SE-164 40 Stockholm
SWEDEN

Akademisk avhandling som med tillstånd av Kungl Tekniska högskolan framlägges till offentlig granskning för avläggande av teknologie doktorsexamen måndagen den 15 november 2010 klockan 9.00 i Sal D, Forum, Kista.

Tryck: Universitetsservice US AB

Abstract

Driven by the increasing capacity of integrated circuits, multiprocessor systems-on-chip (MPSoCs) are increasing widely used in modern consumer electronics devices. In this thesis, the performance analysis and implementation methodologies of predictable streaming applications on these MPSoCs computing platforms are explored. The functionality and application concurrency are described in synchronous data flow (SDF) computational models, and two state-of-the-art architecture templates are proposed as multiprocessor architectures, i.e., network-on-chip (NoC) based MPSoC and hybrid reconfigurable CPU/FPGA platforms. Based on the author's contributions on simulation and formal analytic methods, both modelling framework and design space exploration workflow have been addressed.

A energy efficient design exploration flow is proposed for streaming applications with guaranteed throughput on NoC based MPSoCs, in which both application throughput analysis and system energy calculation are carried out by *simulation* on a multi-clocked synchronous modelling framework. On the other hand, based on event models of data streams, a *formal analytic* scheduling framework for real-time streaming applications with minimal buffer requirement on hybrid CPU/FPGA architectures is exploited. The problem has been formalized declaratively as constraint base scheduling, and solved by a public domain constraint solver. Consecutively, the constraint based analytic method has been extended to solve problems ranging from global computation/communication scheduling and reconfiguration analysis to Pareto efficient design. Finally, a prototype of stream processing system on FPGA based MPSoC is built as a realistic projection on the final implementation, to make the results of theoretical studies in this thesis more meaningful.

ACKNOWLEDGEMENTS

I would first like to thank my supervisors docent Ingo Sander and Prof. Axel Jantsch for their professional supervision. It is a pleasure to be a student of Ingo, who has really put efforts to create a collaborative environment in the group and has encouraged and contributed to my work. Especially, I enjoy various conversations with him on our bicycle trips in the beautiful summer of Stockholm. I thank Axel for originally accepting me as his student and for his inspiring advice on my studies. I have over the years been motivated by his serious scientific spirit in research, and appreciate his efforts to offer the financial support in this Ph.D research.

I thank Prof. Rassul Ayani and Prof. Li-Rong Zheng for welcoming my application to KTH some years ago, and thank Prof. Ahmed Hemani for helping me on the proof reading of this thesis. I thank Mikael Z. Lagerkvist and Dr. Christian Schulte for their fantastic work on *Gecode* constraint solver, which I have substantially used in the last years of my Ph.D study. Over the years, I have sat in the same office room with Kim Petersén, Lars Hellberg, Wajid Hassan, Mohammad Badawi, and Zhipeng Chen respectively, and have worked in the same corridor with: docent Johnny Öberg, Prof. Elena Dubrova, Dr. Tarvo Raudvere, Dr. Zhonghai Lu, Mikael Millberg, Mladen Nikitovic, Jian Liu, Sandro Penolazzi, Fredrik Lundevall, Mikael Collin, Dr. Roshan Weerasekara, and other current or former members in our SAM group. Also, I have the opportunity to worked closely with three Master students Hang Zhang, José Henrique Calado, and Shaochun Guo. Here, I would like to express my joy to meet them and appreciate everyone who has been helpful in my work on the way to this thesis. Especially, I thank Agneta Herling and Alina Munteanu for all the administrative help, and to system group for computer related problems.

My Ph.D study has been inspired by Prof. Twan Basten, Dr. Sander Stuijk, Dr. Marc Geilen, and Yang Yang for their work on SDF models. I thank them for kindly helping me to arrange a research visit to their group and for very useful comments on my work. I thank Prof. David Andrews, Dr. Zonghua Gu, Dr. Kaiyu Chen, and Dr. Juan Chen for all the useful discussions with them.

In particular, I wish to give my special thanks to all my friends and my Chinese fellow students for their support through these years when I am in Sweden.

Finally, I thank my family for their endless love.

Jun Zhu,
Stockholm, Summer 2010

CONTENTS

List of Publications

Articles:

- J. Zhu, I. Sander, and A. Jantsch, Performance analysis of reconfigurations in adaptive real-time streaming applications, in *ACM Transactions in Embedded Computing Systems – Special issue on Embedded Systems for Real-time Multimedia*, 201x (*accepted*).

In Proceedings:

- J. Zhu, I. Sander, and A. Jantsch, Pareto Efficient Design for Reconfigurable Streaming Applications on CPU/FPGAs, in *Proceedings of Design Automation and Test in Europe (DATE '10)*, Dresden, Germany, March 2010, pages 1035-1040.

- J. Zhu, I. Sander, and A. Jantsch, Constrained Global Scheduling of Streaming Applications on MPSoCs, in *Proceedings of the conference on Asia South Pacific Design Automation (ASP-DAC '10)*, Taipei, January 2010, pages 223-228.

- J. Zhu, I. Sander, and A. Jantsch, Buffer minimization of real-time streaming applications scheduling on hybrid CPU/FPGA architectures, in *Proceedings of Design Automation and Test in Europe (DATE '09)*, Nice, France, April 2009, pages 1506-1511.

- J. Zhu, I. Sander, and A. Jantsch, Energy efficient streaming applications with guaranteed throughput on MPSoCs, in *Proceedings of the 7th ACM* *international conference on Embedded Software (EMSOFT '08)*, Atlanta, GA, USA, October 2008, pages 119–128.

- J. Zhu, I. Sander, and A. Jantsch, Performance analysis of reconfiguration in adaptive real-time streaming applications, in *Proceedings of the 6th Workshop on Embedded Systems for Real-Time Multimedia (ESTIMedia '08)*, Atlanta, GA, USA, October 2008, pages 53–58.

Others:

- J. Zhu, Predictable and Concurrent Streaming Applications on MPSoCs: Modelling and Implementations, *Design Automation and Test in Europe (DATE '10) Ph.D. Forum*, March 2010.
- J. Zhu, A. Jantsch, and I. Sander, SDF to synchronous cross domain analysis in ForSyDe stream processing framework, in *2nd HiPEAC Industrial (NXP) Workshop (industrial reviewed)*, Eindhoven, Netherlands, October 2006.

Non-covered in thesis:

- J. Zhu, I. Sander, and A. Jantsch, HetMoC: Heterogeneous Modelling in SystemC, in *Proceedings of Forum for Design Languages (FDL '10)*, Southampton, UK, September 2010 (*accepted*).
- I. Sander, J. Zhu, A. Jantsch, A. Herrholzy, P. A. Hartmanny, and W. Nebelz, High-Level Estimation and Trade-Off Analysis for Adaptive Real-Time Systems, in *Proceedings of the 16th Reconfigurable Architectures Workshop (RAW '09)*, Rome, Italy, May 2009, pages 1-4.

List of Abbreviations

Avalon-MM	Avalon Memory-Mapped
BRAM	Block Random Access Memory
CA	Communication Assist
CMBS	Constraint based Minimal Buffer Scheduling
FIFO	First-In-First-Out (Buffer)
FPGA	Field-Programmable Gate Array
HSDF	Homogeneous Synchronous Data Flow
IP	Intellectual Property
JIT	Just-In-Time
JTAG	Joint Test Action Group
KPN	Kahn Process Network
LE	Logic Elements
MPEG	Moving Picture Experts Group
MIMO	Multiple Inputs and Multiple Outputs
MoC	Model of Computation
MPSoCs	Multiprocessor Systems on Chip
NI	Network Interface
NoC	Network-on-Chip
NP	Non-deterministic Polynomial time
PAPS	Periodic Admissible Parallel Schedule

PASS	Periodic Admissible Sequential Schedule
PC	Personal Computer
RISC	Reduced Instruction Set Computing
RTC	Real-Time Calculus
RTOS	Real-Time Operation System
RTR	Run-Time Reconfigurable
SDF	Synchronous Data Flow
SDRAM	Synchronous Dynamic Random Access Memory
SoC	System-on-Chip
SRAM	Static Random Access Memory
SW/HW	Software/Hardware
UART	Universal Asynchronous Receiver/Transmitter
WCET	Worst Case Exection Time

List of Figures

LIST OF TABLES

CHAPTER 1

INTRODUCTION

For over five decades[1], the advances in silicon technology and the increasing chip density of integrated circuits have followed Moore's law. As the number of transistors on single chips continues to hit new record high, processor manufacturing has entered a billion-transistor era in recent years. Accordingly, the explosive growth in processor capacity has driven the *parallel* revolution in modern computers. That is, processing systems, composed of two or more individual processors on a single chip, are widely used across general-purpose and embedded computing systems. For instance, the commercial billion-transistor processors, unveiled by major manufacturers in early 2010, have single-chip core counts ranging from 4 to 16, as illustrated in Table 1.1[2].

Table 1.1: Major billion-transistor processors in February 2010.

Manufacturer	**Processor**	**Transistors** [a]	**Cores** [a]
Sun	Niagara 3	1.0 billion	16
IBM	POWER7	1.2 billion	8
Intel	Tukwila	2.0 billion	4

[a] Here, the maximal number in specifications is presented.

[1] That is from the invention of the integrated circuit in 1958.

[2] Only three commercial server-class processors presented in IEEE International Solid-State Circuits Conference (ISSCC) [82, 85, 96] are listed out, although Intel has presented another 1.3 billion 48-core experimental processor in ISSCC '10 [48].

This thesis aims at *embedded systems*, which are pervasive computer systems designed to perform one or a few dedicated functions. As *parallelism* becomes a global trend in semiconductor industry, embedded multiprocessor systems, consisting of multiple components of processors, custom circuits, memories and communication links (dedicated wires or networks-on-chip) on the same silicon, have become increasingly popular [84]. Although the more processing elements to be placed on chips, the more operations chips can do, it is non-trivial to harness parallelism effectively in the current multi-core era, not even to mention the upcoming many-core era. Embedded multiprocessor systems-on-chip (MPSoCs) platforms exhibit extreme complexity from the following perspectives.

- **Heterogeneity.** Multiple computation, storage and communication components with timing varieties interact on a single-chip. Both hardware and software modules are integrated in an appropriate architecture, which is usually chosen and derived from a set of possibly diversified architecture templates to meet the different requirements in design specifications.
- **Customizability.** According to the application specific *throughput* on demand, *energy* on demand, and the design cost[3] budget, the system components are fully customizable, e.g., the processor *voltage-frequency* levels, memory *size*s, and communication *bandwidth*.
- **Parallelism.** The computation scheduling, communication, and synchronization for the distributed processing elements in MPSoCs exhibit inherent parallelism, which has been the major theme of embedded computing. For many embedded applications, predictable performance needs to be delivered with the high computation power.
- **Programmability and Reconfigurability.** Software-programmable multiprocessors and run-time reconfigurable (RTR) field-programmable gate array (FPGA) hardware devices are widely used. Besides improving design flexibility and productivity, they add new design challenges as well.

1.1 Motivation

Compared with traditional systems-on-board, heterogeneous embedded MPSoCs have enhanced computation power, programmability, and reduced communication

[3]In this thesis, the design cost is measured by the metric of circuit area, in terms of on-chip buffer memory size or equivalent logic elements.

cost, which make them more suitable for concurrent streaming applications ranging from multimedia, digital signal processing (DSP), and telecommunication to network processing domains. Nevertheless, streaming applications on such platforms always have stringent demands on non-functional properties (e.g., energy dissipation, design cost, and timing), besides functional correctness. A system level design methodology is critical for the design and implementation of streaming applications on embedded MPSoCs, which have both application specific performance demands and implementation (area, energy, and cost) efficiency needs.

However, to capture all the design concerns globally, while making proper design decisions at an early system level, is still a big challenge in embedded system design from two aspects.

- Consumer embedded systems, with low-cost (silicon area), low-power, high-performance and high-portability requirements, always have stringent time-to-market demands.

- On the other hand, each product development involves multiple design processes, such as specification refinement, cost analysis, design optimization, and design iteration. More design flexibility in each process makes system design and decision-making more complex and more time consuming.

A systematic way to design efficient streaming applications on embedded MPSoCs in an early design phase is required. However, such an approach demands contributions from different research fields, such as design methodologies, models of computation, system-level modeling and simulation, and formal analysis methods, besides proof-of-concept prototyping.

1.2 System-level design methodologies

Due to the high design complexity and manufacturing cost, new system-level design methodologies for embedded systems have emerged to deal with the increasing time-to-market pressure. Among them, two promising alternatives are Y-chart scheme [6, 56] and platform-based design [55]. Both of them propose the orthogonalization (i.e., separation of concerns) of application and platform, and use an explicit mapping step to relate application models to architecture platforms. This separation allows designers to map a range instances of application models onto one architecture platform more effectively, and vice versa. Unlike traditional software/hardware co-design approaches, which start from a single design specification [74], it can be used for broader design space exploration, as illustrated by the Y-chart methodology in Figure 1.1.

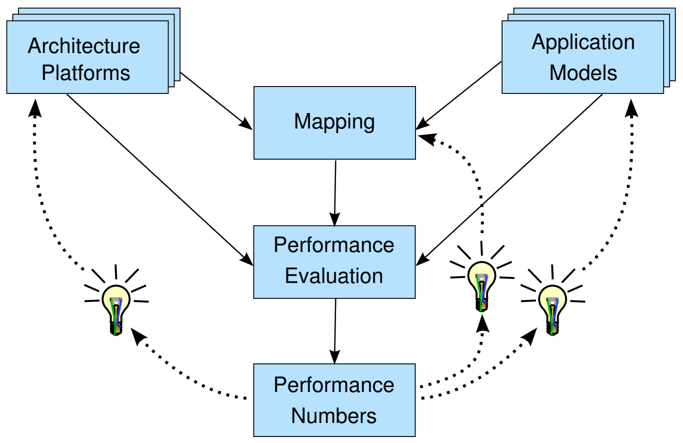


Figure 1.1: Y-chart methodology for design space exploration [56].

In Y-chart methodology, application models are specified as networks of functional blocks based on models of computation [51, 54, 63]. Independently, architecture platforms are characterized as instances of parameterized architecture templates. To assess the mapping of application models onto architecture platforms, a *key* performance evaluation tool is exploited, either by means of simulation or analytical methods. From the resulting performance numbers, the designer may propose improvements on three core design issues: platforms, applications, and mapping decisions, which are iconed as light bulbs in the graph. Such a design process is iterative, until the satisfactory criteria are met.

Y-chart scheme fits various levels of abstraction on applications and platforms. To gain accuracy in performance numbers, the performance evaluation based on more modelling details normally takes longer time. However, too detailed modelling efforts can lead to unacceptable long evaluation time, besides they may not always be available in early design phases. Generally, system-level high abstraction is often used in most Y-chart based design flows for embedded systems in the literature, such as Spade [64], Sesame [75], and CASSE [77].

In this thesis, the conceptions of application and platform have incorporated those in both Y-chart and platform-based design. Based on the orthogonalization of concerns, the application models and architecture platforms are specialized as the following.

- The applications are streaming applications, which can be described as process networks based on synchronous data flow (SDF) models of computation (see Chapter 2.1.1). The SDF models can be used for static scheduling and buffer analysis at compile time, and enforce a clear separation of concerns between computation and communication.
- The platforms are state-of-the-art embedded multiprocessor architectures. In particular, two well-understood MPSoCs templates will be addressed (see Chapter 2.2): tiled MPSoCs with network-on-chip (NoC) communication and hybrid multiprocessor/FPGA. Different architecture instances can be instantiated from parameterized architecture templates, which facilitate the platform reusability in platform-based design.

Individually, the formalism on application functionality and architecture platform can be defined. In the mapping process from functionality to architecture, the result of the performance evaluation is the characterization on the specified performance metrics.

1.3 Outline and contributions

The focus of this thesis is to explore the performance analysis and implementation methodologies of predictable streaming applications on embedded MPSoCs computing platforms, with tight physical (energy, cost, and real-time) constraints. This thesis intends to provide the modelling and design space exploration frameworks for embedded MPSoCs architectures, based on the author's contributions on both simulation and formal analysis methods.

The outline of the thesis is as follows, in which the author's contributions are divided into the corresponding chapters.

Chapter 2

This chapter explains the basics of streaming applications and architecture platforms used in this thesis. Especially, the streaming applications are SDF computational model based.

Chapter 3

This chapter presents a design space exploration flow to achieve energy efficiency for streaming applications on MPSoCs while the specified throughput constraints are met. As the main contributions, the streaming applications are scheduled on a multi-clocked synchronous modeling framework, the application timing properties are guaranteed by throughput analysis, and both processor *voltage-frequency* levels and memory *size*s are customized in the design space to optimize the application pipeline parallelism for energy efficiency.

The public domain simulators Sim-Panalyzer [4] and Cacti [99] are used to estimate the energy dissipations of the parameterized architectural components. Two heuristic algorithms (i.e., greedy and Taboo search) are used during the design optimization process. The experiments show an energy reduction of 21% without any loss in application throughput compared with an ad-hoc approach.

Part of the work was published in

- J. Zhu, I. Sander, and A. Jantsch, Energy efficient streaming applications with guaranteed throughput on MPSoCs, in *Proceedings of the 7th ACM international conference on Embedded Software (EMSOFT '08)*, Atlanta, GA, USA, October 2008, pages 119–128.

- J. Zhu, A. Jantsch, and I. Sander, SDF to synchronous cross domain analysis in ForSyDe stream processing framework, in *2nd HiPEAC Industrial (NXP) Workshop*, Eindhoven, Netherlands, October 2006.

Chapter 4

This chapter addresses the problem of real-time streaming applications scheduling on hybrid CPU/FPGA architectures. The main contribution is a constraint based approach to minimize the buffer requirement for streaming applications on hybrid multiprocessor/FPGA architectures. A novel declarative way of constraint based scheduling for real-time hybrid SW/HW systems is proposed. The event models in analysis are constructed as cumulative functions on data streams, while the application throughput is guaranteed by periodic phases in execution.

Being implemented on the public domain constraint solver *Gecode* [32], a voice-band modem application is used to exemplify the scheduling capabilities of the proposed method. The experimental results show both less buffer requirement and higher throughput guarantees, compared with traditional scheduling methods without constraints on buffer.

Part of the work was published in

- J. Zhu, I. Sander, and A. Jantsch, Buffer minimization of real-time streaming applications scheduling on hybrid CPU/FPGA architectures, in *Proceedings* *of Design Automation and Test in Europe (DATE '09)*, Nice, France, April 2009, pages 1506-1511.

Chapter 5

Based on the extension of the constraint based formalism in Chapter 4, a global computation scheduling and contention-free routing framework for NoC based MPSoCs is proposed. The global scheduling of processors computing and communication transactions are formulated as constraint based problem, to avoid the scheduling overhead in TDMA-like heuristic schemes. Experimental results show that the proposed framework can achieve a high predictable application throughput with minimized buffer cost. For instance, for applications in communication domain, higher throughput (up to 87%) has been observed with less buffer cost, compared to scenarios considering the heuristic scheduling overhead.

Part of the work was published in

- J. Zhu, I. Sander, and A. Jantsch, Constrained Global Scheduling of Streaming Applications on MPSoCs, in *Proceedings of the conference on Asia South Pacific Design Automation (ASP-DAC '10)*, Taipei, January 2010, pages 223-228.

Chapter 6

This chapter proposes a performance analysis framework for adaptive real-time SDF streaming applications on run-time reconfigurable FPGAs. A novel compile-time analysis approach based on iterative timing phases is exploited capture the varying design concerns during reconfigurations. The capabilities of the proposed framework in reconfigurations analysis and design trade-offs analysis are exemplified with experiments.

Part of the work was published in

- J. Zhu, I. Sander, and A. Jantsch, Performance analysis of reconfiguration in adaptive real-time streaming applications, in *Proceedings of the 6th Workshop on Embedded Systems for Real-Time Multimedia (ESTIMedia '08)*, Atlanta, GA, USA, October 2008, pages 53–58.

- J. Zhu, I. Sander, and A. Jantsch, Performance analysis of reconfigurations in adaptive real-time streaming applications, in *ACM Transactions in Embedded Computing Systems – Special issue on Embedded Systems for Real-time Multimedia*, 201x (*accepted*).

Chapter 7

The main contribution is a multi-dimensional design optimization method on state-of-the-art run-time reconfigurable CPU/FPGA platforms. It is a design Pareto-point calculation flow to allocate applications with optimized buffer requirement and software/hardware implementation cost. The capability of the constraint based application allocation, scheduling, and Pareto efficient design has been exemplified by two cases studies on applications from media and communication domains.

Part of the work was published in

- J. Zhu, I. Sander, and A. Jantsch, Pareto Efficient Design for Reconfigurable Streaming Applications on CPU/FPGAs, in *Proceedings of Design Automation and Test in Europe (DATE '10)*, Dresden, Germany, March 2010, pages 1035-1040.

Chapter 8

While the author's theoretical studies have shown the potential of system-level methods on predictable streaming applications design, a FPGA based MPSoCs prototype is used in this chapter, to fill in the gap between system level design methodology proposed in this thesis and final implementation. The prototyping of streaming applications onMPSoCs is built on Altera Stratix II FPGA [2], as a realistic projection to evaluate various properties of the design.

Chapter 9

Finally, this chapter concludes the thesis and gives prospective research directions in the future.

In particular, the workstation used for experiments from Chapter 4 to Chapter 8 is a HP xw4600 Linux workstation with Quad-Core 2.40GHZ processor and 4GB memory.

CHAPTER 2

APPLICATION AND PLATFORM PRELIMINARIES

In system level design, it is essential to capture the functional behavior and architectural characteristics independently for performance analysis and design space exploration. In this thesis, the functionality and application concurrency are described in models of computation (MoCs) based approaches, and two state-of-the-art architecture templates with implementation parallelism are proposed as heterogeneous multiprocessor architectures.

This chapter introduces terminology and notations of the streaming application MoCs and the architecture platforms used in this thesis.

2.1 Application models

A model of computation (MoC) is an abstraction of a computational system. It defines how concurrent computation processes interact. Using different levels of abstractions to leave out the irrelevant properties and details in the system, the MoCs can reconcile essential properties in different purposes of analysis. Therefore, it is possible to use certain MoCs to analyze the system properties of embedded systems [34, 53], such as functional correctness and performance.

In this thesis, the applications are based on synchronous data flow (SDF) models [61].

2.1.1 Synchronous data flow

In SDF graphs, processes execute in a data driven manner, and communicate with each other via FIFO channels. The parallelism in the application is explicitly expressed. A SDF application model with three-stage pipeline is depicted in Figure 2.1, which is used as the tutorial example in this thesis. In the graph, nodes denote the concurrent computation processes, and edges associated with FIFOs denote the communication channels with buffer storage. In each execution or firing, processes read tokens from the input-side FIFOs, operate (compute) on the data within a specified amount of time, and emit the resulting tokens to the output-side FIFOs. The amount of input (output) tokens is fixed in each firing of a process, and is called input (output) *rate*. For instance, when process p_j fires, it reads rate $m_{i,j}$ tokens from $FIFO_{i,j}$ via signal s_2 and writes rate $n_{j,k}$ to $FIFO_{j,k}$ via signal s_3. Obviously, as the signal source or sink, process p_i or p_j does not need the corresponding input- or output-side FIFOs. When all the input (output) rates are 1 in the graph, such a special SDF model is called homogeneous SDF (HSDF) model. A general SDF model can be converted into an equivalent HSDF model [84], but this transformation dramatically increases the problem size. In this thesis, the general SDF models, which are also call multi-rate SDF models, will be considered in later chapters.

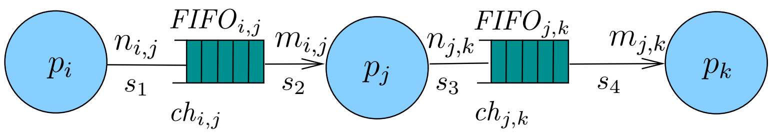


Figure 2.1: An example streaming application model.

In SDF semantics, processes use *blocking read* and *non-blocking write*, i.e., it implicitly assumes infinite buffer space. However, finite buffer storage is always utilized in implementation. A process is enabled and ready for execution, only when both the input-side FIFO(s) has sufficient data tokens and the output-side FIFO(s) has enough vacant space. While a process is computing, the data tokens remain on the input-side FIFO(s) until the computation is completed [87]. At the end of each execution, the output results are available in the output-side FIFO(s). Furthermore, multiple instances of a process are not allowed to execute concurrently as in [42, 87], which is called *auto-concurrency avoidance*. In implementation, a process *executes* only when it is enabled and allowed by the scheduling policy.

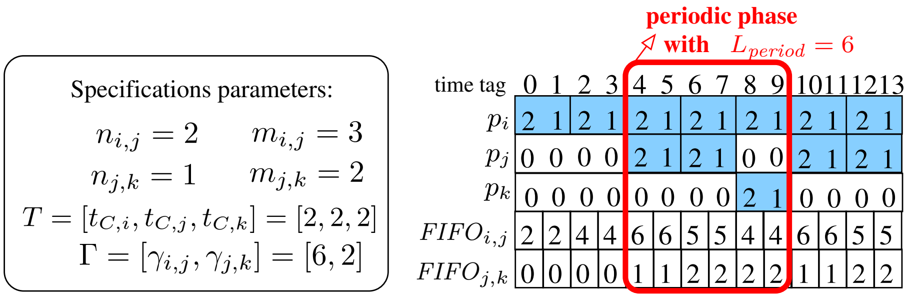
In this thesis, a *consistent* subset of SDF models are considered, which can run indefinitely with bounded buffer [60]. Given the communication channel $ch_{i,j}$ between process p_i and p_j with the input rate $n_{i,j}$ and output rate $m_{i,j}$, for consistent SDF models, p_i and p_j can run in a repetitive pattern with non-trivial (non-zero) firing times[1] r_i and r_j, where r_i and r_j are the minimum integer solutions of a set of balance equations

$$r_i \cdot n_{i,j} = r_j \cdot m_{i,j} \tag{2.1}$$

for all the communication channels. This subset of SDF models are also known to be live and bounded [37]. SDF models provide the suitability for streaming applications scheduling and buffer dimensioning at compile time, and are often used to design predictable parallel embedded systems [60].

To distinguish from regular untimed SDF MoC [60], a *timed-SDF* model [42, 87] has been proposed to analyze timing related properties. A process computation *latency* list T, which contains the time $t_{C,x}$ in time slots (abstract clock cycles) to execute each process p_x once, is used to quantify the process computation. A FIFO *size* list Γ, which contains the storage capacity $\gamma_{y,z}$ in amount of tokens for each buffer $FIFO_{y,z}$, is used to quantify the FIFO storage capabilities. For instance, the three-stage pipelined example application in Figure 2.1 has $T = [t_{C,i}, t_{C,j}, t_{C,k}]$ and $\Gamma = [\gamma_{i,j}, \gamma_{j,k}]$.

2.1.2 Simulation based scheduling and analysis


Specifications parameters:
$n_{i,j} = 2$ $m_{i,j} = 3$
$n_{j,k} = 1$ $m_{j,k} = 2$
$T = [t_{C,i}, t_{C,j}, t_{C,k}] = [2, 2, 2]$
$\Gamma = [\gamma_{i,j}, \gamma_{j,k}] = [6, 2]$
periodic phase
with $L_{period} = 6$

time tag	0	1	2	3	4	5	6	7	8	9	10	11	12	13
p_i	2	1	2	1	2	1	2	1	2	1	2	1	2	1
p_j	0	0	0	0	2	1	2	1	0	0	2	1	2	1
p_k	0	0	0	0	0	0	0	0	2	1	0	0	0	0
$FIFO_{i,j}$	2	2	4	4	6	6	5	5	4	4	6	6	5	5
$FIFO_{j,k}$	0	0	0	0	1	1	2	2	2	2	1	1	2	2

Figure 2.2: A self-timed schedule (on the right) of the example application in Figure 2.1 with specified specification parameters.

[1]A vector of the non-trivial firing times for each process in the model is called *repetition vector*.

For SDF streaming application, simulation based scheduling can be used to analyze the essential timing and buffer properties. Here, the self-timed scheduling [84] is used, which means a process *executes* as soon as it is enabled; otherwise, it *stalls*. The example application specification is instantiated with computation latency list $T = [2, 2, 2]$, storage size list $\Gamma = [6, 2]$, and process input/output token numbers $n_{i,j} = 2$, $m_{i,j} = 3$, $n_{j,k} = 1$ and $m_{j,k} = 2$, as illustrated on the left of Figure 2.2. Thus, the application process repetition vector is $< r_i, r_j, r_k > = < 3, 2, 1 >$.

The corresponding self-timed schedule based on simulation is illustrated on the right side, in which the process and FIFO status are listed in separated rows. The time evolution is depicted in corresponding columns and advances 1 per column. At each time tag, a process in $executing$ (shadowed) state has a number to denote the remaining execution time slots, a $stalling$ (non-shadowed) process status is denoted as 0, and a FIFO status is denoted as the occupied storage space (existing tokens plus self-timed reservation) in number of tokens.

At time tag 0, the process status list is $T'_0 = [2, 0, 0]$, in which p_i is executing with 2 time slots left and p_j and p_k are stalled; in the meantime the FIFO status list is $\Gamma'_0 = [2, 0]$, with only 2 tokens space reserved at $FIFO_{i,j}$. At time tag 2, p_i finishes the previous computation, emits 2 tokens into $FIFO_{i,j}$, and reserves another 2 tokens space from $FIFO_{i,j}$ for a new execution; thus, $T'_2 = [2, 0, 0]$ and $\Gamma'_1 = [4, 0]$. At time tag 4 ($T'_4 = [2, 2, 0]$ and $\Gamma'_4 = [6, 1]$), besides the similar status changes in p_1 and $FIFO_1$, p_2 is enabled and starts to compute. As the schedule advances to time tag 10, the application encounters the same process and FIFO status list (T' and Γ') as at time tag 4 (i.e., $T'_{10} = T'_4 = [2, 2, 0]$ and $\Gamma'_{10} = \Gamma'_4 = [6, 1]$), and enters a periodic phase. The periodic phase extends from time tag 4 to 9, with length $L_{period} = 6$, in which the process p_i, p_j, and p_k are guaranteed to run 3, 2, and 1 times respectively.

Consequently, the schedule guarantees an average output throughput $\rho_k = \frac{1 \cdot m_{j,k}}{L_{period}} = \frac{1}{3}$ at process p_j and requires buffer storage $\Gamma = [6, 2]$, which are the maximum buffer usages at each FIFO.

2.2 Architecture Platforms

A number of heterogeneous SoC architectures have been proposed in recent years [5, 8, 47, 58, 79, 90, 100]. In this thesis, the author will consider two state-of-the-art SoCs architecture templates, i.e., multiprocessor systems-on-chip (MPSoCs) with network-on-chip (NoC) communication [104, 107] and hybrid multiprocessor/FPGA with run-time reconfigurability [105, 106, 108, 109]. Based on these two generic templates, the designer can derive different architecture instances by assigning parameters and constraints to physical resources, such as number and frequency of processors, memory capacities, and interconnection bandwidth.

2.2.1 NoC based MPSoC

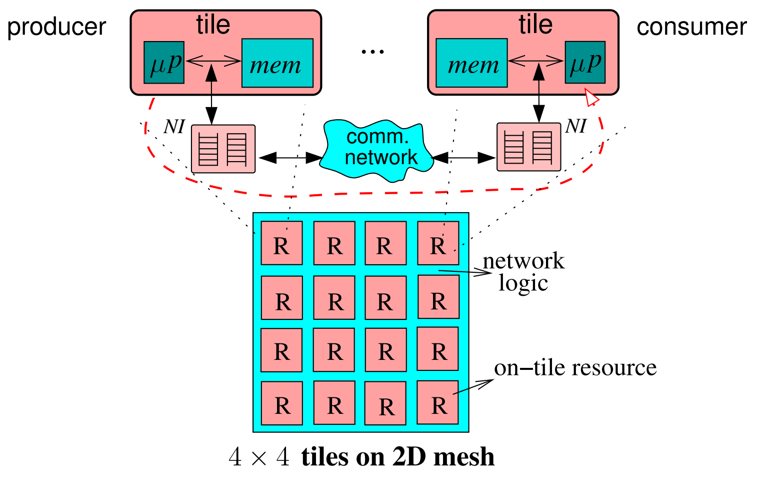


Figure 2.3: An example of NoC based MPSoC architecture template.

The NoC based MPSoC architecture consists of on-tile resources and a packet switched NoC communication [24, 48]. The example architecture template with 4×4 tiles on a 2D mesh NoC topology is illustrated in Figure 2.3.

Each on tile resource (R) contains one processor (μp) and one local distributed SRAM memory (mem). Each processor has a single-cycle access to the local SRAM memory, and no cache is needed. Each on tile resource can be a signal producer, consumer, or both, and is decoupled from the communication network through the network interface (NI). The on tile resources interact with each other via the 2D mesh on-chip communication network.

Here, the multiple processor and memory modules are heterogeneous, in the sense that they have customizable running speed and buffer sizes respectively. The NoC communication provides guaranteed bandwidth and bounded latency, which is achieved with a hard real-time communication backbone [40, 68].

The multiple processors provide programmability as well as high performance, and are suitable to process concurrent streams of data associated with signal processing or multi-media services. The NoC based communication architecture provides scalability, flexibility, and high bandwidth, compared with the traditional bus-based communication. However, a systematic way to customize these system resources efficiently, while the design cost budget and timing related requirement (e.g., energy minimization and throughput guarantees) are still met, becomes crucial.

2.2.2 Hybrid reconfigurable multiprocessor/FPGA

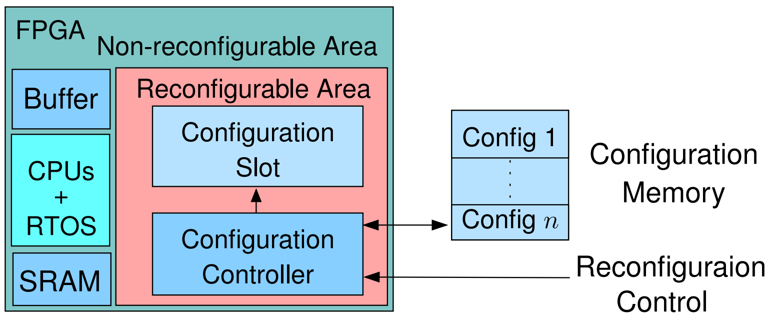


Figure 2.4: Overview of a partially RTR hybrid multiprocessor/FPGA using JIT reconfiguration.

A FPGA platform offers many potential advantages, including high performance, high integration, short time to market and easy field upgrades. Now with the advent of large, fast, and cheap FPGAs, it is possible to place multiple CPUs and custom circuits on the same FPGA die. The so-called hybrid *CPU/FPGA* or *multiprocessor/FPGA* architecture provides an excellent platform for developing custom MPSoCs [5]. The example platforms are Xilinx [101] Virtex and Altera [2] Stratix FPGAs, embedded with either PowerPC processors and MicroBlaze soft cores [101] or Nios soft cores [2].

On the other hand, FPGAs, which are partially run-time reconfigurable (RTR), are also very popular in today's embedded systems, such as Xilinx [101] Virtex-4. The architecture template of such a partially RTR hybrid multiprocessor/FPGA platform is illustrated in Figure 2.4.

On this platform, the applications can be implemented as software running on the host CPUs, while the critical portions are migrated into custom hardware circuits. The hybrid software (CPU) and hardware (FPGA) parts can use direct communication channels to communicate and buffer data, e.g., the Stratix multi-channel shared memory FIFO core. It combines the merits of cost and energy efficiency of FPGAs and the easy programmability of CPUs.

For discussion, the reconfiguration-related FPGA area is divided into two parts, excluding the non-reconfigurable area.

1. The *configuration memory* is used to store the bit stream of all configurations in different working modes in a compressed (with the ratio k_C) format. It can either be a local on-chip memory, or an external memory (Flash, DDRAM, and SDRAM).

2. The *reconfigurable area* is the space for configurations that are only needed for a limited amount of time at run-time. It can be used to store several configurations at the same time. However, this thesis focuses on a "just-in-time" (JIT) approach, in which a single[2] *configuration slot* is shared by different configurations at run-time. Every time a new system function is needed, the *configuration controller* enables the reconfiguration, and the bit stream of the new hardware implementation is loaded from the configuration memory into the reconfiguration slot.

With the run-time reconfigurability, RTR FPGAs allow part of their hardware tasks to be loaded dynamically, and they can be used to built cost-effective hardware platform for stream processing with high flexibility [57]. However, the hybrid CPU/FPGA architecture demands not only new design paradigms [5], but also novel scheduling polices on real-time operation systems (RTOS). Furthermore, the performance analysis for reconfigurable FPGAs is more complex compared with methodologies on traditional non-reconfigurable systems and adds new challenges.

2.3 Discussions

Using deterministic scheduling policies, such as self-timed execution, the above simulation based way can construct schedules at design time for SDF applications without computation resource constraints (i.e., each process is mapped onto an individual dedicated processing element). From the periodic phases (see the schedule

[2]For some applications with predictable MCR, it may be possible to use multiple configuration slots and pre-load the useful configurations to overcome the JIT reconfiguration stall. Such a mechanism *prefetch* adaptation, which remains to be addressed in the author's future work (see Section 9.2).

in Figure 2.2), the application throughput can be guaranteed [38, 42]. Accordingly, the buffer requirement can be analyzed in a finite length of time[3], without everlasting simulation.

However, in the implementation process, i.e., mapping applications onto platforms, the designers still need to deal with the scheduling on multiprocessor architectures within resources constraints, including the computation, communication, and storage physical limits. While the problem has been known to be NP-hard [84], a systematic way to find application schedules according to different design concerns, such as energy efficiency and buffer minimization, is still lacking, especially when multi-clocked domains or run-time reconfigurations on the heterogeneous platforms need to be considered.

In the following chapters, the thesis will present the author's methods to address these issues. In particular, the scheduling policies are restricted to be non-preemptive, to reduce the preemption overhead in implementation for real-time embedded systems. For pre-emptive scheduling on MPSoCs, the work in [97] is referred.

[3]For the Proposition and Proof, see Section 3.4.4

CHAPTER 3

ENERGY EFFICIENCY OF MULTI-CLOCKED MPSOCS

Multiprocessor systems-on-chip based on network-on-chip (NoC) communication have been increasingly adopted as the physical architectures for streaming applications [41, 95]. In streaming applications, a process corresponds to the given computation running on processors. Processes are connected with each other through communication channels, and operate on data streams in a pipelined fashion. The processors can be heterogeneous, each customized for specific tasks, and are running simultaneously to obtain high computation power.

However, performance is not the only concern in a streaming environment and processors are not required to run as fast as possible. Processors, running faster than demanded by the application requirements, will only produce data streams ahead of time and lead to consequent redundant storage buffers [27]. Instead, processors often run at a potentially slower frequency according to the throughput requirements and avoid otherwise higher energy consumption. Nevertheless, to achieve extreme energy efficiency on MPSoCs is non-trivial. The architectural computation, storage, and communication components, which are fully customizable, all contribute to the system energy dissipation. To take all their impacts into consideration in the energy efficiency design, appropriate system models are needed to capture both the parallel nature of streaming applications and the inherent heterogeneous timing properties of MPSoCs as well.

The author's main contribution in this chapter is a design space exploration flow with the following characteristics.

1. The streaming applications are scheduled on a multi-clocked synchronous modeling framework.

2. Timing properties are guaranteed by application throughput analysis.

3. High system energy efficiency on MPSoCs architectures is achieved with customized processor and memory components from optimized pipeline parallelism.

3.1 Related work

Various models of computation (MoCs) [52, 63] have been developed for streaming applications. They are distinguished by their individual formalisms on how processes interact and how or whether time is represented. One is the Kahn process network (KPN) [62]. In [30], a design space exploration framework based on KPN investigates the performance and power trade-offs in MPSoC systems. However, limited by the unbounded FIFO channels in the KPN semantics, it cannot dimension the memory modules and analyze the effects on the system energy cost that arise from parameterized memories.

In the synchronous data flow (SDF) MoC, the static input/output computation tokens allow for the construction of periodic schedules with bounded memory size at compile time [60]. Based on the static memory allocation for digital signal processing (DSP) applications in the SDF MoC, the energy consumption is exploited on the algorithmic level [9]. However, the SDF MoC is untimed, which means the trade-offs between different timing related properties, such as throughput and energy, cannot be explored within this early system model.

A timed extension has been applied to SDF in [42, 87]. Using the timed SDF MoC, a method to minimize the memory size by scheduling the processes appropriately for maximal application throughput is introduced in [42]. In [87], the trade-offs between any specified throughput constraints and the corresponding minimal memory requirements are further studied. However, memory is only one of several factors to impact the total system energy cost, besides computation and communication logics. Nevertheless, the single-unit time assumption in timed SDF [42, 87] does not suit to model the heterogeneous computation and communication timing on MPSoCs systematically.

The synchronous MoC has an explicit global order of events [52]. It has met industrial success to program safety critical reactive systems in Esterel [16], Lustre [44] and Signal [59]. Especially, Lustre and Signal/Polychrony [43] define multi-clocked models for heterogeneous embedded systems. In this chapter, a multi-clocked synchronous MoC framework is proposed in Section 3.4 to analyze timing related properties at the system level in the design space exploration for multi-clocked heterogeneous MPSoCs.

Recently, systematic methods for mapping and scheduling streaming applications with real-time requirements onto a MPSoC platform, where communication happens via NoC backbone, have been addressed in several research groups [46, 65, 88]. In contrast to existing work, this chapter leverages the degrees of customizability of both processor *voltage-frequency* levels and memory *size*s, captures the heterogeneous timing on a multi-clocked synchronous MoC framework, and investigates the minimal energy consumption of streaming applications. High system energy efficiency is achieved without losing throughput guarantees, as confirmed in the experiments.

3.2 Motivation

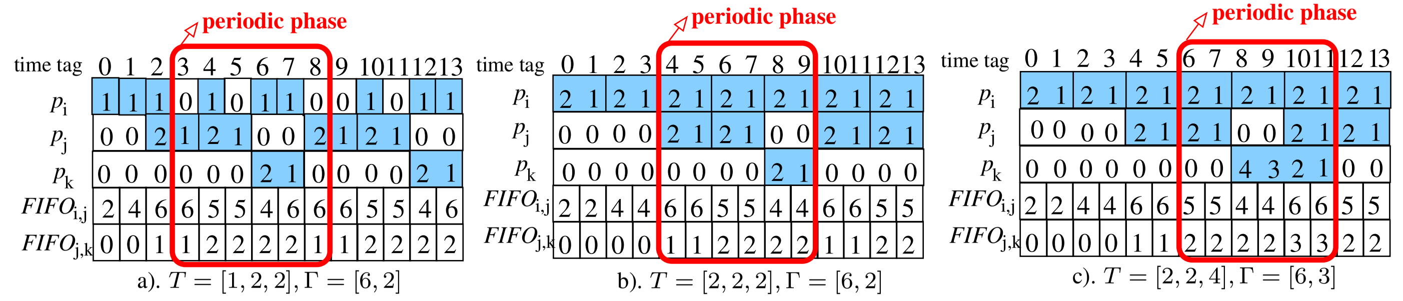
Here the streaming application model in Figure 2.1 is instantiated with input and output rates $n_{i,j} = 2$, $m_{i,j} = 3$, $n_{j,k} = 1$, and $m_{j,k} = 2$. Different design options are explored to motivate the energy efficiency design upon some assumed energy models[1]. For clarification, the problem is motivated in a single domain synchronous MoC, i.e., using one clock with the same time-scale.

3.2.1 Design exploration

In design option *a)*, the computation latency list is assigned as $T = [1, 2, 2]$. Using the memory minimization techniques in [87], the self-timed schedule achieved, with $\Gamma = [6, 2]$, is shown in Figure 3.1.a). In the graph, the process and FIFO status are listed in separated rows as illustrated in Section 2.1.2, and time evolution is depicted in corresponding columns. Using the unit abstract clock, the time tag advances 1 per column.

At time tag 0, the process status list is $T_0' = [1, 0, 0]$, in which p_i is executing with 1 time slot left and p_j and p_k are stalled; in the meantime the FIFO status

[1]Some more practical energy models based on instruction set simulation are adopted in the proposed design space exploration flow in Section 3.3


periodic phase
periodic phase
periodic phase

time tag	0	1	2	3	4	5	6	7	8	9	10	11	12	13
p_i	1	1	1	0	1	0	1	1	0	0	1	0	1	1
p_j	0	0	2	1	2	1	0	0	2	1	2	1	0	0
p_k	0	0	0	0	0	0	2	1	0	0	0	0	2	1
FIFO$_{i,j}$	2	4	6	6	5	5	4	6	6	6	5	5	4	6
FIFO$_{j,k}$	0	0	1	1	2	2	2	2	1	1	2	2	2	2

a). $T = [1, 2, 2], \Gamma = [6, 2]$

time tag	0	1	2	3	4	5	6	7	8	9	10	11	12	13
p_i	2	1	2	1	2	1	2	1	2	1	2	1	2	1
p_j	0	0	0	0	2	1	2	1	0	0	2	1	2	1
p_k	0	0	0	0	0	0	0	0	2	1	0	0	0	0
FIFO$_{i,j}$	2	2	4	4	6	6	5	5	4	4	6	6	5	5
FIFO$_{j,k}$	0	0	0	0	1	1	2	2	2	2	1	1	2	2

b). $T = [2, 2, 2], \Gamma = [6, 2]$

time tag	0	1	2	3	4	5	6	7	8	9	10	11	12	13
p_i	2	1	2	1	2	1	2	1	2	1	2	1	2	1
p_j	0	0	0	0	2	1	2	1	0	0	2	1	2	1
p_k	0	0	0	0	0	0	0	0	4	3	2	1	0	0
FIFO$_{i,j}$	2	2	4	4	6	6	5	5	4	4	6	6	5	5
FIFO$_{j,k}$	0	0	0	0	1	1	2	2	2	2	3	3	2	2

c). $T = [2, 2, 4], \Gamma = [6, 3]$

process	t_x		$\hat{\mathcal{P}}_{\mu p_y, p_x}(v_y, f_y)$	
	(v_H, f_H)	(v_L, f_L)	(v_H, f_H)	(v_L, f_L)
p_i	1	2	1	0.125
p_j	2	4	1	0.125
p_k	2	4	1	0.125

d). Computation latency and energy

FIFO	$\hat{\mathcal{E}}_{mem_y}(\gamma_x)$
FIFO$_{i,j}$	$1 * \gamma_{i,j}$
FIFO$_{j,k}$	$1 * \gamma_{j,k}$

e). Memory energy

design option	energy			$\eta[\%]$
	E_U	E_M	E_{Sum}	
a)	9	8	17	50
b)	6.75	8	14.75	67
c)	5.25	9	14.25	78

f). Energy and computation efficiency analysis

Figure 3.1: Self-timed schedules and energy efficiency analysis for different design options on an instance of the example application ($n_{i,j} = 2$, $m_{i,j} = 3$, $n_{j,k} = 1$, and $m_{j,k} = 2$).

list is $\Gamma'_0 = [2, 0]$, with only 2 tokens space reserved at $FIFO_{i,j}$. At time tag 1, p_i finishes the previous computation, emits 2 tokens into $FIFO_{i,j}$, and reserves another 2 tokens space from $FIFO_{i,j}$ for a new execution; thus, $T'_1 = [1, 0, 0]$ and $\Gamma'_1 = [4, 0]$. At time tag 2 ($T'_2 = [1, 2, 0]$ and $\Gamma'_2 = [6, 1]$), besides the similar status changes in p_i and $FIFO_{i,j}$, p_j is enabled and starts to compute. As the schedule advances to time tag 9, the application encounters the same process and FIFO status list as at time tag 3 ($T'_9 = T'_3 = [0, 1, 0]$ and $\Gamma'_9 = \Gamma'_3 = [6, 1]$) and enters a periodic phase. The periodic phase extends from time tag 3 to 8, with the length 6 time slots and process firing times vector $< 3, 2, 1 >$.

To make both the process computation *latency* and FIFO *size* customizable, the design space is extended to explore another two design options *b)* and *c)*. Compared with *a)*, *b)* is just to run p_i at half speed as $T = [2, 2, 2]$ and keeps the same FIFO sizes Γ; *c)* uses $T = [2, 2, 4]$ and $\Gamma = [6, 3]$ to run both p_i and p_k at half speed and assign one more storage element to $FIFO_{i,j}$. Accordingly, their individual self-timed schedules are shown in Figure 3.1.b) and c). Both schedules encounter a periodic phase, with the same length and the same processes firing times vector, as *a)* does. In this sense, all the design options can deliver the same guaranteed application throughput[2].

3.2.2 Energy efficiency evaluation

In the periodic phase, the energy efficiency of different design options is evaluated, based on the following *assumed*[3] computation latency and energy models (only the dynamic energy is considered).

1. Each process p_x is bound to one individual processor μp_y. Each processor operating voltage v_y is set at two voltage levels v_H and v_L (v_L is half of v_H) to explore the design space. The processor operating frequency f_y changes proportionally to v_y ($f_y \propto v_y$); thus, f_L is at half speed of f_H. The computation latency t_x of p_x changes accordingly to $t_x \propto f_y^{-1}$.

2. For each executing time slot of p_x the *dynamic* power consumption on μp_y is $\hat{\mathcal{P}}_{\mu p_y, p_x}(v_y, f_y) = \alpha \cdot v_y^2 \cdot f_y$, where α is the average switching capacitance. Since $f_y \propto v_y$ is known, $\hat{\mathcal{P}}_{\mu p_y, p_x} \propto f_y^3$ is attained as a cubic function of f_y. For n computation time slots the processor energy consumption $E_{\mu p_y}$ is $E_{\mu p_y} = n \cdot \hat{\mathcal{P}}_{\mu p_y, p_x}(v_y^2, f_y)$.

[2]For the formal definition, see Section 3.4.4

[3]Here, the assumed models are only used for illustration. In practical experiments, the computation latency and energy dissipation are got from simulation (see Section 3.3).

3. Each $FIFO_x$ is bound to one individual memory mem_y, and the memory size equals to the FIFO size γ_x. Within a fixed accessing pattern, the memory energy consumption is $E_{mem_y} = \hat{\mathcal{E}}_{mem_y}(\gamma_x) \propto \gamma_x$, where the memory dynamic energy function $\hat{\mathcal{E}}_{mem_y}$ is proportional to memory size.

4. As the baseline, in design option *a)* all the processor *voltage-frequency* levels are set to (v_H, f_H), and for each time slot the processor dynamic power consumption is normalized to 1, as shown in column 4 of Figure 3.1.d); the memory energy consumption in the periodic phase is assigned to 1 per storage element, as shown in Figure 3.1.e).

Thus, the computation latency, computation energy and memory energy characteristics of other design options have been derived in Figure 3.1.d-e). There are three processors and two memory modules in the physical architecture. For design option *a)* ($T = [1, 2, 2]$, $\Gamma = [6, 2]$ and $< r_i, r_j, r_k > = < 3, 2, 1 >$), the total processor energy E_U and total memory energy E_M are

$$\begin{aligned} E_U &= E_{\mu p_1} + E_{\mu p_2} + E_{\mu p_3} = \textstyle\sum_{\forall \mu p_y} t_x \cdot r_x \cdot \hat{\mathcal{P}}_{\mu p_y, p_x}(v_H, f_H) \\ &= 1 \cdot 3 \cdot 1 + 2 \cdot 2 \cdot 1 + 2 \cdot 1 \cdot 1 = 9 \\ E_M &= E_{mem_1} + E_{mem_2} = \textstyle\sum_{\forall mem_y} \hat{\mathcal{E}}_{mem_y}(\gamma_j) \\ &= 1 \cdot 6 + 1 \cdot 2 = 8 \end{aligned}$$

The total system energy consumption is $E_{Sum} = E_U + E_M = 17$. Similarly, the energy consumptions for design options *b)* and *c)* are calculated, as shown in Figure 3.1.f).

Design option *c)* has the most efficient system energy E_{Sum}. In addition, it achieves the highest average processor utilization η (ratio of the shadowed part in the periodic phase of process schedules), which means the best pipelined parallelism. However, higher pipelined parallelism does not always mean lower system energy dissipation, as demonstrated in Section 3.6.

3.3 Design space exploration flow

An energy efficiency design space exploration flow for streaming applications with guaranteed throughput on MPSoCs is proposed, as shown in Figure 3.2. The inputs of the design exploration flow (three boxes on the top) are the application benchmark C program, the static mapping from the application onto the MPSoC architecture and the architectural design options in the configuration file.

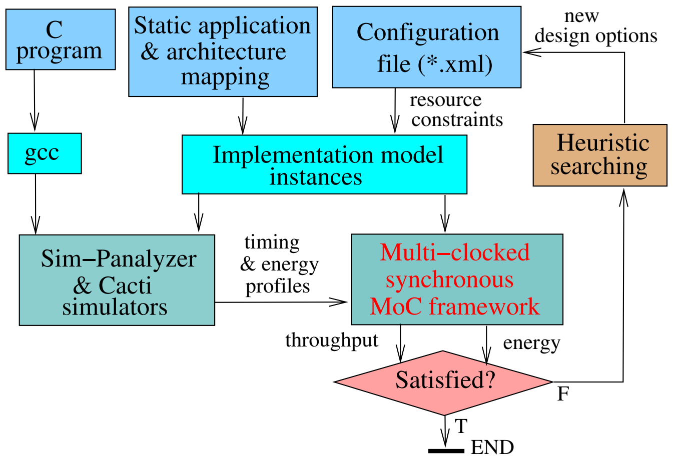

C program
Static application & architecture mapping
Configuration file (*.xml)
new design options
resource constraints
gcc
Implementation model instances
Heuristic searching
Sim–Panalyzer & Cacti simulators
timing & energy profiles
Multi–clocked synchronous MoC framework
throughput
energy
Satisfied?
F
T
END

Figure 3.2: Energy efficiency design exploration flow with guaranteed throughput.

The C program for each computation process is cross-compiled into a binary using the ARM gcc toolchain [1]. From the application to architecture mapping and the resource constraints defined in the configuration file, implementation model instances with different architectural parameters are initialized. The design space is explored by customizing both the processor *voltage-frequency* levels and memory *sizes*, as discussed in Section 3.2.1.

Instead of using the assumed latency and energy models in the previous section, the process binary computation timing and energy dissipations on the architectural component instances are estimated on Sim-Panalyzer [4] and Cacti [99] simulators. Based on the individual process timing and energy profiles, the application throughput and system energy dissipations on MPSoC architecture are analyzed systematically in the multi-clocked synchronous MoC framework. The performance of the application can be determined and optimized at design-time based on the following assumptions:

- Predictable NoC with guaranteed bandwidth and bounded latency.
- Predictable Network Interface (NI).
- Bounded computation time of each process on processing elements.

The design objective is to minimize the total system energy dissipation while meeting the application throughput requirement. As the design space increases exponentially with the problem size, heuristic algorithms (greedy and Taboo [25] search) are used during the design options searching until the design goals are met.

3.4 Multi-clocked synchronous MoC framework

Here, a synchronous MoC framework, which preserves the static computation input/output rates in the SDF MoC, is adopted for timing related properties analysis, e.g., application throughput analysis and system energy evaluation. Furthermore, multi-clocked domains are considered and exploited to relate the heterogeneous timing in different system domains (similar to Lustre [44]), which suit different clocks in heterogeneous MPSoCs well.

3.4.1 Synchronous MoC framework

Preserving the static producing and consuming token rate properties, the SDF models can be transformed into synchronous models [66, 104], in which the timing of process firing and timing-related properties can be captured. In the synchronous MoC, systems are described as a set of concurrent processes, which communicate through synchronous signals.

Stream models

A signal (data stream) s with clock clk_s is an indexed[4] set of events,

$$s = \cup\{e_{(n)}\}^{clk_s} = \{e_{(\mathit{0})}, e_{(\mathit{1})}, \cdots, e_{(n)}, \cdots\}^{clk_s}, \forall n \in \mathbb{N}_0. \tag{3.1}$$

The signal clock $clk_s \in \mathbb{Q}^+$ is the abstract cycle (time slot) period between two adjacent events. It is the elementary time unit, in which time is measured and for which timing related properties (e.g., throughput and energy) are evaluated. Each event $e_{(n)} = (g_{(n)}, \vec{v}_{(n)})$ has a time tag $g_{(n)}$ and a value $\vec{v}_{(n)}$[5]. Time tags are used to model the global order of events, and are implicitly given by the event indexes in the signal, with $g_{(n)} = n \cdot clk_s$; thus, a signal can be simply denoted as $s = \cup\{\vec{v}_{(n)}\}^{clk_s}$. The timing relation of the events in signals with different clocks is visualized in Figure 3.3. Two signals s_1 and s_2 have the clock timing relation $\frac{clk_{s_\mathit{1}}}{clk_{s_\mathit{2}}} = \frac{3}{2}$. The global order of the events in both signals are maintained by time tags.

[4]In this chapter, the subscripted numbers in parentheses are used especially for indexing purpose.

[5]It is an event model both for performance analysis and for functional correctness analysis.

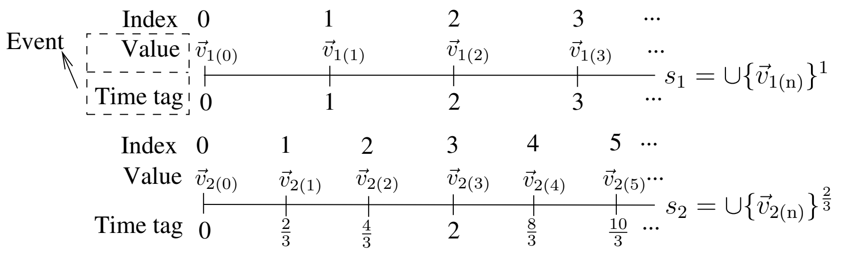

Event
Index
Value
Time tag
$s_1 = \cup\{\vec{v}_{1(\mathrm{n})}\}^1$
$s_2 = \cup\{\vec{v}_{2(\mathrm{n})}\}^{\frac{2}{3}}$

Figure 3.3: Timing relations of events in signals with different clocks.

Process

To capture the input (output) rate in process *executing* cycles[6] and *absent*s in *stalling* cycles, values are a vector $\vec{v}$ of regular tokens, extended with a pseudo value $\perp$. $\vec{v}_{(n)}$ is $\vec{v}$ when the required tokens are *present*, or $\perp$ when *absent*. The number of the regular tokens contained in $\vec{v}$ is fixed. For instance, a synchronous signal $s_1 = \{\perp, <1,1,2>, <2,3,4>, \cdots\}^{\frac{1}{2}}$ has integer token number 3 and clock period $\frac{1}{2}$.

Processes operate on signals. For a set of processes with the same clock of signals, they are said to be in a single domain; otherwise, they are in different multi-clocked domains, which is to be introduced in Section 3.4. In each evaluation cycle, processes consume one event from the input signals and output one event to the output signals. In perfect synchrony [52], the computation and communication are executed in zero time and the computation states are maintained in explicit delay process statements. A synchronous model is composed of combinational processes $p_{comb}(f)$ and unit-cycle delay processes $p_{\Delta}(\vec{v}'_{(0)})$, in which function f specifies the mapping from input events to output events and the given initial state $\vec{v}'_{(0)}$ is the output event at time tag 0 to defer the input events one cycle.

While $p_{comb}(f)$ fits to describes algorithmic function flow, its combination with $p_{\Delta}(\vec{v}'_{(0)})$ can be used to construct control logic and more complex components. For instance, the τ-cycle ($\tau \in \mathbb{N}$) delay process is a combination of τ unit delay process with $p_{\Delta,\tau}(\perp) = \underbrace{p_{\Delta}(\perp) \circ \cdots \circ p_{\Delta}(\perp)}_{\tau}$ [7]; a combinational process with τ-cycle computation latency is $p_{comb,\tau}(f) = p_{comb}(f) \circ p_{\Delta,\tau}(\perp)$; and a *mealy* state machine, with the state function f_{state} and output function f_{out}, has

[6]It is the worst case execution time (WCET) of the process, which is the maximum length of time takes to execute on a specific hardware platform by analysis of a process program flow [20].

[7]The process composition operator $\circ$ has the formal definition $p_1 \circ p_2(s_1) = p_1(p_2(s_1))$

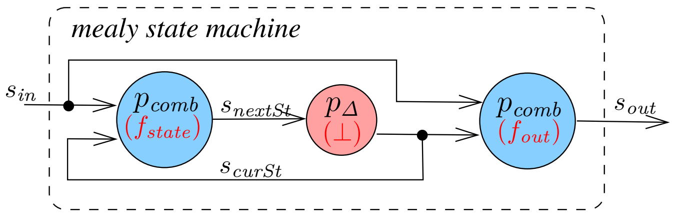


Figure 3.4: Process skeleton of a mealy state machine.

the process skeleton shown in Figure 3.4. The *mealy* state machine is the base to construct the control logic of a finite size FIFO [80].

3.4.2 Multi-clocked synchronous domains and domain interfaces

In the synchronous model, processes using the same abstract clock are said to be in the same synchronous domain. To model the heterogeneous timing properties in embedded systems, such as the parallel computation on several processors running at different frequency levels, multi-clocked synchronous domains are introduced.

When multi-clocked domains exist, i.e., different domain clocks have a rational ratio to each other, asynchronous domain interfaces (DI) are needed to maintain the global timing. In the upper-left part of Figure 3.5, two domains D_A and D_B, with different clock periods clk_A and clk_B, have a rational clock ratio $\lambda_{A:B} = \frac{clk_A}{clk_B} = \frac{m_A}{m_B}, \forall m_A, m_B \in \mathbb{N}, m_A \neq m_B$, and m_A and m_B are relatively prime. The domain interface $DI_{A:B}$, which establishes the clock ratio $\lambda_{A:B}$ from *A* to *B* is defined as $DI_{A:B} = downDI(m_B) \circ upDI(m_A)$.

As the start point at index 0, all the signal events have the consistent time tag 0. Thus, at tag 0 $upDI(m_A)$ and $downDI(m_B)$ output the value $\vec{v}_{(0)}$ from the input signal. Otherwise, $upDI(m_A)$ is up-sampling the input signal clock m_A times by inserting $m_A - 1$ absent events before each event; and $downDI(m_B)$ is down-sampling the input signal clock m_B times by merging every m_B events. They have the following definitions:

$$
\begin{aligned}
&upDI\,(m_A)\left(\{\vec{v}_{(0)}, \vec{v}_{(1)}, \vec{v}_{(2)}, \cdots\}^{clk_x}\right) = \\
&\{\vec{v}_{(0)}, \underbrace{\perp, \cdots, \perp}_{m_A-1}, \vec{v}_{(1)}, \underbrace{\perp, \cdots, \perp}_{m_A-1}, \vec{v}_{(2)}, \cdots\}^{\frac{clk_x}{m_A}} \qquad (3.2)
\end{aligned}
$$

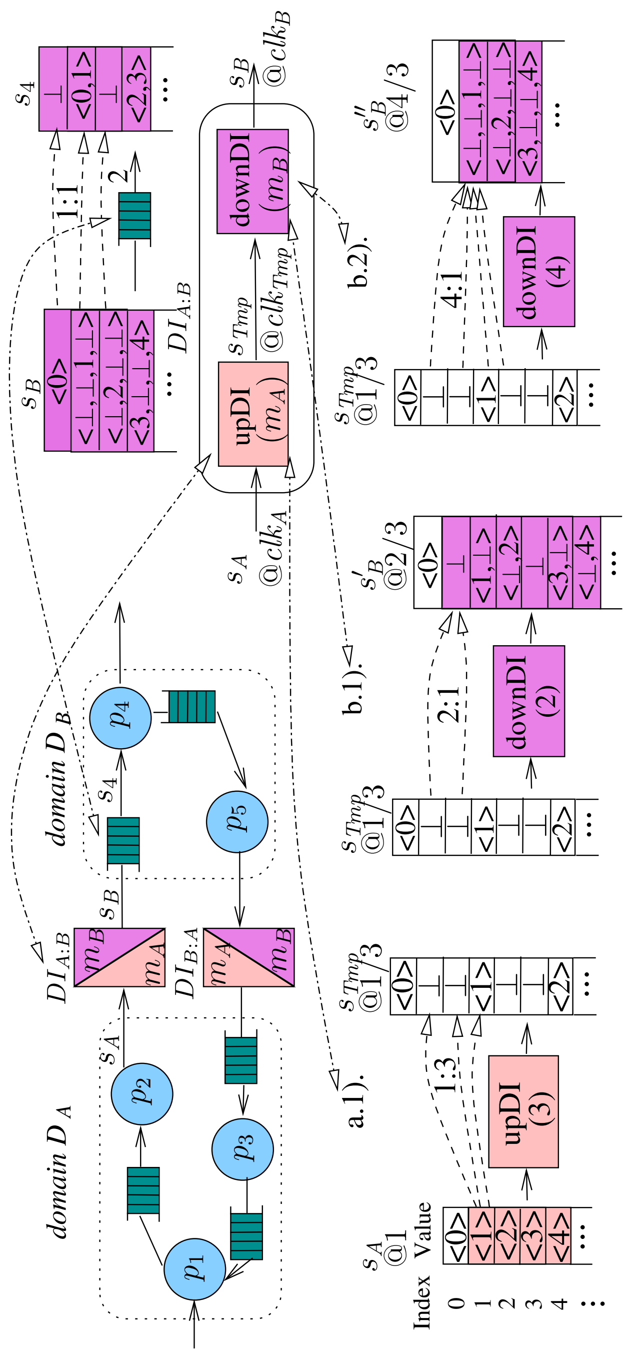


Figure 3.5: Multi-clocked synchronous domains and domain interfaces.

$$\begin{gathered} downDI(m_B)(\{\vec{v}_{(0)},\cdots,\vec{v}_{(m_B)},\vec{v}_{(m_B+1)},\cdots\}^{clk_x}) = \\ \{\vec{v}_{(0)},<\vec{v}_{(1)},\cdots,\vec{v}_{(m_B)}>,<\vec{v}_{(m_B+1)},\cdots,\vec{v}_{(2m_B)}>,\cdots\}^{m_B\cdot clk_x} \\ where \quad <\underbrace{\perp,\cdots,\perp}_{m_B}> \quad is\ reduced\ to\ \perp \end{gathered} \tag{3.3}$$

In Figure 3.5.a.1) and b.1-2), the functionalities of several different instances of up- and down-sampling components are illustrated by a unit rate input signal s_A and two instances of output signal s'_B and s''_B, in which s''_B has the slowest clock with $clk_B = \frac{4}{3}$ (simply denoted as $s''_B@\frac{4}{3}$). The signal events are listed in the ascending order of time tags. Without losing or gaining data tokens, the specified input events (shadowed) of s_A are mapped to the output events (shadowed) of s'_B or s''_B in a different clock domain. When $\lambda_{A:B} = \frac{3}{2} > 1$ and clk_B is faster than clk_A, the output data token pattern is getting more sparse, but the original rate (the number of the *non-absent* values in $\vec{v}$) is kept. Otherwise, $\lambda_{A:B} = \frac{3}{4} < 1$ and it leads to increased token rate; especially, when $m_B \neq 1$, the rate of the output events can vary, as shown in Figure 3.5.b.2). However, as the output data stream is always buffered by a FIFO, it does not violate the static token rate assumption for process computation. As shown in the upper-right part of Figure 3.5, the output signal s_4 of the FIFO always provide a constant input rate 2 required by p_4. To ensure consistent timing, $DI_{B:A}$ from domain D_B to D_A has the reversed sampling ratio $\lambda_{B:A}$.

Domain interfaces act as the glue processes between different synchronous domains. When a domain clock time changes, only the domain interface sampling ratios need to be reconfigured, which greatly facilitates the modelling and design space exploration on heterogeneous MPSoCs.

Domain interface causality

Although the domain interface has asynchronous features, its input and output signals do not violate the causality condition of the demand driven simulation.

Proposition 1. *Domain interface $DI_{A:B}$ has input signal s_A at domain D_A and output signal s_B at domain D_B, as shown in Figure 3.6.* $\forall a_1 \in \mathbb{N}_0$, $\exists b_1 \in \mathbb{N}_0$,

$$\begin{aligned} DI_{A:B}(\{\cdots,\vec{v}_{A(a_1)},\cdots\}^{clk_A}) &= \{\cdots,\vec{v}_{B(b_1)},\cdots\}^{clk_B}, \\ where \quad \{\vec{v}_{B(b_1)}\}^{clk_B} &= \{<\cdots,\vec{v}_{A(a_1)},\cdots>\}^{clk_B}. \end{aligned}$$

s.t. the timing relation $Tag(\vec{v}_{\mathrm{B(b_1)}}) \geqslant Tag(\vec{v}_{\mathrm{A(a_1)}})$ exists, in which the operator Tag is to get the time tag $g_{(n)}$ for a specified signal value $\vec{v}_{(n)}$. Hence, $DI_{A:B}$ preserves causality between its input and output signals.

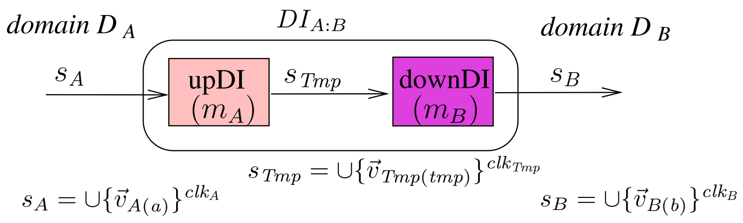

domain D A
$DI_{A:B}$
domain D B
s_A
upDI
(m_A)
s_{Tmp}
downDI
(m_B)
s_B
$s_{Tmp} = \cup\{\vec{v}_{Tmp(tmp)}\}^{clk_{Tmp}}$
$s_A = \cup\{\vec{v}_{A(a)}\}^{clk_A}$
$s_B = \cup\{\vec{v}_{B(b)}\}^{clk_B}$

Figure 3.6: Domain interfaces $DI_{A:B}$.

Proof. In case of $\mathrm{a}_1 = \mathrm{b}_1 = 0$, $Tag(\vec{v}_{B(0)}) \equiv Tag(\vec{v}_{A(0)}) \equiv 0$ meets the timing relation. Otherwise, $\mathrm{a}_1 \geqslant 1$ and $\mathrm{b}_1 \geqslant 1$. From the definition of $upDI(m_A)$ in (3.2), it is known that $clk_{Tmp} = \frac{clk_A}{m_A}$. From the definition of $downDI(m_B)$ in (3.3), it is known that $\forall b_1 \in \mathbb{N}_0$

$$\begin{aligned}\{\vec{v}_{B(b_1)}\}^{clk_B} &= \{< \cdots, \vec{v}_{A(a_1)}, \cdots >\}^{clk_B} \\ &= downDI(m_B) \\ &\quad (\{\vec{v}_{Tmp(m_B\cdot(b_1-1)+1)}, \cdots, \vec{v}_{Tmp(m_B\cdot b_1)}\}^{clk_{Tmp}})\end{aligned}$$

Tag is monotonically increasing, thus

$$Tag(\vec{v}_{A(a_1)}) \leqslant Tag(\vec{v}_{Tmp(m_B\cdot b_1)})$$

From the definition of Tag, it is known that

$$\begin{aligned}Tag(\vec{v}_{B(b_1)}) &= b_1 \cdot clk_B \\ Tag(\vec{v}_{Tmp(m_B\cdot b_1)}) &= m_B \cdot b_1 \cdot clk_{Tmp} = b_1 \cdot (m_B \cdot \frac{clk_A}{m_A}) \\ &= b_1 \cdot clk_B.\end{aligned}$$

Thus, it concludes $Tag(\vec{v}_{B(b_1)}) \geqslant Tag(\vec{v}_{A(a_1)})$. □

3.4.3 Scheduling state and cross domain analysis

As discussed in Section 3.2, in a single synchronous domain D_N, at time tag $g_{(n)}$ $(n \in \mathbb{N}_0)$ the process status list $T'_{N(n)}$ associates with each process the remaining number of time slots when it $executes$, or 0 when it $stalls$; meanwhile, the FIFO status list $\Gamma'_{N(n)}$ associates with each channel the amount of FIFO storage space used. The scheduling state in D_N is a tuple $(T'_{N(n)}, \Gamma'_{N(n)})$.

A multi-clocked application consists of a set of synchronous domains **D**, in which domain D_N with the slowest domain clock clk_N is chosen as the logic mold domain. Each component (process or FIFO) status signal in D_K ($\forall D_K \in \mathbf{D}$) can be cast across domain boundary into the single D_N via $DI_{K:N}$, where $\lambda_{K:N} \leq 1$. Such a single component status signal casting ($\lambda_{K:N} = \frac{3}{4}$) can be illustrated by Figure 3.5.a.1) and b.2). s_A is looked as the status signal in D_K and s''_B the status signal in the mold domain D_N. Each value at time tag $g_{(n)}$ in s''_B is called a scheduling pattern.

To be consistent with the scheduling state definition in the single domain, the scheduling patterns are encoded into incrementing numbers starting from 0, and the same numbers are used to denote the revisited patterns. The functionality of such a encoding module $patternEnc$ is shown in Figure 3.7. In $s_{patternSt}$, the scheduling patterns at time tags 1 and 4 are duplicated, and are encoded as the same 1 in s_{encSt}. In this way, the timing properties for multi-clocked applications can be analyzed by casting the system scheduling states in multi-clocked domains into the single mold domain D_N.

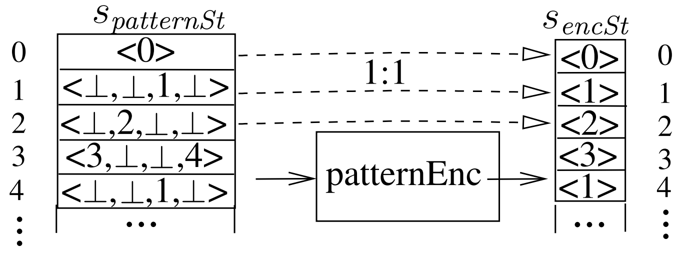


Figure 3.7: Scheduling patterns encoding.

3.4.4 Application throughput analysis and throughput guarantees

Proposition 2. *(Throughput guarantees) For a consistent SDF streaming application (see Section 2.1.1), a periodic phase in its schedule always exists. The required application throughput is guaranteed by the output periodic properties during this period.*

Proof. A consistent SDF streaming application could run indefinitely. However, the application scheduling status (process and FIFO status, see Section 2.1.2) space is always finite. Thus, some scheduling status will be re-visited in a non-terminating schedule. As only deterministic scheduling policies are considered, the application schedule enters a periodic phase when a repeated scheduling status is met. The output throughput during this period could sustain indefinitely, which guarantees the application throughput. □

In the logic mold domain D_N, when the scheduling state $(T'_{N(n_2)}, \Gamma'_{N(n_2)})$ for the application at time tag $g_{(n_2)}$ meets a repeated one as at $g_{(n_1)}(n_1 < n_2, T'_{N(n_2)} = T'_{N(n_1)}, \Gamma'_{N(n_2)} = \Gamma'_{N(n_1)})$, the application schedule enters a periodic phase with length $g_\triangle = g_{(n_2)} - g_{(n_1)}$. In this periodic phase, the process throughput of p_n is defined as

$$Thru(p_n) = \frac{r_n \cdot n_{p_n} \cdot Sz_{token}}{g_\triangle} \tag{3.4}$$

in which r_n is the firing times of p_n, n_{p_n} the output rate considered and Sz_{token} the data size per token. When Sz_{token} in bit and the corresponding physical time to $g_\triangle$ are known, equation (3.4) defines the process throughput in bit/s. Caused by the static token rates, the throughput of various processes in the models have a fixed ratio. The throughput of the sink process is simply used as the application throughput, which is the speed the application delivers outputs.

3.5 Energy efficiency design on MPSoCs

The design objective is to minimize the system energy dissipation while meeting the required application throughput.

3.5.1 Implementation model

The architecture model is an MPSoC template, which consists of on-tile components and a packet switched communication network-on-chip (NoC), as shown in the lower part of Figure 3.8. Each tile contains one processor (μp) and one local SRAM memory (mem), and is decoupled from the communication network through the network interface (NI). Each processor has a single-cycle access to the local SRAM memory, and no cache is needed.

Given an architecture model with a set of processors **U** and a set of memories **M**, the implementation model (a resource-aware refinement of the application model) has the mappings from a set of computation processes **P** onto **U** and a set of FIFOs **F** onto **M**. For simplicity, FIFOs are assumed to be mapped to disjoint memory regions. The techniques in [42], which allow buffer sharing among the FIFOs mapped to the same memory module, are not considered.

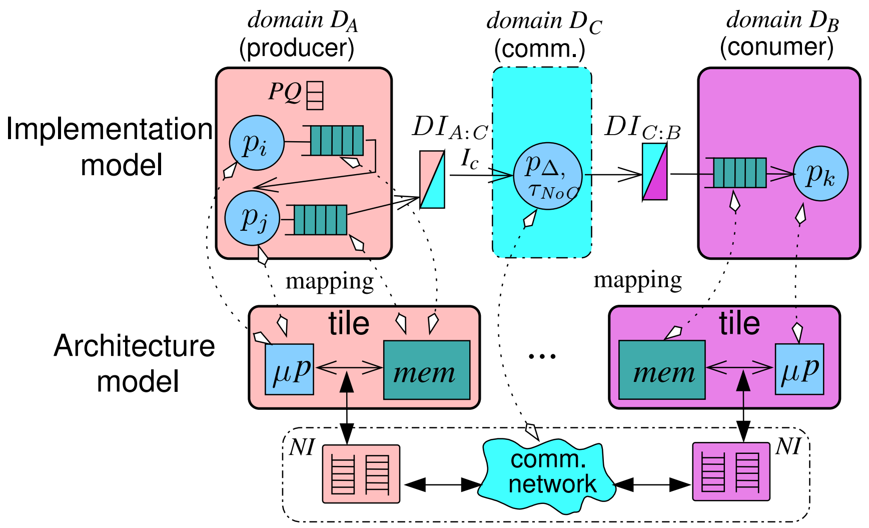


Figure 3.8: Mapping of the implementation model of the example application (Figure 2.1) onto a NoC based MPSoC architecture model.

As the mapping optimization on the NoC communication has been studied in [49] and is out of the scope of this work, an empirical mapping strategy is used as the start of the work in this chapter. Furthermore, the design flexibility in the communication backbone is not investigated and the design option in the NoC communication logic is simply considered to be fixed; instead, this chapter concentrates on exploring the design alternatives of on-tile components.

The mapping of the implementation model of the example application (Figure 2.1) onto a two-tile NoC based MPSoC architecture is shown in Figure 3.8. The MPSoC architectural characteristics are captured in the implementation model, with the following strategies.

1. Each process p_x is mapped to one on-tile processor μp_y, each FIFO $FIFO_x$ to one memory mem_y, and inter-tile channels to NoC communication.

2. Each tile and the NoC communication are modeled in individual domains, and they interact via domain interfaces. In Figure 3.8, the sampling ratios of $DI_{A:C}$ and $DI_{C:B}$ correspond to the heterogeneous timing between different architecture modules.

3. When multiple processes are mapped onto one processor (see domain D_A), their executions are scheduled sequentially according to the non-preemptive assignments in a priority queue (PQ). Thus, the scheduling decision on a single processor only changes when data dependency changes.

4. When multiple FIFOs are mapped onto one memory (see domain D_A), they use independent logic addresses without space sharing, and the memory *size* is the summation of the FIFO *size*s.

5. The NoC logic is assumed to provide a guaranteed bandwidth based on a hard real-time communication backbone [68] between different tiles. In Figure 3.8, the inter-tile data token transmission delay quantified by process $p_{\Delta,\tau_{NoC}}$ is predictable, with

$$\tau_{NoC} = \left\lceil \frac{\frac{m_{p_\Delta} \cdot Sz_{token}}{w_{NoC}} + t_{hop}}{clk_C} \right\rceil \tag{3.5}$$

in which m_{p_Δ} is the input rate of $p_{\Delta,\tau_{NoC}}$, w_{NoC} the reserved NoC bandwidth and t_{hop} the network hop delay.

3.5.2 Design objective: energy efficiency

The architectural design options on the processor *voltage-frequency* levels are customized by the domain interface configurations, and memory *size*s by the parameterized FIFOs. For each instance of the implementation model, the processes computation timing and energy properties are profiled by the cycle-accurate processor energy simulator Sim-Panalyzer [4] and the memory simulator Cacti [99]. As shown in Figure 3.9, Sim-Panalyzer is configured with certain *voltage-frequency* levels, and provides the computation timing and dynamic energy dissipation for the gcc cross-compiled binary of each process. It is assumed that the static power of the processor during customization is constant and is ignored in calculation. Meanwhile, the memory accessing patterns are profiled in an execution trace file, based on the static and dynamic energy dissipations of the specified memory estimated with Cacti.

With each process computation latency resolved from the computation timing profiles and each communication delay from the guaranteed bandwidth NoC communication logic, the timing of each process on the implementation model is specified. While the processes bounded on the same tile are scheduled sequentially, the application self-timed schedule is determined by its scheduling state. The same

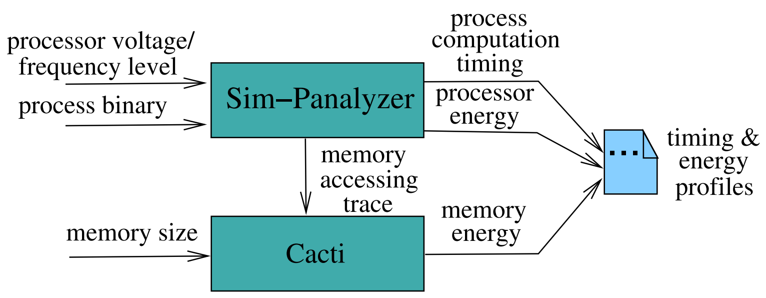


Figure 3.9: Process timing and energy profiling.

techniques as introduced in Section 3.4.4 are used for the resource-aware application throughput analysis.

Within a given throughput requirement, the amount of stream data transmitted via the NoC communication is fixed during some period of time. In [49], the *bit energy* model reveals that the average energy consumption to send one bit on NoC is proportional to the Manhattan distance between two tiles. From this metric, the NoC energy dissipation for streaming applications with guaranteed throughput is static, since a static mapping strategy is used. Thus, the NoC energy dissipation is not counted in the system energy analysis.

The process and memory energy dissipations are estimated for each time slot in a state based way. When all the processes mapped onto a processor μp_y are stalled, the processor is in $idle$ mode and only consumes the static energy $\dot{E}_{\mu p_y}$ (to be ignored as mentioned), so does the local memory mem_y consumes $\dot{E}_{mem_y}$; otherwise, the processor and memory are in $active$ mode, they also consume the dynamic energy ($\hat{E}_{\mu p_y}$ and $\hat{E}_{mem_y}$) of the $executing$ process profiled by Sim-Panalyzer and Cacti.

While satisfying a given application throughput requirement $Thru_0$ upon the sink process p_{sink}, the aim is to minimize the overall system energy E_{Sum}, which is the energy summation of all the processor and memory modules. The design objective and constraints are formalized as the following:

$$\begin{aligned}\min \quad E_{Sum} = & \sum_{\forall \mu p_y \in \mathbf{U}} \hat{E}_{\mu p_y} \\ & + \sum_{\forall mem_y \in \mathbf{M}} (\hat{E}_{mem_y} + \dot{E}_{mem_y}) \qquad (3.6)\end{aligned}$$

$$\begin{aligned} where \quad & \hat{E}_{\mu p_y} = \sum_{\forall p_y \in \mathbf{P}} (t_x \cdot r_x \cdot \hat{\mathcal{P}}_{\mu p_y, p_x}(v_y^2, f_y) \cdot \varphi_{p_x, \mu p_y}) \\ & \hat{E}_{mem_y} = \hat{\mathcal{E}}_{mem_y}(\sum_{\forall FIFO_x \in \mathbf{F}} (\gamma_x \cdot \varphi_{FIFO_x, mem_y})) \\ & \dot{E}_{mem_y} = g_{(\Delta)} \cdot \dot{\mathcal{P}}_{mem_y}(\sum_{\forall FIFO_x \in \mathbf{F}} (\gamma_x \cdot \varphi_{FIFO_x, mem_y})) \end{aligned}$$

$$\begin{aligned} \text{subject to} \quad & Thru(p_{sink}) \geq Thru_0 \\ & \sum_{\forall \mu p_y \in \mathbf{U}} \varphi_{p_x, \mu p_y} = 1, \qquad \forall \varphi_{p_x, \mu p_y} \in \{0, 1\} \\ & \sum_{\forall mem_y \in \mathbf{M}} \varphi_{FIFO_x, mem_y} = 1, \ \forall \varphi_{FIFO_x, mem_y} \in \{0, 1\} \end{aligned}$$

in which $\varphi_{p_x, \mu p_y}$ and φ_{FIFO_x, mem_y} are the decision variables (equals to 0 or 1) to determine the mapping from **P** onto **U** and **F** onto **M**, $\dot{\mathcal{P}}_{mem_y}$ the *static* power function of the specified mem_y, and $g_{(\Delta)}$ the length of the periodic phase.

The design space exploration has the complexity $\mathcal{O}(n^{|U|+|F|})$, which is NP-hard regarding the problem size. Thus, the widely used heuristic algorithms (i.e., greedy and Taboo [25] search) are used for the design options optimization.

3.6 Experimental results

To evaluate the potential of the proposed design flow in energy efficiency design, experiments have been applied on a software FM radio application (presented in [41]) on a NoC based 4×4 mesh tiles MPSoC.

The application model has 44 concurrent processes, which are clustered into 14 partitions in a five-stage pipeline, as illustrated on the left side of Figure 3.10. The pipeline consists of a signal source, a low-pass filter (LPF), a demodulator (Dem), an equalizer (Equ) with 10 children modules over a range of frequencies, and a signal sink. The arrows marked with numbers show the logic connections and data rates between different partitions.

With each partition allocated to one tile, an empirical mapping from the clustered application to the 14 tiles (the other two are left unused) is shown in Figure 3.10. Each pipeline stage of the application is modeled in one individual synchronous clock domain, so is the network logic. The dashed-lines between each pipeline stage stand for the implicit NoC communication and domain interfaces. With the static clock abstraction in the network communication domain, it is taken

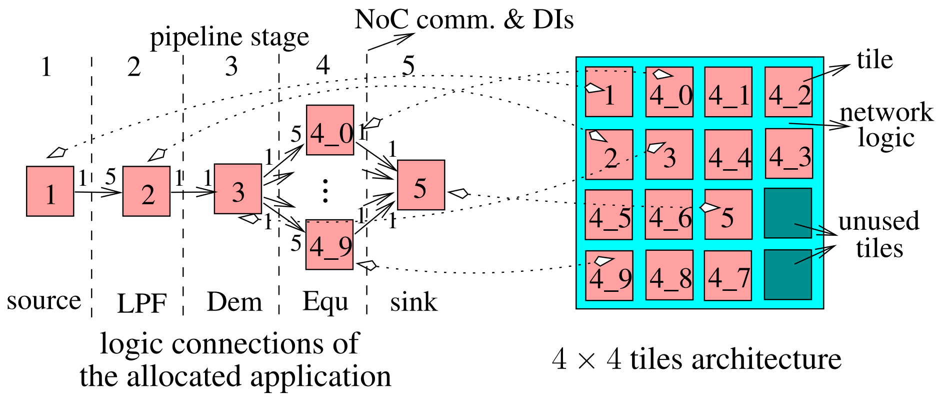


Figure 3.10: FM radio application mapped onto 4×4 tiles.

Table 3.1: Voltage-Frequency levels.

Voltage(V)	2.5	1.8	1.4	1.2
Frequency(MHZ)	233	180	140	100

as the logic mold domain in the multi-clocked synchronous model. To decrease the problem size, symmetrical on-tile resources and NoC communication configurations are used for the 10 children modules in stage 4.

Each on-tile processor is a StrongARM SA-1100 with the customizable *voltage-frequency* levels shown in Table 3.1, and each local memory is SRAM with the *size* given by the FIFO buffers to be implemented on this tile. The C program task of each process is compiled into binary using the ARM gcc toolchain and used as the input of Sim-Panalyzer and Cacti for architectural energy estimations. To make the energy dissipations estimated from Sim-Panalyzer and Cacti consistent, the same .18μm technology node is used in the configurations for both simulators.

In addition, as the energy results for each customizable module is independent of each other, they could be simulated off-line in a linear time proportional to the problem size, and saved in a look-up table for design exploration.

At the application sink process, a unit data token is a compound data type containing 512 32-bit values. According to the sink process throughput, design optimizations have been performed based on three different application workloads. They have the required application throughput $640\,\mathrm{kb/s}$ (*Thru-1*), $533\,\mathrm{kb/s}$ (*Thru-2*) and $400\,\mathrm{kb/s}$ (*Thru-3*) respectively.

Table 3.2: Experimental results.

	Thru-1					*Thru-2*					*Thru-3*				
	E_U[a]	E_M	E_{Sum}	δ[%]	η[%]	E_U	E_M	E_{Sum}	δ[%]	η[%]	E_U	E_M	E_{Sum}	δ[%]	η[%]
Ad-hoc	28.1	181.2	209.2	-	37.5	31.0	174.4	205.4	-	32.0	34.1	174.5	208.6	-	28.5
Min-memory	33.9	152.3	186.2	11.0	33.0	32.9	152.2	185.2	9.8	29.7	34.1	152.4	186.5	10.6	27.0
Greedy	9.3	159.0	168.3	19.6	61.5	9.8	158.8	168.6	17.9	53.0	8.2	155.8	163.9	21.4	47.5
Taboo	10.3	155.6	165.8	20.7	55.3	12.8	152.4	165.3	19.5	40.7	10.9	152.6	163.5	21.6	38.2

[a]The unit of energy is mJ.

As the baseline, an *Ad-hoc* design method[8] is served as the reference. The minimal memory requirements (*Min-Memory*) technique in [87] does not have constraints on the energy consumption of the whole, but only optimize the design in memory usage. Both heuristic *Greedy* and *Taboo* [25] search are implemented in the design exploration flow for design optimizations based on customizable voltage-frequency levels and memory sizes. The termination criteria of both methods are a specified number of iterations, e.g., 10 iterations, during which the objective function Eq. 3.6 has no improvements on its value.

Within all the workloads, the computation schedules on multiple processors achieved by the design options with different methods are shown in Figure 3.11. Being aware of the global timing, the schedules modeled in different clock domains are elaborated on a normalized time axis. Each processor μp_i corresponds to the processor on the tile marked with i in Figure 3.10. In addition, the computations on all the μp_{4_n} $(0 \leqslant n \leqslant 9)$ in the pipeline stage 4 are symmetrical and have the same scheduling behaviors. Only one sample is used in this stage to evaluate the system energy and processor computation efficiency. All the design options meet the application throughput requirements. The heavier the workload is, the denser the periodic execution pattern is. The design options, which require the processors to run at a relatively lower frequency, get the higher computation efficiency(the average $executing$ ratio in the processor schedules), as shown in the η columns of Table 3.2.

The results of the experiments, to deliver a specified amount of application output data in the periodic schedules, are summarized in Table 3.2, in which δ

[8]From 100 randomly selected design choices with satisfied application throughput, the one with median energy consumption is chosen as the *Ad-hoc* design option, similar to the approach Hu et al. [49] adopts.

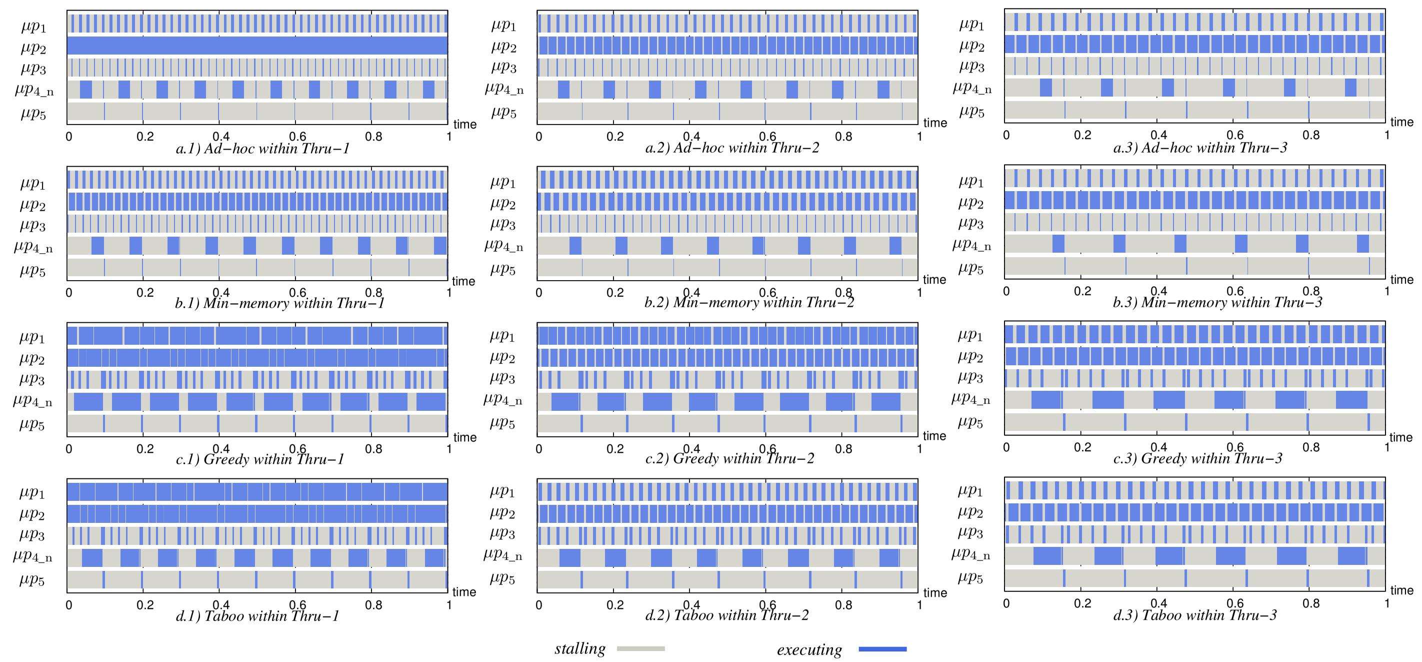


Figure 3.11: Periodic schedules of the FM radio application on multiple processors, in which the three workloads considered are *Thru-1*=640 kb/s, *Thru-2*=533 kb/s and *Thru-3*=400 kb/s. The schedule of μp_{4_n} $(0 \leqslant n \leqslant 9)$ represents the symmetrical computations in the pipeline stage 4.

denotes the energy saving ratio achieved by each other design method according to the *Ad-hoc* approach. In all the experiments, the heuristics methods get the optimized design option in 10^3 searches, which is trivial compared to the searching space, i.e., $> 5 \times 10^9$. Analyzing the experiments the following results can be derived:

1. With minimal memory usage, the *Min-Memory* method consumes the smallest memory energy E_M. However, it ignores the customizability of processors, and achieves only a limited energy saving at around 10%.

2. Higher computation efficiency η does not always mean higher system energy efficiency E_{Sum}. Instead, an energy efficient design assigns the minimum energy budget on both processor and memory modules with a properly pipelined parallelism. For example, in workload *Thru-2* of Table 3.2 *Taboo* gets the η of 40.7%, which is lower than 53.0% of *Greedy*, but its design option is more energy efficient in E_{Sum} (3.3 mJ less).

3. The proposed design flow with *Greedy* search shows the energy savings δ at around 19%. Furthermore, using the proposed design flow with *Taboo* search, which could escape from the local optima in the searching space, even better solutions at 21% can be found.

It is concluded that the proposed design flow with the *Taboo* search takes the advantage of the customizable computation and storage modules, and can be used to design energy efficient streaming applications with guaranteed throughput on MPSoCs.

3.7 Concluding remarks

This chapter proposes an energy efficient design exploration flow for streaming applications with guaranteed throughput on MPSoCs. Both application throughput analysis and system energy calculation have been carried out on a multi-clocked synchronous MoC framework. Instead of only analyzing the memory efficiencies or processor utilizations, the design intent is to minimize the overall energy cost. The degrees of customizability of both processor *voltage-frequency* levels and memory *size*s have been leveraged to investigate the minimal energy consumption of streaming applications. High system energy efficiency is achieved without losing throughput guarantees, as illustrated in experiments.

The investigations suggest that focusing on either best memory efficiencies or processor utilizations is more likely to result in less optimized implementations. Using the heuristic Taboo search, better solutions in terms of total energy cost can be found, which is also supported by experiments.

CHAPTER 4

CONSTRAINT BASED SCHEDULING WITH BUFFER MINIMIZATION

The current trend toward systems-on-chip (SoCs) consisting of several modules of processor, custom circuit and memory (e.g., the hybrid CPU/FPGA architecture [5]) makes the global analysis of heterogeneous software/hardware (SW/HW) systems essential. Streaming applications based on synchronous data flow (SDF) model has been widely used to model and analyze streaming applications on single-/multi-processors [60, 61].

To map the tutorial example application model (see Figure 2.1) onto the hybrid CPU/FPGA architecture below, as illustrated in Figure 4.1, process p_i and p_k are implemented as SW and scheduled sequentially by the real-time operating system (RTOS) on the same CPU, and process p_j is implemented as HW custom circuit. The computation modules (both SW and HW) communicate via dedicated FIFO channels for data buffering. Needless to mention, the general architecture platform considered may have multiple CPU or custom circuit modules.

Due to the static nature of SDF models, sophisticated algorithms can be used to compute optimized schedules at compile-time. This chapter proposes a constraint based scheduling methodology for real-time streaming applications with minimal buffer requirements on such a hybrid SW/HW platform.

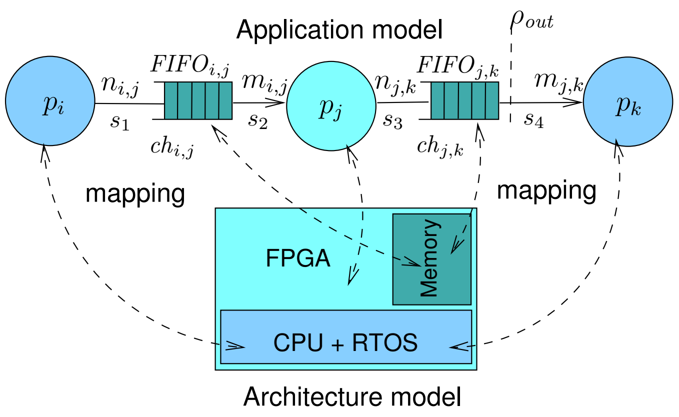


Figure 4.1: The example application (see Figure 2.1) mapped onto a hybrid CPU/FPGA architecture.

4.1 Related work

Lee and Messerschmitt [60] present techniques to construct periodic admissible sequential schedules (PASS) on single-processors or periodic admissible parallel schedules (PAPS) on multi-processors. Later, Bhattacharyya et al. [12] have taken buffer minimization into consideration using heuristics in PASS (but not PAPS) construction.

To provide timing guarantees, Govindarajan et al. [42] and Stuijk et al. [87] exploit a timed-SDF model[1]. Govindarajan et al. address the buffer minimization of SDF applications to obtain schedules with maximal throughput. Furthermore, Stuijk et al. investigate the buffer minimization of applications upon different specified throughput requirements. Nevertheless, neither of the techniques considers the computation resource constraints when multiple processes mapped onto a single processing element (processor), and can handle the global optimization of both sequential SW (RTOS) and parallel HW scheduling on a hybrid CPU/FPGA architecture.

The proposed constraint formulation in this chapter is close in spirit to the work in [42], in which the authors establish their linear constraints based on process start execution time and data dependency. However, they relax the integer constraints on buffer sizes (the integer formulation is NP-complete) in implementation to utilize

[1]To analyze timing related properties, the timed-SDF model is a timed extension to the regular untimed SDF model in [60].

the efficiency of linear programming. It is argued to dimension the correct *minimal* buffer requirement. Furthermore, they use heuristics to estimate buffer requirement before a valid schedule is found, and need iterations to finalize the scheduling. In this chapter, the proposed constraint based scheduling framework constructs the event models as cumulative functions on the number of tokens in data streams, similar as in [23]. The streaming application execution semantics on a limited number (less than the number of processes) of processing resources and design concerns on throughput requirement, buffer properties and scheduling decisions are globally captured as a constraint satisfaction problem with composable constraints. The constraint solver (Gecode) is utilized to solve the NP-complete scheduling problem [31], without the relaxation of integer variables.

In [67], Madsen et al. validate the sanity of several existing scheduling policies (rate monotonic and earliest deadline first) of the multi-processor RTOS on a SystemC model. However, a systematic way to explore and find out an optimal schedule according to the required throughput is still an open issue.

The author aims to provide buffer minimized schedules for real-time SDF streaming applications on hybrid CPU/FPGA architectures. Different from all the previous work mentioned, this chapter focuses on describing the constraint based scheduling problems in a declarative way, and apply the existing successful constraint optimization techniques [32] for problem solving. The optimal schedules found have preserved sanity and minimized buffer requirement.

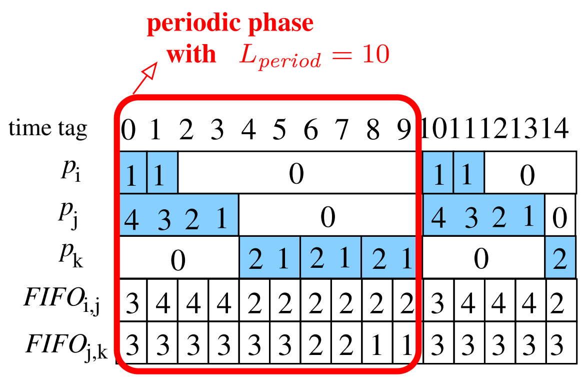
4.2 Motivation

The work in this chapter is motivated by several schedules with varying buffer requirements and throughput guarantees.

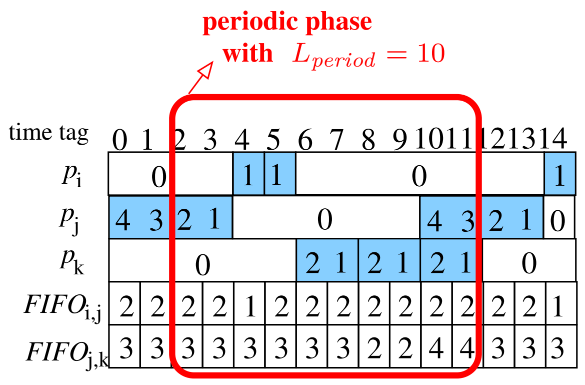
4.2.1 Schedules with varying buffer requirements

Here, the example application in Figure 4.1 is instantiated with computation latency list $T = [1, 4, 2]$ and process input/output token numbers $n_{i,j} = 1$, $m_{i,j} = 2$, $n_{j,k} = 3$ and $m_{j,k} = 1$. Thus, the process repetition vector (see Section 2.1.1) is $< r_i, r_j, r_k >=< 2, 1, 3 >$ for the instance of the example application. It is assumed that there are some initial tokens in buffers $FIFO_{i,j}$ and $FIFO_{j,k}$, which are denoted as $B^0_{i,j} = 2$ and $B^0_{j,k} = 0$ respectively.

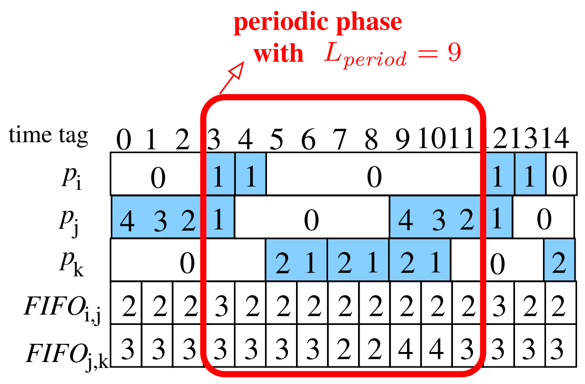
Using the application to architecture mapping as shown in Figure 4.1, three valid periodic schedules are illustrated in Figure 4.2. Figure 4.2a is the PAPS [60]


periodic phase with $L_{period} = 10$
time tag 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
p_i 1 1 0 1 1 0
p_j 4 3 2 1 0 4 3 2 1 0
p_k 0 2 1 2 1 2 1 0 2
$FIFO_{i,j}$ 3 4 4 4 2 2 2 2 2 2 3 4 4 4 2
$FIFO_{j,k}$ 3 3 3 3 3 3 2 2 1 1 3 3 3 3 3

(a) PAPS with $\rho_{out} = \frac{3}{10}$


periodic phase with $L_{period} = 10$
time tag 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
p_i 0 1 1 0 1
p_j 4 3 2 1 0 4 3 2 1 0
p_k 0 2 1 2 1 2 1 0
$FIFO_{i,j}$ 2 2 2 2 1 2 2 2 2 2 2 2 2 2 1
$FIFO_{j,k}$ 3 3 3 3 3 3 3 3 2 2 4 4 3 3 3

(b) A minimal buffer schedule with $\rho_{out} = \frac{3}{10}$


periodic phase with $L_{period} = 9$
time tag 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
p_i 0 1 1 0 1 1 0
p_j 4 3 2 1 0 4 3 2 1 0
p_k 0 2 1 2 1 2 1 0 2
$FIFO_{i,j}$ 2 2 2 3 2 2 2 2 2 2 2 2 3 2 2
$FIFO_{j,k}$ 3 3 3 3 3 3 3 2 2 4 4 3 3 3 3

(c) A minimal buffer schedule with $\rho_{out} = \frac{3}{9}$

Figure 4.2: Comparison of different schedules of an instance of the example application, in which p_i and p_k are mapped onto the same processor and can only be scheduled sequentially.

with unroll factor[2] $J = 1$. As the schedule advances to time tag 10, the application encounters the same status lists as at time tag 0 (i.e., $T'_{10} = T'_0$ and $\Gamma'_{10} = \Gamma'_0$), and enters a periodic phase. The periodic phase has length $L_{period} = 10$, in which the sink process p_k always runs 3 times. Consequently, the schedule guarantees an average output throughput $\rho_{out} = \frac{3 \cdot m_{j,k}}{L_{period}} = \frac{3}{10}$ at process p_j and requires buffer storage $\Gamma = [4, 3]$, which are the maximum buffer usages at each FIFO.

However, a periodic parallel schedule (not the PAPS in [60]) with minimized buffer (i.e., $\Gamma = [2, 4]$), which guarantees the same throughput $\rho_{out} = \frac{3}{10}$, does exist in Figure 4.2b. Furthermore, a schedule with 10% higher throughput guarantee $\rho_{out} = \frac{3}{9}$ can still be achieved in Figure 4.2c, which has the same buffer cost $\Gamma = [3, 4]$.

Although the application throughput can be improved by increasing J in PAPS, the implementation cost of the periodic phase and buffer requirement both increase accordingly (see the case study in Section 4.6), and a systematic way is still lacking [60]. This chapter intends to construct optimal schedules systematically with minimal buffer requirement and higher throughput guarantees.

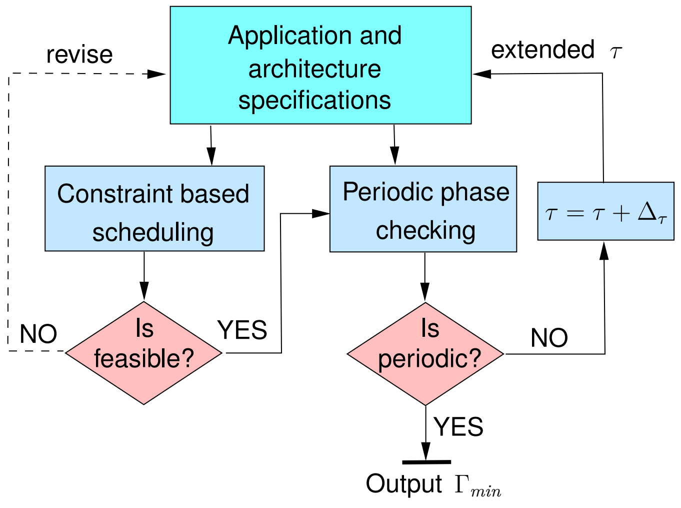


Figure 4.3: Generic constraint based scheduling work flow.

[2]The iteration number of the repetitive vector patterns.

4.3 Work flow

In Figure 4.3, a generic scheduling work flow is proposed. The work flow inputs are the application and architecture specifications, e.g., the application model, specified application and architecture mapping, required throughput, an initial time period τ considered for scheduling. It is generic from the sense that it can be used for different design purposes with customizable constraints and objectives.

The work flow can be described as follows.

Step 1: When the constraint based scheduling problem is feasible, a pending schedule with minimized buffer is got; otherwise, the specifications need to be revised (which is out of the scope of this thesis).

Step 2: The throughput guarantees are checked for the pending schedule (whether a periodic phase could be found). If the throughput guarantee or maximum execution time is met, it stops and outputs the valid schedule with minimal buffer sizes Γ_{min}; otherwise, it increases the considered τ with Δ_τ and goes back to **Step 1**.

Apparently, only when the throughput guarantees are met in Step 2, the output results are valid. The initial values of τ and Δ_τ are application dependent and are given empirically. In this chapter, this work flow is adopted for streaming applications scheduling with minimal buffer and throughput guarantees.

4.4 Streaming application scheduling

In this section, the event models for streaming data flows are illustrated. Subsequently, constraint based scheduling problems are formalized. The time tag t in the following formulation is discrete numbers with $t \in \mathbb{N}_0$, when it is not otherwise clarified.

4.4.1 Event models

The data stream s is captured as a timed indexed synchronous signal as defined in Section 3.4.1. The event models are constructed as cumulative functions on the number of tokens in data steams, i.e., only the number information of the symbolic tokens is considered in performance analysis but not their values.

Without loss of generality, the input/output workloads of each communication channel and the processing capabilities of the computation processes are characterized based on the example application in Figure 4.1 as follows.

Definition 1. *(Arrival function) The arrival function $R_{i,j}(t)$ of the communication channel $ch_{i,j}$ is defined as the sum of tokens arriving from the input data stream during the time interval* $[0, t], t \in \mathbb{N}_0$.

For instance, $R_{i,j}(t) = \sum_0^t s_1$ in Figure 4.1.

Definition 2. *(Output function) The output function $R'_{i,j}(t)$ from process p_i to the communication channel $ch_{i,j}$ equals to the arrival function $R_{i,j}(t)$ of $ch_{i,j}$.*

For instance, $R'_{i,j}(t) = \sum_0^t s_1 = R_{i,j}(t)$ in Figure 4.1. This equivalence forms the basis of compositional analysis for cascaded processes.

Definition 3. *(Service function) The service function $C_{i,j}(t)$ of the communication channel $ch_{i,j}$ by process p_j is defined as the sum of tokens served and removed from the buffer FIFO$_{i,j}$ via the data stream by p_j during the time interval* $[0, t], t \in \mathbb{N}_0$.

For instance, $C_{i,j}(t) = \sum_0^t s_2$ in Figure 4.1.

4.4.2 Buffer properties

While a process is executing, the extra buffer space reservation in scheduling test (see the schedule in Section 2.1.2) can be modelled with the *demand function*:

Definition 4. *(Demand function) The demand function $D_{i,j}(t)$ of the communication channel $ch_{i,j}$ is defined as the sum of $R'_{i,j}(t)$ and the demanding space $d_{i,j}(t)$ at time tag t on FIFO$_{i,j}$ from the producer process p_i, i.e., $D_{i,j}(t) = R'_{i,j}(t) + d_{i,j}(t), d_{i,j}(t) \in \{0, n_{i,j}\}$.*

For instance,

$$D_{i,j}(t) = \begin{cases} \sum_0^t s_1 + n_{i,j} & \text{if } p_i \text{ is executing} \\ \sum_0^t s_1 & \text{if } p_i \text{ is stalling} \end{cases}$$

in Figure 4.1.

A graphical interpretation of the definitions of $R_{i,j}(t)$, $C_{i,j}(t)$ and $D_{i,j}(t)$ is illustrated in Figure 4.4, which is consistent with the schedule in Figure 4.2a. $R'_{i,j}(t)$ is ignored for its equivalence to $R_{i,j}(t)$ (see Definition 2).

Consequently, the following buffer properties can be derived.

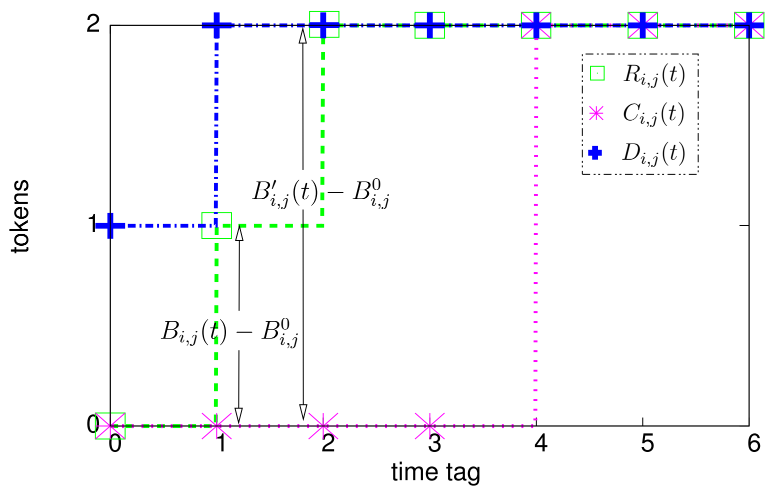


Figure 4.4: Cumulative functions and buffer properties for the PAPS in Figure 4.2a.

Property 1. *(Backlog) The backlog $B_{i,j}(t)$ (tokens arrived but not yet served) in buffer FIFO$_{i,j}$ is the vertical distance between $R_{i,j}(t)$ and $C_{i,j}(t)$ plus an offset of the initial buffer tokens $B^0_{i,j}$ at time tag 0.*

$$B_{i,j}(t) = R_{i,j}(t) - C_{i,j}(t) + B^0_{i,j}, \quad \forall t \in \mathbb{N}_0 \tag{4.1}$$

Property 2. *(Buffer usage) In scheduling, the buffer space in use $B'_{i,j}(t)$ for FIFO$_{i,j}$ (equals to $B_{i,j}(t)+d_{i,j}(t)$) is the vertical distance between $D_{i,j}(t)$ and $C_{i,j}(t)$ plus an offset of the initial buffer tokens $B^0_{i,j}$ at time tag 0.*

$$B'_{i,j}(t) = D_{i,j}(t) - C_{i,j}(t) + B^0_{i,j}, \quad \forall t \in \mathbb{N}_0 \tag{4.2}$$

4.4.3 Streaming application execution semantics

Based on the definitions and properties above, a full list of constraints to formalize the execution semantics of SDF streaming applications are listed out as follows. These constraints hold during the whole life-time of streaming applications ($\forall t \in \mathbb{N}_0$). Meanwhile, the designers can specify some specification dependent parameters, e.g., the initial (tokens) offset $B^0_{i,j}$ for $B'_{i,j}$.

Constraint 1. *(Token ratios) For process p_j, the static input and output token rate ratios can be formalized by $R'_{j,k}(t)$ and $C_{i,j}(t)$ as the following.*

$$R'_{j,k}(t) \cdot m_{i,j} = C_{i,j}(t) \cdot n_{j,k} \tag{4.3}$$

Constraint 2. *(Computation latency) Process p_j has computation latency $t_{C,j}$ in each execution instance.*

$$\begin{aligned} C_{i,j}(t+t_{C,j}) - C_{i,j}(t) &= m_{i,j} \cdot K_j(t+t_{C,j}) && (4.4)\\ D_{j,k}(t+t_{C,j}) - D_{j,k}(t) &= n_{j,k} \cdot K_j(t+t_{C,j}) && (4.5)\\ \textit{where} \quad K_j(t+t_{C,j}) &\in \{0,1\} \end{aligned}$$

in which $K_j(t+t_{C,j})$ denotes the incremental properties of $C_{i,j}(t+t_{C,j})$ and $D_{j,k}(t+t_{C,j})$, i.e., $K_j(t+t_{C,j})$ has value '1' if process p_j finishes one instance of execution exactly at time tag $t+t_{C,j}$, otherwise it has value '0'.

Constraint 3. *(Space reservation) In the communication channel $ch_{j,k}$, the demand function of process p_j reserves vacant space $t_{C,j}$ slots in advance, which corresponds to the semantics that the process can only execute when there are enough space in output-size FIFO(s).*

$$D_{j,k}(t) = R_{j,k}(t+t_{C,j}) \tag{4.6}$$

Constraint 4. *(Asynchronous buffer) The incoming tokens in buffer FIFO$_{i,j}$ takes at least $t_{C,j}$ slots to be served by process p_j, which models the buffering behavior determined by the consumer process(es).*

$$R_{i,j}(t) - C_{i,j}(t+\Delta_t) \geqslant 0, \quad \forall \Delta_t \in [1, t_{C,j}] \tag{4.7}$$

Constraint 5. *(Buffer requirement) The buffer size $\gamma_{i,j}$ of buffer FIFO$_{i,j}$ meets the maximum buffer space requirement, which guarantees a conservative buffer dimensioning.*

$$\gamma_{i,j} \geqslant B'_{i,j}(t) \tag{4.8}$$

4.4.4 Resource limits and throughput requisites

Instead of describing the actual algorithms (the how) used to find the solution, the declarative method in this chapter formalizes the properties (the what) of the desired solution as constraints.

Constraint 6. *(Sequential execution) In a set of processes $\boldsymbol{P}_a$ mapped onto the same processor CPU$_a$, at any time at most one can execute (sequentially) accord-*

ing to:

$$\sum_{p_j \in \boldsymbol{P}_a} W_j(t) \in \{0,1\}. \quad \forall t \in \mathbb{N}_0 \tag{4.9}$$

$$\textit{where} \quad W_j(t) = max(L_{i,j}(t), L_{i,j}(t+\Delta_t)),\ \forall \Delta_t \in [1, t_{C,j}]$$

$$L_{i,j}(t+1) = \frac{C_{i,j}(t+1) - C_{i,j}(t)}{m_{i,j}} \in \{0,1\}$$

in which $W_j(t)$ denotes the computing or stalling 1-0 *status of process p_j (different from the fine-grained process status exemplified in Figure 4.2), $L_{i,j}(t+1)$ denotes the incremental properties (step) of the service function $C_{i,j}(t+1)$.*

In scheduling, a predetermined application throughput needs to be met and guaranteed, as formalized in Constraint 7 and 8.

Constraint 7. *(Application output throughput) After a transient phase τ_0 ($\tau_0 > 0$) with no stable output tokens, a specified output throughput ρ_{out} should be sustained at the application sink process p_j, which is guaranteed by a periodic phase (see fig 5.1) with length L_{period}.*

$$C_{j,k}(\tau_0 + c \cdot L_{period}) - C_{j,k}(\tau_0) = c \cdot J \cdot r_k \cdot m_{j,k}, \tag{4.10}$$

$$L_{period} = \lceil \frac{J \cdot r_k \cdot m_{j,k}}{\rho_{out}} \rceil, J \in \mathbb{N} \backslash \{\infty\}, c \in \mathbb{N}_0$$

in which c specifies the iteration number of periodic phases, r_k is the repetition firing rate *of process p_k (see Eq. 3.1), and J (called unroll factor) denotes how many cycles processes fire as stated in the repetition vector in one periodic phase.*

For a consistent SDF streaming application, a periodic phase in its schedule always exists (see Section 3.4.4). The required application throughput is guaranteed by the periodic properties during this period. However, the problem to determine the length of L_{period} which can provide optimal buffer cost in this formulation is NP-complete itself. Its length is determined by a designer specified J, and an incremental q leads to an increasing L_{period}.

Constraint 8. *(Periodic phase) The repeated process and FIFO status at time tag t' and $t' + L_{period}$ determines a periodic phase between them with length L_{period}.*

$$B'_{i,j}(t') = B'_{i,j}(t' + L_{period}), \quad \forall FIFO_{i,j} \tag{4.11}$$

$$W'_j(t') = W'_j(t' + L_{period}), \quad \forall p_j \tag{4.12}$$

$$\textit{where} \quad W'_j(t') = \sum_{k=1}^{t_{C,j}} k \cdot \frac{C_{i,j}(t'+k) - C_{i,j}(t'+k-1)}{m_{i,j}}$$

in which variables $W'_j(t')$ and $W'_j(t' + L_{period})$ are process status (denoted as numbers for each process in Figure 4.2).

In implementation, Constraint 7 and 8 can only guarantee that the buffer cost is minimized (optimal) using the given length of L_{period}. On the other hand, the length of L_{period} can be pre-specified (if validly) by the designer, which corresponds to the cost to implement a periodic static schedule.

The **scheduling objective** is to find the minimal total buffer sizes, subject to the corresponding Constraint 1 - 8, as the following.

$$\min : \gamma_{Sum} = \sum_{\forall FIFO_{i,j} \in \mathbf{F}} \gamma_{i,j} \tag{4.13}$$

in which **F** is the set of the buffers being considered and $\gamma_{i,j}$ is the size of buffer $FIFO_{i,j}$.

4.4.5 Extension to MIMO and cyclic models

The proposed formulation does not take into account *models with multiple input and multiple output (MIMO) processes* or *cyclic models*. However, the extension of the proposed reconfiguration analysis methodology to such models is intuitive. Without loss of generality, the MIMO process p_j in Figure 4.5[3] is used for illustration.

For MI channels $ch_{i,j}$ and $ch_{l,j}$ of process p_j, the service functions are associated with each other caused by the static input data token rate $m_{i,j}$ and $m_{l,j}$ of the same consumer process p_j. They have the linear relations as follows.

[3]For clarity in the graph, the FIFO modules on communication channels are omitted. Also, a number of dots are used to denote the initial buffer token numbers.

Constraint 9. *(MI linear relation)*

$$\frac{C_{i,j}(t)}{m_{i,j}} = \frac{C_{l,j}(t)}{m_{l,j}} \tag{4.14}$$

Similarly, the MO channels $ch_{j,k}$ and $ch_{j,l}$ have the linear relations on the output and demand functions according to output data token rate of the same producer process p_j as follows.

Constraint 10. *(MO linear relation)*

$$\frac{R'_{j,k}(t)}{n_{j,k}} = \frac{R'_{j,l}(t)}{n_{j,l}}, \frac{D_{j,k}(t)}{n_{j,k}} = \frac{D_{j,l}(t)}{n_{j,l}} \tag{4.15}$$

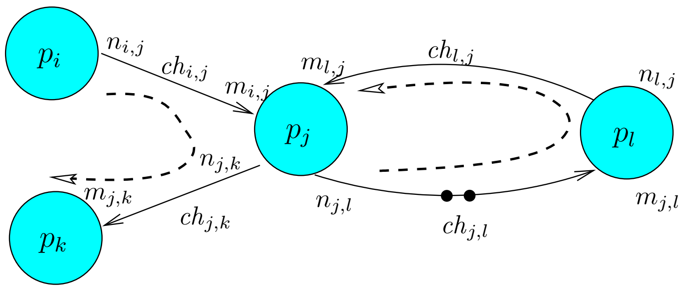


Figure 4.5: A cyclic MIMO application model.

A MIMO model can be analyzed by traversing it with a set of paths, where each path is a sequence of communication channels such that the output channels of a process always succeed its input channels.

A set of paths are complete only when all the communication channels are covered. For instance, the paths ("$ch_{i,j} \to ch_{j,k}$" and "$ch_{j,l} \to ch_{l,j}$") in dashed lines complete the MIMO model in Figure 4.5. Based on the complete set of paths, the scheduling methodology fits the MIMO application models well.

For directed cyclic graphs as shown in Figure 4.5, the data tokens required for loop initialization can be explicitly captured as the initial token offsets in Eq. 4.1 and 4.2. For instance, two initial tokens of communication channel $ch_{j,l}$ are denoted as $B^0_{j,l} = 2$ and the actual backlog of $ch_{j,l}$ is $B_{j,l}(t) + B^0_{j,l}$.

4.5 Constraint programming techniques

Inspired by the success of solving NP-complete problems with constraint programming techniques, the proposed constraint base scheduling framework has been implemented on a public domain constraint solver *Gecode*[4] [32], which is a C++ library.

Both the SDF execution semantics and the scheduling problem on the hybrid multiprocessor/FPGA are encoded. Especially, some modeling techniques have been conducted to improve the computation efficiency in solutions finding:

- **Domain and constraints reduction.** A lower bound for any FIFO [36] can be computed as

$$n_{i,j} + m_{i,j} - gcd(n_{i,j}, m_{i,j}) + B^0_{i,j} \; mod \; gcd(n_{i,j}, m_{i,j}) \tag{4.16}$$

  to prune infeasible (dead-lock) variable value domains. Furthermore, although $t \in \mathbb{N}_0$ holds for all timing related constraints, we only implement and evaluate them in time period $[0, \tau_0 + L_{period}]$ once periodic phase is found.

- **Branching and exploration.** To construct the search tree, the branching variables are prioritized as follows: γ_{Sum}, $\gamma_{i,j}$, and $C_{i,j}$. Empirically, the exploration starts with minimal values for all variables, which also guarantees that the first solution found has minimized buffer cost γ_{Sum}.

4.6 Case study

To evaluate the potential of the proposed methodology, an application of voice-band data modem [61], which has 9 processes and 11 FIFOs, is used in the case study. The application model with customized specification parameters is illustrated in Figure 4.6.

The experiments start from a manual mapping from the application to a multi-processor CPU/FPGA architecture. The labeled processes in the application model are partitioned and mapped onto multi-processors, i.e., p_0 and p_2 mapped onto CPU_1, and p_3 and p_5 mapped onto CPU_2. The rest processes are mapped onto parallel executing custom hardware.

[4]In the left chapters, *Gecode* has been used as the constraint solver as well.

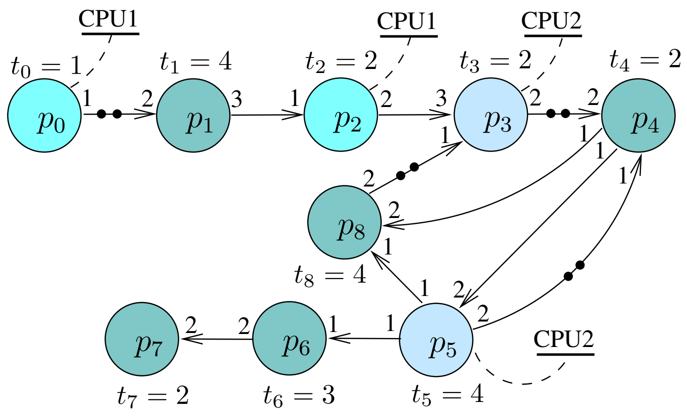


Figure 4.6: The modem application partitioned for a multiprocessor/FPGA architecture.

The proposed Constraint based Minimal Buffer Scheduling (CMBS) methodology is implemented on the public domain constraint solving toolkit *Gecode* [32], which is a library written in C++. CMBS is compared with the reference scheduling method, a trivially customized PAPS [5]. To make this comparison more reasonable, the reference PAPS method is implemented in C++ as well and run both methods on the workstation for experiments (see Section 1.3) to solve the scheduling problems for the modem application. However, both the implementations of the reference PAPS and the constraint based scheduling on *Gecode* solver are not parallelized, only one core is actually utilized.

The experimental results are shown in Table 4.1, which compares and quantifies the buffer requirement and the experimental execution time achieved by the schedules using PAPS and CMBS.

For PAPS, the application throughput may be improved by increasing the unroll factor J, i.e., the cases from $\#1$ to $\#6$. Correspondingly, the results of CMBS are reported with some competitive throughput (no less than in PAPS).

However, the implementation cost of the periodic schedule increases with higher J and it is still in lack of a systematic way to find a finite J [60] yielding an optimal schedule. J is simply increased by 1 each time until the throughput improvement is negligible (i.e., less than 1e-4 to the throughput at $J = 1$), which is the ‘optimal’

[5]Instead of each process being scheduled onto any of the computation resource, a fixed mapping is adopted, i.e., a process can only be scheduled onto a particular CPU or custom circuit. This customization makes PAPS (proposed in [60]) fit the hybrid CPU/FPGA platform.

Table 4.1: Comparison of scheduling methods.

	PAPS				CMBS		
	J	throughput	buffer	time[a]	throughput	buffer	time[a]
#1	1	4.8e-2	26	13	4.8e-2	23	270
#2	2	6.1e-2	29	16	6.3e-2	23	230
#3	3	6.7e-2	31	21	6.7e-2	23	240
#4	6	7.4e-2	32	26	7.4e-2	23	195
#5	8	7.6e-2	35	34	7.7e-2	23	190
#6	22	8.1e-2	49	61	8.3e-2	24	190
#7	100	8.3e-2	127	4202	9.1e-2	24	170

[a] It is the execution time (ms) in solutions finding.

case $\#7$ of PAPS with $J = 100$. In case $\#7$ of CMBS, the results achieved by a schedule with the maximal throughput guarantees are reported.

From the experimental results, some observations made are summarized as follows.

- CMBS always requires less buffer storage space upon the equivalent application throughput guarantees.

- In some case (when required throughput is high), CMBS can achieve higher throughput guarantees than PAPS with much less buffer requirement. For instance, in case $\#7$ CMBS requires 19% of buffer storage demanded by PAPS but gets 8% higher throughput guarantees.

- CMBS is more flexible to meet the vary required throughput guarantees. However, the throughput guarantees of PAPS are determined by the chosen J, which has quite limited options.

- The execution time in CMBS is not sensitive to different throughput guarantees. In fact, when the throughput requirement is higher, the timing of constraint based analysis is shorter and might lead to less execution time. On the contrary, the execution time of PAPS increases fast when J and throughput are relatively higher.

- PAPS is faster in execution time when J is relatively small. However, CMBS surpasses PAPS upon higher throughput requirement, e.g., in case $\#7$.

4.7 Concluding remarks

This chapter has studied the problem of constructing schedules for real-time streaming applications with minimal buffer requirement on hybrid CPU/FPGA architectures. The problem has been formalized declaratively as constraint base scheduling, and can be effectively solved by constraint solvers. The experimental results show that the proposed CMBS methodology performs significantly better than the traditional PAPS method in terms of buffer requirement. It is also flexible in the sense that it can be used to construct schedules to guarantee the required (feasible) throughput.

CHAPTER 5

GLOBAL SCHEDULING ON NoC BASED MPSoCs

Nowadays, multiprocessor systems-on-chip (MPSoCs) are very popular computing platforms for modern embedded systems [100]. To explore the large number of cores built-in, network-on-chip (NoC) communication has been widely used to manage the parallel processing architecture in a scalable manner [26, 40, 68, 90]. While the NoC based MPSoCs enable distributed processing, it is extremely challenging to handle the global computation and communication scheduling of embedded concurrent streaming applications, when different concerns need to be taken into account, e.g., processor availability, interconnection bandwidth, and buffer space.

Given one instance of the tutorial example (see Section 2.1.1) with specified token rates in Figure 5.1(a), it is then allocated it onto a dual-processor architecture, as illustrated in Figure 5.1(b). Each process p_x has a worst case execution time (WCET) $t_{C,x}$, and each channel $ch_{x,y}$ is implemented as finite FIFO buffer with token storage $\gamma_{x,y}$. The hard real-time inter-processor communication latency is captured by an identity process p_δ with delay $t_{C,\delta}$. While processes are *enabled* when they have enough input data tokens and output buffer space, p_i and p_j can only be scheduled sequentially in one single processor μp_1. The scheduling problem on such kind of multi-processors with resource constraints has been known to be NP-complete [31]. Recently, heuristic algorithms have been proposed to provide predictable performance on MPSoCs [88, 29], yet as argued below, they may lead to sub-optimal solutions.

Motivation. Let p_i and p_j to be scheduled with a heuristic time-division multiple-access (TDMA) scheme [88], as marked horizontally on the timeline in Figure 5.1(c-1). Each process is not allowed to fire only when it is *enabled*, but it should also be the process which gets the allocated TDMA time slots. Accordingly, a schedule is built below. The running processes on each processor and the FIFO usage are listed out vertically. At time tag 0, p_i starts the execution and requires space 1 for one output token on $FIFO_{i,j}$. At time tag 1, p_i finishes the previous firing, outputs 1 result token, and starts a new firing. As the scheduling evolves, p_δ and p_k execute once they are *enabled*, and p_i and p_j are scheduled in TDMA scheme. However, the TDMA assigned processes can be stalled, when they violate resource constraints (bounded buffer capacities or no enough input tokens). For instance, p_j is stalled ($\gamma_{j,\delta} = 3$) at time tag 5, and so is p_i ($\gamma_{i,j} = 2$) at time tag 6 and 7. From time tag 2 to 7, the schedule enters a periodic phase with length $L_{period} = 6$, in which the application throughput is guaranteed by process firing patterns. On the other hand, in Figure 5.1(c-2), another optimized schedule exists without using TDMA scheme in μp_1. A 50% throughput gain in this proposed optimized cyclic static schedule is observed, i.e., $L_{period} = 4$, on the same platform.

Although TDMA-like or list scheduling heuristics can be used to design predictable distributed systems [88, 29], they have the following drawbacks:

- They can not avoid the overhead in time slots allocation, which degrades application performance (Section 5.4);
- They are in lack of global optimization mechanisms on MPSoCs, where numerous processors or communication links are concurrently shared by different processes.

As the **contribution in this chapter**, the author proposes a new scheduling framework on MPSoCs to build global schedules for both processors computing (process execution) and communication transactions (real-time traffic-flow). While buffer cost is minimized, a high predictable application throughput is guaranteed based on optimized computation and contention-free routing.

5.1 Related work

Eles et al. [29] first address the scheduling on distributed systems with communication protocols optimization. With optimized bus accessing, application through-

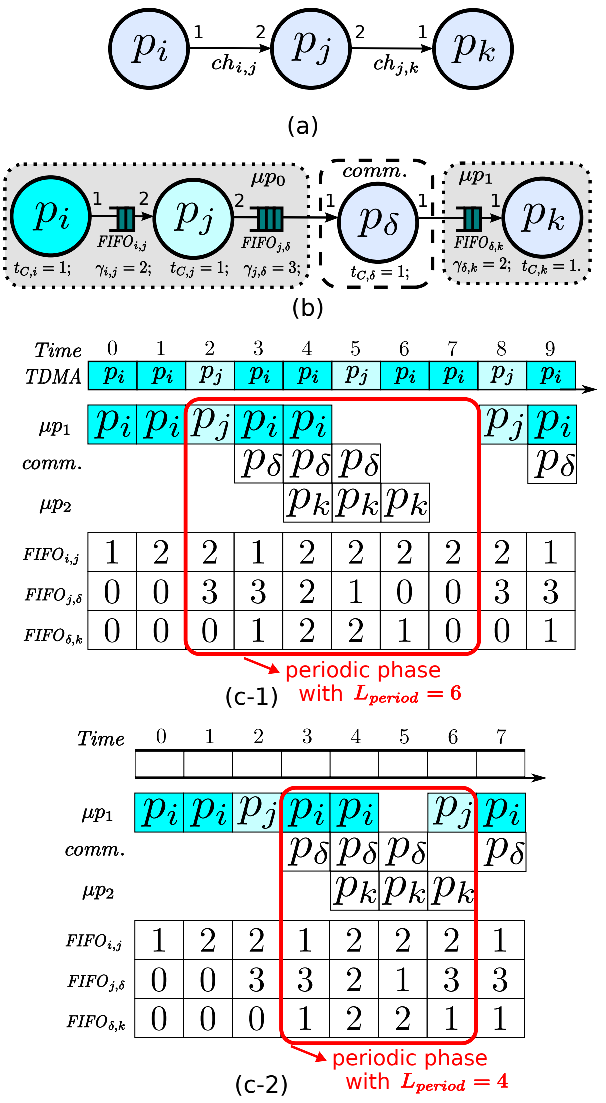


Figure 5.1: Allocation and scheduling of an illustrative application. (a) an illustrative application; (b) application instance allocated onto buffer constrained dual-processor; (c-1) Scheduling with TDMA scheme on μp_1; (c-2) Throughput-optimized scheduling.

put is maximized based on list scheduling (heuristic order) for task graph models. While task graphs can be viewed as special cases of acyclic SDF models with no overlap between different iterations of execution, buffer cost has not been addressed as resource constraints. In [88], Stuijk et al. propose a mapping and TDMA/list scheduling design flow for throughput constrained SDF applications on MPSoCs. As both papers are based on heuristic TDMA or list scheduling, it is argued that there is a lack of global optimization on performance metrics, e.g., the application throughput as motivated in Figure 5.1(c-1).

Inspired by the success of (model-checking, SAT and constraint programming) techniques in solving NP-complete problems, Geilen et al. [36] first use model-checker to determine the minimal deadlock-free buffer cost to schedule SDF models (no computation constraints). Liu et al. [65] use SAT-solver to explore the mapping and scheduling of homogeneous SDF (HSDF) models However, a regular SDF model must be transformed (expanded) to a HSDF model to apply their techniques[1], which dramatically increases the problem size [84]. Constraint programming tools are used as well in the scheduling of task graphs on MPSoCs, without violating computation capacity and communication bandwidth [10, 45]. In chapter 4, constraint based paradigm has been proposed for SDF models scheduling on hybrid CPU/FPGA platforms, in which communicate happens via dedicated (not shared) FIFO channels with ignored delay. Yet, all the work above does not consider the global optimization of distributed computing and communication transactions on MPSoCs.

5.2 Architecture platform

The cornerstone of streaming applications with predictable performance is the architecture platform with deterministic real-time properties. In this chapter, the architectural template is the regular 2-D mesh tiled MPSoCs (see Section 2.2.1) with the communication happens via hard real-time networks-on-chip (NoCs). Nevertheless, the design method to be proposed is not limited to this particular platform.

As exemplified in Figure 5.2, each tile $tile_n$ ($n \in \mathbb{N}_0$) consists of a processor (μp_n), an application memory, a token buffer $buffer_n$, and a network interface (NI). $tile_n$ is labeled as (x_n, y_n) in the mesh topology, where x_n and y_n correspond to the row and column indexing numbers respectively. The work starts with a fixed allocation from processes to tiles. Hence, all application memory modules

[1]In [65], models contain up to 30 HSDF processes have been considered, but the equivalent HSDF H263 model in our experiments (Section 5.4) has 4754 processes.

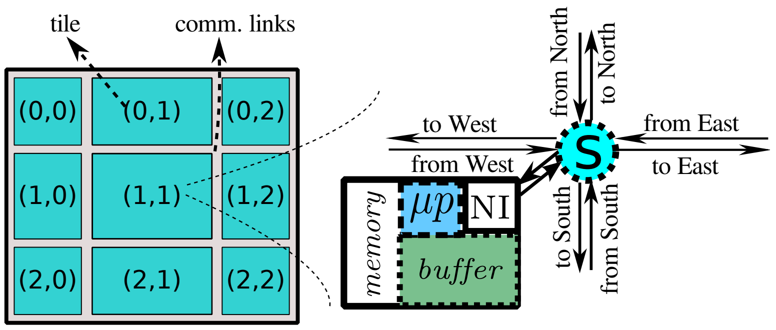


Figure 5.2: A 3×3 regular mesh MPSoC architecture.

are pre-specified on the mesh. Preferably, the work is on the analysis of routing, scheduling, and buffer properties for the dashed modules in Figure 5.2, i.e., NoC switches, processors, and token buffers.

Tiles are connected to the communication network through switches (s), and communicate with each other via unidirectional communication links.

5.3 Constraint based scheduling

The binding of streaming applications onto the target MPSoC architectures is a refinement process with real-time requirement and resource limitations. For streaming applications with real-time constraints, the efficient execution means the streaming services are delivered on-demand to the end-user, neither too fast nor too slow. Thus, a predetermined application throughput needs to be guaranteed, as formalized in Chapter 4 (see Constraint 7 and 8).

In this section, the constraint based formulation is used to model the left application scheduling process, which includes architecture mapping, communication routing, flow control, and computation scheduling. As each (acyclic or cyclic) SDF model can be decomposed into a set of concurrent producer-consumer pairs, the problem is analyzed and formalized based on a general producer-consumer processes pair, as illustrated in the upper part of Figure 5.3.

5.3.1 Mapping

To be aware of the pre-specified mapping decisions, two sets of variables: α and ω are used. A boolean variable $\alpha_{i,\mu p_n}$ denotes the presence of p_i on a processor μp_n. Assuming that different instances of a process can only execute on one dedicated processor, this *single residence* constraint can be formalized as the following.

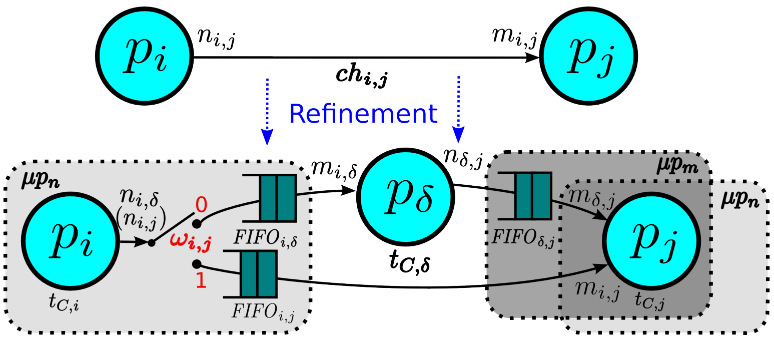


Figure 5.3: A template of refined producer-consumer pair, in which $\omega_{i,j}$ denotes whether channel $ch_{i,j}$ is implemented as intra-processor communication.

Constraint 11. *(Single residence) Each process p_i needs one (and only one) specified processor for computation.*

$$\sum_{\mu p_n \in \mathbf{U}} \alpha_{i,\mu p_n} \equiv 1, \quad \forall p_i \in \mathbf{P} \tag{5.1}$$

in which **P** *is the set of processes in application models, and* **U** *is the set of processors in the architecture platform.*

Redundantly, a boolean variable $\omega_{i,j}$ denotes how a channel $ch_{i,j}$ is implemented, as illustrated in Figure 5.3. When $\omega_{i,j} = 0$, p_i and p_j are mapped onto different processors (μp_n and μp_m), $ch_{i,j}$ is implemented as inter-processor communication with buffers $FIFO_{i,\delta}$ and $FIFO_{\delta,j}$, and the hard real-time communication is captured by process p_δ with bounded latency $t_{C,\delta}$ and without packet loss ($m_{i,\delta} = n_{\delta,j}$). Otherwise, $\omega_{i,j} = 1$, p_i and p_j are mapped onto the same tile, and the intra-tile communication $ch_{i,j}$ is implemented as $FIFO_{i,j}$ with ignorable latency. Processors in the architecture platform are *homogeneous* in the sense that they are the same type and each p_j has the same WCET $t_{C,j}$ being mapped onto any processors. $\omega_{i,j}$ can be defined as the following.

Constraint 12. *(Correlated mapping decision) To be correlated with $\alpha_{i,\mu p_n}$ and $\alpha_{j,\mu p_m}$, $\omega_{i,j}$ denotes whether p_i and p_j are mapped onto the same processor.*

$$\omega_{i,j} = (x_n == x_m) \wedge (y_n == y_m), \ \alpha_{i,\mu p_n} = 1, \alpha_{j,\mu p_m} = 1. \tag{5.2}$$

Event models based on cumulative functions have been used to capture working load and pressing capabilities of streaming applications, and accordingly the execution semantics of SDF models can be formalized (refer to chapter 4). In this chapter, the process scheduling status derived from event models needs to be refined to be aware of the mapping decisions.

Constraint 13. *(Mapping & scheduling association) All the processes assigned to each processor can only execute (be scheduled) sequentially at any time. This mapping and scheduling association is formalized as:*

$$\sum_{p_i \in \mathbf{P}} \alpha_{i,\mu p_n} W_j(t) \in \{0,1\},\ \forall \mu p_n \in \mathbf{U}, t \in \mathbb{N}_0 \tag{5.3}$$

in which $W_j(t)$ denotes the 1-0 *(computing or stalling) status of each process p_j (see Constraint 6).*

5.3.2 Template based Buffering Analysis

Here, some buffer properties and constraints can be formulated from the event models and mapping-aware decisions, based on the refined producer-consumer template in Figure 5.3.

Property 3. *(Buffer usage) The buffer usages of FIFO$_{i,\delta}$, FIFO$_{\delta,j}$, and FIFO$_{i,j}$ in the template (Figure 5.3) at time tag t are denoted as $B_{i,\delta}(t)$, $B_{\delta,j}(t)$, and $B_{i,j}(t)$ respectively, which are defined as:*

$$B_{i,\delta}(t) = D_{i,\delta}(t) - \neg\omega_{i,j}(C_{i,\delta}(t) - B^0_{i,\delta}) - \omega_{i,j}(C_{i,j}(t) - B^0_{i,j}) \tag{5.4}$$

$$B_{\delta,j}(t) = \neg\omega_{i,j}B'_{\delta,j}(t) = D_{\delta,j}(t) - \neg\omega_{i,j}(C_{\delta,j}(t) - B^0_{\delta,j}) \tag{5.5}$$

$$B_{i,j}(t) = \omega_{i,j}B_{i,\delta}(t) \tag{5.6}$$

in which $B^0_{i,\delta}(t)$, $B^0_{\delta,j}(t)$, and $B^0_{i,j}(t)$ are the initial data tokens (at time tag 0) in each buffer.

Furthermore, the asynchronous behaviors of the buffers, caused by data buffering or processing latency (computation and communication), are captured as the following.

Constraint 14. *(Asynchronous buffer) The incoming data tokens in buffers take at least the WCET of the consumer process to be consumed. For buffers FIFO$_{i,\delta}$, FIFO$_{\delta,j}$, and FIFO$_{i,j}$ in the template (Figure 5.3), the asynchronous behaviors can be formalized as:*

$$R_{i,\delta}(t) \geqslant \neg\omega_{i,j}C_{i,\delta}(t + t_{C,\delta}) + \omega_{i,j}C_{\delta,j}(t + t_{C,j}) - B^0_{i,\delta} \tag{5.7}$$

$$R_{\delta,j}(t) = \neg\omega_{i,j}R_{\delta,j}(t) \geqslant \neg\omega_{i,j}C_{\delta,j}(t + t_{C,j}) - B^0_{\delta,j} \tag{5.8}$$

$$R_{i,j}(t) = \omega_{i,j}R_{i,\delta}(t) \tag{5.9}$$

Depending on the buffer organization, there are two measures of buffer requirement when different FIFOs are assigned to the same tile (buffer) [36], i.e., the FIFO implementations of different communication channels are either partitioned disjointly or sharing buffer space. Accordingly, the buffer requirement of these two mechanism can be formalized in the following.

Property 4. *(Buffer cost – disjoint partition) When FIFOs are organized on each tile as disjoint buffer partitions, the buffer cost $\gamma_{Sum'}$ in the platform is simply the sum of individual FIFO sizes.*

$$\gamma_{Sum'} = \sum_{i,j,\delta} (\omega_{i,j}\gamma_{i,j} + \neg\omega_{i,j}(\gamma_{i,\delta} + \gamma_{\delta,j})),\ \forall p_i, p_j \in \mathbf{P} \tag{5.10}$$
$$where \quad \gamma_{i,j} \geqslant B_{i,j}(t), \gamma_{i,\delta} \geqslant B_{i,\delta}(t), \gamma_{\delta,j} \geqslant B_{\delta,j}(t),\ \forall t \in \mathbb{N}_0$$

in which $\gamma_{i,j}$, $\gamma_{i,\delta}$, and $\gamma_{\delta,j}$ denote the sizes of the disjoint FIFOs in the template (Figure 5.3).

Property 5. *(Buffer cost – shared partition) When FIFOs are sharing space on each tile, the buffer cost $\gamma_{Sum''}$ in the platform is the sum of the shared buffer space on individual tiles.*

$$\gamma_{Sum''} = \sum_{n} \gamma_{tile_n} \tag{5.11}$$
$$where\ \gamma_{tile,n} \geqslant \sum_{i,j} (a_{i,\mu p_n} B_{i,\delta}(t) + a_{j,\mu p_n} B_{\delta,j}(t) +$$
$$a_{i,\mu p_n}\omega_{i,j} B_{i,j}(t)),\ \forall p_i, p_j \in \mathbf{P}, \forall t \in \mathbb{N}_0$$

in which γ_{tile_n} denotes the size of the buffer as one shared partition on tile $tile_n$.

5.3.3 Routing and contention-free flow control

When processes in a producer-consumer pair are assigned to different processors, routing is needed for the inter-tile data communication. Here, deterministic *X*-*Y* routing is assumed on NoC infrastructure. The bidirectional routing decisions are capture by two sets of triple-value (± 1 and 0) variables β^r and β^c on the direction of *rows* and *columns* of physical links respectively, i.e., +1 (-1) represents routing a packet flow on the positive (negative) direction of a link on *X* or *Y* axis, and 0 represents no packet flow. β^r and β^c can be associated with the mapping variables α in the following.

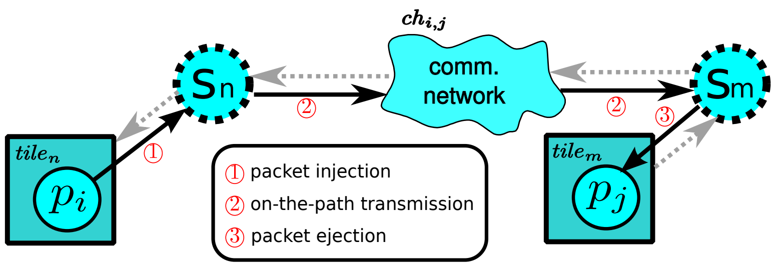


Figure 5.4: Three phases to route a packet in inter-tile channel $ch_{i,j}$.

Constraint 15. *(X-Y routing) Let $\beta^r_{ch_{i,j},l_k}$ represent the routing decision on a row link l_k between tiles (x_k, y_k) and $(x_k + 1, y_k)$, i.e., how channel $ch_{i,j}$ is assigned to l_k. The routing on the primary X axis is formalized as:*

$$\beta^r_{ch_{i,j},l_k} = \begin{cases} +1, & x_m > x_n, y_k = y_n, \forall x_k \in [x_n, x_m); \\ -1, & x_n > x_m, y_k = y_n, \forall x_k \in [x_m, x_n); \\ 0, & \textit{otherwise.} \end{cases} \tag{5.12}$$

$$\forall p_i, p_j \in \mathbf{P}, \alpha_{i,\mu p_n} \equiv 1, \alpha_{j,\mu p_m} \equiv 1, m \neq n.$$

Similarly, the subsequent routing decisions $\beta^c_{ch_{i,j},l_k}$ on column links (Y axis) can be defined.

When a channel $ch_{i,j}$ is assigned to NoC , the communication time needed depends not only on the number of transferred data tokens but also the dynamic link bandwidth during the transaction. In hard real-time NoCs, contention-free routing at the traffic flow level is usually adopted, in which a packet (data token) reserves a circuit switching before the transmission finishes, e.g., in Æthereal [40]. For one application flow (channel), the routing of a packet from source tile to destination tile via NoC switches and links consists of three phases: packet *injection*, *on-the-path* transmission, and packet *ejection*, as illustrated in Figure 5.4. When different application flows are sharing the same NoC resources (switches and links), the packet *injections* (*ejections*) congest spatially when multiple producer (consumer) processes are mapped onto the same tile. Similarly, the *on-the-path* transmissions congest when different application flows are sharing the same links. However, different application flows can be scheduled temporally to avoid the competition on the same links at the same time. Such kind of contention-free scheduling can be formalized as the following.

Constraint 16. *(Contention-free traffic flow scheduling) When communication channels are assigned to inter-tile physical links, the packet injection, ejection, and* *inter-tile traffic flow on rows of communication links are scheduled to avoid contention, as formalized in Eq. 5.13, 5.14, and 5.15 respectively.*

$$\sum_i \alpha_{i,\mu p_n} W_\delta(t) \in \{0,1\}, \tag{5.13}$$

$$\sum_j \alpha_{j,\mu p_n} W_\delta(t) \in \{0,1\}, \tag{5.14}$$

$$\sum_{i,j} \beta^r_{ch_{i,j},l_k} W_\delta(t) \in \{0,\pm 1\}, \sum_{i,j} |\beta^r_{ch_{i,j},l_k}| W_\delta(t) \in \{0,1,2\}, \tag{5.15}$$

$$\forall p_i, p_j \in \mathbf{P}, \mu p_n \in \mathbf{U}, t \in \mathbb{N}_0$$

with $W_\delta(t)$ (see Eq. 4.9) to denote the data transmission 1-0 *(working or idle) status on NoC modeled by p_δ. Similarly, the scheduling of traffic flow on columns of communication links can be formalized as in Eq. 5.15, which is omitted for clarity.*

5.4 Experimental results

The experiments have been done on a few benchmarks to demonstrate how the proposed Constraint based Minimal Buffer Scheduling (CMBS) approach works with Gecode solver in practice. First, the effects of scheduling overhead (OH) which exists in TDMA-like scheduling are evaluated, and then a comparison with the heuristic PAPS [60] in buffer cost is done.

5.4.1 Evaluation of scheduling overhead

Usually, the predictable time slots allocation in TDMA or list scheduling is modeled by increasing the computation or communication latency with the postponed time, i.e., the OH is caused by the improper TDMA time wheel allocation before the computation or communication can happen. Here, the OH is slightly (compared with [88]) specified as 50% in computation and 100% in communication latency. The proposed *CMBS* approach is then used to estimate the best scheduling quality of heuristics, denoted as *CMBS-OH*. Two benchmarks from communication domains, i.e., a Modem application from [13], and a Wireless application from [69], are used. The architecture platform is a 3×3 mesh-based MPSoC.

Table 5.1 shows the experimental results. Each time two instances of the same applications are allocated onto the MPSoC empirically with specified throughput requirement, and the unroll factor J (see Constraint 7) is fixed to 1. The search tree

Table 5.1: Comparison of scenarios with varying OH (3×3 platform).

specification				*SCP-OH*[a]		*SCP*	
app.	#[b]	*thru. req.*	*J*	$\gamma_{Sum'}(\gamma_{Sum''})$	*time*[c]	$\gamma_{Sum'}(\gamma_{Sum''})$	*time*[c]
	2	3.125e-2	1	-[d]	352	98(45)	3.0e3
Modem	2	1.667e-2	1	98(46)	5.0e3	92(41)	1.4e3
	2	2.5e-2	1	-	422	121(53)	4.7e4
Wireless	2	1.7e-2	1	123(49)	6.4e5	120(48)	1.2e3

[a] 50% OH in computation latency, plus 100% OH in communication latency.
[b] The number of application instances mapped onto platform.
[c] The solutions finding time (ms), branched and explored with $\gamma_{Sum'}$ and γ.
[d] The problem is infeasible.

is branched and explored with buffer storage $\gamma_{Sum'}$ and γ_{tile_n}, then $\gamma_{Sum''}$ (Property 5) is calculated. Compared with *CMBS-OH*, *CMBS* can meet much higher (up to 87%) throughput requirement, without much increase in buffer cost. Therefore, it is argued that the OH in heuristic scheduling on MPSoCs can cause performance degradation. Furthermore, buffer costs measured in $\gamma_{Sum''}$ show a great reduction ($55\sim65\%$) on $\gamma_{Sum''}$. Although, to branch and explore with $\gamma_{Sum''}$ and γ_{tile_n} has the potency to further reduce $\gamma_{Sum''}$, it dramatically increase the problem complexity. For instance, using the specification $\#2$ of Wireless, a $\gamma_{Sum''}$ with value 30 (instead of 49 in Table 5.1) can be found in 3hrs8mins. But all other experiments fail to find a solution in 4 hrs. The memory usages for solutions finding are in $47\sim163$Mb. Interestingly, the infeasible cases on both applications take the least memory usage and time, which is opposed to the model-checking method in [36].

5.4.2 Comparison with PAPS

Here, another performance metrics (i.e., buffer cost), which has not been well constrained in heuristics for MPSoCs, is evaluated. A PAPS implemented in C++ is used as the reference method, which has no buffer constraints, similar as [88, 29]. A fixed ideal delay (no-contention aware) in inter-tile communication is used in the reference PAPS, as communication protocols was not considered. Three benchmarks are used, i.e., a Bipartite model [13], a Cd2dat rate converter [13], and a H263 decoder [88]. The architecture platform is a 2×2 mesh-based MPSoC.

Table 5.2 shows the experimental results. In *CMBS*, the search tree is branched and explored with two types of buffer measurements. The solutions finding time is reasonable for such a medium problem size. Although a NoC communication protocol is modeled, CMBS shows a significant buffer saving (6% of PAPS in

Table 5.2: Comparison of scheduling methods (2×2 platform).

specification			*PAPS*			*CMBS*		
app.	*#*	*thru. req.*	*J*	$\gamma_{Sum'}(\gamma_{Sum''})$	*time*	*J*	$\gamma_{Sum'}(\gamma_{Sum''})$	*time'(time")*[a]
Bipartite	1	1.101e-1	3	510(510)	120	3	36(31)	2.6e3(2.3e5)
	1	1.096e-1	2	352(352)	50	1	36(31)	1.8e3(1.4e5)
	1	1.082e-1	1	194(194)	20	1	36(28)	1.9e3(2.1e5)
Cd2dat	2	2.462e-1	-	-	-	1	68(34)	1.9e3(4.7e4)
	2	1.553e-1	2	2926(1504)	430	1	66(34)	1.8e3(3.3e5)
	2	1.550e-1	1	1472(762)	120	1	66(34)	1.9e3(3.3e5)
H263	2	2.103e-4	-	-	-	1	9512(9506)	9.7e3(2.3e5)
	2	2.102e-4	2	19016(19012)	2.0e5	1	9512(9506)	9.5e3(2.3e5)
	2	2.101e-4	1	9512(9508)	2.4e4	1	9512(9506)	9.2e3(2.1e5)

[a] The solutions finding time (ms) branched and explored with two buffer measurements.

the best case), and can provide higher (up to 64%) throughput. Here, the highest memory usage for CMBS in solutions finding is up to 219Mb, caused by a high peak depth 440 in its search tree.

5.5 Concluding remarks

This chapter has presented a constraint based scheduling framework for SDF streaming applications on NoC based MPSoCs. Based on constraint programming techniques, the global scheduling for both processors computing and communication transactions has been achieved. Compared with traditional heuristics scheduling, the proposed method has higher predictable application throughput and less buffer cost.

CHAPTER 6

PERFORMANCE ANALYSIS OF ADAPTIVE SYSTEMS

Partially run-time reconfigurable (RTR) FPGAs, such as Xilinx Virtex-4 [101], are becoming very popular infrastructures in today's embedded systems [19]. They allow part of the hardware tasks to be reprogramed dynamically on the fly while the remaining part continues its operation without being affected. For a number of signal processing and multi-media streaming applications, this reconfigurable property enhances their capability to vary functionalities at run-time in a dynamic environment with varying demands, which significantly reduces the design cost while leveraging the ubiquity of embedded systems. Hence, RTR FPGAs can be used to built cost-effective hardware platform for streaming applications [57], and deliver high flexibility, besides breakthrough performance. However, this combination of flexibility and efficiency does not come for free. Adaptivity adds another dimension of complexity to the design process, while the system performance during reconfiguration still needs to be satisfied. Compared with traditional non-adaptive systems, the reconfigurations in adaptive systems add new challenges in performance analysis and lead to even more complexity.

An run-time reconfigurable streaming application based on adaptive extensions on synchronous data flow (SDF) semantics [60] is illustrated in Figure 6.1, which is used as a tutorial example in this chapter. Besides the regular source and sink processes p_i and p_k, there is an adaptive process p_j, which has N different working modes from M_1 to M_N as illustrated in the dashed box. The mode change is

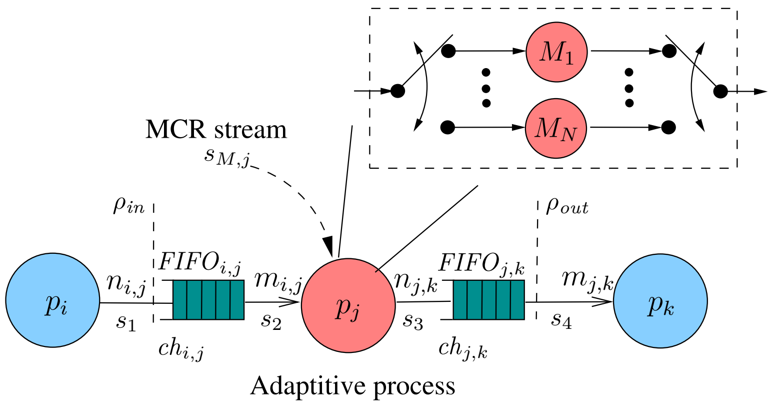


Figure 6.1: A minimal adaptive streaming application model.

initiated by the mode change request (MCR) stream, i.e., $s_{M,j}$ for p_j. Each mode transition circumstance introduces a temporal overhead, during which p_j does not work in either the old or new mode and is thus stalled.

While the stream source p_i provides a peak throughput ρ_{in} (to the input of communication channel $ch_{i,j}$ cutting by the dashed-line), an average output throughput ρ_{out} needs to be guaranteed by the application even during the reconfigurations. Caused by the reconfiguration stalls of adaptive process p_j, it is critical that the backlog tokens in the FIFO(s) between p_j and the consumer process(es), the so called *playout buffer(s)*[1], are sufficient to sustain the application throughput. Therefore, to exploit the full potential of RTR adaptive systems, special resource requirement and scheduling techniques are needed when taking the behaviors and properties of reconfigurations into consideration. This chapter will address performance analysis, without losing throughput guarantees and design efficiency, for real-time adaptive streaming applications. To the extent of the author's knowledge, it is still an open topic, which has not yet been covered by previous work.

6.1 Related work

Formal analysis at design time has been widely used in performance analysis of heterogeneous embedded systems, such as timing properties validation, buffer dimensioning, and scheduling policies optimization. It has been a promising op-

[1]In the application in Figure 6.1, the playout buffer is $FIFO_{j,k}$.

tion to overcome the limitation of simulation methods in incomplete coverage at corner cases, and can thus provide conservative system properties guarantees. In SymTA/S [78], a compositional way for scheduling analysis has been presented based on standard periodic/sporadic event models of data streams. Network calculus [22] and real-time calculus (RTC) [94, 17] are both a collection of methods in deterministic queuing theories. They formalize the incoming workloads and processing capabilities as cumulative functions of time, and suit system analysis of performance guarantees and buffer dimensioning in network domain [15] and real-time distributed embedded system domain [93, 92] respectively. In the subsequent extension of RTC, Chakraborty et al. [18] and Phan et al. [73] present a mode based RTC to handle the execution dependence between processing resources and buffers caused by their state information (i.e., the fill-level of buffers and its effect on processing resources). However, none of them takes adaptive systems into consideration and do not aim at design cost (area) analysis and buffer dimensioning as the author does.

Schedulability analysis and reconfiguration methods for multi-mode (adaptive) real-time systems has been studied in [83, 76], where each mode consists of different set of tasks. They develop mode change protocols in mode transition stages, and exploit analysis techniques to ensure that no deadlines are violated during the transition periods. For adaptive systems with pre-specified (known at compile-time) reconfiguration scenarios, they can be modeled and simulated in a similar way as in [81]. However, this chapter aims at adaptive systems with reconfiguration requests known at run-time. The simulation based way has inherent limitations to cover the corner cases which have not been simulated. Therefore, it can lead to too optimistic buffer dimensioning.

Bilsen et al. [14] first present cyclo-static dataflow model, which supports cyclically changing of the number of tokens produced and consumed by processes. Since the mode changing behavior is predefined at compile-time, static schedules can be constructed when the necessary and sufficient conditions for scheduling hold. Furthermore, the buffer requirement is analyzed for cyclo-static dataflow models according to the specified throughput requirement in [89], similar as in SDF models [87]. However, the mode changing problem to be addressed in this paper is more challenging in the sense that the reconfigurations are only known at *run-time* (unpredictable at compile-time). Accordingly, the buffer requirement can be analyzed for specified throughput requirement [89], similar as in SDF models [87]. Wiggers et al. [98] propose an algorithm to compute sufficient buffer capacities for task graphs with data dependent execution conditions (consumption token rate), but they do not consider the run-time overhead between the transition of different execution conditions. On the other hand, the adaptive systems to be addressed in this chapter has more flexibility from the sense that the reconfigurations are only known at *run-time* (unpredictable at compile-time). In [91], simulation based techniques have been introduced for the analysis of different performance metrics of scenario-aware SDF models with stochastic mode changes. However, they do not dimension the buffer requirement for RTR applications with throughput guarantees.

This chapter proposes, in contrast to the existing work, a novel approach for run-time reconfigurations analysis based on iterative timing phases, and present a performance analysis framework for adaptive real-time streaming applications on RTR FPGAs. It is based on the refinement of the author's previous work [105], which is a hybrid approach combined integer linear programming (ILP) analysis and simulation on synchronous model of computation. In the implementation, the existing successful optimization techniques in constraint programming (CP) domain, i.e., *Gecode* [32] solver, has been exploited for problem solving (to get the minimal buffer requirement without losing throughput guarantees).

6.2 Reconfiguration preliminaries

In this section, without losing generality, the reconfiguration preliminaries used in this chapter are defined based on the example application in Figure 6.1.

6.2.1 Definitions and assumptions

First, the reconfiguration definitions and the main assumptions used in this work are introduced.

For the adaptive process p_j, a mode change is triggered by an MCR, i.e., the stream $s_{M,j}$ in Figure 6.1, which might either come from an external controller or be retrieved from the input data streams. During the mode transition stage, the old configuration is deleted and released for the loading of new ones. The reconfiguration transition from an old mode M_1 to a new mode M_2 takes non-zero time $t_{R,j}^{M_1,M_2}$. $t_{R,j}^{M_1,M_2}$ depends on the circuit size of different reconfiguration modes and is usually non-ignorable.

For convenience, to avoid the use of $t_{R,j}^{M_x,M_y}$, $t_{R,j}$ is simply used to implicitly mean $t_{R,j}^{M_x,M_y}$ when the mode switching is known from context. Similarly, $t_{C,j}$ is used to implicitly mean the computation time $t_{C,j}^{M_1}$ or $t_{C,j}^{M_2}$ of process p_j in the respective working modes.

An MCR may occur during the execution of the system in a particular mode, but never during the transition stages. In the worst case, two succeeding configurations have the minimal interval $t_{interR,i}$ to avoid violating the application throughput requirement caused by too-close consecutive reconfiguration stalls.

The input data stream arrives at a peak or average[2] throughput ρ_{in}. Meanwhile, a required average output throughput ρ_{out} is applied to the sink process p_k, to denote the stable application throughput requirement during the lifetime of adaptive systems even in reconfiguration transition stage.

6.2.2 Consistency

The consistency of SDF models (see Section 2.1.1) is known to be a necessary condition to allow them to be executed within bounded memory without deadlock [37]. To execute adaptive SDF streaming applications within bounded memory without deadlock, the model consistency needs to be preserved. Besides the balance equations as defined in Eq. 3.1 for communication channels between non-adaptive processes, furthermore, for each communication channel $ch_{i,j}$ from non-adaptive process p_i to adaptive process p_j, the following equation holds.

$$r_i \cdot n_{i,j} = \sum_{x=1}^{n} r_j^{M_x} \cdot m_{i,j}^{M_x} \tag{6.1}$$

in which $r_j^{M_x}$ and $m_{i,j}^{M_x}$ are the firing rate and consumption input rate for process p_j in each working mode M_x.

Reconfiguration protocols for adaptive SDF applications, similar as in [76] for task graphs, are needed to guarantee that such an equation holds at run-time. Since to develop such protocols is out of the scope of this chapter, it is simply assumed that all configurations of the same adaptive process have the same input/output token rate which meets Eq. 3.1, i.e., the reconfiguration scenarios addressed always comply with the application consistency requirement. Instead, this work focuses on the performance analysis of adaptive systems. However, the proposed approach should be able to handle more general SDF applications, once the reconfiguration protocols are available and can be captured as extra constraints in the analysis framework.

[2]Both situations will be covered in the proposed analysis approach.

6.2.3 Design cost on architecture model

The architecture template is a partially reconfigurable FPGA, as illustrated in Figure 2.4. Since the pre-specified application throughput needs to be sustained during the whole time period of reconfigurations, the consequence may be that there is a need for extra buffer space, which includes the output buffer to sustain the output throughput during reconfiguration transitions and possibly the buffer to store input data tokens. The only unit used for area is logic elements (LEs). Area requirements in form of memory elements are converted into LEs. Given the hardware implementations for each mode of the process, the cost of configuration controller A_{CC}, configuration slot A_C, and configuration memory $\sum_{i=1}^{i=n} A_{M,i}$ are static (fixed). For the "just-in-time" (JIT) configuration on RTR FPGAs in Figure 2.4, the total design cost in terms of area is

$$A_{JIT}=A_{CC}+\sum_{i=1}^{i=n} A_{M,i}+\underbrace{k_C \cdot max(A_{M,1},\dots,A_{M,n})}_{A_C}+A_{Buffer} \qquad (6.2)$$

in which A_{Buffer} is the area cost of buffers, i.e., $FIFO_{i,j}$ and $FIFO_{j,k}$ in Figure 6.1.

To design JIT reconfigurable systems efficiently, it is critical to dimension the minimal conservative buffer size and the corresponding cost A_{Buffer}, and to explore the implementation trade-offs of different design options.

6.3 Reconfigurations analysis framework

In the following, a constraint based framework based on iterative timing phases is to be proposed for reconfiguration analysis. fits well to capture both the streaming application execution semantics and the varying design concerns during run-time reconfigurations.

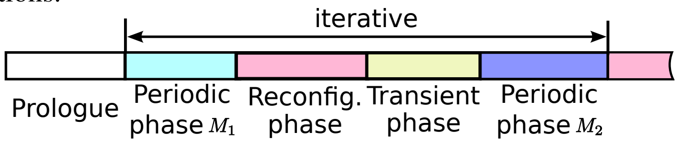


Figure 6.2: Timing phases of reconfiguration analysis. In this graph, only a particular mode transition from M_1 to M_2 is shown.

The staged timing phases in reconfiguration analysis, as illustrated in Figure 6.2, are the following.

Prologue. It is the start-up phase with no throughput guarantees. The length of the prologue phase can be specified by τ_0 in Constraint 7 (see Section 4.4.4).

Periodic phase $M_1(M_2, \cdots, M_N)$. They are phases with guaranteed throughput in working mode $M_1(M_2)$ of the adaptive process p_j. While the length L_{period} is throughput relevant, the sustainable throughput requirements can be distinct in different working modes and are specified in Constraint 7-8.

Reconfiguration phase. It starts with an initial working mode M_1 upon the reconfiguration request MCR. The mode transmission starting time tag t' (determined by a reconfiguration decision variable $\xi(t)$ to be defined in Section 6.4) is explored in a specified periodic phase L_{period}[3] to find the optimal reconfiguration with minimized *objective function*. The reconfiguration stall (working mode transmission to M_2) takes $t_{R,j}$ time slots (Constraint 19). The length of this phase is specified to be the worst case $L_{period} + t_{R,j}$.

Transient phase. It has a length τ_1, in which throughput is met but with no periodic properties in scheduling yet.

Caused by the periodic properties in the scheduling of each working mode, the buffer requirement can be analyzed in a finite length of time, without everlasting analysis. To make the phases in gray (colored) iterative, we can use the timing analysis to traverse all reconfiguration scenarios. The conservative size of each buffer should adopt their worst case dimension respectively.

The design objective is to find the minimal total buffer sizes as defined in Eq. 4.13. In particular, the conservative size $\gamma_{i,j}$ of each buffer $FIFO_{i,j}$ should adopt their worst case dimension respectively in traversing all the reconfiguration scenarios.

6.4 Constraints on reconfigurable systems

For SDF streaming applications, their execution rules follow the semantics as described in Section 4.4.3, with the application output throughput ρ_{out} subject to Constraint 7-8. For the input data stream s_1 to process p_j, a peak or average throughput ρ_{in} is described as follows.

Constraint 17. *(Source input) For the source signal process p_i with computation latency $t_{C,i}$, the data stream provides a peak throughput ρ_{in} according to*

$$R_i(t + t_{C,i}) - R_i(t) \quad \leqslant \quad t_{C,i} \cdot \rho_{in} \tag{6.3}$$

[3]It is based on the assumption that the worst case reconfiguration response (starting) time is L_{period} after the run-time MCR.

Otherwise, a stable average throughput ρ_{in} can be verified at specified time instances as follows

$$R_i(t_0 + k \cdot \Delta_{t_0}) - R_i(t_0) = k \cdot \Delta_{t_0} \cdot \rho_{in}, \quad \forall k \in \mathbb{N} \tag{6.4}$$

in which the given t_0 specifies the starting time instance for average throughput checking, and Δ_{t_0} determines the interval between different time instances.

To capture the run-time *reconfiguration* of the reconfigurable process, a boolean function $\xi(t)$ is used to denote the working mode at time tag t of the reconfigurable process p_j, i.e., $\xi(t) = 0$ indicates that p_j works in mode M_1, otherwise p_j is in (or being reconfigured to) mode M_2. Thus, the computation latency $t_{C,j}$ for the adaptive process p_j can be formulated.

Constraint 18. *(Computation latency - adaptive) To be aware of the reconfiguration decision $\xi(t)$, the computation latency $t_{C,j}$ of the adaptive process p_j is*

$$t_{C,j} = \neg\xi(t) \cdot t_{C,j}^{M_1} + \xi(t) \cdot t_{C,j}^{M_2} \tag{6.5}$$

in which $t_{C,j}^{M_1}$ and $t_{C,j}^{M_2}$ denote the computation latency in working mode M_1 and M_2 for p_j respectively,

However, Eq. 6.5 can not be implemented in Gecode directly, since $t_{C,j}$ is not an explicit variable in our model. Accordingly, the constraints for adaptive process p_j containing $t_{C,j}$ need to be rewritten correspondingly. For instance, Eq. 4.4 is equivalent to the following constraint applicable in Gecode.

$$\begin{aligned} \neg\xi(t) \cdot C_{i,j}(t + t_{C,j}^{M_1}) + \xi(t) \cdot C_{i,j}(t + t_{C,j}^{M_2}) - C_{i,j}(t) = \\ m_{i,j} \cdot (\neg\xi(t) \cdot K_j(t + t_{C,j}^{M_1}) + \xi(t) \cdot K_j(t + t_{C,j}^{M_2})) \end{aligned} \tag{6.6}$$

Especially, the multiplication of two variables in Eq. 6.6 can be captured by a non-linear arithmetic constraint (*mult*) in Gecode. Similarly, other SDF execution semantics on computation and buffer resources can be extended to be reconfiguration aware (for the reconfigurable process), which are omitted for clarity in the thesis.

Furthermore, in the reconfigurations analysis of the adaptive process p_j, its computation stall during the reconfiguration transition stage needs to be considered. Such a stall takes $t_{R,j}$ time, and can be described as a constraint as follows.

Constraint 19. *(Reconfiguration stall) The computation of the adaptive process p_j stalls for $t_{R,j}$ time period after the reconfiguration starts at time tag t'.*

$$W_j(t' + \Delta_{t'}) = 0, \quad \forall \Delta_{t'} \in [0, t_{R,j}) \tag{6.7}$$

$$W_j(t) = \sum_{\Delta_t} \tfrac{C_{i,j}(t+\Delta_t+1) - C_{i,j}(t+\Delta_t)}{m_{i,j}}, \quad {\scriptstyle \Delta_t \in [0, t_{C,j})} \tag{6.8}$$

in which $W_j(t)$ denotes the 1-0 *(computing or stalling) status of process p_j at any time tag t.*

6.5 Experimental results

In this section, the proposed reconfigurations analysis framework is implemented on *Gecode*, and several experiments have been done on both the example application of Figure 6.1 and an industrial application from Thales Communications.

6.5.1 The example application

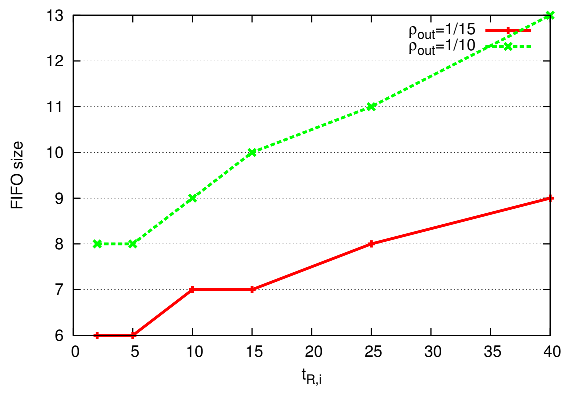
It is assumed that the adaptive process p_j in the example application has two different modes, and both configurations have the same properties (i.e., the same input and output rates, computation time and reconfiguration time). Thus, two iterations of the timing phases from Phase I to II (see Figure 6.2) in reconfiguration analysis can traverse all the reconfiguration scenarios.

The specification model shown in Figure 6.1 is instanced with concrete parameters (i.e., $n_{i,j} = 2$, $m_{i,j} = 3$, $n_{j,k} = 1$ and $m_{j,k} = 2$), with a minimum interval $t_{interR,j} = 50$ time slots between the two succeeding reconfigurations of p_j.

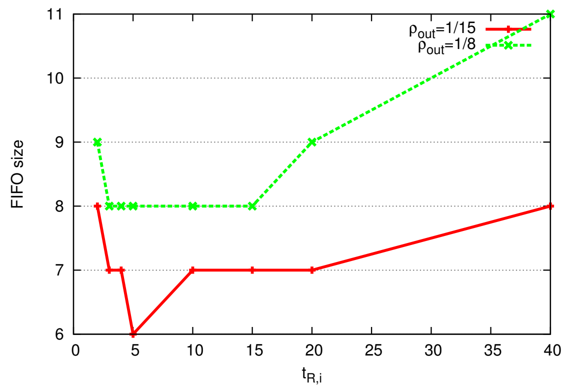
options	#1	#2	#3	#4	#5	#6	#7	#8
$t_{R,j}$	**2**	**3**	**4**	**5**	**10**	**15**	**20**	**40**
$t_{C,j}$	16	13	12	**10**	8	6	5	4
A_M	0.5	0.8	1.0	1.3	2.5	3.8	5.0	10

Table 6.1: Design options for the adaptive process p_j.

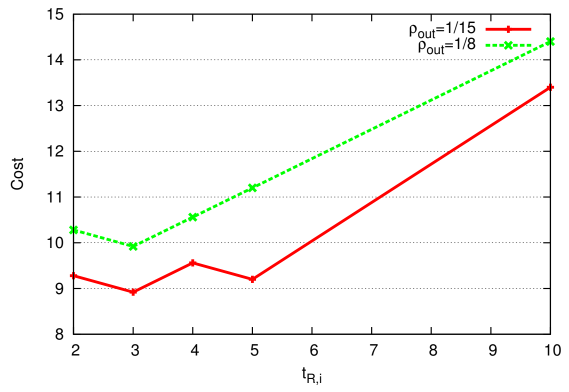
First, it is assumed that different design options have varying reconfiguration time $t_{R,j}$, as listed out in the second row of Table 6.1, but have a fixed latency $t_{C,j}$ (i.e., $t_{C,j} = 10$), and evaluate the FIFO sizes requirement. Figure 6.3a shows the minimal FIFO sizes needed upon different $t_{R,j}$, corresponding to different output throughput ρ_{out}. To consider the two design concerns $t_{R,j}$ and ρ_{out} separately,



(a) FIFO size upon varying $t_{R,j}$ & fixed $t_{C,j}$



(b) FIFO size upon varying $t_{R,j}$ & $t_{C,j}$



(c) Design cost upon varying $t_{R,j}$ & $t_{C,j}$

Figure 6.3: Experimental results of the example application.

apparently, higher ρ_{out} ($\rho_{out} = 1/10$ in the graph) demands larger FIFO sizes, so do the design options with higher $t_{R,j}$ caused by the longer computation stall during reconfigurations.

Instead, in the following scenario, the design options are chosen according to the reconfiguration properties listed out in Table 6.1. These design options show different implementation strategies in the speed and area trade-offs, e.g., an adder can be implemented as carry-lookahead adder (optimized for speed) or a ripple-adder (optimized for area). We assume $t_{R,j} = k_R \cdot A_{M,j}$ with $k_R = 10.0$ and $t_{C,j} \propto \lceil \frac{1}{\sqrt{t_{R,j}}} \rceil$. Although, higher ρ_{out} ($\rho_{out} = 1/8$) still demands larger FIFO sizes, the FIFO sizes are not monotonic to $t_{R,j}$ any more, as both $t_{C,j}$ and $t_{R,j}$ can affect the buffer requirement to sustain ρ_{out} during reconfiguration. The design options with $t_{R,j}$ close to 5 need less buffer.

Using a given compression ratio $k_C = 4.0$, the design costs are evaluated. As the design options after #5 simply show a fast monotonically increasing cost, only the design costs of design option #1-5 are shown for clarity in Figure 6.3c. Higher throughput requirement still leads to larger design costs. However, the design costs heavily depend on the $t_{C,j}$ and $t_{R,j}$ trade-off, i.e., the speed and time trade-off, and #2 with $t_{R,i} = 3$ shows the minimum cost.

6.5.2 An industrial application

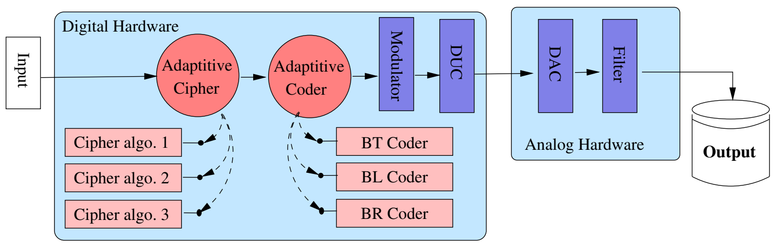


Figure 6.4: The adaptive coding and modulation case study synopsis.

An industrial adaptive coding and modulation application from Thales communication, as illustrated in Figure 6.4, is used to evaluate the potential of the proposed methodology. The diagram shows a mix of digital and analogue modules, with a channel Coder preceded by a Cipher block, and followed by a digital modulator, a digital up converter, a digital to analogue converter and an analogue filter. The experiments focus on the reconfigurable part, i.e., the Coder with 3 modes of bursts BR, BL or BT and the Cipher with 3 algorithms 1-3. The input source for the Cipher algorithm is a packet of bits, and the output stream from the Coder needs to sustain a stable throughput. The design objective in performance analysis is to minimize the buffer requirement without losing the stable output transmission, when either or both of the two modules are in reconfiguration.

The adaptive part of the abstract application model is illustrated in Figure 6.5a, in which the specification parameters are omitted for clarity. There are two adaptive processes (modules): the Cipher p_j and the Coder p_k. Each of them receives the adaptation control signal $s_{M,j}$ or $s_{M,k}$ from the environment, and can change the working modes among three possibilities (i.e., algorithm 1-3 for the Cipher, and BT/BL/BR coder for the Coder). The input date stream has a peak throughput ρ_{in}, and an average output throughput ρ_{out} is demanded.

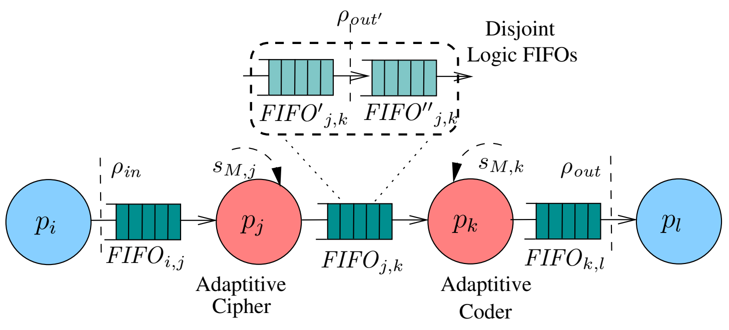
It is assumed that the reconfigurations of two adaptive Cipher and Coder are independent of each other. To decouple the analysis, the buffer $FIFO_{j,k}$ between p_j and p_k has been partitioned into two disjoint logic FIFOs $FIFO'_{j,k}$ and $FIFO''_{j,k}$, as shown in the dashed box, to be analyzed individually. From the static input (output) rates of the SDF model and ρ_{out}, the average output throughput requirement $\rho_{out'}$ of the Cipher can be derived, which is also the average input throughput to the Coder. For the Cipher, the peak input throughput and average output throughput are ρ_{in} and $\rho_{out'}$ respectively. In this way, the analysis of two reconfigurable modules can be decoupled as follows.

1. To find the minimum buffer sizes for the Cipher buffers $FIFO_{i,j}$ and $FIFO'_{j,k}$ to meet the average output throughput requirement $\rho_{out'}$ upon the peak input throughput ρ_{in} (subject to Eq. 6.3 in Constraint 17).

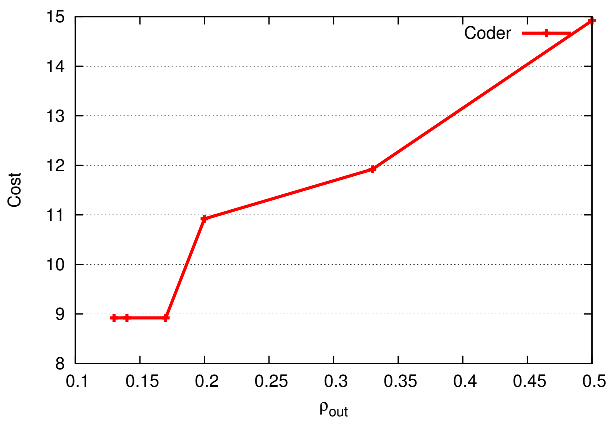
2. To find the minimum play-out buffer $FIFO''_{j,k}$ and $FIFO_{k,l}$ for the Coder module to meet the average output requirements ρ_{out} upon the average input throughput $\rho_{out'}$ (subject to Eq. 6.4 in Constraint 17).

Although the decoupled analysis (based on disjoint logic FIFOs) has the possibility to over-dimension the physical FIFO size, it is a conservative approach without restrictions on the reconfiguration of different adaptive modules, i.e., independent of reconfiguration protocols. For instance, using this analysis approach, the Cipher and Coder modules can be even reconfigured at the same time, when protocols of the reconfiguration of consecutive modules are still lacking.

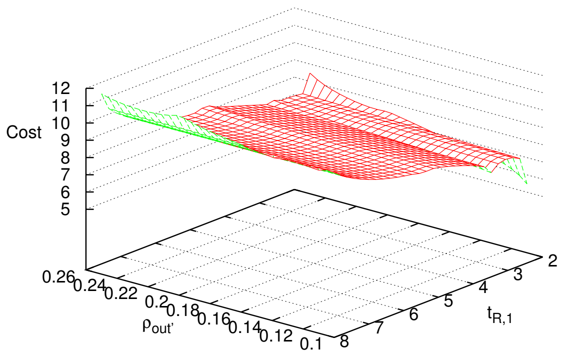
For both the Cipher and Coder, all the reconfiguration transition possibilities are traversed (i.e., $3 \times (3-1) = 6$), and the worst cases are chosen as the conservative buffer requirement.



(a) Abstract model of industrial application (with disjoint logic FIFOs in the dashed box).



(b) Design cost of JIT adaptation on the Coder.



(c) Design cost of JIT adaptation on the Cipher.

Figure 6.5: Industrial application and experimental results.

In both Coder and Cipher modules, the design cost increases with the output throughput, as shown in Figure 6.5b and 6.5c. The design costs of different implementation strategies for the Cipher with varying $t_{R,j}$ are also shown in Figure 6.5c, and the design costs are not monotonic to $t_{R,j}$. It exemplifies that the proposed reconfigurations anslysis framework suits design trade-offs analysis in design options exploration, and can be applied on a series of compositional adaptive processes (modules) as well.

Furthermore, the complexity of solution finding increases exponentially with problem size (compared with the example application). For the Coder, the solutions finding time is 247-708ms and the peak memory is 22.4-54.4MB. For the Cipher, they are 352-408ms and 27.7-36.4MB respectively.

6.6 Concluding remarks

This chapter presents a constraint based performance analysis framework for adaptive real-time streaming applications. Based on the implementation on constraint solver, the experimental results show that the proposed framework suits well reconfigurations analysis and design trade-offs analysis. It can be used to exploit the reconfigurability of adaptive real-time streaming applications, without losing efficiency. Especially, the industrial case study illustrates the capability of the proposed methodology to cope with the cascaded composition of adaptive modules.

Meanwhile, the author foresees the possibility to compact the physical buffer size of the disjoint logic FIFOs when the mode change protocols of multiple reconfigurable modules are specified. Such a technique can further reduce the design cost of adaptive systems, which remains to be the future work.

CHAPTER 7

PARETO EFFICIENT DESIGN FOR RECONFIGURABLE MULTIPROCESSOR/FPGAS

To deliver high performance streaming media applications with reduced design costs and time-to-market, there are trends in embedded system design to use combined components of multi-CPU and custom circuits in commercial off-the-shelf (COTS) FPGAs, the so called hybrid multiprocessor/FPGAs [5].

While free FPGA gates are customized as application specific reconfigurable components for performance critical functions and CPUs as execution engines for software, such COTS chips can be used as embedded platforms with high throughput and run-time reconfiguration (RTR) requests [7]. One realistic application of them, for instance, is being used in a roaming smart-phone with reconfigurable communication protocols.

Unfortunately, the design space exploration techniques and optimization methodologies on hybrid multiprocessor/FPGAs are still immature. The difficulties exit in two categories:

- While applications scheduling with resource constraints on multi-processors has been know to be NP-complete [31], the reconfiguration analysis even increases the design complexity.

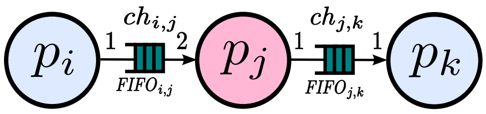
Table 7.1: Extract of Xilinx Virtex-5 FPGAs [101].

Device	BRAM (16kb)	Power PC #	CLB	
			slices	DRAM
XC5VFX30T	**136**	**1**	**5,120**	**380**
XC5VFX100T	**456**	**2**	**16,000**	**1,240**
XC5VFX200T	**912**	**2**	**30,720**	**2,280**

- To design systems that use less resources (cost) without losing performance guarantees, the decision-making over diversified COTS platforms remains difficult. For instance, the XC5VFX devices in Xilinx Virtex-5 family cover a wide range of $380 \sim 2280$KB reconfigurable logic blocks and up to two PowerPC cores [101], as illustrated in Table 7.1.

For an example adaptive synchronous data flow (SDF) application as depicted in Figure 7.1(a), the computation of p_j is reconfigurable with two working modes. Each process (working mode) has a worst case execution time (WCET) when it is partitioned as either SW or HW. While software (SW) reconfigurations correspond to process context switch on processors, the overhead (OH) is negligible compared with process execution time. The hardware (HW) reconfiguration OH to manipulate reconfigurable logics on FPGAs at run-time is non-trivial, during which the computation is stalled. Assuming a specified number of processors on each chip, the implementation cost of each process is either the size of SRAM storage for SW or the area of logic blocks and (reconfiguration) memory for HW. In Figure 7.1(b), the design specifications (normalized) to implement the illustrative application on a single-CPU/FPGA platform are exemplified.

Motivation. Given the example application and its design specifications in Figure 7.1, three design options and their reconfiguration scheduling are explored and analyzed in the following. With design option A (p_i and p_j partitioned to HW and p_k to SW), the schedule is illustrated in Figure 7.2(a). The time range of the scheduling evolves horizontally, while the process and FIFO status are listed out vertically, i.e., at each time tag a process in *executing* state has a number to denote the remaining execution time slots, a FIFO status is denoted as the using storage space in tokens, and processes with *stalled* computation or FIFOs not used have otherwise blank status. At the beginning, the application with the adaptive process p_j working in mode A (with WCET 2 slots) can enter a periodic phase



(a) Example application model
(p_j is reconfigurable)

	HW reconfig. OH	Cost SW	Cost HW	WCET SW	WCET HW
p_i	-	-	1	-	1
p_j	2	1	2	3(2)	2(1)
p_k	-	1	2	2	1

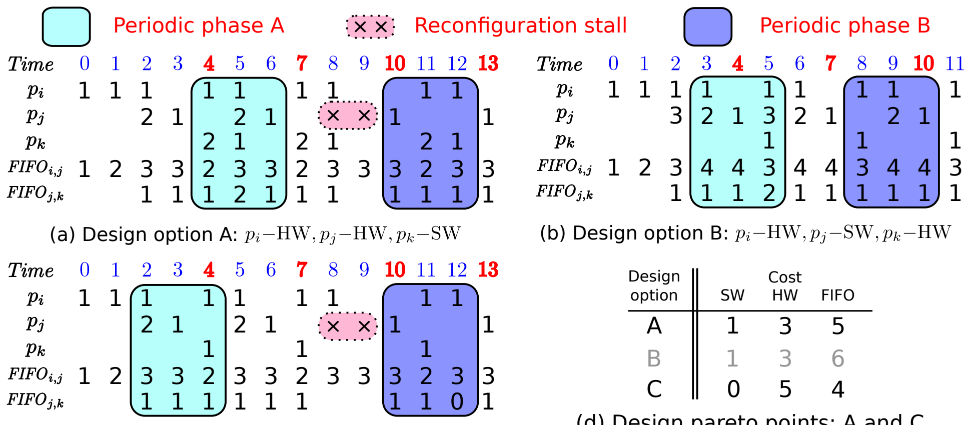
(b) Design specifications
(p_j has two working modes)

Figure 7.1: An example reconfigurable SDF application model and its design specifications on a CPU/FPGA platform.

from time tag 4 to 6. From time tag 8, p_j starts the reconfiguration[1] with an OH 2 and switches to working *mode B* (with WCET 1 slot). Again, the schedule enters another periodic phase from time tag 10 to 12. The application throughput is guaranteed in both periodic phases, since all processes have the same execution iterations in a time period 3. In addition, the application throughput is guaranteed even during mode transition, e.g., between time tags 4, 7, and 10 the sink process p_k always executes once. For each FIFO, the required FIFO size is the maximal buffer usage in scheduling. Similarly, the reconfiguration scheduling of two other design options are illustrated in Figure 7.2(b-c), in which p_j is implemented in SW in design option B with neglected reconfigurable OH. Apparently, all design options have the same application throughput guarantees.

Accordingly, the SW/HW cost, and FIFO buffer requirement of all design options are evaluated in Figure 7.2(d). Although these quantities have a partial order to indicate the precedence of design options, A dominates B in the sense that the former one is always preferred with not worse (bigger) evaluations on all criteria. The set of optimal design options A and C are called Pareto points [35], i.e., either one has at least one criteria better than the other (A has less HW cost, and B has

[1]For clarity, the reconfiguration time slot is fixed at 8. But, the method to be presented in Section 7.3 is more general and fits run-time reconfigurations.



(a) Design option A: p_i−HW, p_j−HW, p_k−SW

(b) Design option B: p_i−HW, p_j−SW, p_k−HW

(c) Design option C: p_i−HW, p_j−HW, p_k−HW

Design option	Cost SW	Cost HW	Cost FIFO
A	1	3	5
B	1	3	6
C	0	5	4

(d) Design pareto points: A and C
(A dominates B)

Figure 7.2: Example application scheduling (p_j has two working modes A and B) and design Pareto points.

less SW and FIFO cost). In this chapter, Pareto points will be used to indicate the trade-offs of multiple criteria in design space exploration, which can not be optimized independently. However, a systematic way to calculate design Pareto points in reconfigurable streaming applications is still lacking.

As the **contribution in this chapter**, a new Pareto efficient design framework is proposed for reconfigurable SDF streaming applications on hybrid multi-CPU/FPGAs platforms. While the SW cost, HW cost and buffer requirement are characterized as multiple objectives, the pruned Pareto-optimal points found can be used in cost-efficient selection from variably priced CPU/FPGA platforms.

7.1 Related work

In [91], simulation based techniques have been introduced for the analysis of different performance metrics of scenario-aware SDF models with stochastic mode changes. Schedulability analysis and reconfiguration methods for multi-mode (adaptive) real-time systems has been studied in [83, 76], where each mode consists of different tasks. They develop mode change protocols in mode transition stages, and exploit analysis techniques to ensure that no deadlines are violated during the transition periods. On RTR hybrid CPU/FPGAs, Yuan et al. [103] address SW/HW partitioning and scheduling of task graphs with the objective of maximizing application throughput. All these work do not address multi-dimensional optimization on platform resources, as in this chapter.

An inspiring Pareto calculator has been proposed for the optimization of multi-dimensional space of attributes [35], which is a tool for general compositional computations. It has been used to calculate optimized design options for MPEG-4 media on mobile devices via a wireless connection, with the trade-offs on the quality of the video, energy consumption and transmission latency. Based on the author's previous work on performance analysis of SDF applications on reconfigurable FPGAs in Chapter 6 a design Pareto-point calculation flow for SW/HW and buffer cost efficient design is proposed for SW/HW allocation, scheduling, and run-time reconfiguration analysis on multiprocessor/FPGA platforms.

7.2 Application and Architectural assumptions

In the multiprocessor/FPGA computing platforms as illustrated in Figure 2.4, there are CPUs dedicated to SW processes and custom circuits to HW processes. While the cost of memory (SRAM) and custom circuits on FPGAs is determined by application allocation decision, the buffer requirement depends heavily on scheduling policies (RTOS). Given a reconfigurable process allocated as HW, new functionality can be loaded from the configuration memory into the configuration slot (reconfigurable circuits in FPGA) specified by the reconfiguration control. Since this loading takes time (reconfiguration stall), the system timing might be violated without reconfiguration analysis. On the other hand, processes improperly implemented as SW might degrade system performance, especially when the number of processors is limited. It is critical to find the Pareto efficient design options while the throughput guarantees are met during the run-time reconfiguration transitions.

The Pareto efficient design on partially RTR multiprocessor/FPGAs is based on the following assumptions:

- The reconfiguration of one process does not interfere the working of other running processes on such a partially RTR CPU/FPGAs platform.
- The hard-core multi-processors are *homogeneous*, on which each process has the same WCET being mapped onto different processors, e.g., PowerPC on Virtex-5.
- The cost to implement processes as either SW or HW is known at design-time in specifications.

7.3 Pareto efficient design framework

Here, the constraint based Pareto efficient design framework is proposed, which includes application allocation, scheduling, and Pareto points calculation, besides reconfiguration analysis.

7.3.1 Allocation and mapping

A set of boolean decision variables μ_i denote whether each process p_i is allocated to CPUs ($\mu_i = 0$) or FPGAs ($\mu_i = 1$). Assuming different instances (computation iterations) of a process are only allocated onto one processor, a set of boolean variables $\alpha_{i,\mu p_n}$ indicate the presence of p_i on the specified processor μp_n, with $\alpha_{i,\mu p_n} \equiv 0$ for processes allocated to FPGAs. Processes allocation and mapping can be formalized as the following.

Constraint 20. *(Allocation & mapping) While each process p_i can be allocated to CPUs or FPGAs, a process allocated to CPUs needs one (and only one) processor for different process instances.*

$$\sum_{\mu p_n \in \boldsymbol{U}} \alpha_{i,\mu p_n} = \neg \mu_i, \quad \forall p_i \in \mathbf{P} \tag{7.1}$$

in which **P** *is the set of processes in application models, and* **U** *is the set of processors in the architecture platform.*

The mapping problem on homogeneous multi-processors contains symmetries, i.e., for some mapping decisions, there are duplicated equivalent solutions in the searching space. Thus, a stronger restriction (Eq. 7.2) is applied to order the processors in allocation to exclude revisiting symmetrically equivalent states.

$$\sum_i 2^i \alpha_{i,\mu p_n} \geqslant \sum_i 2^i \alpha_{i,\mu p_{n+1}}, \quad \forall \mu p_n, \mu p_{n+1} \in \mathbf{U} \tag{7.2}$$

in which the equality holds when neither of the consecutive processors μp_n and μp_{n+1} has processes allocated.

7.3.2 Extended execution semantics

Here, the execution semantics of SDF applications (see Section 4.4.3) based on cumulative event models is extended to be *allocation* aware. Besides the allocation decision variables μ_i, we use a boolean function $\xi(t)$ to denote the working mode at time tag t of the reconfigurable process p_j, i.e., $\xi(t) = 0$ indicates that p_j works in mode A , otherwise p_j is in (or being reconfigured to) mode B. For instance, the process computation latency is formulated in the following constraint.

Constraint 21. *(Computation latency) While each process can be allocated to CPUs or FPGAs, its WCET in implementation is denoted as $t_{C_{sw},j}$ or $t_{C_{hw},j}$ with the following constraints.*

$$C_{i,j}(t + t_{C,j}) = C_{i,j}(t) + m_{i,j}K_j(t), \quad {\scriptstyle \forall K_j(t)\in\{0,1\}} \tag{7.3}$$

$$where \quad t_{C,j} = \neg\mu_j t_{C_{sw},j} + \mu_j t_{C_{hw},j} \tag{7.4}$$

in which $K_j(t)$ denotes the incremental properties of $C_{i,j}(t)$. Especially, for a reconfigurable process p_j, the WCET varies from $t^A_{C_{sw},j}(t^A_{C_{hw},j})$ to $t^B_{C_{sw},j}(t^B_{C_{hw},j})$ during the reconfiguration from working mode A to B. Thus, $t_{C,j}$ in Eq. 7.3 can be defined:

$$\begin{aligned} t_{C,j} =& \neg\xi(t)(\neg\mu_j t^A_{C_{sw},j} + \mu_j t^A_{C_{hw},j}) + \\ & \xi(t)(\neg\mu_j t^B_{C_{sw},j} + \mu_j t^B_{C_{hw},j}) \end{aligned} \tag{7.5}$$

However, the constraints in Eq. (7.3-7.5) can not be implemented in Gecode solver directly and need to be rewritten. For instance, Eq. (7.3-7.4) are equivalent to the following constraint applicable in Gecode.

$$\begin{aligned} &\neg\mu_j C_{i,j}(t + t_{C_{sw},j}) + \mu_j C_{i,j}(t + t_{C_{hw},j}) \\ &\quad = C_{i,j}(t) + m_{i,j}K_j(t), \quad \forall K_j(t) \in \{0,1\} \end{aligned} \tag{7.6}$$

Without being explicitly mentioned in this chapter, $t_{C,j}$ has the same definitions as in Eq. (7.3-7.5) to be allocation decision and reconfiguration (for the reconfigurable process) aware. Similarly, other SDF execution semantics on computation and buffer resources can be extended, which are omitted for clarity.

Furthermore, the process scheduling with computation and buffer resource constraints is refined to be aware of the allocation and mapping decisions.

Constraint 22. *(Allocation & scheduling association) While the processes allocated to FPGAs are concurrent in scheduling, the processes allocated to one single processor can only execute sequentially. This mapping and scheduling association is formalized as:*

$$\sum_{p_j \in \boldsymbol{P}} \alpha_{j,\mu p_n} W_j(t) \in \{0,1\}, \ \forall \mu p_n \in \boldsymbol{U} \tag{7.7}$$

$$W_j(t) = \sum_{\Delta_t} \tfrac{C_{i,j}(t+\Delta_t+1) - C_{i,j}(t+\Delta_t)}{m_{i,j}}, \quad {\scriptstyle \Delta_t\in[0,t_{C,j})} \tag{7.8}$$

in which $W_j(t)$ denotes the 1-0 *(computing or stalling) status of process p_j at time tag t on CPUs or FPGAs, and $t_{C,j}$ has the same definitions as in Eq. (7.3-7.5) to be allocation decision and reconfiguration aware.*

7.3.3 Design Pareto points calculation

In design specifications, the cost to implement each process p_i in SW (if feasible) on CPUs is measured as the code size π_i^{SW} in SRAM or the area of custom circuits π_i^{HW} in HW. The only unit used for HW area cost is the number of logic elements (LE). The memory cost for a configuration stored in the configuration memory is thus calculated as the equivalent LE cost which is technology dependent. The SW and HW implementation cost on the platform are defined.

Property 6. *(SW & HW implementation cost) The process allocation determines the total SW cost π_{Sum}^{SW} and total HW cost π_{Sum}^{HW}.*

$$\pi_{Sum}^{SW} = \sum_{p_i \in \boldsymbol{P}} \neg\mu_i \cdot \pi_i^{SW}, \ \pi_{Sum}^{HW} = \sum_{p_i \in \boldsymbol{P}} \mu_i \cdot \pi_i^{HW} \tag{7.9}$$

Assuming different FIFOs are implemented disjointly, the buffer cost on the platform is formalized.

Property 7. *(Buffer cost) In scheduling, the buffer cost in total corresponds to the sum of the maximal buffer usage of all FIFOs.*

$$\gamma_{Sum} = \sum_{\forall ch_{i,j}} \max_{\forall t} B'_{i,j}(t) \tag{7.10}$$

To evaluate the quality of design options in solutions finding, a constraint on SW/HW implementation cost and buffer requirement is formulated to prune design options.

Constraint 23. *(Design options pruning) Given a set of design Pareto points $ParetoSet$ found in design space exploration. Besides meeting all other constraints formulated above, a solution sol should not be dominated ($\nprec$) by any Pareto points ϕ.*

$$\begin{aligned} sol \nprec \phi \Longleftrightarrow & sol.\pi_{Sum}^{SW} < \phi.\pi_{Sum}^{SW} \vee sol.\pi_{Sum}^{HW} < \phi.\pi_{Sum}^{HW} \\ & \vee sol.\gamma_{Sum} < \phi.\gamma_{Sum}, \quad \forall \phi \in ParetoSet \end{aligned} \tag{7.11}$$

in which $sol \nprec \phi$ always holds when $ParetoSet = \emptyset$.

To maintain a set of Pareto points in design space exploration, a calculation flow is presented as in Algorithm 1. In line 4, while all the formulated constraints are used in solutions finding, a $ParetoSet$ is dynamically maintained[2] during exploration and used in Constraint 23. From line 8 to 10, if a sol dominates ($\prec$) a current Pareto point, the dominated design option is moved into another $DominatedSet$.

Algorithm 1: Design Pareto-point calculation flow.

```
   Output: $ParetoSet$
1  $ParetoSet \longleftarrow \emptyset$;
2  $DominatedSet \longleftarrow \emptyset$;
3  /* A dynamic $ParetoSet$ is used in Constraint 23                 */
4  while $(sol = solutionsFinding(ParetoSet)) \neq Null$ do
5      if $|ParetoSet| > 0$ then
6          for $k \leftarrow 1$ to $ParetoSet.size$ do
7              /* If sol dominates a Pareto point                    */
8              if $sol \prec ParetoSet.at(k)$ then
9                  $DominatedSet.insert(ParetoSet.at(k))$;
10                 $ParetoSet.erase(k)$;
11     $ParetoSet.insert(sol)$
```

Discussion. The SW/HW and buffer cost in the proposed formulation are based on high-level estimations, e.g., buffer cost based on symbolic token units. However, it is possible to extend this method within a practical design flow, once these vendor and technology dependent factors can be formalized as design constraints. For instance, assuming FIFO buffers are implemented as block RAM (BRAM), a more practical buffer cost γ_{Sum}^{BRAM} can be re-formalized from Eq. 7.10.

$$\gamma_{Sum}^{BRAM} = k_{BRAM} \sum_{\forall ch_{i,j}} d_{BRAM} \lceil \frac{\max_{\forall t} B'_{i,j}(t)}{d_{BRAM}} \rceil \tag{7.12}$$

in which k_{BRAM} and d_{BRAM} are the cost factor and depth of BRAM.

[2]To our best knowledge, it is infeasible with ILP modeling techniques.

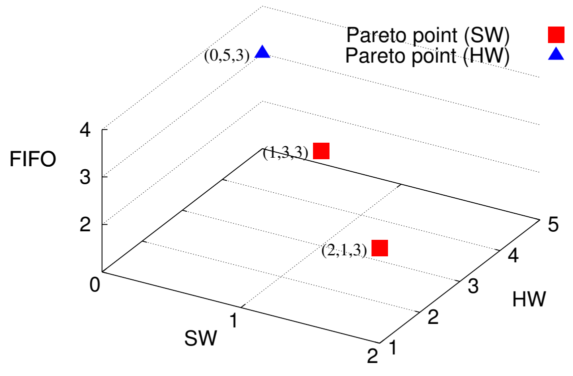
7.4 Experimental results

To demonstrate the potential in Pareto efficient design, the methodology has been used on a *Cd2dat* [13] application from the media domain, and a *Wireless* [69] application from the communication domain, besides the example application Figure 7.1(a).

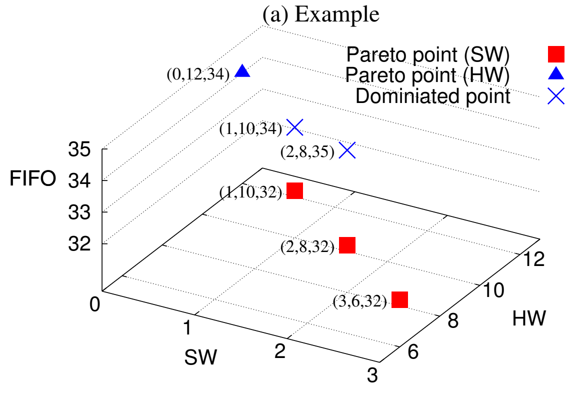
A reconfigurable FPGA platform with two processors, e.g., Xilinx Virtex-5 FPGAs, is adopted. The design specifications of different SDF applications are presented in Table 7.2. For each application, the number of SDF processes and the equivalent HSDF process number are listed out. Especially, part of the SDF processes have been pre-allocated on the platform, either as SW or HW modules. In each application, one process is specified to be reconfigurable with two working modes. Assuming the same specified (feasible) application throughput requirement needs to be sustained (not only in different working modes but also in reconfiguration phases), the application allocation, scheduling, and reconfiguration analysis between mode transitions are explored. However, within the scope of this chapter, the impacts of varying application throughput on design Pareto points are not investigated.

The peak memory and running time on the experimental workstation have been measured in the solver, which are shown in Table 7.2 as well. The memory and time usage increase exponentially with the problem size. The proposed method has been computation efficient to solve the NP-complete allocation and scheduling problem with a reasonable problem size, e.g., in 1.9s for the small example case and 105s for *Cd2dat*. The design Pareto points for different application specifications are illustrated in Figure 7.3. For the example application, Pareto points with even less FIFO cost than in Figure 7.2 can be found, since a lower throughput requirement has been specified in Table 7.2 (0.2 instead of 0.33). To distinguish whether to implement the reconfigurable process in SW or HW in the graph, the Pareto points are marked as ‘Pareto point (SW)’ or ‘Pareto point (HW)’ respectively. The buffer costs found are very tight[3], in the sense that for each application the minimal buffer cost in all Pareto points is the same as the minimal dead-lock avoiding bounds [36]. Another set of temporal Pareto points $DominatedSet$ dominated by new solutions found in design space exploration (line 8 to 10 in Algorithm 1) are presented. However, more design options which are pruned by Pareto points are discarded. For instance, 263 failure nodes are pruned for *Wireless*, while each node corresponds to a set of

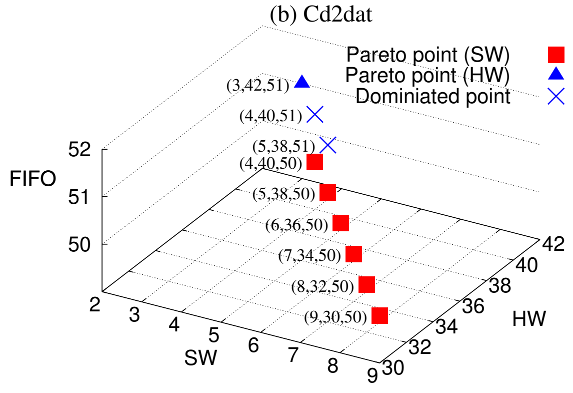
[3]They are throughput and reconfiguration stall $t_{R,j}$ relevant. For *Cd2dat* and *Wireless*, we specify $t_{R,j}$ to be 1-2 orders-of-magnitude higher than $t_{C,j}$.



(a) Example



(b) Cd2dat



(c) Wireless

Figure 7.3: Design Pareto points for different applications.

Table 7.2: Design specifications and experimental results.

application	*process #* [a]		*pre-allocation* [b]			*thru. req.*	*mem.* [c]	*time* [d]
	SDF	*HSDF*	*SW*	*HW*	*SW/HW*			
Example	3	4	0	1	2	0.2	1.29e3	1.90e3
Cd2dat	6	612	0	3	3	0.5	4.18e6	1.05e5
Wireless	24	32	3	15	6	0.028	3.80e6	1.94e4

[a] The number of SDF processes and equivalent HSDF processes.
[b] The number of processes pre-allocated as SW or HW on the platform.
[c] The peak memory consumption (KB) in solutions finding.
[d] The solutions finding time (ms).

pruned design options.

Memory issues. In the experiments, the peak memory usage increases dramatically with the problem size, e.g., $4.18e6$KB (very close to the 4GB RAM capacity on our workstation) is used for *Cd2dat*. On the other hand, memory usage up to 2GB has also been reported in a model checking method in [36], in which a relatively simpler problem (only the scheduling of SDF applications) is considered. In another experiment, the processes allocation is fixed to either SW or HW (i.e., process number allocated to SW/HW is 0 in Table 7.2) for *Wireless*. A Pareto point has been found in 2.1s with peak memory 414MB. Thus, it might be possible to use heuristics in the constraint based techniques to improve the searching efficiency, in terms of computation time and memory usage.

7.5 Concluding remarks

In this chapter, a Pareto efficient design method for reconfigurable streaming applications on off-the-shelf CPU/FPGA platforms has been presented. The problem is formulated as constraint based application allocation, scheduling, and reconfiguration analysis. A design Pareto-point calculation flow for SW/HW and buffer cost is implemented on a public domain constraint solver Gecode [32], and is exemplified by two case studies from different application domains.

CHAPTER 8

PROTOTYPING OF STREAMING APPLICATIONS ON MPSOCS

Synchronous data flow (SDF) models can be scheduled statically, which makes their run-time implementations efficient [60]. In Chapter 3-7, the author has proposed performance analysis and design space exploration frameworks for predictable streaming applications on multiprocessor systems-on-chip (MPSoCs) platforms. Using the optimized design options and scheduling policies, the designer can get even higher efficiency on the non-functional properties of streaming applications, in particular on real-time performance, energy dissipation, and buffer requirements. Although both simulation and formal analysis based approaches in this thesis have shown the potential of compile time optimization techniques, a realistic projection on the final implementation has the added advantage to make the results in theoretical studies more meaningful. In this chapter, prototyping stream processing systems on FPGA is used as such a compelling method, to build a physical platform that entails the designer to evaluate various properties of a design.

FPGA is an integrated circuit designed to be configurable after manufacturing, which facilitates the deployment of a multiprocessor platform on a single chip [50]. In this chapter, to ease a rapid prototyping, the Stratix II EP2S60 FPGA in the Altera family [2] is used as the programmable device. With the inherent flexibility to integrate existing intellectual property (IP) components, such as embedded microprocessors, DSP, and interconnect fabric modules, a modern Stratix II FPGA has simplified the design process of a MPSoC architecture with on-chip interconnection in this chapter. Such a prototype provides a demo platform to fill in the gap between system level design methodology and final implementation, and it is used to validate the compile time design optimization methods on predictable streaming applications design.

8.1 MPSoC architecture platform

8.1.1 Architecture template

The architecture template of the tile based [8, 24] MPSoC is illustrate in Figure 8.1. A logic tile consists of processor(μp), instruction memory (*I-mem*), data memory (*D-mem*), FIFO buffer, and communication assist (CA). The processor has SRAM based local *I-mem* and *D-mem*. As a single-cycle accessing time is needed for the processor to access *I-mem* or *D-mem*, no cache is needed. The explicit FIFO channels between different computation processes of streaming applications are allocated onto a local data buffer, and multiple FIFOs on the same tile share a physical buffer in a disjoint way. A CA is a controller that assists the inter-tile incoming and outgoing communication operations. That is, each tile is connected with others via CA across the on-chip interconnection.

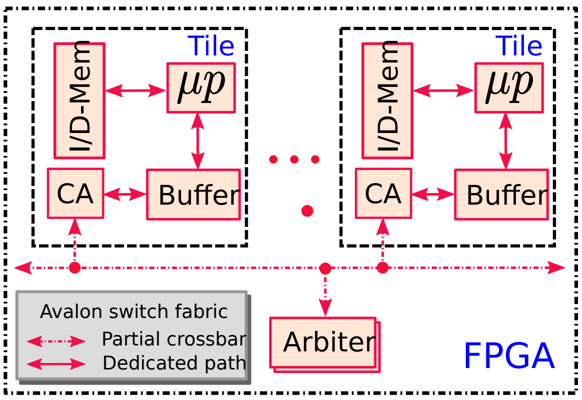


Figure 8.1: Architecture template of the tile based MPSoC on FPGA.

8.1.2 FPGA based MPSoC

Based on Altera Quartus II tool suite to configure Stratix II FPGA, various IP cores can be used to build the prototyping architecture platform, including vender pre-verified cores and custom cores.

- The processor is the Nios II, which is a 32-bit RISC soft-core designed specifically for Altera FPGAs with a Harvard architecture, i.e., it has separated instruction and data buses.
- The *I-mem*, *D-mem*, and the circular FIFO based data buffer are all implemented in SRAM on-chip memory. Each Nios processor has dedicated connection from master ports to the slave ports of each module with high memory bandwidth.
- CA is a custom core employed for run-time flow control. It moves data tokens from the producer processes to consumer processes, when their respective output-side FIFOs and input-side FIFOs are mapped onto different tiles. The *communication* performed via CAs is decoupled from the *computation* performed on processors by the asynchronous FIFO buffers.
- The JTAG UART core provides the way for programming and debug on the FPGA device via the serial communication connected to the host PC.
- Avalon switch fabric provides a collection of interconnect and logic resource to manage the connection of different components. Here, a common Avalon memory-mapped (Avalon-MM) interface is used for the read/write interfaces on master/slave components. Compared with traditional bus, it provides flexibility and supports multiple masters operations simultaneously, using a slave-side arbitration scheme.

In the FPGA based MPSoC in Figure 8.1, besides the dedicated path in the intra-tile interconnect, the inter-tile interconnect (in dashed line) is based on partial crossbar switch, in which each master has the connectivity to a subset of slaves. When multiple master ports contend for a single slave port, an arbiter grants fare access in a round-robin order with minimum bandwidth guarantee.

8.1.3 Communication model

When an inter-tile logic channel is implemented as a shared Avalon inter-tile interconnection, the communication delay can be captured by the latency-rate server

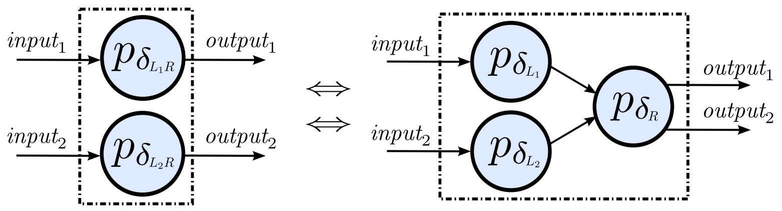


Figure 8.2: Refinement of two communication processes $p_{\delta_{L_1 R}}$ and $p_{\delta_{L_2 R}}$ based on the latency-rate server model [86].

model [86]. Given two such channels sharing the same communication interconnection, their point-to-point inter-tile communication delay can be modelled by process $p_{\delta_{L_1 R}}$ and $p_{\delta_{L_2 R}}$ respectively on the left side of Figure 8.2. The refined delay model consists of two parts, as illustrated on the right side of the graph.

1. The constant contributions from interconnection by process $p_{\delta_{L_1}}$ and $p_{\delta_{L_2}}$. They are the same as regular computation processes with specified computation latency (see Section 4.4.3).

2. The variable contributions caused by communication resource sharing is explicitly captured by p_{δ_R}, with bounded communication bandwidth as formalized in Constraint 24.

Constraint 24. *(Communication bandwidth) For all the logic channels sharing the same inter-tile interconnection (shared server p_{δ_R}), the bounded communication bandwidth ω enforces extra constraints on the service function $C_{\delta_{L_x},\delta_R}(t)$ of each channel as the following.*

$$\sum_{\forall p_{\delta_{L_x}}} (C_{\delta_{L_x},\delta_R}(t+1) - C_{\delta_{L_x},\delta_R}(t)) \leqslant \omega \tag{8.1}$$

8.2 Run-time implementation

Based on a static application to architecture mapping, the run-time implementation process is the scheduling of streaming applications on MPSoCs. In this thesis, the applications do not allow run-time process migration on the platform.

8.2.1 Scheduling strategies

The scheduling of SDF applications on MPSoCs involves both processes execution *order* and *start-time* assignments on each processor. Different implementation strategies are distinguished from the ways to carry out these two assignments, either in a fixed or dynamic manner, as illustrated in Table 8.1 [8].

Table 8.1: Different run-time implementation strategies.

Implementation strategies	**Fixed order?**	**Fixed start-time?**
Full static	Yes	Yes
Static order	Yes	No
Run-time order	No	No

To reduce the run-time computation overhead, it is preferred that both assignments could be fixed at compile-time. However, a fully static schedule is volatile to the varying computation or communication delays at run-time, which makes it infeasible for distributed systems. In this chapter, two run-time scheduling strategies, *static order* and *run-time order* [8], are exploited. Both do not require a global notion of time or clock, when such a global time is not easily known on MPSoCs.

As a subset of Kahn process networks [54], SDF models are deterministic and the model will produce the same output values for the same input streams without being affected by the timing semantics of data streams, i.e., without being affected by the varying run-time *computation latencies*. When processes execute in a data driven manner on a multiprocessor architecture, the execution requirement on the exact start-time of processes is relaxed.

- In the static order scheduling, the order in which processes execute in each processor is determined at compile time such that all data precedence constraints are met.

- In the run-time order scheduling, the sequential execution oder on each processor is determined at run-time based on a static checking list in priority queue, similar as in list scheduling [29].

In each processor, the processes execute consecutively in a non-preemptive manner according to the static assignment, and no operating system is needed. Since the producer-consumer synchronization is performed at run-time, both scheduling strategies are robust with respect to variances in *communication latencies* as well.

Based on the optimization approaches proposed in this thesis, the static order strategy can be optimized subject to the design objective, which is to minimize the buffer cost according to the specified application throughput requirement. On the other hand, the run-time order scheduling, without optimization on buffer and timing constraints, is used as a reference method.

8.2.2 Scheduling algorithms

Without losing generality, the run-time scheduling algorithms are exemplified by an example 4-stage pipelined application mapped onto a 2-core MPSoC platform as shown in Figure 8.3.

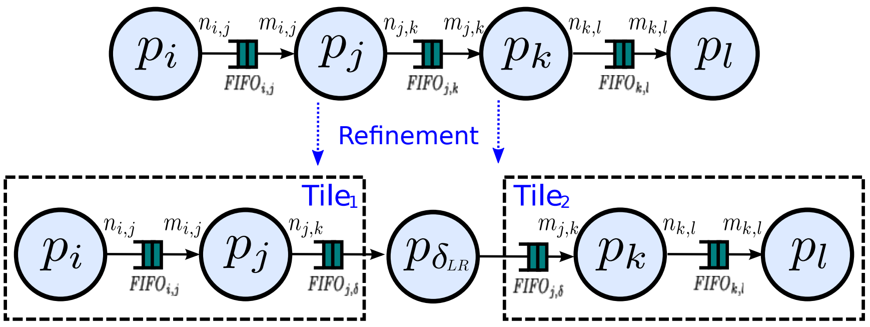


Figure 8.3: Mapping and refinement of a 4-stage pipelined application onto a 2-core MPSoC platform, in which the communication delay is captured by process $p_{\delta_{LR}}$ based on latency-rate model.

Since process p_j and p_k are mapped onto different tiles, the buffer $FIFO_{j,k}$ on the inter-tile channel is decomposed into two on-tile buffers $FIFO_{j,\delta}$ and $FIFO_{\delta,k}$, and the communication delay is captured by $p_{\delta_{L_1R}}$ as a latency-rate server model.

Given a user specified order or priority, the static order and run-time order schedules on each tile can be constructed respectively, as illustrated in the scheduling Algorithm 8.2.1-8.2.4. In the run-time order scheduling, a process executes as soon as it is enabled when its input tokens become available and there are output buffer space. In both strategies, when the producer and consumer processes of a single logic channel are mapped onto different tiles, the inter-tile message passing uses communication primitives for synchronization, and the producer or consumer process needs to check at run-time for input data token availability or output FIFO capacity.

Algorithm 8.2.1: STATICORDERTILE1(c)

```
while (1)
  do { $p_i()$;
       if ($\gamma_{j,\delta} - B'_{j,\delta}(t) \geqslant n_{j,k}$)
         then $p_j()$;
```

Algorithm 8.2.2: STATICORDERTILE2(c)

```
while (1)
  do { if ($B_{j,k}(t) \geqslant m_{j,k}$)
         then $p_k()$;
       $p_l()$;
```

Algorithm 8.2.3: RUNTIMEORDERTILE1(c)

```
while (1)
  do { if ($\gamma_{i,j} - B'_{i,j}(t) \geqslant n_{i,j}$)
         then $p_i()$;
       if ($B_{i,j}(t) \geqslant m_{i,j}$ and $\gamma_{j,\delta} - B'_{j,\delta}(t) \geqslant n_{j,k}$)
         then $p_j()$;
```

Algorithm 8.2.4: RUNTIMEORDERTILE2(c)

```
while (1)
  do { if ($B_{j,k}(t) \geqslant m_{j,k}$ and $\gamma_{k,l} - B'_{k,l}(t) \geqslant n_{k,l}$)
         then $p_k()$;
       if ($B_{k,l}(t) \geqslant m_{k,l}$)
         then $p_l()$;
```

8.3 Experimental results

To evaluate the scheduling capability of static order and run-time order strategies, a prototype of the *cd2dat* application [13] on a 3-core MPSoC platform is built. The application has 6 pipelined processes, with the process to platform mapping illustrated in Figure 8.4.

For both strategies, the application C code is cross compiled for release using the optimization level “-Os” to optimize for size. The constraint based method proposed in this thesis is used to optimized the order in the static order scheduling, and the run-time order scheduling with user specified priority queues is used as

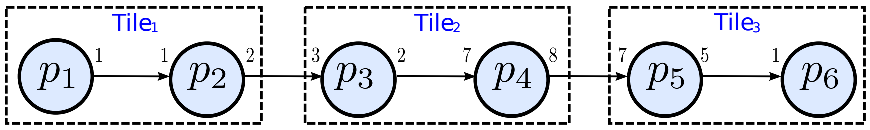


Figure 8.4: Mapping of the *cd2dat* application [13] onto a 3-core MPSoC platform.

Table 8.2: Comparison of different scheduling strategies.

$\mathcal{F}$[c]	**Static order**		**Run-time order A**[a]		**Run-time order B**[b]	
	buffer	*throughput*	*buffer*	*throughput*	*buffer*	*throughput*
10	34	115.86	34	75.844	67	76.092
100	34	55.533	34	51.926	43	51.980
1000	34	8.7086	34	8.6485	40	8.6513

[a] It is using run-time order scheduling with reduced buffer cost.
[b] It is uisng run-time order scheduling with big enough buffer.
[c] It is a factor to specify the computation latency of processes.

reference method. In the experiments, the control logic overhead is only scheduling strategy relevant once the application is defined. However, the process computation latencies imitated by dummy operations can vary and are proportional to a factor $\mathcal{F}$.

To compare different scheduling strategies, the experimental results are shown in Table 8.2. For the run-time order scheduling, two variations are used. That is, 'Run-time order A' corresponds to the strategy with the same bounded buffer space and buffer usage as in 'Static order', and 'Run-time order B' corresponds to the strategy with big enough buffer space[1]. In the table, *buffer* denotes the total buffer usage during run-time, and *throughput* is the guaranteed application throughput obtained. Compared with run-time order scheduling, the static order scheduling optimized by the constraint based scheduling method proposed in this thesis can achieve improved throughput with minimized buffer cost. Especially, when the factor of computation latency is small ($\mathcal{F} = 10$), it can get up to 153% throughput gains and down to 51% buffer cost reduction.

[1]Each buffer space is three times to the one in 'Static order', but the run-time buffer usage is reported in Table 8.2

CHAPTER 9

CONCLUSIONS AND FUTURE WORK

9.1 Conclusions

Driven by the increasing capacity of integrated circuits, parallelism has been a global architectural theme in embedded computing systems. Multiprocessor systems-on-chip (MPSoCs), consisting of multiple components of computation, storage, and communication elements on the same silicon, are increasing widely used in modern consumer electronics devices. With the enhanced computation power, program-ability, and reduced communication cost, MPSoCs are suitable for concurrent streaming applications ranging from multimedia, digital singla processing (DSP), and telecommunication to network processing domains. However, it is non-trivial to harness the inherent high processing power effectively, when there are both application specific performance demands and implementation (area, energy, and cost) efficiency needs. A systematic way to design efficient streaming applications on embedded MPSoCs in an early design phase is thus required.

This thesis explores the performance analysis and implementation methodologies of predictable streaming applications on these MPSoCs computing platforms. The functionality and application concurrency are described in synchronous data flow (SDF) computational models, and two state-of-the-art architecture templates with implementation parallelism are proposed as heterogeneous multiprocessor architectures, i.e., network-on-chip (NoC) based MPSoC and hybrid reconfigurable multiprocessor/FPGA platforms. Based on the author's contributions on simulation and formal analytic methods, both modelling frameworks and design space exploration workflows for embedded MPSoCs architectures have been proposed.

In Chapter 3, a *simulation* based energy efficient design exploration flow is proposed for streaming applications with guaranteed throughput on NoC based MPSoCs. Both application throughput analysis and system energy calculation have been carried out on a multi-clocked synchronous modelling framework. The degrees of customizability of both processor *voltage-frequency* levels and memory *size*s have been leveraged to investigate the minimal energy consumption of streaming applications. In experiments, heuristic search (greedy and Taboo) algorithms are used to find efficient design options in terms of total energy dissipations.

In Chapter 4, a *formal analytic* scheduling framework for for real-time streaming applications with minimal buffer requirement on hybrid CPU/FPGA architectures is proposed. Based on event models of data streams, the problem has been formalized declaratively as constraint base scheduling, and solved by a public domain constraint solver *Gecode* [32]. Experiments have shown the capability of the proposed approach in constructing schedules with high (feasible) throughput guarantees and minimized buffer requirement.

Consecutively, the constraint based analytic method has been extended in the following chapters.

- In Chapter 5, a global scheduling and contention-free routing framework for NoC based MPSoCs is addressed.

- Based on adaptive extensions on SDF semantics and iterative timing phases, a performance analysis framework is proposed in Chapter 6 for adaptive real-time streaming applications on run-time reconfigurable FPGAs.

- In Chapter 7, a design Pareto-point calculation flow is exploited for multi-dimensional design optimization, with optimized buffer requirement and software/hardware implementation cost on run-time reconfigurable multi-processor/FPGA platforms.

Finally, a prototype of stream processing systems on FPGA based MPSoCs is built in Chapter 8 as a realistic projection on the final implementation, to make the results of theoretical studies in this thesis more meaningful.

9.2 Future work

Although several issues on performance analysis and implementation of predictable MPSoC computing systems have been address in this thesis, like any other research projects, there are more unfulfilled research problems, which are worthy to be explored in the future.

- The computational models of streaming applications considered in this thesis are based on SDF models, which are quite restrictive in semantics. To extend the SDF semantics to be more expressive, the model can be used to describe streaming applications with more dynamic properties, such as the adaptive extensions in Chapter 6, cyclo-static data flow [14], extended SDF [72], SDF scenarios [33], heterogeneous SDF (HSDF) [39], and parameterized SDF (PSDF) [11]. On the other hand, the model is required to preserve the static time analyzability in predictable systems design. Typically, analysis techniques based on more expressive models are preferably to achieve a more efficient design [33]. It is worthy to find a balance between expressiveness and analyzability of computational models, according to the application specific requirement.

- Both heuristics and constraint programming techniques have been exploited to solve the NP-complete problems in this thesis. A comparison between them and some other techniques, such as evolutionary algorithms, model checking and SAT solvers [3, 21, 28], is worthy to be addressed. Besides, to improve the propagation and searching efficiency of the constraint models can help to solve problems with scaled-up problem size or complexity. Preferably, the domain specific knowledge as used in [87, 102] is promising to be considered and modelled.

- From version 3.1.0, the constraint solver *Gecode* starts to support parallel search in multiple threads. However, the searching speed on multi-thread heavily depends on whether the search tree can be distributed to each thread efficiently, and it takes more memory than single thread searching [32]. To consider using multiple threads in searching with a reasonable peak memory in the exploration of search tree remains to be the future work.

- Theoretical studies have considered memory sharing between different logic buffers in Chapter 5, similar as in [70, 71, 42, 36]. Compared disjoint partitioning, it shows a great reduction in memory size. However, a general way to conduct memory sharing is non-trivial in implementation, which has not been seen in the literature and remains to be addressed.

- Based on worst case scenarios, performance analysis in this thesis has been done for hard real-time applications. For applications with soft real-time requirements, probabilistic analysis based on average case is needed. Potentially, it can lead to more efficient resource usage.
- In Chapter 8, a prototype of streaming applications on Avalon bus based MPSoCs has been built. Another interconnection alternative is the NoC communication, as covered in theoretical studies in Chapter 3 and Chapter 5. When more scalability on the MPSoC platform is required, the prototype needs to be extended to NoC architecture.

BIBLIOGRAPHY

[1] ARM Ltd. `http://www.arm.com`.

[2] Altera Ltd. `http://www.altera.com`.

[3] The SPIN website. `http://spinroot.com`.

[4] The SimpleScalar-ARM power modeling project. `http://www.eecs.umich.edu/~panalyzer/`.

[5] D. Andrews, D. Niehaus, R. Jidin, M. Finley, W. Peck, M. Frisbie, J. Ortiz, E. Komp, and P. Ashenden. Programming models for hybrid FPGA-CPU computational components: A missing link. *IEEE Micro*, 24(4):42–53, 2004.

[6] F. Balarin, Y. Watanabe, H. Hsieh, L. Lavagno, C. Passerone, and A. Sangiovanni-Vincentelli. Metropolis: An integrated electronic system design environment. *Computer*, 36(4):45–52, 2003.

[7] M. Barr. A reconfigurable computing primer. *Multimedia System Design*, pages 44–47, Septemer 1998.

[8] M. Bekooij, O. Moreira, P. Poplavko, B. Mesman, M. Pastrnak, and J. V. Meerbergen. Predictable embedded multiprocessor system design. In *Proceedings of Workshop on Software and Compilers for Embedded Systems (SCOPES '04)*. Springer, 2004.

[9] L. Benini, M. Ferrero, A. Macii, E. Macii, and M. Poncino. Supporting system-level power exploration for DSP applications. In *Proceedings of the 10th Great Lakes Symposium on VLSI (GLSVLSI '00)*, pages 17–22, New York, NY, USA, 2000. ACM.

[10] L. Benini, M. Lombardi, M. Milano, and M. Ruggiero. A constraint programming approach for allocation and scheduling on the cell broadband engine. In *Proceed-*

ings of the 14th International Conference on Principles and Practice of Constraint Programming (CP '08), pages 21–35, Berlin, Heidelberg, 2008. Springer-Verlag.

[11] B. Bhattacharya and S. Bhattacharyya. Parameterized dataflow modeling of DSP systems. In *IEEE International Conference on Acoustics, Speech, and Signal Processing*, volume 6, pages 3362 –3365 vol.6, 2000.

[12] S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee. *Software Synthesis from Dataflow Graphs*. Kluwer Academic Press, Norwell, MA, USA, 1996.

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

[14] G. Bilsen, M. Engels, R. Lauwereins, and J. A. Peperstraete. Cyclo-static dataflow. *IEEE Transactions on Signal Processing*, 2(44):397–408, 1996.

[15] J.-Y. L. Boudec and P. Thiran. *Network Calculus: A Theory of Deterministic Queuing Systems for the Internet*. Springer, LNCS, 2001.

[16] F. Boussinot and R. De Simone. The ESTEREL language. *Proceedings of the IEEE*, 79(9):1293–1304, September 1991.

[17] S. Chakraborty, S. Kunzli, and L. Thiele. A general framework for analysing system properties in platform-based embedded system designs. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '03)*, pages 190–195, Washington, DC, USA, 2003. IEEE Computer Society.

[18] S. Chakraborty, L. T. X. Phan, and P. S. Thiagarajan. Event count automata: A state-based model for stream processing systems. In *Proceedings of the 26th IEEE International Real-Time Systems Symposium (RTSS '05)*, pages 87–98, Washington, DC, USA, 2005. IEEE Computer Society.

[19] S. Chaudhuri, S. Guilley, F. Flament, P. Hoogvorst, and J.-L. Danger. An 8x8 run-time reconfigurable FPGA embedded in a SoC. In *Proceedings of the 45th annual Conference on Design Automation (DAC '08)*, pages 120–125, New York, NY, USA, 2008. ACM.

[20] K. Chen, D. I. August, and S. Malik. Retargetable static timing analysis for embedded software. In *Proceedings of the 14th International Symposium on Systems Synthesis (ISSS '01)*, pages 39–44, Los Alamitos, CA, USA, 2001. IEEE Computer Society.

[21] A. Cimatti, E. M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. Nusmv 2: An opensource tool for symbolic model checking. In *Proceedings of the 14th International Conference on Computer Aided Verification (CAV '02)*, pages 359–364, London, UK, 2002. Springer-Verlag.

[22] R. L. Cruz. A calculus for network delay. *IEEE Transactions on Information Theory*, 37(1):114–141, 1991.

[23] R. L. Cruz. Quality of service guarantees in virtual circuit switched networks. *IEEE Journal on Selected Areas in Communications*, 13(6):1048–1056, 1995.

[24] D. Culler, J. P. Singh, and A. Gupta. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann, August 1998. ISBN 1558603433.

[25] D. Cvijovic and J. Klinowski. Taboo Search: An Approach to the Multiple Minima Problem. *Science*, 267:664–666, Feb. 1995.

[26] W. Dally and B. Towles. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003.

[27] M. Duranton. The challenges for high performance embedded systems. In *Proceedings of the 9th EUROMICRO Conference on Digital System Design (DSD '06)*, pages 3–7, Washington, DC, USA, 2006. IEEE Computer Society.

[28] N. Eén and N. Sörensson. An extensible SAT-solver. In E. Giunchiglia and A. Tacchella, editors, *SAT*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003.

[29] P. Eles, Z. Peng, P. Pop, and A. Doboli. Scheduling with bus access optimization for distributed embedded systems. *IEEE Trans. Very Large Scale Integr. Syst.*, 8(5): 472–491, 2000.

[30] C. Erbas. *System-Level Modeling and Design Space Exploration for Multiprocessor Embedded System-on-Chip Architectures*. PhD thesis, 2006.

[31] M. R. Garey and D. S. Johnson. *Computers and intractability : a guide to the theory of NP-completeness*. W. H. Freeman, January 1979.

[32] Gecode. Generic Constraint Development Environment, 2009. `http://www.gecode.org/`.

[33] M. Geilen. Synchronous dataflow scenarios. *ACM Transactions in Embedded Computing Systems – Special issue on Model-driven Embedded System Design*, 2010. (accepted for publication).

[34] 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.

[35] M. Geilen and T. Basten. A calculator for pareto points. In *Proceedings of the Conference on Design Automation and Test in Europe (DATE '07)*, pages 285–290, San Jose, CA, USA, 2007.

[36] M. Geilen, T. Basten, and S. Stuijk. Minimising buffer requirements of synchronous dataflow graphs with model checking. In *Proceedings of the 42nd annual Conference on Design Automation (DAC '05)*, pages 819–824, New York, NY, USA, 2005.

ACM.

[37] A. H. Ghamarian, M. C. W. Geilen, T. Basten, B. D. Theelen, M. R. Mousavi, and S. Stuijk. Liveness and boundedness of synchronous data flow graphs. In *Proceedings of the Formal Methods in Computer Aided Design (FMCAD '06)*, pages 68–75, Washington, DC, USA, 2006. IEEE Computer Society.

[38] 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 *Proceedings of the International Conference on Application of Concurrency to System Design (ACSD '06)*, pages 25–36, Washington, DC, USA, 2006. IEEE Computer Society.

[39] A. Girault, B. Lee, and E. A. Lee. Hierarchical finite state machines with multiple concurrency models. *IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems*, 18:742–760, 1999.

[40] K. Goossens, J. Dielissen, and A. Radulescu. Æthereal network on chip: Concepts, architectures, and implementations. *IEEE Des. Test*, 22(5):414–421, 2005.

[41] M. I. Gordon, W. Thies, and S. Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. *Source ACM SIGOPS Operating Systems Review archive*, 40(5):151–162, 2006.

[42] R. Govindarajan, G. R. Gao, and P. Desai. Minimizing buffer requirements under rate-optimal schedule in regular dataflow networks. *Journal of VLSI Signal Processing*, 31(3):207–229, July 2002.

[43] P. L. Guernic, J. Talpin, and J. L. Lann. Polychrony for system design. *Journal of Circuits, Systems and Computers. Special Issue on Application Specific Hardware Design*, 2002.

[44] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[45] P.-E. Hladik, H. Cambazard, A.-M. Déplanche, and N. Jussien. Solving a real-time allocation problem with constraint programming. *J. Syst. Softw.*, 81(1):132–149, 2008.

[46] P. K. F. Hölzenspies, G. J. M. Smit, and J. Kuper. Mapping streaming applications on a reconfigurable mpsoc platform at run-time. In *Proceedings of the International Symposium on System-on-Chip (ISSoC '07)*, pages 74–77, Tampere, Finland, November 2007. IEEE Circuits and Systems Society.

[47] A. Hormati, M. Kudlur, D. Bacon, S. Mahlke, and R. Rabbah. Optimus: Efficient realization of streaming applications on FPGAs. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES '08)*, Atlanta, USA, October 2008.

[48] J. Howard and et. al. A 48-Core IA-32 Message-Passing Processor with DVFS in 45nm CMOS. In *IEEE International Solid-State Circuits Conference (ISSCC '10)*, pages 19–21, 2010.

[49] J. Hu and R. Marculescu. Energy-aware mapping for tile-based NoC architectures under performance constraints. In *Proceedings of the conference on Asia South Pacific Design Automation (ASP-DAC '03)*, pages 233–239, New York, NY, USA, 2003. ACM.

[50] K. Huang, D. Grunert, and L. Thiele. Windowed fifos for fpga-based multiprocessor systems. In *ASAP*, pages 36–41, 2007.

[51] A. Jantsch. *Modeling Embedded Systems and SoC's: Concurrency and Time in Models of Computation*. Morgan Kaufmann, San Francisco, CA, 2003.

[52] A. Jantsch. Models of embedded computation. In R. Zurawski, editor, *Embedded systems handbook*. CRC Press, 2005.

[53] A. Jantsch and I. Sander. Models of computation and languages for embedded system design. In *IEE Proceedings on Computers and Digital Techniques*, pages 114–129, 2005.

[54] G. Kahn. The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, *Information Processing: Proceedings of the IFIP Congress*, pages 471–475. North-Holland, New York, NY, 1974.

[55] K. Keutzer, S. Malik, S. Member, A. R. Newton, J. M. Rabaey, and A. Sangiovanni-vincentelli. System-level design: Orthogonalization of concerns and platform-based design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19:1523–1543, 2000.

[56] B. Kienhuis, E. Deprettere, K. Vissers, and P. van der Wolf. An approach for quantitative analysis of application-specific dataflow architectures. In *Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures and Processors*, page 338, Washington, DC, USA, 1997. IEEE Computer Society.

[57] V. Kirischian, V. Geurkov, and L. Kirischian. A multi-mode video-stream processor with cyclically reconfigurable architecture. In *Proceedings of the Conference on Computing Frontiers (CF '08)*, pages 105–106, New York, NY, USA, 2008. ACM.

[58] T. Kogel and H. Meyr. Heterogeneous MP-SoC: the solution to energy-efficient signal processing. In *Proceedings of the 41st annual Conference on Design Automation (DAC '04)*, pages 686–691, New York, NY, USA, 2004. ACM.

[59] P. Le Guernic, T. Gautier, M. Le Borgne, and C. Le Marie. Programming real-time applications with SIGNAL. *Proceedings of the IEEE*, 79(9):1321–1335, September 1991.

[60] E. A. Lee and D. G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on Computers*, C-36(1): 24–35, January 1987.

[61] E. A. Lee and D. G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, September 1987.

[62] E. A. Lee and T. M. Parks. Dataflow process networks. *IEEE Proceedings*, 83(5): 773–799, May 1995.

[63] E. A. Lee and A. Sangiovanni-Vincentelli. A framework for comparing models of computation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(12):1217–1229, December 1998.

[64] P. Lieverse, P. Van Der Wolf, K. Vissers, and E. Deprettere. A methodology for architecture exploration of heterogeneous signal processing systems. *Journal of VLSI Signal Processing Systems*, 29(3):197–207, 2001.

[65] W. Liu, M. Yuan, X. He, Z. Gu, and X. Liu. Efficient SAT-based mapping and scheduling of homogeneous synchronous dataflow graphs for throughput optimization. In *Proceedings of the 28th IEEE international real-time systems symposium (RTSS '08)*, Barcelona, Spain, November 2008. IEEE Computer Society.

[66] R. Lublinerman and S. Tripakis. Translating data flow to synchronous block diagrams. In *Proceedings of the 6th Workshop on Embedded Systems for Real-Time Multimedia (ESTIMedia '08)*, pages 101–106, Atlanta, USA, October 2008.

[67] J. Madsen, K. Virk, and M. J. Gonzalez. A SystemC-based abstract real-time operating system model for multiprocessor system-on-chip. In *Multiprocessor System-on-Chip*. Morgan Kaufmann, 2004.

[68] M. Millberg, E. Nilsson, R. Thid, and A. Jantsch. Guaranteed bandwidth using looped containers in temporally disjoint networks within the Nostrum network on chip. In *Proceedings Design Automation and Test in Europe (DATE '04)*, page 20890, Washington, DC, USA, 2004. IEEE Computer Society.

[69] O. Moreira, F. Valente, and M. Bekooij. Scheduling multiple independent hard-real-time jobs on a heterogeneous multiprocessor. In *Proceedings of the 7th ACM & IEEE International conference on Embedded Software (EMSOFT '07)*, pages 57–66, New York, NY, USA, 2007. ACM.

[70] P. K. Murthy and S. S. Bhattacharyya. Shared buffer implementations of signal processing systems using lifetime analysis techniques. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 20:177–198, 2001.

[71] P. K. Murthy and S. S. Bhattacharyya. Buffer merging—a powerful technique for reducing memory requirements of synchronous dataflow specifications. *ACM Trans. Des. Autom. Electron. Syst.*, 9(2):212–237, 2004.

[72] C. Park, J. Chung, and S. Ha. Extended synchronous dataflow for efficient DSP system prototyping. In *Proceedings of the IEEE International Workshop on Rapid System Prototyping (RSP '99)*, page 196, Washington, DC, USA, 1999. IEEE Computer Society.

[73] L. T. Phan, S. Chakraborty, and P. Thiagarajan. A multi-mode real-time calculus. In *Proceedings of the 28th IEEE international Real-Time Systems Symposium (RTSS '08)*, Barcelona, Spain, November 2008. IEEE Computer Society.

[74] A. D. Pimentel, L. O. Hertzberger, P. Lieverse, P. van der Wolf, and E. F. Deprettere. Exploring embedded-systems architectures with artemis. *Computer*, 34(11):57–63, 2001.

[75] A. D. Pimentel, C. Erbas, and S. Polstra. A systematic approach to exploring embedded system architectures at multiple abstraction levels. *IEEE Trans. Comput.*, 55(2):99–112, 2006.

[76] J. Real and A. Crespo. Mode change protocols for real-time systems: A survey and a new proposal. *Real-Time Systems*, 26(2):161–197, 2004.

[77] V. Reyes, W. Kruijtzer, T. Bautista, G. Alkadi, and A. Nunez. A unified system-level modeling and simulation environment for mpsoc design: Mpeg-4 decoder case study. In *Proceedings of Design Automation and Test in Europe (DATE '04)*, volume 1, page 101, Los Alamitos, CA, USA, 2006. IEEE Computer Society.

[78] K. Richter, D. Ziegenbein, M. Jersak, and R. Ernst. Model composition for scheduling analysis in platform design. In *Proceedings of the 39th annual Conference on Design Automation (DAC '02)*, pages 287–292, New York, NY, USA, 2002. ACM.

[79] M. J. Rutten, J. T. J. van Eijndhoven, E. G. T. Jaspers, P. van der Wolf, E.-J. D. Pol, O. P. Gangwal, and A. Timmer. A heterogeneous multiprocessor architecture for flexible media processing. *IEEE Design & Test of Computers*, 19(4):39–50, 2002.

[80] I. Sander and A. Jantsch. System modeling and transformational design refinement in ForSyDe. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 23(1):17–32, 2004.

[81] I. Sander and A. Jantsch. Modelling adaptive systems in ForSyDe. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 200(2):39–54, 2008. First Workshop on Verification of Adaptive Systems (VerAS 2007).

[82] J. Shin, K. Tam, D. Huang, B. Petrick, H. Pham, C. Hwang, H. Li, A. Smith, T. Johnson, F. Schumacher, D. Greenhill, A. Leon, and A. Strong. A 40nm 16-Core 128-Thread CMT SPARC SoC Processor. In *IEEE International Solid-State Circuits Conference (ISSCC '10)*, pages 98–99, 2010.

[83] Y. Shin, D. Kim, and K. Choi. Schedulability-driven performance analysis of multiple mode embedded real-time systems. In *Proceedings of the 37th Conference on Design Automation (DAC '00)*, pages 495–500, New York, NY, USA, 2000.

[84] S. Sriram and S. S. Bhattacharyya. *Embedded Multiprocessors: Scheduling and Synchronization*. Marcel Dekker, Inc., New York, NY, USA, 2000.

[85] B. Stackhouse, B. Cherkauer, M. Gowan, P. Gronowski, and C. Lyles. A 65nm 2-Billion-Transistor Quad-Core Itanium Processor. In *IEEE International Solid-State Circuits Conference (ISSCC '08)*, pages 92–93, 2008.

[86] D. Stiliadis, A. Varma, S. clocked Fair Queueing, W. R. Robin, and D. Round. Latency-rate servers: A general model for analysis of traffic scheduling algorithms. In *IEEE/ACM Transactions on Networking*, pages 111–119, 1996.

[87] S. Stuijk, M. Geilen, and T. Basten. Exploring trade-offs in buffer requirements and throughput constraints for synchronous dataflow graphs. In *Proceedings of the 43th annual Conference on Design Automation (DAC '06)*, pages 899–904, San Francisco, California, USA, July 2006.

[88] S. Stuijk, T. Basten, M. C. W. Geilen, and H. Corporaal. Multiprocessor resource allocation for throughput-constrained synchronous dataflow graphs. In *Proceedings of the 44th annual Conference on Design Automation (DAC '07)*, pages 777–782, New York, NY, USA, 2007. ACM.

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

[90] M. B. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, P. Johnson, J.-W. Lee, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal. The Raw microprocessor: A computational fabric for software circuits and general-purpose programs. *IEEE Micro*, 22(2):25–35, 2002.

[91] B. Theelen, M. Geilen, T. Basten, J. Voeten, S. Gheorghita, and S. Stuijk. A scenario-aware data flow model for combined long-run average and worst-case performance analysis. In *Proceedings of ACM/IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE '06)*, pages 185–194, Los Alamitos, CA, USA, 2006. IEEE Computer Society.

[92] L. Thiele. Performance analysis of distributed embedded systems. In *Proceedings of the 7th ACM & IEEE International Conference on Embedded Software (EMSOFT '07)*, pages 10–10, New York, NY, USA, 2007. ACM.

[93] L. Thiele and E. Wandeler. *Embedded Systems Handbook*, chapter Performance Analysis of Distributed Embedded Systems. CRC Press, 2006.

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

[95] S. wei 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.

[96] D. Wendel, R. Kalla, R. Cargoni, J. Clables, J. Friedrich, R. Frech, J. Kahle, B. Sinharoy, W. Starke, S. Taylor, S. Weitzel, S. Chu, S. Islam, and V. Zyuban. The implementation of POWER7: A highly parallel and scalable multi-core high-end server processor. In *Proceedings of IEEE International Solid-State Circuits Conference (ISSCC '10)*, pages 102–103, 2010.

[97] M. Wiggers, M. Bekooij, P. Jansen, and G. Smit. Efficient computation of buffer capacities for multi-rate real-time systems with back-pressure. In *Proceedings of the 4th International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS '06)*, pages 10–15, New York, NY, USA, 2006. ACM.

[98] M. H. Wiggers, M. J. G. Bekooij, and G. J. M. Smit. Computation of buffer capacities for throughput constrained and data dependent inter-task communication. In *Proceedings of the Conference on Design Automation and Test in Europe (DATE '08)*, pages 640–645, New York, NY, USA, 2008. ACM.

[99] S. J. Wilton and N. P. Jouppi. Cacti: An enhanced cache access and cycle time model. *IEEE Journal of Solid-State Circuits*, 31(5):677–688, 1996.

[100] W. Wolf. The future of multiprocessor systems-on-chips. In *Proceedings of the 41st annual Conference on Design Automation (DAC '04)*, pages 681–685, New York, NY, USA, 2004. ACM.

[101] Xilinx Ltd. `http://www.xilinx.com`.

[102] Y. Yang, M. Geilen, T. Basten, E. Stuijk, and H. Corporaal. Automated bottleneck-driven design-space exploration of media processing systems. In *Proceedings of Design Automation and Test in Europe (DATE '10)*, pages 1041–1046, Dresden, Germany, March 2010.

[103] M. Yuan, X. He, and Z. Gu. Hardware/software partitioning and static task scheduling on runtime reconfigurable FPGAs using a SMV solver. In *Proceedings of the 2008 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS '08)*, pages 295–304, Washington, DC, USA, 2008. IEEE Computer Society.

[104] J. Zhu, I. Sander, and A. Jantsch. Energy efficient streaming applications with guaranteed throughput on MPSoCs. In *Proceedings of the International Conference on Embedded Software (EMSOFT '08)*, pages 119–128, Atlanta, USA, October 2008.

[105] J. Zhu, I. Sander, and A. Jantsch. Performance analysis of reconfiguration in adaptive real-time streaming applications. In *Proceedings of the 6th Workshop on Embedded Systems for Real-Time Multimedia (ESTIMedia '08)*, pages 53–58, Atlanta, USA, October 2008.

[106] J. Zhu, I. Sander, and A. Jantsch. Buffer minimization of real-time streaming applications scheduling on hybrid CPU/FPGA architectures. In *Proceedings of Design Automation and Test in Europe (DATE '09)*, pages 1506–1511, Nice, France, April 2009.

[107] J. Zhu, I. Sander, and A. Jantsch. Constrained global scheduling of streaming applications on MPSoCs. In *Proceedings of the conference on Asia South Pacific Design Automation (ASP-DAC '10)*, pages 223–228, Taipei, January 2010.

[108] J. Zhu, I. Sander, and A. Jantsch. Pareto efficient design for reconfigurable streaming applications on CPU/FPGAs. In *Proceedings of Design Automation and Test in Europe (DATE '10)*, pages 1035–1040, Dresden, Germany, March 2010.

[109] J. Zhu, I. Sander, and A. Jantsch. Performance analysis of reconfigurations in adaptive real-time streaming applications. *ACM Transactions in Embedded Computing Systems – Special issue on Embedded Systems for Real-time Multimedia*, 2010. (accepted for publication).