Energy and Design Cost Efficiency for Streaming Applications on Systems-on-Chip

JUN ZHU

Licentiate Thesis
Stockholm, Sweden 2009

© Jun Zhu, January 2009

Tryck: Universitetsservice US AB
Abstract

With the increasing capacity of today’s integrated circuits, a number of heterogeneous system-on-chip (SoC) architectures in embedded systems have been proposed. In order to achieve energy and design cost efficient streaming applications on these systems, new design space exploration frameworks and performance analysis approaches are required. This thesis considers three state-of-the-art SoCs architectures, i.e., the multi-processor SoCs (MPSoCs) with network-on-chip (NoC) communication, the hybrid CPU/FPGA SoCs, and the run-time reconfigurable (RTR) FPGAs. The main topic of the author’s research is to model and capture the application scheduling, architecture customization, and buffer dimensioning problems, according to the real-time requirement. While these problems are NP-complete, both heuristic algorithms and the constraint solver are exploited in solutions finding.

On NoC communication based MPSoCs, an approach to optimize the real-time streaming applications with customized processor voltage-frequency levels and memory sizes is presented. A multi-clocked synchronous model of computation (MoC) framework is proposed in heterogeneous timing analysis and energy estimation. Using heuristic searching (i.e., greedy and taboo search), the experiments show an energy reduction (up to 21%) without any loss in application throughput compared with an ad-hoc approach.

On hybrid CPU/FPGA architectures, the buffer minimization scheduling of real-time streaming applications is addressed. Based on event models, the problem has been formalized declaratively as constraint base scheduling, and solved by the public domain constraint solver Gecode. Compared with traditional PAPS method, the proposed method needs much less buffer (19% of PAPS in the best case), while high throughput guarantees can still be achieved.

Furthermore, a novel compile-time analysis approach based on iterative timing phases is proposed for run-time reconfigurations in adaptive real-time streaming applications on RTR FPGAs. Finally, the reconfigurations analysis and design trade-offs analysis capabilities of the proposed framework have been exemplified with experiments on both example and industrial applications.
Acknowledgements

I would first like to thank my supervisors Prof. Axel Jantsch and Dr. Ingo Sander for their help and support. I am thankful to Axel for having over the years been encouraged and motivated by the inspiring spirit from his advice and supervision, and I appreciate his efforts to offer the financial support in this research. It is also a pleasure to be a student of Ingo. He really puts efforts to create a good research environment in our group, and encourages me during my studies. Especially, we enjoy quite a lot of conversations together on our bicycle trips in the beautiful summer of Stockholm, on both research and private topics.

Meanwhile, 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 would also like to express my joy of having the opportunity to work together with my colleagues: Dr. Tarvo Raudvere, Dr. Zhonghai Lu, Kim Petersén, Mikael Millberg, Mikael Lagerkvist, Mladen Niki- tovic, Jian Liu, Mikael Collin, docent Johnny Öberg, Fredrik Lundevall, Roshan Weerasekara, Sandro Penolazzi, and both current and former members in our SAM group. I appreciate all the interesting discussions with them. Thanks to system group for computer related problems, and Agneta Herling and Alina Munteanu for all administrative help.

Thanks to Dr. Sander Stuijk, Prof. Twan Basten, Prof. David Andrews, Dr. Zonghua Gu, Dr. Kaiyu Chen, and Dr. Juan Chen for the inspiring discussions, which have been very helpful in my work on the way to this thesis. Especially, I thank Prof. Twan Basten for very useful comments in the quality checking of this thesis and for being the opponent in my defense.

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 abroad.

Finally, I thank my family for their endless love.

Jun Zhu,
Stockholm, January 2009
## Contents

<table>
<thead>
<tr>
<th>Contents</th>
<th>vi</th>
</tr>
</thead>
<tbody>
<tr>
<td>List of Abbreviations</td>
<td>ix</td>
</tr>
<tr>
<td>List of Figures</td>
<td>x</td>
</tr>
<tr>
<td>List of Tables</td>
<td>xii</td>
</tr>
<tr>
<td>1 Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Background</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Architecture models</td>
<td>2</td>
</tr>
<tr>
<td>1.2.1 NoC based MPSoC</td>
<td>2</td>
</tr>
<tr>
<td>1.2.2 Hybrid CPU/FPGA</td>
<td>3</td>
</tr>
<tr>
<td>1.2.3 Run-time reconfigurable FPGA</td>
<td>4</td>
</tr>
<tr>
<td>1.3 Contributions and outline</td>
<td>5</td>
</tr>
<tr>
<td>2 Streaming Application Models Preliminaries</td>
<td>9</td>
</tr>
<tr>
<td>2.1 Synchronous data flow</td>
<td>9</td>
</tr>
<tr>
<td>2.2 Synchronous MoC framework</td>
<td>10</td>
</tr>
<tr>
<td>2.2.1 Stream models</td>
<td>11</td>
</tr>
<tr>
<td>2.2.2 Process</td>
<td>11</td>
</tr>
<tr>
<td>2.3 Simulation based on synchronous MoC</td>
<td>12</td>
</tr>
<tr>
<td>2.4 Discussions</td>
<td>13</td>
</tr>
<tr>
<td>3 Energy Efficiency of Multi-Clocked MPSoCs</td>
<td>15</td>
</tr>
<tr>
<td>3.1 Introduction</td>
<td>15</td>
</tr>
<tr>
<td>3.2 Related work</td>
<td>16</td>
</tr>
<tr>
<td>3.3 Motivation</td>
<td>17</td>
</tr>
<tr>
<td>3.3.1 Design exploration</td>
<td>17</td>
</tr>
<tr>
<td>3.3.2 Energy efficiency evaluation</td>
<td>19</td>
</tr>
<tr>
<td>3.4 Design space exploration flow</td>
<td>20</td>
</tr>
<tr>
<td>3.5 Multi-clock synchronous MoC framework</td>
<td>21</td>
</tr>
<tr>
<td>3.5.1 Multi-clock synchronous domains and domain interfaces</td>
<td>21</td>
</tr>
</tbody>
</table>
# List of Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMBS</td>
<td>Constraint based Minimal Buffer Scheduling</td>
</tr>
<tr>
<td>DDRAM</td>
<td>Double Data Random Access Memory</td>
</tr>
<tr>
<td>FIFO</td>
<td>First-In-First-Out (Buffer)</td>
</tr>
<tr>
<td>FPGA</td>
<td>Field-Programmable Gate Array</td>
</tr>
<tr>
<td>JIT</td>
<td>Just-In-Time</td>
</tr>
<tr>
<td>KPN</td>
<td>Kahn process network</td>
</tr>
<tr>
<td>LE</td>
<td>logic elements</td>
</tr>
<tr>
<td>MIMO</td>
<td>Multiple Inputs and Multiple Outputs</td>
</tr>
<tr>
<td>MoC</td>
<td>Model of Computation</td>
</tr>
<tr>
<td>MPSoCs</td>
<td>Multi-Processor Systems on Chip</td>
</tr>
<tr>
<td>NI</td>
<td>Network Interface</td>
</tr>
<tr>
<td>NoC</td>
<td>Network-on-Chip</td>
</tr>
<tr>
<td>NP</td>
<td>Non-deterministic Polynomial time</td>
</tr>
<tr>
<td>PAPS</td>
<td>Periodic Admissible Parallel Schedule</td>
</tr>
<tr>
<td>PASS</td>
<td>Periodic Admissible Sequential Schedule</td>
</tr>
<tr>
<td>RTC</td>
<td>Real-Time Calculus</td>
</tr>
<tr>
<td>RTOS</td>
<td>Real-Time Operation System</td>
</tr>
<tr>
<td>RTR</td>
<td>Run-Time Reconfigurable</td>
</tr>
<tr>
<td>SDF</td>
<td>Synchronous Data Flow</td>
</tr>
<tr>
<td>SoC</td>
<td>System-on-Chip</td>
</tr>
<tr>
<td>SRAM</td>
<td>Static Random Access Memory</td>
</tr>
<tr>
<td>SW/HW</td>
<td>Software/Hardware</td>
</tr>
<tr>
<td>WCET</td>
<td>worst case execution time</td>
</tr>
</tbody>
</table>
List of Figures

1.1 An example of NoC based MPSoC architecture template. . . . . . . . . . 3
1.2 The hybrid CPU/FPGA architecture built with tightly coupled memory. 3
1.3 Overview of a reconfigurable FPGA using “just-in-time” (JIT) reconfig-

uration with a single configuration slot. . . . . . . . . . . . . . . . . . . 4
2.1 Streaming application model example. . . . . . . . . . . . . . . . . . . . 10
2.2 Timing relations of events in signals with different clocks. . . . . . . . . 11
2.3 Process skeleton of a mealy state machine. . . . . . . . . . . . . . . . . . 12
2.4 A self-timed schedule of the example application (see Figure 2.1) with
specified specification parameters. . . . . . . . . . . . . . . . . . . . . . . 13
3.1 Self-timed schedules and energy efficiency analysis for different design
options on an instance of the example application (\( n_{i,j} = 2, m_{i,j} = 3, \\
\quad n_{j,k} = 1, \text{ and } m_{j,k} = 2 \)). . . . . . . . . . . . . . . . . . . . . . . . . . . 18
3.2 Energy efficiency design exploration flow with guaranteed throughput. 19
3.3 Multi-clock synchronous domains and domain interfaces. . . . . . . . . 22
3.4 Domain interfaces \( DI_{A:B} \). . . . . . . . . . . . . . . . . . . . . . . . . . . . 24
3.5 Scheduling patterns encoding. . . . . . . . . . . . . . . . . . . . . . . . . 25
3.6 Mapping of the implementation model of the example application (Fig-

ure 2.1) onto a NoC based MPSoC architecture model. . . . . . . . . . . 27
3.7 Process timing and energy profiling. . . . . . . . . . . . . . . . . . . . . . 28
3.8 FM radio application mapped onto \( 4 \times 4 \) tiles. . . . . . . . . . . . . . 30
3.9 Periodic schedules of the FM radio application on multiple processors,
in which the three workloads considered are \( Thru-1=640 \text{ kb/s}, \text{ Thru-} \\
2=533 \text{ kb/s and Thru-3}=400 \text{ kb/s} \). The schedule of \( \mu p_{4-n} (0 \leq n \leq 9) \)
represents the symmetrical computations in the pipeline stage 4. . . . . . . 32
4.1 The example application (see Figure 2.1) mapped onto the hybrid CPU/FPGA
architecture. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 35
4.2 Comparison of different schedules of an instance of the example appli-
cation, in which \( p_i \) and \( p_k \) are mapped onto the same processor and can 
only be scheduled sequentially. . . . . . . . . . . . . . . . . . . . . . . . . 38
4.3 Generic constraint based scheduling work flow. . . . . . . . . . . . . . . 39
### List of Figures

4.4 Cumulative functions and buffer properties for the PAPS in Figure 4.2a. 41
4.5 A cyclic MIMO application model. 44
4.6 The modem application partitioned for a multiprocessor CPU/FPGA architecture. 45

5.1 A minimal adaptive streaming application model. 48
5.2 Timing phases of reconfiguration analysis. 52
5.3 Experimental results of the example application. 54
5.4 The adaptive coding and modulation case study synopsis. 55
5.5 Industrial application and experimental results. 57
List of Tables

3.1 Voltage-Frequency levels. ........................................ 31
3.2 Experimental results. ........................................... 33
4.1 Comparison of scheduling methods. .......................... 45
5.1 Design options for the adaptive process $p_j$. ............. 53
Chapter 1

Introduction

1.1 Background

Over the last five decades\(^1\), the progress in electronics has followed Moore’s law. Accordingly, the measures of the capabilities of digital electronic devices evolve roughly exponentially, e.g., processing speed, memory capacity, communication bandwidth. This trend will continue to be the driving force of the upcoming innovations in embedded systems. Among them, one is the emergence of heterogeneous systems-on-chip (SoCs).

Heterogeneous SoCs consist of multiple components of multi-processors, custom circuits, memories and communication links (dedicated or network-on-chip) on the same silicon. Currently, they exhibit extreme complexity, besides merits, 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 chosen from a diversified possible architecture templates to meet the different requirements of design specifications and time-to-market plans.

- **Customizability.** The system components are fully customizable (e.g., the processor *voltage-frequency* levels, memory *sizes*, communication *bandwidth*) according to the application *throughput* on demand, *energy* on demand, and the design cost\(^2\) budget.

- **Programmability and Reconfigurability.** Software-programmable multi-processors and run-time reconfigurable (RTR) field-programmable gate arrays (FPGAs) hardware devices are widely used. They not only improve the design flexibility and productivity but also add new design challenges.

---

\(^1\)That is from the invention of the integrated circuit in 1958.

\(^2\)In this thesis, the design cost is measured by metrics of on-chip buffer memory requirement and circuits area.
Compared with traditional systems, e.g., systems-on-board, the heterogeneous SoCs have enhanced computation power and flexibility, which make them more suitable for streaming applications in the different domains of digital signal processing, multi-media, network processing, as well as encryption and decryption. Meanwhile, the real-time streaming applications on such platforms always have stringent demands on non-functional properties (e.g., timing, design cost, energy dissipation), besides functional correctness. An early system level methodology is critical for the efficient design of streaming applications on heterogeneous SoCs.

However, to capture all the design concerns globally, while making optimal design decisions at an early system level, is still a big challenge, especially when consumer embedded systems with low-cost (silicon area), low-power, high-performance and high-portability requirements, always have stringent time-to-market demands. Furthermore, each product development involves specification refinement, cost analysis, design optimization, design iteration, and other design processes. More design flexibility makes system design more complex.

To design efficient streaming applications on heterogeneous SoCs systematically, it demands contributions from several research fields, e.g., formal methods, system modeling and analysis, simulation, models of computation. This thesis intends to provide performance analysis and design optimization frameworks for heterogeneous SoCs based on the author’s contributions on both simulation and formal analysis methods.

1.2 Architecture models

A number of heterogeneous SoC architectures have been proposed in recent years [7, 8, 33, 38, 62, 67]. In this thesis, the author will consider three state-of-the-art SoCs architectures, i.e., multi-processor SoCs (MPSoCs) with network-on-chip (NoC) communication [68], hybrid CPU/FPGA architectures [70], and FPGAs with runtime reconfigurability [69].

1.2.1 NoC based MPSoC

The NoC based MPSoC architecture consists of on-tile resources and a packet switched NoC communication. The example architecture template with 4 × 4 tiles on a 2D mesh NoC topology is illustrated in Figure 1.1.

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
1.2. ARCHITECTURE MODELS

NoC communication provides guaranteed bandwidth and bounded latency, which is achieved with a hard real-time communication backbone [47],

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 meeting the design cost budget and timing related requirement (e.g., energy minimization and throughput guarantees), becomes crucial.

1.2.2 Hybrid CPU/FPGA

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 architecture provides an excellent platform for developing custom MPSoCs [7]. The example architectures are Xilinx [6] Virtex FPGAs and Altera [1] Stratix FPGAs, embedded with either PowerPC processors and MicroBlaze soft processor cores [6] or Nios soft processor cores [1], as illustrated in Figure 1.2. The applications can be implemented as software running on the host CPUs, while the critical portions are migrated into custom hardware circuits. It combines the merits of cost and energy efficiency of FPGAs and the easy programmability of CPUs.

It is assumed that this hybrid software (CPU) and hardware (FPGA) architecture uses direct communication channels to communicate and buffer data, e.g., the Stratix multi-channel shared memory FIFO core. The hybrid CPU/FPGA architecture demands not only new design paradigms [7], but also novel scheduling polices on real-time operation systems (RTOS).

1.2.3 Run-time reconfigurable FPGA

![Diagram of a reconfigurable FPGA using “just-in-time” (JIT) reconfiguration with a single configuration slot.]

Partially run-time reconfigurable (RTR) FPGAs, such as Xilinx [6] Virtex-4, are very popular in today’s embedded systems. The architecture template of a partially RTR FPGA is illustrated in Figure 1.3. For discussion, the reconfiguration-related FPGA area (excluding the non-reconfigurable area) is divided into two parts.

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 memory, or an external memory (Flash, DDRAM, SRAM, etc.).
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”-approach (JIT), which uses a single configuration slot. 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 their 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 [37]. However, the performance analysis for reconfigurable FPGAs is more complex compared with methodologies on traditional non-reconfigurable systems and adds new challenges.

### 1.3 Contributions and outline

This thesis deals with the design cost and energy efficiency of real-time streaming applications on the proposed heterogeneous SoC architectures. In particular, the streaming applications are modeled with synchronous data flow (SDF) model of computation. The outline of the thesis is as follows, in which the author’s contributions are divided into the corresponding chapters.

#### Chapter 2

This chapter introduces terminology and notations of the streaming application models used in this thesis. In particular, the SDF model and the synchronous model are introduced.

#### Chapter 3

This chapter presents a design space exploration flow to achieve energy efficiency for streaming applications on MPSoCs while meeting the specified throughput constraints. The public domain simulators Sim-Panalyzer and Cacti are used to estimate the energy dissipations of the parameterized architectural components. As the main contributions, the streaming applications are scheduled on a multi-clock synchronous modeling framework, the application timing properties are guaranteed by throughput analysis, and both processor voltage-frequency levels and memory sizes are customized in the design space to optimize the application pipeline parallelism for energy efficiency. Two widely used 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


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 CPU/FPGA architectures. A novel declarative way of constraint based scheduling for real-time hybrid SW/HW systems is proposed, while the application throughput is guaranteed by periodic phases in execution. Being implemented on the public domain constraint solver Gecode, 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 the traditional PAPS method.

Part of the work was published in


Chapter 5

This chapter proposes a performance analysis framework for adaptive real-time synchronous data flow streaming applications on run-time reconfigurable FPGAs. As the main contributions, it presents a constraint based way to capture both streaming application execution semantics and the varying design concerns during reconfigurations, and a novel compile-time analysis approach based on iterative timing phases for run-time reconfigurations is exploited. The event models in analysis are constructed as cumulative functions on data streams, and the periodic properties in deterministic scheduling is exploited to ensure throughput guarantees. Finally, 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

1.3. CONTRIBUTIONS AND OUTLINE


Chapter 6

This chapter concludes the thesis and gives prospective research directions in the future.
Chapter 2

Streaming Application Models
Preliminaries

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 always reconcile properties that are essential in different purposes of analysis. Therefore, it is possible to use certain MoCs to analyze the system properties of embedded computational systems [24, 36], such as their functional correctness, performance, scheduling, and buffer dimensioning.

This chapter introduces terminology and notations of the streaming application MoCs used in this thesis. The synchronous data flow (SDF) MoC is introduced as a un-timed model in Section 2.1. The synchronous MoC is introduced as a timed-model in Section 2.2. Further, the timed simulation in the synchronous model is illustrated in Section 2.3. Finally, Section 2.4 discusses the pros and cons of the simulation based method in performance analysis for streaming applications.

2.1 Synchronous data flow

A SDF application model with three-stage pipeline is depicted in Figure 2.1, which is used as the tutorial example in this thesis. Nodes denote the computation processes and edges associated with FIFOs denote the communication channels with buffer storage. 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 $\text{FIFO}_{i,j}$ via signal $s_2$ and writes rate $n_{j,k}$ to $\text{FIFO}_{j,k}$ via signal $s_3$.

In SDF semantics, processes use blocking read and non-blocking write (infinite buffers). On the other hand, finite buffer storage is always utilized in implementa-
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. A process executes only when it is enabled and allowed by the scheduling policy. Further, two instances of a process are not allowed to execute concurrently, as in [29, 60]. While a process is computing, the data tokens remain on the input-side FIFO(s) until the computation is completed [60]. At the end of each execution, the output results are available in the output-side FIFO(s).

In this thesis, a subset of SDF models are considered, which are said to be consistent [40] and can run indefinitely with bounded buffer. 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 $r_i$ and $r_j$, where $r_i$ and $r_j$ are the minimum integer solutions of a set of balance equations

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

for all the communication channels. This subset of SDF models are also known to be live and bounded [26].

To quantify the process computation and FIFO storage capabilities of the application model, a process computation latency list $T$ contains computation time $t_{C,i}^2$ in time slots to execute each process $p_i$ once. A time slot equals to an abstract clock cycle. A FIFO size list $\Gamma$ contains the storage capacity $\gamma_{y,z}$ in amount of tokens for each FIFO $FIFO_{y,z}$. For instance, the example application in Figure 2.1 has $T = [t_{C,i}, t_{C,j}, t_{C,k}]$ and $\Gamma = [\gamma_{i,j}, \gamma_{j,k}]$.

### 2.2 Synchronous MoC framework

Preserving the static producing and consuming token properties, the SDF models can be transformed into synchronous models [68, 45], in which the timing of process firing and timing-related properties (e.g., throughput and energy) can be captured. In the synchronous MoC, systems are described as a set of concurrent processes, which communicate through synchronous signals.

---

1. A vector of the non-trivial firing times for each process in the model is called *repetition vector*.
2. To distinguish from regular untimed SDF MoC [40], this model is called *timed-SDF* model. It has been used in [29, 60] to analyze timing related properties.
2.2. SYNCHRONOUS MOC FRAMEWORK

2.2.1 Stream models

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

$$s = \cup \{e(n)\}_{clk_s} = \{e(0), e(1), \cdots, e(n), \cdots\}_{clk_s}, \forall n \in \mathbb{N}_0. \quad (2.2)$$

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 (throughput, energy, etc.) are evaluated. Each event $e(n) = (g(n), \vec{v}(n))$ has a time tag $g(n)$ and a value $\vec{v}(n)$. 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 events for signals with different clocks is visualized in Figure 2.2. Two signals $s_1$ and $s_2$ have the clock timing relation $\frac{clk_{s_1}}{clk_{s_2}} = \frac{3}{2}$. The global order of the events in both signals are maintained by time tags.

![Figure 2.2: Timing relations of events in signals with different clocks.](image)

2.2.2 Process

To capture the input (output) rate in process executing cycles and absents in stalling cycles, values are a vector $\vec{v}$ of regular tokens, extended with a pseudo value $\bot$. $\vec{v}(n)$ is $\vec{v}$ when the required tokens are present, or $\bot$ when absent. The number of the regular tokens contained in $\vec{v}$ is fixed. For instance, a synchronous signal $s_1 = \{\bot, < 1, 1, 2, >, < 2, 3, 4 >, \cdots \}$ 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-clock domains, which is to be introduced in Section 3.5. In each evaluation cycle,

---

3In this chapter, the subscripted numbers in parentheses are used especially for indexing purpose.

4It is an event model both for performance analysis and for functional correctness analysis.

5It 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 [16].
processes consume one event from the input signals and output one event to the
output signals. In perfect synchrony [35], 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
\( p_{\text{comb}}(f) \) and unit-cycle delay processes \( p_\Delta(v_{(0)}) \), in which function \( f \) specifies the
mapping from input events to output events and the given initial state \( v_{(0)} \) is the
output event at time tag 0 to defer the input events one cycle.

**Figure 2.3: Process skeleton of a mealy state machine.**

While \( p_{\text{comb}}(f) \) is suited for describing algorithmic function flow, its combination
with \( p_\Delta(v_{(0)}) \) can be used to construct control logic and more complex components.
For instance, the \( \tau \)-cycle \( (\tau \in \mathbb{N}) \) delay process is \( p_\Delta,\tau(\perp) = p_\Delta(\perp) \circ \cdots \circ p_\Delta(\perp) \); a combinational process with \( \tau \)-cycle computation latency is \( p_{\text{comb},\tau}(f) = p_{\text{comb}}(f) \circ p_\Delta,\tau(\perp) \); and a **mealy** state machine, with the state function \( f_{\text{state}} \) and output function \( f_{\text{out}} \), has the process skeleton shown in Figure 2.3. The **mealy** state machine
is the base to construct the control logic of a finite size FIFO [54].

### 2.3 Simulation based on synchronous MoC

Here, the self-timed scheduling [58] 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 in the left of Figure 2.4. 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.

\[ \text{The process composition operator } \circ \text{ has the formal definition } p_1 \circ p_2(s_1) = p_1(p_2(s_1)) \]
2.4 DISCUSSIONS

Specifications parameters:

\[ n_{i,j} = 2 \quad m_{i,j} = 3 \]
\[ n_{j,k} = 1 \quad m_{j,k} = 2 \]
\[ T = [t_{C,i}, t_{C,j}, t_{C,k}] = [2, 2, 2] \]
\[ \Gamma = [\gamma_{i,j}, \gamma_{j,k}] = [6, 2] \]

Figure 2.4: A self-timed schedule of the example application (see Figure 2.1) with specified specification parameters.

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 \( \text{FIFO}_{i,j} \). At time tag 2, \( p_i \) finishes the previous computation, emits 2 tokens into \( \text{FIFO}_{i,j} \), and reserves another 2 tokens space from \( \text{FIFO}_{i,j} \) 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 \( p_i \) and \( \text{FIFO}_1 \), \( p_2 \) 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 \( \Gamma' \)) 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 \( L_{\text{period}} = 6 \), 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 \( \rho_k = \frac{1}{L_{\text{period}}} = \frac{1}{6} \) at process \( p_j \) and requires buffer storage \( \Gamma = [6, 2] \), which are the maximum buffer usages at each FIFO.

2.4 Discussions

Using deterministic scheduling policies, such as self-timed execution, the above simulation based way can construct schedules at design time with guaranteed throughput in periodic phases [29, 27] (see the schedule in Figure 2.4). Accordingly, the buffer requirement can be analyzed in a finite length of time\(^7\), without everlasting simulation.

However, a systematic way to find application schedules accordingly to different design concerns, such as energy efficiency and buffer minimization, is still lacking, especially when multi-clock domains in MPSoCs or run-time reconfigurations in adaptive systems need to be considered.

\( ^7 \) For the Proposition and Proof, see Section 3.5.3
In the following chapters, the thesis will present the author’s methods to address these issues. Especially, they are

- a multi-clocked synchronous MoC framework for energy efficiency on heterogeneous MPSoCs in Chapter 3;
- a buffer minimization method based on data stream event models and constraint analysis to schedule SDF models on a limited number of processors in Chapter 4;
- a performance analysis framework based on iterative timing phases for reconfigurations in adaptive systems in Chapter 5.
Chapter 3

Energy Efficiency of Multi-Clocked MPSoCs

3.1 Introduction

MPSoCs based on NoC communication have been increasingly adopted as the physical architectures for streaming applications [28, 64]. In streaming applications, a process corresponds to the given computation running on an MPSoC processor. 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 MPSoC 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 [20]. 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 in 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-clock synchronous modeling framework.
2. Timing properties are guaranteed by application throughput analysis.
3. High system energy efficiency on MPSoC architecture is achieved with customized processor and memory components from optimized pipeline parallelism.

3.2 Related work

Various models of computation (MoCs) [35, 43] 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) [42]. In [22], 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 dataflow (SDF) MoC, the static input/output computation tokens allow for the construction of periodic schedules with bounded memory size at compile-time [40]. Based on the static memory allocation for 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 [29, 60]. Using the timed SDF MoC, a method to minimize the memory size by scheduling the processes appropriately for maximal application throughput is introduced in [29]. In [60], the trade-offs between any specified throughput constraints and the corresponding minimal memory requirements are further studied. Nevertheless, memory is only one of several factors to impact the total system energy cost, besides computation and communication logics; meanwhile, the single unit-time assumption [29, 60] does not suit to model the heterogeneous computation and communication timing in MPSoCs.

The synchronous MoC has an explicit global order of events [35]. It has met industrial success to program safety-critical reactive systems in Esterel [13], Lustre [31] and Signal [39]. Especially, Lustre and Signal/Polychrony [30] define multi-clock models for heterogeneous embedded systems. In this chapter, the synchronous MoC framework presented in Section 2.2 is extended to multi-clock domains, and is advocated to analyze timing related properties at the system level in the design space exploration for 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 [32, 44, 59]. The performance of the application can be determined and optimized at design-time based on the following assumptions:

- a predictable NoC (guaranteed bandwidth and bounded latency);
3.3. MOTIVATION

- a predictable Network Interface (NI);
- a bounded computation time (by the WCET) of each process on processing elements.

In contrast to existing work, this chapter leverages the degrees of customizability of both processor voltage-frequency levels and memory sizes, captures the heterogeneous timing on a multi-clock 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.

3.3 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.

3.3.1 Design exploration

In design option a), the computation latency list is assigned as $T = [1, 2, 2]$. Using the memory minimization techniques in [60], 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 2.3, 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 $p_1$ is executing with 1 time slot left and $p_2$ and $p_3$ 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 1, $p_1$ 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 $p_1$ and $FIFO_{i,j}$, $p_2$ 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 ($T_3' = [0, 1, 0]$ and $\Gamma_3' = [6, 1]$) 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 $p_1$ at half speed as $T = [2, 2, 2]$ and keeps the same FIFO

\(^1\)Some more practical energy models based on instruction set simulation are adopted in the proposed design space exploration flow in Section 3.4
CHAPTER 3. ENERGY EFFICIENCY OF MULTI-CLOCKED MPSOCs

<table>
<thead>
<tr>
<th>Design</th>
<th>Energy and Computation Efficiency</th>
<th>Memory Efficiency</th>
<th>Process</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>γ = [2, 2, 2]</td>
<td>Ψ = [6, 2]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>4</td>
</tr>
</tbody>
</table>

Figure 3.1: Self-timed schedules and energy efficiency analysis for different design options on an instance of the example application (n\textsubscript{i,j} = 2, m\textsubscript{i,j} = 3, n\textsubscript{j,k} = 1, and m\textsubscript{j,k} = 2).
3.3. MOTIVATION

sizes $\Gamma$; 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 $\text{FIFO}_{i,j}$. 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$^2$.

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

$$E_{\mu p_y} = n \cdot \hat{P}_{\mu p_y,p_x} (v_y^2, f_y).$$

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

$^2$For the formal definition, see Section 3.5.3

$^3$Here, the assumed models are only used for illustration. In practical experiments, the computation latency and energy dissipation are got from simulation (see Section 3.4).
energy $E_U$ and total memory energy $E_M$ are

$$E_U = E_{\mu p_1} + E_{\mu p_2} + E_{\mu p_3} = \sum_{\forall \mu p_y} t_x \cdot r_z \cdot \hat{P}_{\mu p_y, p_z}(v_H, f_H)$$

$$= 1 \cdot 3 \cdot 1 + 2 \cdot 2 \cdot 1 + 2 \cdot 1 \cdot 1 = 9$$

$$E_M = E_{\text{mem}_1} + E_{\text{mem}_2} = \sum_{\forall \text{mem}_y} \hat{E}_{\text{mem}_y}(\gamma_j)$$

$$= 1 \cdot 6 + 1 \cdot 2 = 8$$

The total system energy consumption is $E_{\text{Sum}} = E_U + E_M = 17$. 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 $E_{\text{Sum}}$. In addition, it achieves the highest average processor utilization $\eta$ (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 3.7.

### 3.4 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.

The C program for each computation process is cross-compiled into a binary using the ARM gcc toolchain [2]. 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 3.3.1.

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 [66] 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-clock synchronous MoC framework.

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 [19] search) are used during the design options searching until the design goals are met.

3.5 Multi-clock synchronous MoC framework

The synchronous MoC framework (see Section 2.2) is adopted for both timing related properties analysis, e.g., application throughput analysis and system energy evaluation. The static computation input/output rates in the SDF MoC are preserved in the synchronous model. Furthermore, multi-clock domains are considered and exploited to relate the heterogeneous timing in different system domains (similar to Lustre [31]), which suit different clocks in heterogeneous MPSoCs well.

3.5.1 Multi-clock 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-clock synchronous domains are introduced.

When multi-clock 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 3.3, 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
Chapter 3. Energy Efficiency of Multi-Clocked MPSoCs

Figure 3.3: Multi-clock synchronous domains and domain interfaces.
3.5. MULTI-CLOCK SYNCHRONOUS MOC FRAMEWORK

the following definitions:

\[
\text{upDI}\left(m_A\right)\left(\{\vec{v}(0), \vec{v}(1), \vec{v}(2), \ldots\}^{clk_x}\right) = \\
\left\{\vec{v}(0), \perp, \ldots, \perp, \vec{v}(1), \perp, \ldots, \perp, \vec{v}(2), \ldots\right\}^{clk_x}_{m_A-1}
\]

\[
\text{downDI}\left(m_B\right)\left(\{\vec{v}(0), \ldots, \vec{v}(m_B), \vec{v}(m_B+i), \ldots\}^{clk_x}\right) = \\
\left\{\vec{v}(0), <\vec{v}(1), \ldots, \vec{v}(m_B), >, <\vec{v}(m_B+i), \ldots, \vec{v}(2m_B), >, \ldots\right\}^{clk_x}_{m_B \cdot clk}
\]

where \(<\perp, \ldots, \perp, >\) is reduced to \perp

In Figure 3.3.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, \(\lambda_{A:B} = \frac{3}{2} < 1\) 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 3.3.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 3.3, 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 design space exploration in 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.

Lemma 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 3.4. \(\forall a_1 \in \mathbb{N}_0, \exists b_1 \in \mathbb{N}_0,\)

\[
DI_{A:B}\left(\{\ldots, \vec{v}_{A(a_j)}, \ldots\}^{clk_A}\right) = \{\ldots, \vec{v}_{B(b_j)}, \ldots\}^{clk_B},
\]

where \(\{\vec{v}_{B(b_j)}\}^{clk_B} = \{< \ldots, \vec{v}_{A(a_j)}, \ldots>\}^{clk_B}.\)
s.t. the timing relation \( \text{Tag}(\vec{v}_B(b_1)) \geq \text{Tag}(\vec{v}_A(a_1)) \) exists, in which the operator \( \text{Tag} \) 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.

\[
\text{Tag}(\vec{v}_B(b_1)) = b_1 \cdot \text{clk}_B
\]

\[
\text{Tag}(\vec{v}_{\text{Tmp}}(m_B \cdot b_1)) = m_B \cdot b_1 \cdot \text{clk}_{\text{Tmp}} = b_1 \cdot \text{clk}_B
\]

Thus, it concludes \( \text{Tag}(\vec{v}_B(b_1)) \geq \text{Tag}(\vec{v}_A(a_1)) \).

### 3.5.2 Scheduling state and cross domain analysis

As discussed in Section 3.3, 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-clock application consists of a set of synchronous domains \( D \), in which domain \( D_N \) with the slowest domain clock \( \text{clk}_N \) is chosen as the logic mold domain. Each component (process or FIFO) status signal in \( D_K (\forall D_K \in D) \) can be cast across domain boundary into the single \( D_N \) via \( D_{IK,N} \), where \( \lambda_{K,N} \leq 1 \). Such a single component status signal casting \( (\lambda_{K,N} = \frac{3}{4}) \) can be illustrated by Figure 3.3.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 an encoding module \( \text{patternEnc} \) is shown in Figure 3.5. In \( s_{\text{patternSt}} \), the scheduling patterns at time tags 1 and 4 are duplicated, and are encoded as the same 1 in \( s_{\text{encSt}} \). In this way, the timing properties for multi-clock applications can be analyzed by casting the system scheduling states in multi-clock domains into the single mold domain \( D_N \).

\[
\begin{array}{c|c}
\text{0} & \text{S}_{\text{patternSt}} \\
\hline
\langle 0 \rangle & \langle<0>\rangle \\
\langle<1,1,1,\perp>\rangle & \langle<1,1,1,\perp>\rangle \\
\langle<1,2,1,\perp>\rangle & \langle<1,2,1,\perp>\rangle \\
\langle<3,1,\perp,4>\rangle & \langle<3,1,\perp,4>\rangle \\
\langle<3,1,\perp,1,\perp>\rangle & \langle<3,1,\perp,1,\perp>\rangle \\
\vdots & \vdots \\
\end{array}
\]

\[
\begin{array}{c|c}
\text{S}_{\text{encSt}} & \\
\hline
\langle 0 \rangle & 0 \\
\langle<1,1,1,\perp>\rangle & 1 \\
\langle<1,2,1,\perp>\rangle & 2 \\
\langle<3,1,\perp,4>\rangle & 3 \\
\langle<3,1,\perp,1,\perp>\rangle & 4 \\
\vdots & \vdots \\
\end{array}
\]

\[\text{patternEnc}\]

Figure 3.5: Scheduling patterns encoding.

### 3.5.3 Application throughput analysis and throughput guarantees

**Proposition 1.** (Throughput guarantees) For a consistent SDF streaming application (see Section 4.1), a periodic phase (see Section 4.3.1) 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 4.3.1) 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. \( \square \)
CHAPTER 3. ENERGY EFFICIENCY OF MULTI-CLOCKED MPSOCS

In the logic mold domain $D_N$, when the application scheduling state $(T'_N(n_2), \Gamma'_N(n_2))$ 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 $g_\Delta = g(n_2) - g(n_1)$. In this periodic phase, the process throughput of $p_n$ is defined as

$$Thru(p_n) = \frac{r_n \cdot n_{pn} \cdot Sz_{token}}{g_\Delta} \quad (3.3)$$

in which $r_n$ is the firing times of $p_n$, $n_{pn}$ the output rate considered and $Sz_{token}$ the data size per token. When $Sz_{token}$ in bit and the corresponding physical time to $g_\Delta$ are known, equation (3.3) 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.

3.6 Energy efficiency design on MPSOcs

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

3.6.1 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 3.6. Each tile contains one processor ($\mu p$) 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 [29], which allow buffer sharing among the FIFOs mapped to the same memory module, are not considered.

As the mapping optimization on the NoC communication has been studied in [34] 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.

As the mapping optimization on the NoC communication has been studied in [34] 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 3.6. The MPSoC architectural characteristics are captured in the implementation model, with the following strategies.
3.6. ENERGY EFFICIENCY DESIGN ON MPSOCs

![Diagram showing implementation and architecture models with domain mappings and communication interfaces.]

Figure 3.6: Mapping of the implementation model of the example application (Figure 2.1) onto a NoC based MPSoc architecture model.

1. Each process $p_x$ is mapped to one on-tile processor $\mu p_y$, each FIFO $FIFO_z$ 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 3.6, 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 sizes.

5. The NoC logic is assumed to provide a guaranteed bandwidth based on a hard real-time communication backbone [47] between different tiles. In Figure 3.6, the inter-tile data token transmission delay quantified by process $p_{\Delta,\tau_{NoC}}$ is
predictable, with

\[
\tau_{NoC} = \left\lceil \frac{m_{p_\Delta} S_{slots}}{w_{NoC}} + t_{hop} \right\rceil \cdot \frac{1}{\Delta \cdot k_C} \tag{3.4}
\]

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

### 3.6.2 Design objectives: application throughput and energy efficiency

The architectural design options on the processor voltage-frequency levels are customized by the domain interface configurations, and memory sizes 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 [66]. As shown in Figure 3.7, 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 techniques as introduced in Section 3.5.3 are used for the resource-aware application throughput analysis.
3.6 ENERGY EFFICIENCY DESIGN ON MPSOCS

Within a given throughput requirement, the amount of stream data transmitted via the NoC communication is fixed during some period of time. In [34], 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 $\mu_p_y$ 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 $\text{mem}_y$ consumes $\dot{E}_{\text{mem}_y}$; otherwise, the processor and memory are in active mode, they also consume the dynamic energy ($\dot{E}_{\mu_p_y}$ and $\dot{E}_{\text{mem}_y}$) of the executing process profiled by Sim-Panalyzer and Cacti.

While satisfying a given application throughput requirement $\text{Thru}_0$ upon the sink process $p_{\text{sink}}$, the aim is to minimize the overall system energy $E_{\text{Sum}}$, which is the energy summation of all the processor and memory modules. The design objectives and constraints are formalized as the following:

$$\min \ E_{\text{Sum}} = \sum_{\forall \mu_p_y \in U} \dot{E}_{\mu_p_y} + \sum_{\forall \text{mem}_y \in M} (\dot{E}_{\text{mem}_y} + \dot{E}_{\text{mem}_y})$$

(3.5)

where

$$\dot{E}_{\mu_p_y} = \sum_{\forall p_x \in P} (t_x \cdot r_x \cdot \dot{P}_{\mu_p_y,p_x} (v_x^2, f_y) \cdot \varphi_{p_x,\mu_p_y})$$

$$\dot{E}_{\text{mem}_y} = \dot{E}_{\text{mem}_y} (\sum_{\forall \text{FIFO}_x \in F} (\gamma_x \cdot \varphi_{\text{FIFO}_x,\text{mem}_y}))$$

$$\dot{E}_{\text{mem}_y} = g(\Delta) \cdot \dot{P}_{\text{mem}_y} (\sum_{\forall \text{FIFO}_x \in F} (\gamma_x \cdot \varphi_{\text{FIFO}_x,\text{mem}_y}))$$

subject to

$$\text{Thru}(p_{\text{sink}}) \geq \text{Thru}_0$$

$$\sum_{\forall \mu_p_y \in U} \varphi_{p_x,\mu_p_y} = 1, \ \forall \varphi_{p_x,\mu_p_y} \in \{0,1\}$$

$$\sum_{\forall \text{mem}_y \in M} \varphi_{\text{FIFO}_x,\text{mem}_y} = 1, \ \forall \varphi_{\text{FIFO}_x,\text{mem}_y} \in \{0,1\}$$

in which $\varphi_{p_x,\mu_p_y}$ and $\varphi_{\text{FIFO}_x,\text{mem}_y}$ are the decision variables (equals to 0 or 1) to determine the mapping from $P$ onto $U$ and $F$ onto $M$, $\dot{P}_{\text{mem}_y}$ the static power function of the specified $\text{mem}_y$, and $g(\Delta)$ the length of the periodic phase.
The design space exploration has the complexity $O(n|U|+|F|)$, which is NP-hard regarding the problem size. Thus, the widely used heuristic algorithms (i.e., greedy and taboo [19] search) are used for the design options optimization.

### 3.7 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 [28]) on a NoC based $4 \times 4$ 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 3.8. 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 3.8. 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 as the logic mold domain in the multi-clock 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
3.7. EXPERIMENTAL RESULTS

Table 3.1: Voltage-Frequency levels.

<table>
<thead>
<tr>
<th>Voltage (V)</th>
<th>2.5</th>
<th>1.8</th>
<th>1.4</th>
<th>1.2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frequency (MHz)</td>
<td>233</td>
<td>180</td>
<td>140</td>
<td>100</td>
</tr>
</tbody>
</table>

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 kb/s (Thru-1), 533 kb/s (Thru-2) and 400 kb/s (Thru-3) respectively.

As the baseline, an Ad-hoc design method is served as the reference. The minimal memory requirements (Min-Memory) technique in [60] does not have constraints on the energy consumption of the whole, but only optimize the design in memory usage. Both heuristic Greedy and Taboo [19] 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. 3.5 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 3.9. Being aware of the global timing, the schedules modeled in different clock domains are elaborated on a normalized time axis. Each processor µpi corresponds to the processor on the tile marked with i in Figure 3.8. In addition, the computations on all the µpi,n (0 ≤ n ≤ 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.

---

4From 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. [34] adopts.
Figure 3.9: 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 µp₄ₙ (0 ⩽ n ⩽ 9) represents the symmetric computations in the pipeline stage 4.
3.8 Concluding remarks

Table 3.2: Experimental results.

<table>
<thead>
<tr>
<th></th>
<th>Thru-1</th>
<th>Thru-2</th>
<th>Thru-3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$E_U$</td>
<td>$E_M$</td>
<td>$E_{Sum}$</td>
</tr>
<tr>
<td>Ad-hoc</td>
<td>28.1</td>
<td>181.2</td>
<td>209.2</td>
</tr>
<tr>
<td>Min-memory</td>
<td>33.9</td>
<td>152.3</td>
<td>186.2</td>
</tr>
<tr>
<td>Greedy</td>
<td>9.3</td>
<td>159.0</td>
<td>168.3</td>
</tr>
<tr>
<td>Taboo</td>
<td>10.3</td>
<td>155.6</td>
<td>165.8</td>
</tr>
</tbody>
</table>

*The unit of energy is mJ.

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 $\delta$ 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 $\eta$ 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 $\eta$ 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 $\delta$ 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.

3.8 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-clock
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 sizes 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

Scheduling with Buffer Minimization

4.1 Introduction

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 [7]) makes the global analysis of heterogeneous software/hardware (SW/HW) systems essential. Meanwhile, the real-time streaming applications on such platforms always have stringent demands on non-functional properties (e.g., timing, design cost, energy dissipation), besides functional correctness. To capture all the design concerns globally, while making optimal design decisions at an early system level, is still a big challenge.

Figure 4.1: The example application (see Figure 2.1) mapped onto the hybrid CPU/FPGA architecture.
Synchronous data flow (SDF) has been widely used to model and analyze streaming applications on single-/multi-processors [40, 41]. The tutorial example application model (see Figure 2.1) is mapped 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 tightly coupled memory for data operations. 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.

4.2 Related work

Lee and Messerschmitt [40] 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. [10] have taken buffer minimization into consideration using heuristics in PASS (but not PAPS) construction.

To provide timing guarantees, Govindarajan et al. [29] and Stuijk et al. [60] 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, both of their work do not consider the computation resource constraints when multiple processes mapped onto a single processing element (processor). Thus, none of these techniques 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 [29], 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 the efficiency of linear programming. It is argued to dimension the correct minimal buffer requirement. Furthermore, they use heuristics to estimates buffer requirement before a valid schedule is found, and need iterations to finalize the scheduling. The proposed constraint based scheduling framework constructs the event models as cumulative functions on the number of tokens in data streams, similar as in [18]. The streaming application execution semantics on a limited number (less than the number of processes) of processing resources and design concerns on throughput

\(^1\)To analyze timing related properties, the timed-SDF model is a timed extension to the regular untimed SDF model in [40].
4.3. MOTIVATION

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 [23], without the relaxation of integer variables.

In [46], 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.

In this chapter, the author aims to provide such kind of buffer minimization schedules for real-time SDF streaming applications on hybrid CPU/FPGA architectures. Different from all the previous (operational) work mentioned, this chapter focuses on describing the constraint based scheduling problems (a declarative way), but apply the existing successful constraint optimization techniques [3] for problem solving. The optimal schedules found have preserved sanity and minimized buffer requirement.

4.3 Motivation

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

4.3.1 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) 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 [40] 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 [40]) 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 \( \Gamma = [3, 4] \).

\(^2\)The iteration number of the minimal repetitive pattern (see Section 2.1).
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.
4.4. WORK FLOW

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 [40]. This chapter intends to provide a systematic way to construct optimal schedules with minimal buffer requirement and higher throughput guarantees.

4.4 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 $\tau$ 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 $\Gamma_{\text{min}}$; otherwise, it increases the considered $\tau$ with $\Delta_{\tau}$ and goes back to **Step 1**.

![Figure 4.3: Generic constraint based scheduling work flow.](image-url)
Apparently, only when the throughput guarantees are met in Step 2, the output results are valid. The initial values of \( \tau \) and \( \Delta_\tau \) 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.

### 4.5 Streaming application scheduling

In this section, the event models for streaming data flows are illustrated. Subsequently, constraint based scheduling problems are formalized.

#### 4.5.1 Event models

The data stream \( s \) is captured as a timed indexed synchronous signal as defined in Section 2.2.1. The event models are constructed as cumulative functions on the number of tokens in data streams, 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, \( R_{i,j}(t) = \int_0^t s_1 \) 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, \( R'_{i,j}(t) = \int_0^t s_1 = R_{i,j}(t) \) 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, \( C_{i,j}(t) = \int_0^t s_2 \) in Figure 4.1.

#### 4.5.2 Buffer properties

While a process is executing, the extra buffer space reservation in the scheduling (see Section 2.3) 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 input side 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,

\[ D_{i,j}(t) = \begin{cases} \int_0^t s_1 + n_{i,j} & \text{if } p_i \text{ is executing} \\ \int_0^t s_1 & \text{if } p_i \text{ is stalling} \end{cases} \]

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).

![Graphical interpretation of buffer properties](image)

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

Consequently, the following buffer properties can be derived.

**Property 1.** (Backlog) The backlog \( B_{i,j}(t) \) (tokens arrived but not yet served) in buffer \( \text{FIFO}_{i,j} \) 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.

\[ B_{i,j}(t) = R_{i,j}(t) - C_{i,j}(t) + B_{0,i,j}, \quad \forall t \in \mathbb{N}_0 \quad (4.1) \]

**Property 2.** (Buffer usage) In scheduling, the buffer space in use \( B'_{i,j}(t) \) for \( \text{FIFO}_{i,j} \) (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.

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

### 4.5.3 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 \(B'_{i,j}\) for \(B'_{i,j}\).

**Constraint 1.** (Token ratios) For process \(p_j\), the \(R'_{j,k}(t)\) and \(C_{i,j}(t)\) follow the static input/output tokens ratio.

\[
R'_{j,k}(t) \cdot m_{i,j} = C_{i,j}(t) \cdot n_{j,k}
\]  

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

\[
C_{i,j}(t + t_{C,j}) - C_{i,j}(t) = m_{i,j} \cdot K_j(t + t_{C,j})
\]  

\[
D_{i,j}(t + t_{C,j}) - D_{i,j}(t) = n_{j,k} \cdot K_j(t + t_{C,j})
\]

where \(K_j(t + t_{C,j}) \in \{0, 1\}\)

in which \(K_j(t + t_{C,j})\) denotes the incremental properties of \(C_{i,j}(t + t_{C,j})\) and \(D_{i,j}(t + t_{C,j})\) in a period of \(t_{C,j}\).

**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.

\[
D_{j,k}(t) = R_{j,k}(t + t_{C,j})
\]

**Constraint 4.** (Asynchronous buffer) The incoming tokens in buffer \(FIFO_{i,j}\) takes at least \(t_{C,j}\) slots to be served by process \(p_j\).

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

**Constraint 5.** (Buffer requirement) The buffer size \(\gamma_{i,j}\) of buffer \(FIFO_{i,j}\) meets the maximum buffer space requirement.

\[
\gamma_{i,j} \geq B'_{i,j}(t)
\]

### 4.5.4 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 \(P_a\) mapped onto the same processor \(CPU_a\), at any time at most one can execute (sequentially) according to:

\[
\sum_{p_j \in P_a} W_j(t) \in \{0, 1\}, \quad \forall t \in \mathbb{N}_0
\]

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

\[
L_j(t + 1) = \frac{C_{i,j}(t + 1) - C_{i,j}(t)}{m_{i,j}} \in \{0, 1\}
\]
4.5. STREAMING APPLICATION SCHEDULING

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

**Constraint 7.** (Application output throughput) After some start-up time period \( \tau_0 \) \((\tau_0 > 0)\) with no stable output tokens, a specified output throughput \( \rho_{out} \) should be sustained at the application sink process \( p_k \).

\[
C_k(\tau_0 + c \cdot L_{\text{period}}) \geq \rho_{out} \cdot c \cdot L_{\text{period}}, \quad \forall c \in \mathbb{N}_0.
\]

in which \( L_{\text{period}} \) is the length of the periodic phase in the schedule (see Figure 2.4).

However, the problem to determine the length of \( L_{\text{period}} \) which can provide optimal buffer cost in this formulation is NP-complete itself. Empirically, its length is chosen as \( L_{\text{period}} = q \cdot \lceil \frac{\rho_{out}}{\rho_{out}} \rceil, q \in \mathbb{N} \setminus \{\infty\} \), in which \( q \) is incremental and leads to a valid \( L_{\text{period}} \) with \( \tau_0 + L_{\text{period}} \leq \tau \) (see \( \tau \) in Figure 4.3). Thus, in implementation Constraint 7 is utilized to guarantee that the buffer cost is minimized (optimal) within the a given length of \( L_{\text{period}} \). On the other hand, the length of \( L_{\text{period}} \) corresponds to the cost to implement a periodic static schedule, which can be prespecified (if validly) by the designer.

**Scheduling objective.** The scheduling objective is to find the minimal total buffer sizes as follows.

\[
\min \sum_{\forall \text{FIFO } y,z \in F} \gamma_{y,z} \quad (4.11)
\]

in which \( F \) is the set of the buffers being considered and \( \gamma_{y,z} \) is the size of buffer \( \text{FIFO}_{y,z} \), subject to the corresponding Constraint 1 - 7.

For a consistent SDF streaming application (see Section 4.1), a periodic phase (see Section 4.3.1) in its schedule always exists. The required application throughput is guaranteed by the periodic properties during this period.

4.5.5 Extension to MIMO and cyclic model

The extension of the proposed methodology to multiple input and multiple output (MIMO) models, as shown in Figure 4.5\(^3\), is intuitive. Without loss of generality, the MIMO process \( p_j \) in Figure 4.5 is used for illustration.

For MI channels of \( p_j \), their individual arrival and service functions (i.e., \( R_{j,k}(t), R_{j,l}(t), C_{i,j}(t), \text{ and } C_{l,j}(t) \)) are correlated according to input rates, and have the linear relations as follows.

**Constraint 8.** (MI linear relation)

\[
\frac{R_{j,k}(t)}{n_{j,k}} = \frac{R_{j,l}(t)}{n_{j,l}}, \quad \frac{C_{i,j}(t)}{m_{i,j}} = \frac{C_{l,j}(t)}{m_{l,j}}, \quad \forall t \in \mathbb{N}_0 \quad (4.12)
\]

\(^3\)For clarity in the graph, the FIFO modules on communication channels are omitted. Instead, a number of dots are used to denote the initial buffer token numbers.
Similarly, the MO channels have the linear relations on the output and demand functions according to output rates as follows.

**Constraint 9. (MO linear relation)**

\[
\frac{R'_{j,k}(t)}{n_{j,k}} = \frac{R'_{j,l}(t)}{n_{j,l}}, \quad \frac{D_{j,k}(t)}{n_{j,k}} = \frac{D_{j,l}(t)}{n_{j,l}}, \quad \forall t \in \mathbb{N}_0
\]  

(4.13)

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, e.g., the paths (“\(ch_{i,j} \rightarrow ch_{j,k}\)” and “\(ch_{j,l} \rightarrow ch_{l,j}\)” ) in dashed lines in Figure 4.5. Based on the complete set of paths, the scheduling methodology fits the MIMO application models well.

Furthermore, for directed cyclic graphs as shown in Figure 4.5, the data tokens required for loop initialization can be explicitly modeled as the initial token offsets in Equation 4.1 and 4.2 directly.

### 4.6 Case study

To evaluate the potential of the proposed methodology, an application of voice-band data modem [41], 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.

The proposed constraint based minimal buffer scheduling (CMBS) methodology is implemented on the public domain constraint solving toolkit Gecode [3], which
Table 4.1: Comparison of scheduling methods.

<table>
<thead>
<tr>
<th></th>
<th>PAPS</th>
<th></th>
<th>CMBS</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>throughput</td>
<td>buffer</td>
<td>time</td>
<td>throughput</td>
</tr>
<tr>
<td>#1</td>
<td>1</td>
<td>4.8e-2</td>
<td>26</td>
<td>13</td>
</tr>
<tr>
<td>#2</td>
<td>2</td>
<td>6.1e-2</td>
<td>29</td>
<td>16</td>
</tr>
<tr>
<td>#3</td>
<td>3</td>
<td>6.7e-2</td>
<td>31</td>
<td>21</td>
</tr>
<tr>
<td>#4</td>
<td>6</td>
<td>7.4e-2</td>
<td>32</td>
<td>26</td>
</tr>
<tr>
<td>#5</td>
<td>8</td>
<td>7.6e-2</td>
<td>35</td>
<td>34</td>
</tr>
<tr>
<td>#6</td>
<td>22</td>
<td>8.1e-2</td>
<td>49</td>
<td>61</td>
</tr>
<tr>
<td>#7</td>
<td>100</td>
<td>8.3e-2</td>
<td>127</td>
<td>4202</td>
</tr>
</tbody>
</table>

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

is a library written in C++. CMBS is compared with the reference scheduling method, a trivially customized PAPS. To make this comparison more reasonable, the reference PAPS method is implemented in C++ as well and run both methods on a HP xw4600 Linux workstation with Quad-Core 2.40GHZ processor and 8GB of RAM to solve the scheduling problems for the modem application. Since both Gecode 2.1.1 used and our implementation of the reference PAPS 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.

---

4 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 [40]) fit the hybrid CPU/FPGA platform.
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 $J$ [40] 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’ 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

Performance Analysis of Adaptive Systems

5.1 Introduction

Partially reconfigurable FPGAs allow a part of the hardware tasks to be reprogramed dynamically on the fly, while the rest continue their operation without being affected. This feature enhances the adaptive capability of many embedded signal processing and multi-media streaming applications working in dynamic environments.

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. The performance analysis for adaptive systems is more complex, compared with traditional non-adaptive systems, and adds new challenges.

An adaptive system in SDF streaming application model is illustrated in Figure 5.1, 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 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 $\rho_{in}$ (to the input of communication channel $ch_{i,j}$ cutting by the dashed-line), an average output throughput $\rho_{out}$ needs to be guaranteed by the application even during the reconfiguration stalls of process $p_j$. Therefore, to exploit the full potential of run-time reconfigurable (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.
CHAPTER 5. PERFORMANCE ANALYSIS OF ADAPTIVE SYSTEMS

Adaptative process

MCR stream

\[ s \]

\[ s_2 \]

\[ s_4 \]

\[ ch_{i,j} \]

\[ m_{i,j} \]

\[ n_{i,j} \]

\[ \rho_{in} \]

\[ s_{M,j} \]

\[ \rho_{out} \]

\[ M_1 \]

\[ M_p \]

Adaptive process

Figure 5.1: A minimal adaptive streaming application model.

the extent of the author’s knowledge, it is still an open topic, which has not yet
been covered by previous work.

5.2 Related work

Formal analysis at design time has been widely used in performance analysis of
heterogeneous embedded systems, such as timing properties validation, buffer di-
mensioning, and scheduling policies optimization. It has been a promising option
to overcome the limitation of simulation methods in incomplete coverage at cor-
er cases, and can thus provide conservative system properties guarantees. In
SymTA/S [53], a compositional way for scheduling analysis has been presented
based on standard periodic/sporadic event models of data streams. Network calcu-
lus [12] and real-time calculus (RTC) [14] are both a collection of methods in deter-
mimistic 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 and real-time embedded
system domain respectively. In the subsequent extension of RTC, Chakraborty et
al. [15] and Phan et al. [51] present a mode based RTC to handle the execution
dependence between processing resources and buffers caused by their state infor-
mation (i.e., the fill-level of buffers and its effect on processing resources). However,
one 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 [57, 52], 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.
5.3. RECONFIGURATION PRELIMINARIES

In the cyclo-static dataflow MoC [11], the number of tokens produced and consumed by a single task changes periodically. Since this cyclically changing behavior is known at compile-time, static schedules can be constructed when the necessary and sufficient condition for scheduling holds [11]. Furthermore, the buffer requirement can be analyzed according to the specified throughput requirement in [65, 61], similar as in SDF models [60]. 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 [63], 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. In the implementation, the existing successful optimization techniques in constraint programming (CP) domain, i.e., Gecode [3] solver, has been exploited for problem solving (to get the minimal buffer requirement without losing throughput guarantees).

5.3 Reconfiguration preliminaries

In this section, without losing generality, the reconfiguration preliminaries used in this chapter are defined based on the example application in Figure 5.1.

5.3.1 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 5.1, 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 can not be neglected.

For clarity, to avoid the use of $t_{R,j}^{M_1,M_2}$, $t_{R,j}$ is simply used to implicitly mean $t_{R,j}^{M_1,M_2}$. 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\(^1\) throughput \(\rho_{\text{in}}\). Meanwhile, a required average output throughput \(\rho_{\text{out}}\) 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.

### 5.3.2 Consistency

The consistency of SDF models (see Section 2.1) is known to be a necessary condition to allow them to be executed within bounded memory without deadlock [26]. 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. 2.2 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.

\[
r_i \cdot n_{i,j} = \sum_{x=1}^{n} r_{j}^{M_x} \cdot m_{i,j}^{M_x}
\]

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, similar as in [52], are needed to guarantee that such an equation holds for adaptive models at run-time. However, since to develop such protocols is out of the scope of this chapter, it is simply assumed that the reconfiguration scenarios addressed in this work comply with the application consistency requirement, and this work focuses on the performance analysis of them.

### 5.3.3 Design cost on architecture model

The architecture template is a partially reconfigurable FPGA, as illustrated in Figure 1.3.

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 \(\sum_{i=1}^{n} A_{M,i}\) are static (fixed). For the “just-in-time” (JIT) configuration on RTR FPGAs in Figure 1.3, the total design cost in terms of area is

\[
A_{\text{JIT}}=A_{CC} + \sum_{i=1}^{n} A_{M,i} + k_C \cdot \max(A_{M,1}, \ldots, A_{M,n}) + A_{\text{Buffer}}
\]

\(^1\)Both situations will be covered in the proposed analysis approach.
5.4. RECONFIGURATION ANALYSIS FOR ADAPTIVE SYSTEMS

in which $A_{Buffer}$ is the area cost of buffers, i.e., $FIFO_{i,j}$ and $FIFO_{j,k}$ in Figure 5.1.

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.

5.4 Reconfiguration analysis for adaptive systems

For adaptive systems with pre-specified (known at compile-time) reconfiguration scenarios, they can be modeled and simulated in a similar way as in [55]. 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.

In the following, the author is going to present a constraint based approach which fits well to capture both the streaming application execution semantics and the varying design concerns in different timing phases proposed for reconfiguration analysis.

5.4.1 Constraints on reconfigurable systems

For SDF streaming applications, their execution rules follow the semantics as described in Section 4.5.3.

For the input data stream $s_1$ to process $p_j$, a peak or average throughput $\rho_{in}$ is described as follows.

**Constraint 10.** (Source input) For the source signal process $p_i$ with computation latency $t_{C,i}$, the data stream provides a peak throughput $\rho_{in}$ according to

$$R_i(t + t_{C,i}) - R_i(t) \leq t_{C,i} \cdot \rho_{in} \quad (5.3)$$

or an average throughput $\rho_{in}$ according to

$$R_i(t + t_{C,i}) - R_i(t) = t_{C,i} \cdot \rho_{in} \quad (5.4)$$

The application output throughput $\rho_{out}$ is subject to Constraint 7.

5.4.2 Iterative timing phases for reconfgurations analysis

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

$$C_j(t' + \Delta_t) - C_j(t') \equiv 0, \quad \forall \Delta_t \in [1, t_{R,j}] \quad (5.5)$$
Since the reconfiguration stall of process \( p_j \) can happen any time \( t' \) (known at run-time) after \( t_{\text{interR},i} \) of the previous mode change, a periodic phase analysis (Constraint 12 below) is adopted after \( t_{\text{interR},i} \) to overcome the coverage of some particular reconfiguration scenarios at compile-time. In this periodic phase, the minimal backlog in the playout buffer is guaranteed by Constraint 13.

**Constraint 12.** (Periodic phase) It is specified that, at time tag \( t' \) after \( t_{\text{interR},i} \) of the previous mode change, the process and FIFO status are repeatable at time tag \( t' + L_{\text{period}} \), and the schedule enters a periodic phase with length \( L_{\text{period}} \).

\[
\begin{align*}
B_{i,j}(t') &= B_{i,j}(t' + L_{\text{period}}), \quad \forall \text{FIFO}_{i,j} \\
W_{i}'(t') &= W_{i}'(t' + L_{\text{period}}), \quad \forall p_i
\end{align*}
\]

\[\text{where} \quad W_{i}'(t') = \sum_{k=1}^{L_{\text{period}}} k \cdot C_i(t' + k)\] (5.6) (5.7)

in which the variable \( W_{i}'(t') \) and \( W_{i}'(t' + L_{\text{period}}) \) denote the process status (Section 2.3) of process \( p_i \).

**Constraint 13.** (Periodic minimal backlog) A minimal backlog \( B_{j,k}(t' + t_{\text{interR},j}) \) in the playout buffer for the sink process \( p_k \) happens \( t_{\text{interR},j} \) time period after the previous reconfiguration of adaptive process \( p_j \) at time tag \( t' = \tau_0 \). It is guaranteed by the backlog properties in a periodic phase with length \( L_{\text{period}} \).

\[
B_{j,k}(t' + t_{\text{interR},j} + \Delta t) \geq B_{j,k}(t' + t_{\text{interR},j}), \quad \forall \Delta t \in [1, L_{\text{period}}]
\] (5.8)

![Figure 5.2: Timing phases of reconfiguration analysis](image)

To dimension the necessary conservative buffer sizes for adaptive systems, the timing of reconfiguration analysis is decomposed into different iterative phases as illustrated in Figure 5.2. A stepwise exploration framework is built as follows.

**Phase I (non-reconfiguration)** After a start-up period \( \tau_0 \), a stable application throughput \( \rho_{\text{out}} \) at the sink process \( p_k \) is required in this phase (subject to Constraint 7, which holds in the following timing phases also).
5.5. EXPERIMENTAL RESULTS

**Phase I’ (periodic phase)** By imposing a periodic phase with length $L_{\text{period}}$, the periodic buffer properties are used to validate that the worst case backlog in the playout buffer $\text{FIFO}_{j,k}$ is no less than a certain amount when reconfigurations can happen at any possible time (subject to Constraint 12 and 13). In this sense, the adaptive process $p_j$ might be reconfigured at any unpredictable time with the known worst case backlog.

**Phase II (reconfiguration)** Application throughput is guaranteed when the computation of process $p_j$ is stalled by reconfiguration, which takes $t_{R,j}$ time (subject to Constraint 11).

To make the period of Phase I, I’, and II iterative on different working modes, all the valid reconfiguration scenarios can be explored.

The design objective is to find the minimal total buffer sizes as follows.

$$\min: \sum_{\forall \text{FIFO}_{i,j}} \gamma_{i,j}$$

in which the conservative size $\gamma_{i,j}$ of each buffer $\text{FIFO}_{i,j}$ should adopt their worst case dimension respectively in traversing all the reconfiguration scenarios.

### 5.5 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 5.1 and an industrial application from Thales Communications.

#### 5.5.1 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 5.2) in reconfiguration analysis can traverse all the reconfiguration scenarios.

The specification model shown in Figure 5.1 is instanced with concrete parameters (i.e., $n_{i,j} = 2$, $m_{i,j} = 3$, $n_{j,k} = 1$ and $m_{j,k} = 2$) and it is assumed that the two succeeding reconfigurations of $p_j$ have a minimum interval $t_{\text{interR,j}} = 50$ time slots.

<table>
<thead>
<tr>
<th>options</th>
<th>#1</th>
<th>#2</th>
<th>#3</th>
<th>#4</th>
<th>#5</th>
<th>#6</th>
<th>#7</th>
<th>#8</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_{R,j}$</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>10</td>
<td>15</td>
<td>20</td>
<td>40</td>
</tr>
<tr>
<td>$t_{C,j}$</td>
<td>10</td>
<td>12</td>
<td>14</td>
<td>15</td>
<td>8</td>
<td>10</td>
<td>20</td>
<td>40</td>
</tr>
<tr>
<td>$A_{M}$</td>
<td>0.5</td>
<td>0.8</td>
<td>1.0</td>
<td>1.3</td>
<td>2.5</td>
<td>3.8</td>
<td>5.0</td>
<td>10</td>
</tr>
</tbody>
</table>

Table 5.1: Design options for the adaptive process $p_j$. 

Figure 5.3: Experimental results of the example application.
5.5. EXPERIMENTAL RESULTS

First, it is assumed that different design options have varying reconfiguration time \( t_{R,j} \), as listed out in the second row of Table 5.1, but have a fixed latency \( t_{C,j} \) (i.e., \( t_{C,j} = 10 \)), and evaluate the FIFO sizes requirement. Figure 5.3a shows the minimal FIFO sizes needed upon different \( t_{R,j} \), corresponding to different output throughput \( \rho_{\text{out}} \). To consider the two design concerns \( t_{R,j} \) and \( \rho_{\text{out}} \) separately, apparently, higher \( \rho_{\text{out}} \) (\( \rho_{\text{out}} = 1/10 \) 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 5.1. 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 \( t_{C,j} \propto \left\lceil \frac{1}{\sqrt{t_{R,j}}} \right\rceil \). Although, higher \( \rho_{\text{out}} \) (\( \rho_{\text{out}} = 1/8 \)) 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 \( \rho_{\text{out}} \) 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 5.3c. 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 \( t_{R,j} = 3 \) shows the minimum cost.

5.5.2 An industrial application

![Diagram](image)

An industrial adaptive coding and modulation application from Thales communication, as illustrated in Figure 5.4, 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 5.5a, 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 \( \rho_{in} \), and an average output throughput \( \rho_{out} \) 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 \( \text{FIFO}_{j,k} \) between \( p_j \) and \( p_k \) has been partitioned into two disjoint logic FIFOs \( \text{FIFO'}_{j,k} \) and \( \text{FIFO''}_{j,k} \), as shown in the dashed box, to be analyzed individually. From the static input (output) rates of the SDF model and \( \rho_{out} \), 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 \( \rho_{in} \) 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 \( \text{FIFO}_{i,j} \) and \( \text{FIFO'}_{j,k} \) to meet the average output throughput requirement \( \rho_{out}' \) upon the peak input throughput \( \rho_{in} \) (subject to Eq. 5.3 in Constraint 10).

2. To find the minimum play-out buffer \( \text{FIFO'}_{j,k} \) and \( \text{FIFO''}_{k,l} \) for the Coder module to meet the average output requirements \( \rho_{out} \) upon the average input throughput \( \rho_{out}' \) (subject to Eq. 5.4 in Constraint 10).

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.

In both Coder and Cipher modules, the design cost increases with the output throughput, as shown in Figure 5.5b and 5.5c. The design costs of different implementation strategies for the Cipher with varying \( t_{R,j} \) are also shown in Figure 5.5c, and the design costs are not monotonic to \( t_{R,j} \). It exemplifies that the proposed reconfigurations analysis framework suits design trade-offs analysis in design options exploration, and can be applied on a series of compositional adaptive processes (modules) as well.

### 5.6 Concluding remarks

This chapter presents a constraint based performance analysis framework for adaptive real-time streaming applications. Based on the implementation on constraint
5.6. CONCLUDING REMARKS

(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 5.5: Industrial application and experimental results.
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 6

Conclusions and Future Work

6.1 Conclusions

With the increasing capacity in today’s integrated circuits, more and more functionalities are inherent on systems-on-chip (SoCs) in modern electronics consumer devices. Meanwhile, the real-time streaming applications on SoCs impose rigorous requirement not only on functional correctness but also on non-functional properties, e.g., performance, design cost and energy dissipation. To manage the design complexity and flexibility in embedded systems, design space exploration flows, which take design cost and energy efficiency into consideration, are critical.

This thesis addresses the energy and design cost efficiency problems of real-time streaming applications on heterogeneous SoCs. The work has been implemented to exploit the aspects of computation resources customization and memory storage dimensioning, besides scheduling polices optimization, on three state-of-the-art SoC architectures.

In Chapter 3, an energy efficient design exploration flow is proposed for streaming applications with guaranteed throughput on MPSoCs. Both application throughput analysis and system energy calculation have been carried out on a multi-clock synchronous MoC framework. The degrees of customizability of both processor voltage-frequency levels and memory sizes 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, the problem of constructing schedules for real-time streaming applications with minimal buffer requirement on hybrid CPU/FPGA architectures has been addressed. Based on event models of data streams, the problem has been formalized declaratively as constraint base scheduling, and solved by the constraint solver Gecode. Experiments show the flexibility of the proposed CMBS approach to construct schedules with guaranteed specified (feasible) throughput and minimized buffer requirement.
In Chapter 5, a reconfigurations analysis framework based on iterative timing phases is proposed for run-time reconfigurable (RTR) FPGAs. 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.

6.2 Future work

The energy and design cost efficiency design space exploration flow presented in this thesis can be further extended in the following directions:

- A static process mapping (allocation) from the application model to either processors or custom circuits is adopted in this thesis. In the future, the mapping flexibility can be further addressed to investigate the design trade-offs between mapping strategies and design cost.

- In Chapter 3, a constant bandwidth is assumed for each logic connection, similar as in [59, 44]. Although it can be guaranteed in hard real-time NoC communication when the communication patterns are specified, it makes the allocation work not being aware of the NoC communication contention. To model the time- and contention-aware communication bandwidth in communication protocols, such as the time-triggered NoC [56, 50], can make the future work more practical.

- The buffer dimensioning has been conducted disjointly without considering memory sharing between different logic buffers. However, the memory sizes can be compacted for different logic buffers sharing the same physical memory module. Some buffer sharing algorithms have been proposed in [48, 49, 29, 25], and are still in need to further minimize the memory sizes in the future work of this thesis.

- Both heuristics and constraint programming techniques have been exploited to solve the NP-complete problems in this thesis in a reasonable running time. Some other techniques, such as evolutionary algorithms, model checking and SAT solvers [5, 17, 21], are worthy to be addressed and compared with these techniques. Besides, to improve the propagation and searching efficiency of the constraint models is promising to solve the problem with scaled-up size or complexity (e.g., compared with static allocation strategy, mapping flexibility can increase the problem complexity dramatically).
Bibliography


