On Hard Real-Time Scheduling of Cyclo-Static Dataflow and its Application in System-Level Design

Mohamed A. Bamakhrama

On Hard Real-Time Scheduling of Cyclo-Static Dataflow and its Application in System-Level Design

PROEFSCHRIFT

ter verkrijging van
de graad van Doctor aan de Universiteit Leiden,
op gezag van Rector Magnificus prof.mr. C.J.J.M. Stolker,
volgens besluit van het College voor Promoties
te verdedigen op woensdag 12 maart 2014
klokke 15:00 uur

door

Mohamed Ahmed Mohamed Bamakhrama
geboren te Dubai
in 1983

Promotion Committee

Promotor:	Prof. dr. Ed F. Deprettere	Universiteit Leiden
Co-Promotor:	Dr. Todor P. Stefanov	Universiteit Leiden
Other Members:	Prof. dr. Petru Eles	Linköpings Universitet
	Prof. dr. Rolf Ernst	Technische Universität Braunschweig
	Prof. dr. Marco Bekooij	Universiteit Twente
	Prof. dr. Joost Kok	Universiteit Leiden
	Prof. dr. Farhad Arbab	Universiteit Leiden
	Prof. dr. Harry Wijshoff	Universiteit Leiden

On Hard Real-Time Scheduling of Cyclo-Static Dataflow
and its Application in System-Level Design
Mohamed A. Bamakhrama. -
Dissertation Universiteit Leiden. - With ref. - With summary in Dutch.

Copyright © 2014 by Mohamed A. Bamakhrama. All rights reserved.
This dissertation is licensed under the Creative Common Attribution-Share Alike 3.0 license. You can obtain a copy of this license from the following URL:
`http://creativecommons.org/licenses/by-sa/3.0/`

This dissertation was typeset using LaTeX and version controlled using Git.

ISBN 978-90-9028032-5
Printed in the Netherlands.

My prayer and my sacrifice and my life and my death are all for God,
the Lord of the worlds.

Holy Quran 6:162

Contents

Table of Contents vii

List of Figures xi

List of Tables xiii

1 Introduction 1

1.1 Current Design Challenges and Trends 3

1.1.1 Design of Concurrent Software 4

1.1.2 Designer Productivity 6

1.1.3 Real-Time Guarantees 8

1.2 Problem Statement 10

1.3 Research Contributions 11

1.4 Related Work 13

1.4.1 Hard Real-Time Scheduling of Streaming Programs 14

1.4.2 Design Flows for Hard Real-Time Streaming Systems 19

1.5 Organization of this Dissertation 20

2 Background 23

2.1 Notations 23

2.2 Parallel Execution of Programs 23

2.3 Cyclo-Static Dataflow (CSDF) 26

2.4 Real-Time Scheduling 29

2.4.1 Task Model 29

2.4.2 Scheduling Concepts 30

2.4.3 Uniprocessor Schedulability Analysis 32

2.4.4 Multiprocessor Schedulability Analysis 35

3 Automated Parallelization and Model Construction **39**
3.1 Input Programs 39
3.1.1 Top-Level Part 39
3.1.2 Implementation Part 40
3.2 Automated Parallelization 41
3.3 Model Construction 42

4 Scheduling Framework **47**
4.1 Input Streams 48
4.2 Basic Definitions 50
4.3 Deriving Periods 52
4.4 Deriving Deadlines and Start Times 57
4.5 Deriving Buffer Sizes 61
4.6 Throughput Analysis 65
4.7 Latency Analysis 67
4.8 Deriving Architecture and Mapping Specifications 70

5 System-Level Synthesis **75**
5.1 Hardware 75
5.2 Software 77
5.2.1 Scheduling Infrastructure 78
5.2.2 Communication Infrastructure 83

6 Evaluation and Results **87**
6.1 Experiment I: Evaluating Automated Parallelization and Model Construction 88
6.2 Experiment II: Evaluating Performance and Resource Usage Metrics under Periodic Scheduling 88
6.2.1 Benchmarks 89
6.2.2 Throughput Evaluation 89
6.2.3 Latency Evaluation 90
6.2.4 Processor Requirements Evaluation 91
6.2.5 Memory Requirements Evaluation 93
6.2.6 Summary of Experiment II 93
6.3 Experiment III: Validating Synthesized Systems 95

7 Summary and Future Work **99**
7.1 Suggestions for Future Work 101

Bibliography **103**

Curriculum Vitae **119**

List of Publications **120**

Samenvatting **121**

Acknowledgments **123**

List of Figures

1.1 The challenges involved in designing modern hard real-time multiprocessor streaming systems. 4
1.2 Decidability and expressiveness for popular dataflow MoCs 6
1.3 System-level design of modern embedded systems 7
1.4 Popular real-time task models and the complexity of their feasibility tests 9
1.5 Bridging dataflow MoCs and real-time task models through the proposed scheduling framework 11
1.6 Input and outputs of the proposed scheduling framework 12
1.7 Overview of the proposed design flow 14

2.1 Example of a CSDF graph that corresponds to the SANLP program in Listing 1 29

3.1 Automated parallelization and model construction 40
3.2 The parallel program corresponding to the SANLP shown in Listing 1 42
3.3 The port domains of $\mathcal{P}_{\mathtt{snk}}$ shown in Figure 3.2 43
3.4 The domains of v_1 and v_2 44

4.1 Scheduling framework 48
4.2 Occurrence of $t_{\mathrm{MIT}}(I_{i,j})$ 49
4.3 Computing $t_{\mathrm{buffer}}(I_{i,j})$ 49
4.4 Schedule $\mathbb{S}_1$ 55
4.5 Schedule $\mathbb{S}_2$ 55
4.6 Schedule $\mathbb{S}_{\mathbb{L}}$ 55
4.7 Schedule $\mathbb{S}_{\infty}$ 56
4.8 The periodic schedule for the CSDF graph shown in Figure 2.1 constructed using Theorem 4.3.1 57
4.9 Timeline of A_i and A_j when $t' \geq S_i$ 60
4.10 Timeline of A_i and A_j when $t' < S_i$ 60
4.11 Execution time-lines of A_i and A_j when $S_i \leq S_j$ 63

4.12 Execution time-lines of A_i and A_j when $S_i > S_j$ 63
4.13 Decision tree for scheduling CSDF actors as real-time periodic tasks 71
4.14 The CSDF graph corresponding to the SANLP program shown in Listing 2 72
4.15 Mapping of G_1 and G_2 onto 6 processors assuming EDF and FFD . . 73

5.1 Electronic System-Level Synthesis 76
5.2 Top-level block diagram of the hardware platform considered in this dissertation 76
5.3 Tile organization 77
5.4 Complete MPSoC architecture 78
5.5 Crossbar Topology 79
5.6 Detailed description of function `vTaskDelayUntil` 81
5.7 FIFO layout in memory and the read/write registers 83

6.1 Results of the latency evaluation 92
6.2 Minimum number of processors required by optimal and partitioned schedulers 94
6.3 Zynq-7000 SoC architecture 97

List of Tables

2.1 Summary of mathematical notations 24
2.2 Approximation rations for known bin packing heuristics 37

3.1 Deriving production/consumption rates sequences from the shortest repetitive pattern 45

4.1 Computing $\vec{D}$ and $\vec{S}$ for the CSDF graph shown in Figure 2.1 on page 29 under different values of $\vec{\eta}$. 61
4.2 Computing the buffer sizes for the CSDF graph shown in Figure 2.1 on page 29 under different values of $\vec{\eta}$. 64
4.3 Values of K_i^u and K_j^u defined in (4.49) and (4.50) for the CSDF graph shown in Figure 2.1. 68
4.4 The output paths latencies and graph maximum latency of the CSDF graph shown in Figure 2.1 on page 29 under different values of $\vec{\eta}$. . . . 68
4.5 The taskset parameters for G_1 and G_2 assuming $\mu_G = 1$ and $\vec{\eta} = \vec{1}$ for both graphs 72

6.1 Specifications of the machine on which the experiments were performed 87
6.2 Time needed to parallelize and derive the CSDF model for the benchmark programs 88
6.3 Benchmarks used for evaluating the periodic scheduling framework proposed in Chapter 4 90
6.4 Results of Throughput Comparison 91
6.5 The total amount of memory needed to realize the buffers in the communication channels under periodic and self-timed schedules 95
6.6 Programs used in Experiment III 96
6.7 Hardware platforms used in Experiment III 97
6.8 The set of synthesized systems 98

Chapter 1

Introduction

The computer was born to solve problems that did not exist before.

Bill Gates

THE current period of human history is known as the **information** age. This name stems from the fact that humanity is shifting from the traditional industry-based society that characterized the industrial revolution period between 1700-1900 CE, into a *knowledge-based* society. The knowledge which used to be concentrated only in libraries and accessible only to a small fraction of the society has now become available (mostly for free) to anyone with a computer and Internet access. This dissemination of knowledge has been mainly enabled by the advent of **electronic computers** and all the advances that they enabled and brought such as the Internet. Electronics in general, and computers in particular, have changed many aspects in our behavior and way of thinking. Today, computers resemble the "backbone" of a modern society. They are everywhere starting from mobile phones and MP3 players and all the way to satellites, airplanes, and nuclear reactors. They process daily vast amounts of data to ensure that our societies continue to function properly. Computer systems can be classified based on *their functionality* into two categories:

1. **General-purpose** systems such as Personal Computers (PC). Such systems are flexible and can be controlled directly by the user to perform a variety of tasks such as web browsing, word processing, gaming, etc.
2. **Embedded** systems such as the ones that you can find inside digital TVs, cars, trains, etc. Such systems have specific functionality and they are mostly invisible to the user. Such computer systems are said to be *embedded* within larger systems.

Most of people are familiar with the first category since they use them in their daily

life. However, almost 90% of all computer systems shipped worldwide in 2010 were actually embedded systems [MRP+11]. Embedded systems have become pervasive in our life. They are everywhere, and even though that we mostly do not see them, we still feel their presence through their actions. A very important property of embedded systems is that their correct functionality does not depend only on producing the correct result but also on producing the correct result *at the right time*. Such systems, where time is critical to the correct functionality, are called **real-time systems**. Real-time systems can be either **hard** or **soft**. A hard real-time system is one where the failure to meet the timing requirements leads to a system *failure*. In contrast, a soft real-time system is one where the failure to meet the timing requirements does not lead to a failure but to *degraded* system performance that can be tolerated. Deciding whether a system is hard or soft depends usually on the overall system requirements and the environment where the system is deployed.

Real-time systems can be further classified, based on the *type of programs* they run, into:

1. **Control** programs that wait for external events from the physical world, and then react to these events. Examples include factory automation programs running on Programmable Logic Controllers (PLC) and railway switching systems.
2. **Streaming** programs that are characterized by processing continuous *streams* of data which arrive to the system. Usually, such programs process large amounts of data within short periods of time. Examples of such programs include those used in video and audio processing, digital signal processing, and network protocol processing.

In many cases, a single embedded system can contain both control and streaming programs.

A **hard real-time streaming** system is a hard real-time system that runs a set of streaming programs. This implies that all the streaming programs running on the system have hard timing requirements (i.e., timing requirements that must be always met). Examples of such systems include: (1) collision avoidance and path planning sub-systems in Unmanned Aerial Vehicles (UAV) and self-driving cars, and (2) Software-Defined Radio (SDR) used in wireless communication. For instance, a typical value of required reaction time to new sensor data in self-driving cars is around 300 ms [Thr10]. At the same time, some self-driving cars, such as Google's self-driving car, have been reported to gather around 750 MB of data per second [Woo]. Such tremendous amount of gathered data implies the need for parallel processing in order to meet the required reaction times. In this dissertation, we tackle the problem of designing such streaming systems in an automated and systematic way such that all the programs "provably" meet their timing requirements. In the next section, we explain the current challenges in

designing such systems and their implications.

1.1 Current Design Challenges and Trends

As general-purpose computers have moved from single core processors to multicore processors [HNO97], the same has happened in the embedded systems domain. Today, embedded systems designers integrate multiple processors, hardware peripherals and memories into a single chip. Such chips are referred to as **Multiprocessor System-on-Chip (MPSoC)** [JTW05]. MPSoCs represent a good candidate for running many computationally-intensive streaming programs in hard real-time systems. For example, Thrun reported in [Thr10] that Stanford's Stanley self-driving car, which was used in DARPA 2005 challenge, employed two Intel quad-core processors in order to run the autonomous driving software, including path planning and collision avoidance algorithms. Such algorithms are also highly parallelizable [KKLR13], which means that they benefit from execution on multiprocessor systems. However, such algorithms are specified, most of the time, as sequential programs. This means that such programs must be parallelized in order to meet their timing constraints and utilize the underlying processors. Another challenge is that designing MPSoCs is becoming increasingly complex as the number of processors integrated onto a single chip keeps increasing. According to the International Technology Roadmap for Semiconductors (ITRS) report for 2011 [Int11]:

> *"In the near term, the grand challenges for design technology remain (1) power management, and (2) design productivity and design for manufacturability. In the long term, the grand challenges for design technology have been updated as (1) design of concurrent software, and (2) design for reliability and resilience."*

Therefore, we can represent the problem of designing hard real-time multiprocessor systems that run multiple streaming programs as an intersection of three different problems as shown in Figure 1.1. The gray area represents the area to which the aforementioned design problem belongs. In this gray area, the designer must produce a hard real-time multiprocessor system that runs several streaming programs in parallel with the ability to provide **temporal isolation** between the programs. Temporal isolation is the ability to start/stop programs, at run-time, without violating the timing requirements of other already running programs. In the following three sections, we describe each of the three challenges shown in Figure 1.1 in detail and the current approaches in addressing these challenges.

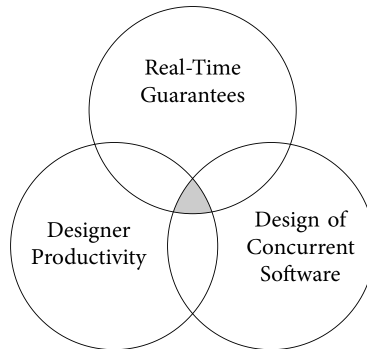


Figure 1.1: *The challenges involved in designing modern hard real-time multiprocessor streaming systems.*

1.1.1 Design of Concurrent Software

Software became a very important component in modern embedded systems. The amount and complexity of software that is running on such systems has increased dramatically over the last decades. For example, the size of embedded software in automotive systems has increased by two orders of magnitude between 1990 and 2010 [EJ09]. According to [EJ09], software is becoming a key differentiator in many domains such as automotive. A complicating factor in designing modern embedded systems is that embedded software must be written as *parallel* software in order to utilize the underlying processors in an MPSoC. Given a sequential program, researchers have identified three possible types of parallelism:

1. **Instruction-Level** Parallelism (ILP): this represents the lowest level of parallelism that is visible to the programmer. Under ILP, multiple instructions within the program may be executed in parallel.
2. **Data-Level** Parallelism (DLP): under DLP, a function (or block) within the program is executed simultaneously on several processors and each copy of the function processes its own stream of data.
3. **Task-Level** Parallelism (TLP): under TLP, the program is split into a set of functions (or *tasks*) and these functions execute in parallel.

It is important to note that a program might contain more than one type of parallelism. For example, a program might exhibit task-level parallelism and data-level parallelism. In this case, the program is split into multiple tasks that execute in parallel and a certain task has several concurrently executing replicas with each replica processing a separate stream of data. In general, identifying parallelism in a sequential program is a tedious task. This tedious task is exacerbated further by the fact that designers need to provide timing guarantees for programs running on hard real-time

multiprocessor systems. Traditionally, embedded software has been designed at the level of Board Support Package (BSP) and high-level Application Programming Interface (API) [HHBT09]. An example of a popular API for designing concurrent software is the POSIX threads (Pthreads) standard [SG13]. However, designing concurrent software at this level is known to be a cumbersome and error-prone task and it is not easy to provide timing guarantees [Lee06]. Therefore, it has been recognized that the designers need to abstract from the actual programs by building *high-level models* of them [EJ09, Tei12]. Then, these models are used to analyze the program performance under different scheduling and mapping decisions. Such design approach is often called **Model-Based Design (MBD)** or **Model-Driven Design (MDD)** and the models used in such approaches are called **Models of Computation (MoC)** [HHBT09].

In the broadest sense, a MoC defines the set of permitted operations used in computation. MoCs can be classified as either *sequential* or *parallel*. In this dissertation, we are interested in parallel MoCs as they are suitable for expressing programs that are mapped onto MPSoCs. In particular, *dataflow* MoCs have been identified as suitable parallel MoCs for expressing streaming programs [TA10]. In general, dataflow MoCs abstract the program in the form of a *directed graph*, where graph nodes represent the tasks of the program and graph edges represent the data dependencies among the tasks. Thus, parallelism is explicitly specified in the model. According to [TA10], almost all streaming programs can be modeled as Synchronous Dataflow (SDF, [LM87]) graphs. Several parallel dataflow MoCs have been proposed in the literature that vary in their *expressiveness* and *decidability* [LN05, JS05]. The expressiveness of a MoC is usually measured by its Turing completeness. On the other hand, decidability refers to the ability to perform scheduling decisions at compile-time [HO10]. If the execution order of tasks can be determined at compile-time, then the designer can decide before running the program if the program has the possibility of buffer overflow or deadlock. Decidable dataflow MoCs achieve their decidability by placing *restrictions* on the semantics of the MoC [HO10]. Generally speaking, expressiveness and decidability are inversely related as shown in Figure 1.2, which shows the expressiveness and decidability for popular dataflow MoCs.

Another development in designing parallel software is the advent of **automated parallelization** tools. Streaming programs, in general, are designed at an algorithmic level using a high-level sequential language such as C or MATLAB. Once the correctness of these sequential specifications is verified, they are passed to subsequent design stages. It has been noted that most of the execution of these sequential specifications is spent in nested loops [Fra10]. Thus, researchers have investigated several techniques for parallelizing such programs. One particular class of nested loop programs that received a lot of attention is *static control and affine indices* programs–also called *Static Affine Nested Loop Programs (SANLP)* [Fea91]. This class has been shown to embody a large

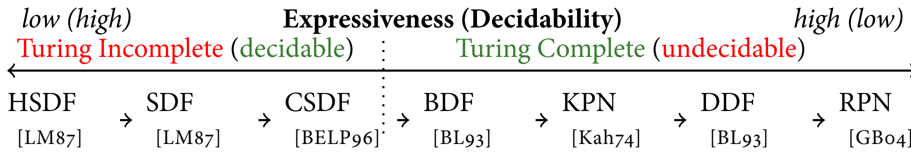


Figure 1.2: *Decidability and expressiveness for popular dataflow MoCs. The arrows between the MoCs indicate that the MoC on the left-side is a subset of the one on the right-side. For example, SDF is a subset of CSDF. The dotted vertical line represents the borderline between decidable and undecidable models.*

Listing 1 Example of a SANLP in C

```
int main() {
  while(1) {
    for(i=1;i<=10;i++) {
      for(j=1;j<=3;j++) {
        src(&img[i][j],&img1[i][j]);

        if(j<=2)
          img[i][j]=f1(img[i][j]);
        else
          img[i][j]=f2(img[i][j]);

        snk(img[i][j],img1[i][j]);
      }
    }
  }
  return 0;
}
```

portion of streaming programs [Bas04]. An example of a valid SANLP is shown in Listing 1. It has been shown in [Fea91] that a SANLP can be automatically analyzed to construct a parallel version of it. Hence, it is important to utilize this property to relieve the designer from the burden of parallelizing such programs manually. Given a sequential program, automated parallelization tools analyze the program and construct a parallel version of it. This parallel version of the program exposes the parallelism present in the original sequential program. Several parallelizing compilers have been proposed for SANLPs, such as the `PNgen` compiler [VNS07].

1.1.2 Designer Productivity

Improving designer productivity has been an active area of research since the advent of electronic systems. This area of research is called **Electronic Design Automation**

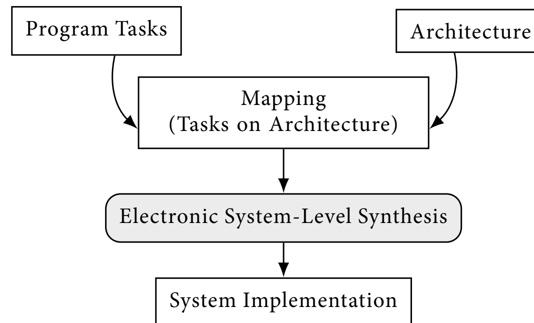


Figure 1.3: *System-level design of modern embedded systems*

(EDA). A common thing among all electronic systems is that they are made of **transistors**. Modern electronics are developed by putting an enormous number of transistors into an **Integrated Circuit (IC)** (commonly known as **chip**) that is manufactured on a single plane of silicon. Gordon Moore has predicted in 1965 [Moo65] that the number of transistors that can be put into an IC doubles approximately every 18 months. The 18 months period has been revised later to 24 months. The rationale behind this prediction is that the physical processes used to manufacture ICs keep improving. This in turn will lead to the ability to shrink the transistors, and hence, put more transistors into the same area. Moore's prediction has survived, till now, the test of time and turned into one of the most important rules of semiconductor industry, known as **Moore's law**. Therefore, as the number of transistors in a chip keeps increasing, it is necessary to design the chip at the *right level of abstraction*. In the 1960s and 1970s, electronic systems were designed at the *gate level*; designers represented their system as a set of Boolean equations and then they translated these equations into circuits composed of gates. With the number of transistors doubling every 24 months, the designers eventually had to deal with a very large number of gates. This forced the shift from *gate-level* design to *Register-Transfer Level (RTL)* in 1980s. At RTL, the designer deals no more with gates, but rather with components such as registers, adders, multipliers, etc. Again, as the number of transistors kept increasing, the designers eventually had to deal with a very large number of such components. Therefore, it became necessary to raise the level of abstraction again from RTL to *system-level* [KM+00]. At system-level, the designer deals with processors, memories, interconnects, and peripherals as the primitive blocks that form the system. System-level design has been identified as a promising design approach for MPSoCs [Hen03]. System-level design abstracts the SoC design by considering it as a process of *mapping* a set of tasks onto a set of processing elements as shown in Figure 1.3. Once such a mapping is determined, for example using design space exploration, **Electronic System-Level (ESL) synthesis** [GHP+09]

tools provide (mostly) automated procedures to generate the hardware descriptions (i.e., RTL) and the parallel software running on the processors. In addition, **Platform-Based Design (PBD)** has emerged as a de facto solution to address the design re-use problem [KM+00]. System-level design and platform-based design are closely related. Under PBD, a system is divided into *three* layers:

1. *Hardware platform*: consists of a set of processors, Input/Output (I/O) peripherals, and accelerators. The hardware components have a largely *fixed* functionality, with some degree of parameterization.
2. *Software platform*: consists of the RTOS, device drivers, and Basic I/O System (BIOS) routines. The software platform offers a set of APIs to the user programs running on the system. The software platform is sometimes called *Hardware-dependent Software (HdS)*.
3. *Programs software*: consists of the users' programs running on the system. These programs communicate with the underlying software platform through the API, which abstracts the underlying hardware platform.

1.1.3 Real-Time Guarantees

As mentioned earlier, several modern streaming programs have very high computational demands together with hard timing requirements. Such programs have two primary **performance metrics** which are **throughput** and **latency**. Throughput measures how many samples (or data-units) a program can produce during a given time interval. Latency measures the time elapsed between receiving a certain input sample and producing the processed sample by the program. Therefore, it is important to provide guaranteed throughput and latency for each program running on the designed system. Providing such guarantees depends on the system hardware and software. The hardware must be **predictable** which means that any hardware operation must have a *bounded* worst-case duration. The same applies to system software such as operating system and device drivers. In addition, the operating system scheduler must be capable of enforcing temporal isolation among the running programs on the system. Temporal isolation, as mentioned earlier, is the ability to start/stop programs, at run-time, without violating the timing requirements of other already running programs. Commercial Off-The-Shelf (COTS) multicore hardware systems have been identified as inadequate for hard real-time embedded systems [WGR+09]. Recently, several attempts have been made to propose predictable multicore hardware architectures that can be used in hard real-time embedded systems (e.g., [HGBH09, Sch09, UCS+10, Liu12]).

On the software side, several scheduling policies have been proposed for scheduling streaming programs on MPSoCs [NVC10]. For a long time, self-timed scheduling was considered the most appropriate policy for streaming programs modeled as dataflow graphs [LH89, SB09]. Under self-timed scheduling, a task is executed *as soon as possible*

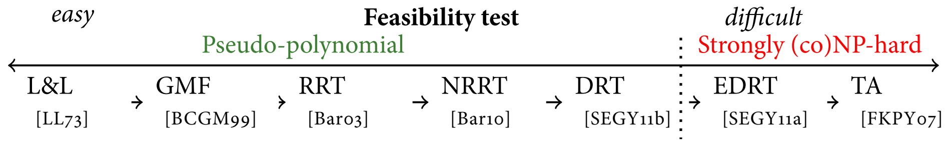


Figure 1.4: *Popular real-time task models and the complexity of their feasibility tests. The arrows between the models indicate that the model on the left-side is a subset of the one on the right-side. For example, L&L is a subset of GMF.*

(i.e., when its input data is available). However, the need to support multiple programs running on a single system without prior knowledge of the properties of the programs (e.g., required throughput, number of tasks, etc.) at system design-time is forcing a shift towards run-time scheduling approaches. Most of the existing run-time scheduling solutions assume programs modeled as task graphs and provide best-effort or soft real-time guarantees [NVC10]. Few run-time scheduling solutions exist which support programs modeled using a MoC and provide hard real-time guarantees [God98, BHM+05, Mor12]. However, these solutions use either simple MoCs such as SDF graphs or Time Division Multiplexing (TDM) scheduling.

Several algorithms from the classical hard real-time multiprocessor scheduling theory [DB11] can perform *fast* admission and scheduling decisions for incoming programs while providing hard real-time guarantees. Moreover, these algorithms enforce temporal isolation between running programs. Another key advantage of using classical hard real-time scheduling algorithms is the ability to derive in an analytical and fast way the minimum number of processors needed to schedule a set of tasks and the mapping of tasks to processors. Hard real-time scheduling theory and algorithms work in a similar way to model-based design; they abstract the programs in the form of a *real-time task model*. Such task models impose restrictions on the timing of the program tasks. As a result, it becomes easier to perform timing analysis of the program and reason about its behavior during the design phase. Several real-time task models have been proposed in the literature which differ in the complexity of their *feasibility tests* as shown in Figure 1.4. A feasibility test deals with the problem of deciding whether a given set of tasks can be scheduled to meet all their deadlines.

The most famous model is the *real-time periodic* task model proposed by Liu and Layland in 1973 [LL73]–often called Liu and Layland (L&L) model. This model has simple feasibility analysis which led to its wide adoption. However, this model places several restrictions on the tasks such as:

- The releases (i.e., invocations) of all tasks are periodic, with constant interval between releases.
- All tasks are *independent* in that releases of a certain task do not depend on the

initiation or completion of releases of other tasks.
- Execution time of each task is constant.

Several models were proposed to extend Liu and Layland model as shown in Figure 1.4. However, these extended models have more sophisticated feasibility analysis.

1.2 Problem Statement

We can summarize the discussion in Section 1.1 as follows. Model-based design and electronic system-level synthesis have emerged as de facto solutions to the problems of designing parallel software for MPSoCs and generating the complete MPSoC, respectively. However, no such de facto solution exists yet for the problem of scheduling parallel streaming programs on MPSoCs used in hard real-time systems. Scheduling has a direct influence on the architecture and mapping specifications needed to perform electronic system-level synthesis as shown in Figure 1.3. One possible and attractive solution is to use classical hard real-time scheduling algorithms due to their benefits mentioned in Section 1.1.3. However, most hard real-time scheduling algorithms assume *independent* periodic or sporadic tasks [DB11]. Such a simple task model is not directly applicable to modern streaming programs which, as mentioned in Section 1.1.1, are typically modeled as directed graphs, where graph nodes represent actors (i.e., tasks) and graph edges represent data-dependencies. The actors in such graphs have *data-dependency constraints* and do not necessarily conform to the periodic or sporadic task models. Therefore, the core problem addressed by this dissertation is to *investigate the applicability of hard real-time scheduling theory for real-time periodic tasks to streaming programs modeled as acyclic Cyclo-Static Dataflow (CSDF, [BELP96]) graphs*.

We argue that in a hard real-time system, one would like to choose, as much as possible, a decidable MoC in order to verify the timing behavior at system design time. At the same time, one would like to use the least complex (in terms of feasibility) real-time task model that is permissive enough in its semantics to model the underlying program. Therefore, in this dissertation, we consider Cyclo-Static Dataflow (CSDF) as the model of computation and the real-time periodic (i.e., L&L [LL73]) model as the real-time task model. The choice of CSDF is motivated by the fact that it is expressive enough to model most streaming programs as demonstrated in [TA10], while the choice of the real-time periodic model is motivated by the fact that it has a simple feasibility test and is widely adopted by existing real-time operating systems. By considering acyclic CSDF graphs, our findings and results are applicable to most streaming programs since it has been shown recently that around 90% of streaming programs can be modeled as acyclic SDF graphs [TA10]. Note that SDF graphs are a subset of the CSDF graphs considered in this dissertation.

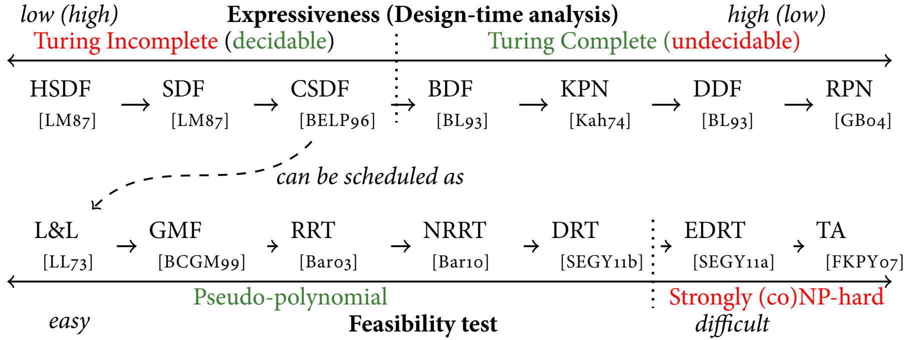


Figure 1.5: *Bridging dataflow MoCs and real-time task models through the proposed scheduling framework. The link indicates that any acyclic CSDF can be scheduled as a set of L&L tasks.*

1.3 Research Contributions

The research contributions of this dissertation can be summarized as follows.

Contribution 1: Proposing a Scheduling Framework that Bridges Dataflow MoCs and Real-Time Task Models

We propose an analytical **scheduling framework** (see [BS11, BS12]) that bridges classical dataflow models and classical real-time task models as shown in Figure 1.5. We prove analytically using this framework that any streaming program, modeled as an acyclic CSDF graph, can be executed as a set of real-time periodic tasks. The proposed framework computes the parameters (i.e., periods, start times, and deadlines) of the periodic tasks corresponding to the graph actors and the minimum buffer sizes of the communication channels such that a valid periodic schedule is guaranteed to exist. This framework shows that the use of both models is possible and that they complement each other; CSDF captures the functional aspects of the program, while the real-time periodic task model captures the timing aspects. Using both models, as demonstrated by our proposed framework, enables the designer to: (1) schedule the tasks to meet certain performance metrics (i.e., throughput and latency), (2) derive analytically the scheduling parameters that guarantee the required performance, and (3) compute analytically the minimum number of processors that guarantee the required performance together with the mapping of tasks to processors. An overview of the inputs and outputs of the proposed scheduling framework is shown in Figure 1.6. Each CSDF is annotated with the *Worst-Case Execution Time (WCET)* of its actors. The WCET values are obtained from the program through either static analysis tools or profiling the program on the target MPSoC platform [WEE+08]. The user input constraints include:

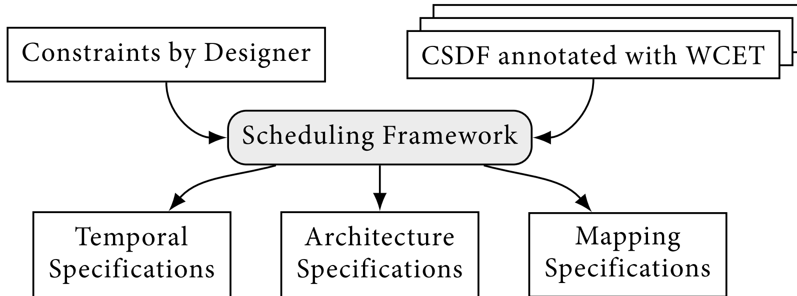


Figure 1.6: *Input and outputs of the proposed scheduling framework*

(i) type of scheduling algorithm, (ii) type of allocation algorithm, and (iii) values of parameters used to control the derivation of periods and deadlines as explained later in Chapter 4. The outputs of the framework are: (i) architecture specifications which describe how many processors are needed to schedule the programs, (ii) mapping specifications which associate each task with the processor on which it runs, and (iii) temporal specifications which consist of the parameters (i.e., periods, start times, and deadlines) of the periodic tasks corresponding to the CSDF actors together with the buffer sizes of the CSDF communication channels.

Additionally, the proposed scheduling framework establishes the following results:

- Matched I/O rates graphs (which correspond to roughly 90% of streaming programs) have a throughput under periodic schedules that is equal to their throughput under worst-case self-timed schedules. Periodic schedules here refers to scheduling the graph actors as real-time periodic tasks. This result opens the door for applying periodic scheduling to streaming programs.
- For certain classes of CSDF graphs, it is possible to achieve throughput and latency, under periodic schedules, that are equal to the throughput and latency under worst-case self-timed schedules. It is also shown that, for CSDF graphs in general, the latency can be reduced via reducing the deadlines of the actors along the critical paths.

Contribution 2: Proposing and Realizing a System-Level Design Flow that Incorporates the Proposed Scheduling Framework

In order to demonstrate the applicability of the proposed scheduling framework, we propose a system-level design flow that incorporates the proposed scheduling framework (see [BZNS12]). This design flow represents a contribution because most of the existing scheduling frameworks for streaming programs represent theoretical frameworks that have limited or no applicability in real design flows. The proposed design flow is

based on an existing state-of-the-art system-level design flow for streaming programs called Daedalus [TNS+07, NTS+08]. Similar to Daedalus, the proposed design flow starts from the sequential specifications of the programs, and then, generates in a fully automated manner the final system implementation, which provably meets the timing requirements of the programs. A complete implementation of the proposed design flow is available for download as an open source framework from http://daedalus.liacs.nl/. This implementation of the proposed design flow is called the Daedalus^{RT} design flow.

The proposed design flow is illustrated in Figure 1.7. It consists, in total, of six steps that are marked inside circles in Figure 1.7. *Step 1* accepts, as input, a set of SANLPs and then uses the `PNgen` compiler to parallelize them and generate the parallel specification of these input programs. The parallel specification consists of the Polyhedral Process Network (PPN, [VNS07]) representation of the program. PPN is a parallel MoC that is useful for code generation and optimizations. However, it is not suitable for analytical performance analysis. This leads us to the next step.

In *Step 2*, the generated PPNs in step 1 are used to construct the performance analysis model, i.e., CSDF model. Given a PPN, we use the algorithm proposed in [BZNS12] to derive a CSDF graph that is equivalent to the given PPN.

In *Step 3*, we perform WCET analysis on the parallel specification of the program. WCET analysis, as mentioned earlier, can be performed through either static analysis tools or profiling the code on the target MPSoC platform.

In *Step 4*, the CSDF models generated in step 2, the WCET values generated in step 3, and the user constraints, which include for example the type of scheduler and other parameters as explained earlier, are fed to the proposed scheduling framework explained in Contribution 1. This results in: (i) the architecture specification, which describes how many processors are needed to schedule the programs, and (ii) the mapping specification, which describes the allocation of tasks to processors.

In *Step 5*, the PPNs together with architecture and mapping specifications are processed by ESPAM [NSD08]. ESPAM is an ESL synthesis tool that supports MPSoC synthesis from PPNs. We have extended ESPAM to support synthesizing the target MPSoC hardware and software. The output from this step is a full MPSoC implementation consisting of the RTL needed to perform low-level synthesis for FPGA or ASIC together with the software running on each processor in the MPSoC.

Step 6 is the last step in the design flow and consists of performing low-level synthesis for FPGA or ASIC together with compiling the code for each processor.

1.4 Related Work

Design of hard real-time streaming systems has been an active research area for a long time. We give in this section a survey of the existing work and how it relates to

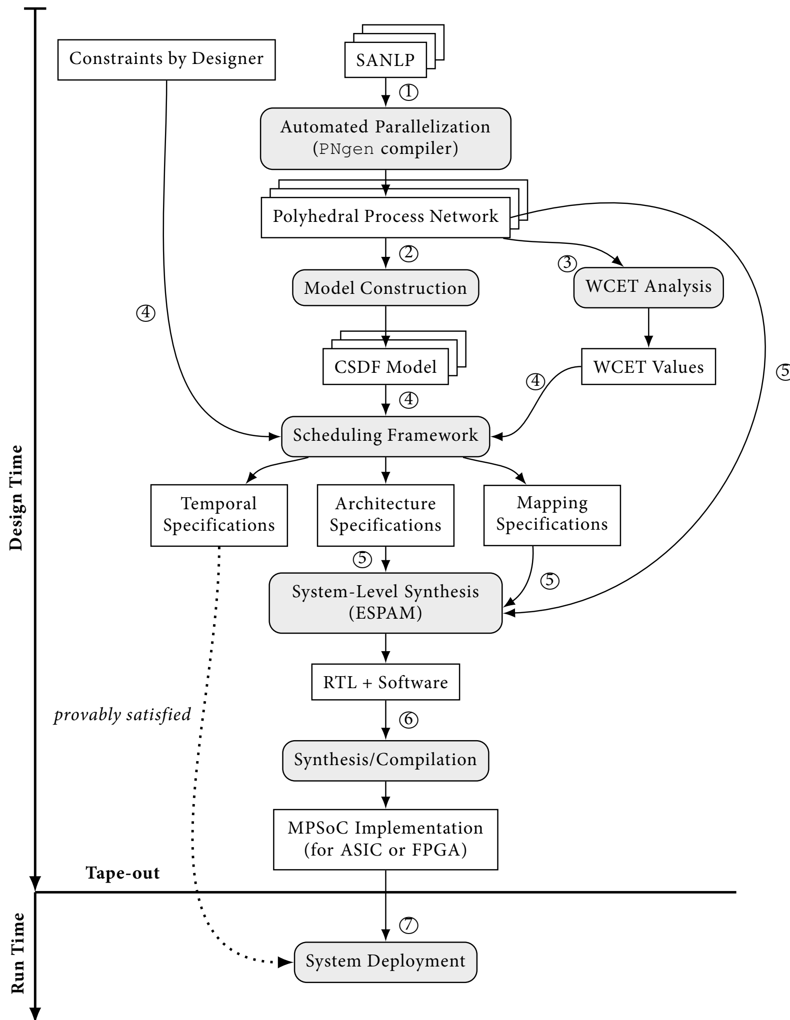


Figure 1.7: *Overview of the proposed design flow*

this dissertation. The related work is organized into two categories: hard real-time scheduling of streaming programs, and design flows for hard real-time streaming systems.

1.4.1 Hard Real-Time Scheduling of Streaming Programs

It is shown in [LM87, BELP96] that any SDF/CSDF graph can be converted into a functionally equivalent Homogenous SDF (HSDF, [LM87]) graph, where HSDF is equivalent to Computation Graphs introduced by Karp and Miller in 1966 [KM66].

Several scheduling techniques (e.g., [Mor12]) utilize this property by converting a given SDF/CSDF into an HSDF graph and then performing the scheduling analysis on the HSDF. However, the resulting HSDF graph has a size that grows exponentially with the size of the input SDF/CSDF. Therefore, such scheduling techniques have two disadvantages. First, the resulting HSDF might be huge which introduces large overhead when the actual HSDF is scheduled on the system. Second, in a periodic schedule of CSDF, each actor must have a start time and a period and each channel must have a buffer size. By deriving the schedule for the HSDF graph, it is possible to derive a start time and a period for each CSDF actor. However, it is not clear how to derive a buffer size for each CSDF channel from the buffer sizes of the HSDF.

Parks and Lee [PL95] studied the applicability of non-preemptive fixed task priority scheduling with rate monotonic priority assignment to streaming programs modeled as SDF graphs. Our work differs in the following aspects. First, they considered non-preemptive scheduling, while we consider only preemptive scheduling. Non-preemptive scheduling is known to be NP-hard in the strong sense even for the uniprocessor case [JSM91]. Second, they considered SDF graphs which are a subset of the more general CSDF graphs.

Verhaegh et al. [VLA+96] proposed Multidimensional Periodic Scheduling (MPS) to schedule digital signal processing programs written as a set of nested loops and modeled using Signal Flow Graphs (SFG). The inputs to MPS are the SFG and a set of explicit timing constraints. Given an SFG, MPS derives a schedule for each operation, where this schedule is described by multiple periods and offsets, and a buffer size for each channel such that the precedence and timing constraints are met. The scheduled operations execute in a strictly periodic manner (similar to the real-time periodic model). Verhaegh et al. showed that MPS is NP-hard and they proposed a two-stage solution approach in [VAvGL01]. The MPS framework is very similar to the framework proposed in Chapter 4 in that both frameworks derive the periods and start times of the tasks and the buffer sizes of the channels. However, the frameworks differ in the considered MoC. MPS considers SFG, while the framework in Chapter 4 considers CSDF which is more general than SFG [PC13]. Another difference is that the framework in Chapter 4 can derive a deadline for each task to reduce the graph latency.

Goddard [God98] studied applying real-time scheduling to streaming programs modeled using the Processing Graphs Method (PGM). He used a task model called *Rate-Based Execution (RBE)* which is a generalization of the real-time periodic task model. For a given PGM, he developed an analysis technique to find the RBE task parameters of each actor and buffer size of each channel. Each channel in PGM is associated with a production/consumption rate (as in SDF) and a consumption threshold. The interpretation is that the consumer consumes a number of tokens equal to the consumption rate only if the channel contains a number of tokens that is greater than

or equal to the consumption threshold. In contrast, CSDF supports a sequence of predefined production/consumption rates. As a result, the analysis technique in [God98] is not applicable to CSDF graphs.

Ziegenbein et al. [ZUE00] proposed a technique to optimize the response time of SDF graphs under Earliest Deadline First (EDF) scheduling assuming jittery input streams. Their technique accepts, as input, the SDF graphs, the average period of each actor, and the jitter bounds of the input streams. Then, they revise the deadline of each actor in such a way that the data dependencies are respected and the total response time is minimized. Our approach differs from [ZUE00] in the following aspects. First, we compute all the scheduling parameters of the tasks including the deadline, while in [ZUE00] the authors assume that the periods are given. Thus, our approach gives the designer greater flexibility in controlling the timing behavior of the tasks. Second, we consider CSDF graphs which extend the SDF model considered in [ZUE00].

Bekooij et al. [BHM+05] analyzed the impact of TDM scheduling on programs modeled as SDF graphs running on embedded real-time multiprocessor systems. Wiggers et al. [WBS09] proposed a classification scheme for run-time scheduling algorithms based on the causes of interference among programs. They identified two causes of interference which are (i) how often other tasks are executed and (ii) what execution time is associated with these executions. According to Wiggers taxonomy [WBS09], run-time scheduling algorithms are classified into:

1. *Non-starvation-free* algorithms: under such algorithms, interference depends on (i) and (ii). Examples include fixed-task priority scheduling.
2. *Starvation-free* algorithms: interference is independent of (i) but depends on (ii). Examples include round-robin scheduling.
3. *Budget-schedulers*: interference is independent of (i) and (ii). Thus, a budget scheduler guarantees every task a minimum amount of time x in every time interval of length y. Examples of budget schedulers include TDM and Constant Bandwidth Server (CBS, [AB98]).

Wiggers et al. defined a subset of dataflow graphs called *functionally deterministic dataflow graphs* and showed that such graphs have time deterministic behavior under budget schedulers. CSDF is an example of a functionally deterministic dataflow graph. In [SBW09], Steine et al. proposed a priority-based budget scheduling algorithm that overcomes some of the limitations of TDM. Recently, Hausmans et al. [HWGB13] extended the analysis in [BHM+05, WBS09] to programs modeled as arbitrary HSDF graphs when they are scheduled using algorithms of the first class according to Wiggers taxonomy. In another work, Hausmans et al. [HGWB13] proposed a two parameter (σ, ρ) workload characterization to reduce the gap between the worst-case throughput determined by the analysis and the actual throughput of the program. The (σ, ρ) workload characterization uses the average execution time of consecutive executions of a

task to provide throughput and latency guarantees. Compared to [BHM+05, HWGB13], the framework in Chapter 4 applies the analysis directly to a more expressive MoC (namely CSDF) and avoids the conversion to a larger HSDF. Compared to [WBS09, SBW09], we use the real-time periodic model which does not restrict the designer to a certain class of algorithms as defined by Wiggers taxonomy. The designer can use any algorithm that supports the underlying task model. Compared to [HGWB13], we consider "classical" hard real-time tasks, where each execution of a task must meet its deadline. In contrast, under the (σ, ρ) workload characterization, the average WCET is used to improve the minimum guaranteed throughput/latency. Thus, an internal task in the dataflow graph may miss its deadline without causing the program to violate its guaranteed throughput/latency.

Thiele and Stoimenov [TS09] proposed an analysis framework for HSDF graphs based on Real-Time Calculus (RTC, [CKT03]). Their analysis framework provides upper and lower bounds on the performance metrics under different scheduling policies (e.g., TDM and fixed task priority scheduling). An advantage of their framework is its ability to handle cyclic graphs. However, their framework acts as a general performance analysis technique that provides only upper and lower bounds on the performance. In contrast, our scheduling framework computes the tasks' parameters that guarantee a certain performance under certain scheduling policies. Moreover, we apply the analysis directly on the more general CSDF model (although acyclic graphs only), while the framework in [TS09] applies the analysis to HSDF graphs. This means that a program modeled as an SDF/CSDF graph must be converted into HSDF in order to apply the analysis in [TS09]. Such conversion has disadvantages as mentioned earlier.

Moreira [Mor12] has investigated temporal analysis of hard real-time radio programs modeled as SDF graphs. He proposed a scheduling framework based on TDM combined with static allocation. He also proved that it is possible to derive a periodic schedule for the actors of a cyclic SDF graph if and only if the periods are greater than or equal to the maximum cycle mean of the graph. He formulated the conditions on the start times of the actors in the equivalent HSDF graph in order to enforce a periodic execution of every actor as a Linear Programming (LP) problem. Our approach differs from [Mor12] in the following aspects. First, we use the periodic task model which allows applying a variety of hard real-time scheduling algorithms for multiprocessors. Second, we use the CSDF model which is more expressive than the SDF model and perform the analysis directly on CSDF instead of converting it into HSDF as done in [Mor12].

Bodin et al. [BMKdD12] studied the throughput of programs modeled as SDF graphs under K-periodic schedules. In a K-periodic schedule, a schedule of K_i occurrences of task i is repeated every O_i time units. It has been shown that self-timed schedules are K-periodic schedules [CMQV89]. Thus, K-periodic schedules achieve

the maximum throughput for a given SDF program. When $K_i = 1$ for all tasks, we obtain 1-periodic schedules which are equivalent to the schedules generated using the real-time periodic task model. Thus, K-periodic schedules serve as a powerful tool to analyze different scheduling policies. However, the realization of such schedules is more complex than the 1-periodic ones. In this dissertation, we prove the existence of 1-periodic schedules for acyclic CSDF graphs. 1-periodic schedules are easier to realize, however, this simplicity comes at the price of extra buffer requirements as shown later in Chapter 6.

Bouakaz et al. [BTV12, BT13] proposed recently a new dataflow model called *Affine Dataflow (ADF)* which extends the CSDF model. They proposed as well an analysis framework similar to ours to schedule the actors in an ADF graph as periodic tasks. They claim also that their analysis framework is capable of handling cyclic ADF graphs. An advantage of their approach is the enhanced expressiveness of the ADF model. The framework proposed in [BTV12, BT13] has been proposed after our framework and the authors in [BTV12, BT13] refer to our framework and compare empirically their framework with ours using the benchmarks explained later in Chapter 6. For most benchmarks, both CSDF and ADF achieve the same throughput and latency while requiring the same buffer sizes. However, in few cases, ADF results in reduced buffer sizes compared to CSDF [BT13].

Benabid-Najjar et al. [BNHMMK12] studied periodic scheduling of SDF graphs. For acyclic graphs, they proved that any acyclic SDF graph can be scheduled as a set of periodic tasks. For cyclic SDF graphs, they showed that the existence of a periodic schedule depends on the number of initial tokens in the graph cycles and provided a framework to derive the graph throughput under a periodic schedule. Compared to [BNHMMK12], our framework proves the existence of periodic schedules for acyclic CSDF graphs, which are more expressive than SDF graphs. Recently, Bodin et al. [BMKdD13] extended the work in [BNHMMK12] to cyclic CSDF graphs by providing a framework to derive the maximum throughput of a CSDF graph under a periodic schedule. Similar to [BT13], the work in [BMKdD13] is very recent and was proposed after our framework.

Geuns et al. [GHB13] proposed a technique to parallelize automatically sequential streaming programs containing while loops and if statements. Their technique derives a Structured Variable-rate Phased Dataflow (SVPDF) model of the parallelized program. After that, they perform scheduling analysis on the model to derive a schedule which ensures that the source and sink tasks can be scheduled in a strictly periodic fashion. A key difference between the analysis framework in [GHB13] and the framework in Chapter 4 is that the analysis in [GHB13] does not impose strict periodicity on all tasks. It is imposed only on the source and sink tasks. In contrast, the framework in Chapter 4 imposes strict periodicity on all the tasks to ensure that they conform

with the real-time periodic task model. Using the real-time periodic task model has the following advantages. First, any scheduling algorithm that supports the real-time periodic task model can be used to schedule the programs. Second, multiple programs can be scheduled, while preserving temporal isolation, as long as the programs' tasks conform to the task model and satisfy the schedulability test of the used scheduling algorithm. Third, the minimum number of processors needed to schedule the programs can be determined in a fast and analytical way.

1.4.2 Design Flows for Hard Real-Time Streaming Systems

Several design flows for automated mapping of streaming programs onto MPSoC platforms are surveyed in [GHP+09]. Most of these flows deal with soft real-time streaming systems. Additionally, these flows assume that the program model is derived manually by the designer/programmer. In contrast, our proposed design flow deals with hard real-time systems and derives the program model in a completely automated manner.

Distributed Operation Layer (DOL, [TBHH07]) is a framework for mapping parallel applications onto tiled MPSoCs. It accepts, as input, an application, which is specified as a process network, and an architecture specification. After that, it uses multi-objective optimization algorithms to perform the mapping of application to architecture. Then, it applies analytical performance analysis based on Real-Time Calculus (RTC, [CKT03]) to estimate the performance of the application after mapping. Our proposed design flow differs from DOL in the following aspects. First, we perform automated parallelization and model construction of the input applications, while DOL assumes that the input is a parallel application. Second, Real-Time Calculus gives worst-case upper and lower bounds on the performance, however, in reality these bounds may be rarely reached. In contrast, our proposed design flow guarantees a certain performance of the programs under certain scheduling policies.

PeaCE [HKL+08] is an integrated hardware/software co-design framework for embedded multimedia systems. It employs Synchronous Piggybacked Dataflow (SPDF) for computation tasks and Flexible Finite State Machines (fFSM) for control tasks. PeaCE uses hardware/software co-simulations during the design phase in order to meet certain timing constraints. In contrast, our proposed flow avoids these iterative steps by applying hard real-time multiprocessor scheduling theory to guarantee temporal isolation and a given throughput of each application running on the target MPSoC.

CA-MPSoC [SKS+10] is an automated design flow for mapping multiple applications modeled as SDF graphs onto Communication Assist (CA) based MPSoC platform. The flow uses non-preemptive scheduling to schedule the applications. In contrast, we consider only preemptive scheduling, because non-preemptive scheduling to meet all the deadlines is known to be NP-hard in the strong sense even for the uniprocessor

case [JSM91]. Moreover, we consider a more expressive MoC, namely the CSDF model.

CompSOC [GAC+13] is a platform and an associated design flow for running applications modeled as CSDF graphs. The platform part (called CoMPSoC [HGBH09]) provides predictability and *composability*, which means that the applications running on the system are completely isolated in terms of execution time, power, and access to shared resources. Similar to CoMPSoC, the hardware architecture proposed in Chapter 5 is designed to provide predictability. For the software side, CompSOC uses a custom OS called Compose that implements two-level hierarchical scheduling. In the first (or base) level, it divides the processor time into fixed intervals and uses TDM scheduling to provide complete isolation between the applications running on different intervals. In the second level, each interval may use a different scheduling policy (e.g., EDF or round-robin) to schedule the tasks executed within the interval. The associated design flow with CompSOC accepts, as input, the CSDF models of the application. Then, it uses SDF^3 [SGB06] to derive the buffer sizes and TDM interval sizes needed to guarantee a certain performance. Therefore, CompSOC provides a platform for executing given parallel applications, while our proposed design flow is concerned with providing a complete integrated design flow that parallelizes the applications and then derives the scheduling and platform parameters that guarantee a certain performance.

MAPS [Cas13] is a design flow for mapping dataflow applications onto MPSoCs. The design flow accepts, as input, a set of sequential programs written in a variant of C called *C for Process Networks (CPN)*. After that, it parallelizes these programs and generates a performance analysis model based on Kahn Process Networks (KPN, [Kah74]). Then, it uses a simulation-based composability analysis to provide certain performance guarantees on the target platform. Our proposed design flow differs from MAPS in that our flow provides hard real-time guarantees to the programs, while MAPS provides soft real-time guarantees.

MAMPSx [FSH+13] is an automated design flow for mapping applications modeled as SDF graphs onto heterogeneous MPSoCs while providing performance guarantees. The flow requires the designer to specify a sequential implementation (in C) of each actor in an SDF graph. The generated implementation uses either self-timed scheduling or TDM scheduling to ensure meeting the throughput constraints. In contrast, our proposed design flow starts from sequential applications written in C, and consequently, the parallelized programs are automatically extracted from the initial C programs. Moreover, we schedule the actors as real-time periodic tasks, which enables applying very fast schedulability analysis to determine the minimum number of required processors. Instead, [FSH+13] applies design space exploration techniques to determine the minimum number of processors and the mapping. Finally, our methodology supports multiple applications to run simultaneously on an MPSoC, while the work in [FSH+13] does not support multiple applications.

1.5 Organization of this Dissertation

The rest of this dissertation is organized as follows:

1. Chapter 2 presents an overview of dataflow models and hard real-time scheduling theory. This overview is necessary to understand the subsequent chapters.
2. Chapter 3 presents the first two stages in the proposed design flow: automated parallelization and model construction.
3. Chapter 4 presents the key contribution of this dissertation: scheduling framework for streaming programs. This framework constitutes the third stage in the proposed design flow.
4. Chapter 5 presents the fourth stage of the proposed design flow (i.e., ESL synthesis) and explains the hardware and software parts of the synthesized systems.
5. Chapter 6 presents the results of empirical evaluation of the proposed scheduling framework and design flow. This empirical evaluation is performed through a set of experiments.
6. Chapter 7 ends this dissertation with conclusions and suggestions for future work.

Chapter 2

Background

Essentially, all models are wrong, but some are useful.

George E. P. Box

THIS chapter introduces the notations, definitions, and existing results that are used in the subsequent chapters. It also contains material from the theory of dataflow models and hard real-time scheduling that is needed to understand the subsequent chapters.

2.1 Notations

We present in Table 2.1 a summary of the mathematical notations used throughout this dissertation.

Definition 2.1.1 (Partition of a Set)**.** Let V be a set. An x-partition of V is a set, denoted by ${}^{x}V$, where

$${}^{x}V = \{{}^{x}V_1, {}^{x}V_2, \cdots, {}^{x}V_x\},$$

such that each subset ${}^{x}V_i \subseteq V$, and

$$\bigcap_{i=1}^{x} {}^{x}V_i = \varnothing \text{ and } \bigcup_{i=1}^{x} {}^{x}V_i = V$$

2.2 Parallel Execution of Programs

We start by defining what we mean by a *program*.

Table 2.1: *Summary of mathematical notations*

Symbol	Meaning
$\mathbb{N}$	The set of natural numbers excluding zero
$\mathbb{N}_0$	$\mathbb{N} \cup \{0\}$
$\mathbb{Z}$	The set of integers
$\mathbb{Q}$	The set of rational numbers
$\lvert x \rvert$	The cardinality (i.e., size) of a set x
$\hat{x}$	The maximum value of x
$\check{x}$	The minimum value of x
lcm	The least common multiple operator
gcd	The greatest common divisor operator
$\div$	The integer division operator
mod	The integer modulo operator
${}^{x}V$	An x-partition of a set V (see Definition 2.1.1)

Definition 2.2.1 (Program)**.** A **program** (also called **application**) is a sequence of operations (also called statements) that transform a given input to an output.

A statement can be a simple expression (e.g., `z = x + y`), an invocation of a function (e.g., `z = f(x,y)`), or a control statement (e.g., `if(x>1)`). For some programs, the statements need to be executed in a strictly sequential way in order to maintain the correct functionality of the program. For some other programs, the statements can be executed in a parallel fashion while maintaining the correct functionality. In general, the main objective of executing the statements of a given program in parallel is to achieve a **speedup**. Let Δ_1 be the time needed to run the program on one processor, and Δ_m be the time needed to run the program on m processors. We define the speedup as:

$$\text{speedup} = \frac{\Delta_1}{\Delta_m} \tag{2.1}$$

An ideal parallel implementation of a program running on m processors achieves a speedup equal to m. However, Amdahl in 1967 [Amd67] observed that, in reality, any program consists of two portions: a *parallelizable* portion, and a *sequential* portion. The statements in the sequential portion *can not* be executed in parallel, and hence, do not benefit from execution on multiprocessor systems. Let $f \in [0,1]$ be a fraction that denotes the relative size of the parallelizable portion of a program. Amdahl showed that the actual speedup is given by:

$$\text{speedup} = \frac{1}{(1-f) + \frac{f}{m}} \tag{2.2}$$

For example, for a program where $f = 0.9$ (i.e., 90% of the program is parallelizable), the maximum speedup is:

$$\text{maximum speedup} = \lim_{m \to \infty} \frac{1}{(1-0.9) + \frac{0.9}{m}} = \frac{1}{0.1} = 10 \tag{2.3}$$

This means that the maximum speedup that can be obtained by executing the program on a multiprocessor system is 10. Therefore, executing this program on more than 10 processors does not result in an extra speedup.

The ability to execute two program statements in parallel is constrained by the *data dependencies* between them. For example, if a statement S_j requires the data produced by statement S_i, then S_j must be executed after S_i has completed its execution. To find all data dependencies in a given program, one needs to perform **dependency analysis**. Dependency analysis reveals, for a given program, all data dependencies among the statements of the program. Let S be a program where S_i and S_j represent two statements of the program. Additionally, let $\text{INPUT}(S_i)$ be the set of resources[1] read by S_i, and $\text{OUTPUT}(S_i)$ be the set of resources written to by S_i. We denote the sequential execution of S_j after S_i by $S_i \rightarrow S_j$, while we denote the parallel execution of S_i and S_j by $S_i \parallel S_j$. Bernstein [Ber66] showed that $S_i \rightarrow S_j$ and $S_i \parallel S_j$ are equivalent provided that:

1. $\text{OUTPUT}(S_i) \cap \text{OUTPUT}(S_j) = \varnothing$
2. $\text{OUTPUT}(S_i) \cap \text{INPUT}(S_j) = \varnothing$
3. $\text{OUTPUT}(S_j) \cap \text{INPUT}(S_i) = \varnothing$

The three conditions above are known in the literature as **Bernstein's conditions**. They form the basis of how we can analyze a given program to determine the statements that can be executed in parallel. During the execution of a program S, we say that a statement S_i *precedes* a statement S_j, denoted by $S_i \prec S_j$, if S_i is executed before S_j. Given a program S where S_i and S_j are two statements and $S_i \prec S_j$, one can identify the following types of data dependencies:

- **Flow (True) Dependence**: If $\text{OUTPUT}(S_i) \cap \text{INPUT}(S_j) \neq \varnothing$
- **Anti-Dependence**: If $\text{INPUT}(S_i) \cap \text{OUTPUT}(S_j) \neq \varnothing$
- **Output Dependence**: If $\text{OUTPUT}(S_i) \cap \text{OUTPUT}(S_j) \neq \varnothing$
- **Input Dependence**: If $\text{INPUT}(S_i) \cap \text{INPUT}(S_j) \neq \varnothing$

Standard dataflow analysis [ALSU86] is a body of techniques that derive the data dependencies among the statements of a given program. Array dataflow analysis [Fea91] is a technique which performs dataflow analysis for SANLP programs. Feautrier [Fea91] showed that array dataflow analysis can be used to construct a *parallel* version of a given SANLP program. This means that programs written in SANLP form can be

[1] *resource* here means hardware resources used to store data such as memory locations and registers.

automatically analyzed and parallelized. An example of a tool which implements such automated analysis and parallelization is the `PNgen` compiler [VNS07]. The details of how a parallel program is derived using automated parallelization are explained in Chapter 3.

2.3 Cyclo-Static Dataflow (CSDF)

As mentioned earlier in Chapter 1, we use in this dissertation the Cyclo-Static Dataflow (CSDF) model for modeling streaming programs. In this section, we introduce this model and its properties.

CSDF is a dataflow model that extends the well-known Synchronous Dataflow (SDF, [LM87]) model. It is defined in [BELP96] as a directed graph $G = (A, E)$, where A is a set of **actors** that correspond to the graph nodes and $E \subseteq A \times A$ is a set of **communication channels** that correspond to the graph edges. Actors represent statements in the program that transform incoming data streams into outgoing data streams, while communication channels represent data dependencies among the actors. The communication channels carry streams of data, and an atomic data object is called a **token**. A channel $E_u \in E$ is a first-in, first-out (FIFO) queue with unbounded capacity defined by a tuple $E_u = (A_i, A_j)$. The tuple means that E_u is directed from A_i (called *source*) to A_j (called *destination*). An actor receiving an input stream of the program is called *input actor*, and an actor producing an output stream of the program is called *output actor*. A **path** W_k between actors A_a and A_z is an ordered sequence of channels defined as $W_k = \{(A_a, A_b), (A_b, A_c), \cdots, (A_y, A_z)\}$. A path W_k is called *output path* if the starting actor A_a is an input actor and the ending actor A_z is an output actor. For a graph G, we use W to denote the set of all output paths in G. Each actor $A_i \in A$ is associated with two sets of channels and two sets of actors. The sets of channels are the *input channels set*, denoted by $\text{inp}(A_i)$, which consists of all the input channels to A_i, and the *output channels set*, denoted by $\text{out}(A_i)$, which consists of all the output channels from A_i. The sets of actors are the *successors set*, denoted by $\text{succ}(A_i)$, and the *predecessors set*, denoted by $\text{prec}(A_i)$. They are given by:

$$\text{succ}(A_i) = \{A_j \in A : \exists E_u = (A_i, A_j) \in E\} \quad (2.4)$$

$$\text{prec}(A_i) = \{A_j \in A : \exists E_u = (A_j, A_i) \in E\} \quad (2.5)$$

We assume that: (1) for any input actor A_i, $\text{prec}(A_i) = \varnothing$, and (2) for any output actor A_j, $\text{succ}(A_j) = \varnothing$.

Every actor $A_j \in A$ has an **execution sequence** $[f_j(1), f_j(2), \cdots, f_j(\mathcal{N}_j)]$ of length $\mathcal{N}_j$. The interpretation of this sequence is: the nth time that actor A_j is fired, it executes the code of function $f_j(((n-1) \bmod \mathcal{N}_j) + 1)$. Similarly, production and consumption rates of tokens are also *sequences* of length $\mathcal{N}_j$ in CSDF. The token

Algorithm 1 $\text{LEVELS}(G)$

Require: Acyclic CSDF graph $G = (A, E)$
1: $i \leftarrow 1$
2: **while** $A \neq \varnothing$ **do**
3: $\quad {}^{\mathbb{L}}A_i \leftarrow \{A_j \in A : \text{prec}(A_j) = \varnothing\}$
4: $\quad E'_i \leftarrow \{E_u \in E : \exists A_k \in {}^{\mathbb{L}}A_i \text{ such that } E_u = (A_k, A_l)\}$
5: $\quad A \leftarrow A \setminus {}^{\mathbb{L}}A_i$
6: $\quad E \leftarrow E \setminus E'_i$
7: $\quad i \leftarrow i + 1$
8: **end while**
9: $\mathbb{L} \leftarrow i - 1$
10: **return** $\mathbb{L}$-partition of A given by ${}^{\mathbb{L}}A = \{{}^{\mathbb{L}}A_1, {}^{\mathbb{L}}A_2, \cdots, {}^{\mathbb{L}}A_{\mathbb{L}}\}$.

production of actor A_j on channel E_u is represented as a sequence of constant integers $[x_j^u(1), x_j^u(2), \cdots, x_j^u(\mathcal{N}_j)]$. The nth time that actor A_j is fired, it produces $x_j^u(((n-1) \bmod \mathcal{N}_j) + 1)$ tokens on channel E_u. The consumption of actor A_k is completely analogous; the token consumption of actor A_k from a channel E_u is represented as a sequence $[y_k^u(1), y_k^u(2), \cdots, y_k^u(\mathcal{N}_k)]$. The firing rule of a CSDF actor A_k is evaluated as "true" for its nth firing if and only if all its input channels contain at least $y_k^u(((n-1) \bmod \mathcal{N}_k) + 1)$ tokens. The total number of tokens produced by actor A_j on channel E_u during the first n invocations is denoted by $X_j^u(n)$ and given by $X_j^u(n) = \sum_{l=1}^{n} x_j^u(l)$. Similarly, the total number of tokens consumed by actor A_k from channel E_u during the first n invocations is denoted by $Y_k^u(n)$ and given by $Y_k^u(n) = \sum_{l=1}^{n} y_k^u(l)$.

An acyclic CSDF graph has a **number of levels**, denoted by $\mathbb{L}$, and is given by Algorithm 1. Algorithm 1 builds an $\mathbb{L}$-partition of A, denoted by ${}^{\mathbb{L}}A$, by partitioning it in a way similar to *topological sort*. The actors belonging to subset ${}^{\mathbb{L}}A_i$ are said to be *level-i actors*.

An important property of the CSDF model is its *decidability*, which is, as mentioned in Chapter 1, the ability to derive at compile-time a schedule for the actors. This is formulated in Definition 2.3.1 and Theorem 2.3.1.

Definition 2.3.1 (Valid Static Schedule [BELP96])**.** Given a connected CSDF graph G, a **valid static schedule** for G is a finite sequence of actors invocations that can be repeated infinitely on the incoming sample stream while the amount of data in the buffers remains bounded. A vector $\vec{q} = [q_1, q_2, \cdots, q_{|A|}]^T$, where $q_j > 0$, is a **repetition vector** of G if each q_j represents the number of invocations of an actor A_j in a valid static schedule for G. The repetition vector of G with the smallest norm is called the **basic repetition vector** of G and is denoted by $\vec{\breve{q}}$. G is **consistent** if there exists a repetition vector. If a deadlock-free schedule can be found, G is said to be **live**. Both consistency

and liveness are required for the existence of a valid static schedule.

Bilsen et al. [BELP96] proved the following theorem:

Theorem 2.3.1. *Let G be a CSDF graph. A repetition vector* $\vec{q} = [q_1, q_2, \cdots, q_{|A|}]^T$ *of G is given by*

$$\vec{q} = \Theta \cdot \vec{r}, \qquad \text{with} \qquad \Theta_{jk} = \begin{cases} \mathcal{N}_j & \text{if } j = k \\ 0 & \text{otherwise} \end{cases} \tag{2.6}$$

where $\vec{r} = [r_1, r_2, \cdots, r_{|A|}]^T$ *is a positive integer solution of the balance equation*

$$\Gamma \cdot \vec{r} = \vec{0} \tag{2.7}$$

and where the topology matrix $\Gamma \in \mathbb{Z}^{|E| \times |A|}$ *is defined by*

$$\Gamma_{uj} = \begin{cases} X_j^u(\mathcal{N}_j) & \text{if actor } A_j \text{ produces on channel } E_u \\ -Y_j^u(\mathcal{N}_j) & \text{if actor } A_j \text{ consumes from channel } E_u \\ 0 & \text{Otherwise.} \end{cases} \tag{2.8}$$

Definition 2.3.2. For a consistent and live CSDF graph G, an **actor iteration** is the invocation of an actor $A_i \in A$ for q_i times, and a **graph iteration** is the invocation of *every* actor $A_i \in A$ for q_i times, where $q_i \in \vec{q}$.

Corollary 2.3.1 (From [BELP96])**.** *If a consistent and live CSDF graph G completes k iterations, where* $k \in \mathbb{N}$*, then the net change to the number of tokens in the buffers of G is zero.*

Lemma 2.3.1. *Any acyclic consistent CSDF graph is live.*

Proof. Bilsen et al. proved in [BELP96] that a CSDF graph is live if and only if every cycle in the graph is live. Equivalently, a CSDF graph deadlocks only if it contains at least one cycle. Thus, absence of cycles in a CSDF graph implies its liveness. ∎

Example 2.3.1. Figure 2.1 shows an example of a CSDF graph. This CSDF graph models the SANLP program shown in Listing 1. The graph has four actors which correspond to the functions in the program. The correspondence between the actors and the functions is as follows: A_1 corresponds to `src`, A_2 corresponds to `f1`, A_3 corresponds to `f2`, and A_4 corresponds to `snk`. The graph has five edges which represent the data dependencies between the functions in the program. In this graph, there is one input actor (i.e., A_1) and one output actor (i.e., A_4). The graph has three output paths given

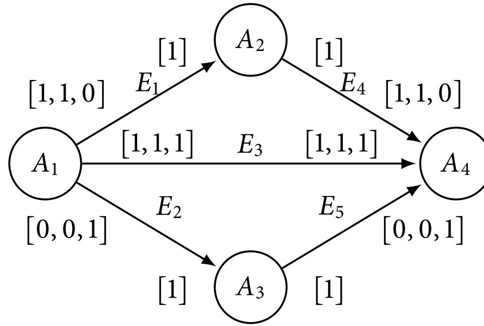


Figure 2.1: *Example of a CSDF graph that corresponds to the SANLP program in Listing 1*

by $W = \{W_1 = \{(A_1, A_2), (A_2, A_4)\}, W_2 = \{(A_1, A_3), (A_3, A_4)\}, W_3 = \{(A_1, A_4)\}\}$. Based on Theorem 2.3.1 on page 28, we compute the basic repetition vector as follows:

$$\Gamma = \begin{bmatrix} 2 & -1 & 0 & 0 \\ 1 & 0 & -1 & 0 \\ 3 & 0 & 0 & -3 \\ 0 & 1 & 0 & -2 \\ 0 & 0 & 1 & -1 \end{bmatrix}, \vec{r} = \begin{bmatrix} 1 \\ 2 \\ 1 \\ 1 \end{bmatrix}, \Theta = \begin{bmatrix} 3 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 3 \end{bmatrix}, \text{and } \vec{\breve{q}} = \begin{bmatrix} 3 \\ 2 \\ 1 \\ 3 \end{bmatrix}$$

We show later in Chapter 3 how such a graph can be automatically derived from a given SANLP program. □

2.4 Real-Time Scheduling

In this section, we introduce the real-time periodic task model, some important real-time scheduling concepts, and schedulability analysis for uniprocessor and multiprocessor systems.

2.4.1 Task Model

A system is composed of a set of m identical processors $\{\pi_1, \pi_2, \cdots, \pi_m\}$. These processors execute a set of n tasks $T = \{T_1, T_2, \cdots, T_n\}$ and a task T_i may be preempted at any time. A task T_i corresponds to a CSDF actor A_i and we model the tasks using the **real-time periodic** task model. Under the periodic task model, each task is a *recurrent* one with a constant inter-arrival time. A **task** $T_i \in T$ is characterized by a 4-tuple of integers $T_i = (S_i, C_i, P_i, D_i)$. The tuple parameters are interpreted as follows: $S_i \geq 0$ is the *start time* of T_i in absolute time units, $C_i > 0$ is the *Worst-Case Execution Time (WCET)* of T_i, $P_i \geq C_i$ is the *task period* (i.e., inter-arrival time) in relative time units, and D_i, where $C_i \leq D_i \leq P_i$, is the *deadline* of T_i in relative time units.

A periodic task T_i is invoked at time instants $r_{i,k} = S_i + kP_i$ for all $k \in \mathbb{N}_0$. When T_i is invoked, we say that T_i *releases* a **job**. The kth job (or invocation) of T_i is denoted by $T_{i,k}$. Upon invocation, a task must execute for C_i time-units. The **deadline** D_i is interpreted as follows: job $T_{i,k}$ has to finish its execution before time $d_{i,k} = r_{i,k} + D_i$ for all $k \in \mathbb{N}_0$. If $D_i = P_i$, then T_i is said to have an *implicit-deadline*. If $D_i < P_i$, then T_i is said to have a *constrained-deadline*. If all the tasks in a taskset T are implicit-deadline tasks, then we say that T is an *implicit-deadline taskset*. Otherwise, we say that T is a *constrained-deadline taskset*. Similarly, if all the tasks in a taskset T have the same start time, then we say that T is *synchronous*. Otherwise, we say that T is *asynchronous*. For synchronous tasksets, we assume that their start time is 0.

The **utilization** of a task T_i is $u_i = C_i/P_i$. For a taskset T, the total utilization of T is $u_{\text{sum}}(T) = \sum_{T_i \in T} u_i$ and the maximum utilization factor of T is $\hat{u}(T) = \max_{T_i \in T} u_i$. Similarly, the **density** of a task T_i is $\delta_i = C_i / \min(D_i, P_i)$, the total density of T is $\delta_{\text{sum}}(T) = \sum_{T_i \in T} \delta_i$, and the maximum density of T is $\hat{\delta}(T) = \max_{T_i \in T} \delta_i$. Note that the density is equivalent to the utilization for implicit-deadline tasks.

The **processor demand**, denoted by $\text{demand}(T_i, t_1, t_2)$, of a task T_i over a time interval $[t_1, t_2]$ is the total computation time of all the jobs of T_i having activation time and deadline within $[t_1, t_2]$. According to [BRH90], $\text{demand}(T_i, t_1, t_2)$ is given by:

$$\text{demand}(T_i, t_1, t_2) = \zeta(T_i, t_1, t_2) \cdot C_i \tag{2.9}$$

where $\zeta(T_i, t_1, t_2)$ is the total number of T_i jobs that are activated in the interval $[t_1, t_2]$ and have a deadline within the interval $[t_1, t_2]$. The authors in [BRH90] showed that $\zeta(T_i, t_1, t_2)$ is given by:

$$\zeta(T_i, t_1, t_2) = \max\{0, \left\lfloor \frac{t_2 - S_i - D_i}{P_i} \right\rfloor - \max\{0, \left\lceil \frac{t_1 - S_i}{P_i} \right\rceil\} + 1\} \tag{2.10}$$

For a taskset T, the processor demand of T over the time interval $[t_1, t_2]$ is denoted by $\text{demand}(T, t_1, t_2)$ and given by:

$$\text{demand}(T, t_1, t_2) = \sum_{T_i \in T} \text{demand}(T_i, t_1, t_2) \tag{2.11}$$

2.4.2 Scheduling Concepts

Given a system and a taskset T, a **correct schedule** is one that allocates a processor to a task $T_i \in T$ for exactly C_i time units in the interval $[S_i + kP_i, S_i + kP_i + D_i)$ for all $k \in \mathbb{N}_0$, with the restriction that a task may not execute on more than one processor at the same time. The schedule itself can be constructed either **offline** (i.e., at system design-time) or **online** (i.e., at system run-time). Offline scheduling allows the designer to perform computationally expensive optimizations to produce the best possible schedule

according to some criteria (e.g., minimizing energy). However, offline scheduling lacks the flexibility to deal with new events at run-time. In contrast, online scheduling can deal with such new events at run-time. However, online scheduling introduces extra overheads due to the scheduling decisions taken during run-time. In the rest of this dissertation, we assume online scheduling algorithms unless we explicitly mention otherwise.

According to [DB11], real-time multiprocessor scheduling algorithms can be viewed as attempting to solve two problems:

- The **priority assignment** problem: when and in what order with respect to other tasks, each job should execute.
- The **allocation** problem: on which processor a task should execute and whether a task can migrate between processors.

As a result, real-time multiprocessor scheduling algorithms can be classified based on **priority** as follows [DB11]:

- *Fixed task priority*: Each task has a single priority that is used for all its jobs.
- *Fixed job priority*: The jobs of a single task may have different priorities, however, each job has a single priority. An example of an algorithm using this policy is the Earliest Deadline First (EDF, [LL73]) algorithm.
- *Dynamic priority*: A single job might have different priorities during the course of its execution. An example is the Least Laxity First (LLF) algorithm.

Based on **allocation**, algorithms can be classified as follows [DB11]:

- *No migration*: Each task is allocated to a processor and no migration is permitted.
- *Task-level migration*: The jobs of a task may execute on different processors, however, a single job can only execute on a single processor.
- *Job-level migration*: A job may execute on more than one processor, however, it may not execute on more than one processor at the same time.

A scheduling algorithm that does not permit migration at all is said to be a **partitioned** scheduling algorithm, while an algorithm that permits all tasks to be migrated between all processors is said to be a **global** algorithm, and finally an algorithm that permits a subset of tasks to be migrated among a subset of processors is said to be a **hybrid** algorithm.

Graham in 1969 [Gra69] showed that a system may exhibit unexpected scheduling "anomalies" even though the system operates under a "better" set of conditions. For example, he showed that decreasing the WCET of tasks may result in an increase in the schedule length. Such anomalies are known in the literature as **Graham's anomalies** (or **scheduling anomalies**). A scheduling algorithm under which scheduling anomalies can never occur is said to be **anomaly-free** [AJ02]. Examples of anomaly-free uniprocessor scheduling algorithms include EDF and fixed-task priority scheduling with rate monotonic priority assignment. Partitioned multiprocessor scheduling algorithms in

which each processor is scheduled using an anomaly-free uniprocessor scheduling algorithm are known to be anomaly-free as well [AJ02].

In this dissertation, we focus only on partitioned scheduling due to the following reasons: (1) task migration on distributed-memory MPSoCs has a non-negligible overhead, (2) partitioned scheduling has been shown to be the most suitable type for hard real-time systems [Bra11], and (3) partitioned scheduling where each processor is scheduled using an anomaly-free uniprocessor scheduling algorithm is anomaly-free as mentioned earlier.

A taskset T is said to be **feasible** on a system if there exists a scheduling algorithm that can construct a correct schedule for T on the system. A taskset T is said to be **schedulable** using a scheduling algorithm $\mathcal{A}$ if $\mathcal{A}$ can construct a correct schedule for T on the system. Finally, a scheduling algorithm $\mathcal{A}$ is said to be **optimal** with respect to a task model and a system if and only if it can schedule all tasksets that comply with the task model and are feasible on the system.

On uniprocessor systems, EDF scheduling is known to be optimal for all periodic tasksets [But11]. On multiprocessor systems, several global and hybrid algorithms are known to be optimal for implicit-deadline periodic tasksets (e.g., [BCPV96, CRJ10, FLS+11, RLM+12]) and, in contrast, optimal online scheduling of constrained-deadline tasksets is impossible [FGB10]. Finally, partitioned scheduling is known to be non-optimal for periodic tasksets [CFH+04].

A **schedulability test** for a scheduling algorithm $\mathcal{A}$ decides, given a task set T, whether T is schedulable using $\mathcal{A}$. Schedulability tests can be classified further into [DB11]:

- **Sufficient**: If all tasksets that are deemed schedulable by the test are in fact schedulable.
- **Necessary**: If all the tasksets that are deemed unschedulable by the test are in fact unschedulable.
- **Exact**: If the test is both sufficient and necessary.

2.4.3 Uniprocessor Schedulability Analysis

On uniprocessor systems, the most popular classes of priority assignment are the *fixed-task* priority and the *fixed-job* priority. In particular, EDF scheduling is the most popular algorithm in the fixed-job class. Therefore, we present in this section the schedulability tests for EDF and fixed-task priority scheduling.

Earliest Deadline First (EDF)

Under EDF, an exact test for implicit-deadline periodic tasksets is formulated in the following theorem:

Theorem 2.4.1 (From [LL73]). *An implicit-deadline periodic taskset T is schedulable under EDF if and only if*

$$u_{sum}(T) \leq 1 \tag{2.12}$$

For constrained-deadline periodic tasks, Baruah et al. [BRH90] derived in 1990 an exact schedulability test for EDF on uniprocessor systems. The test is formulated in the following lemma:

Lemma 2.4.1 (From [BRH90]). *A periodic task set T is feasible on one processor if and only if $u_{sum}(T) \leq 1$ and $demand(T, t_1, t_2) \leq (t_2 - t_1)$ for all $0 \leq t_1 < t_2 < \hat{S} + 2H$, where $\hat{S} = \max\{S_1, \cdots, S_n\}$ and $H = \mathrm{lcm}\{P_1, \cdots, P_n\}$.*

The exact test in Lemma 2.4.1 is known to be co-NP-hard in the strong sense [BRH90]. A more efficient exact test for synchronous periodic tasksets has been derived in 2009 by Zhang and Burns [ZB09]. The test is known as the *Quick-convergence Processor-demand Analysis (QPA)*. Let T be a synchronous periodic taskset and let $\check{D} = \min\{D_1, D_2, \cdots, D_n\}$. Let L_a be defined as

$$L_a = \max\left\{D_1, D_2, \cdots, D_n, \frac{\sum_{i=1}^{n}(P_i - D_i)u_i}{1 - u_{\mathrm{sum}}(T)}\right\} \tag{2.13}$$

Let L_b be a value computed by the following recurrence equation:

$$w^0 = \sum_{i=1}^{n} C_i, \quad w^{k+1} = \sum_{i=1}^{n} \left\lceil \frac{w^k}{P_i} \right\rceil C_i \tag{2.14}$$

where the recurrence stops when $w^{k+1} = w^k$, and then $L_b = w^{k+1}$. Now, let L^* be given by

$$L^* = \begin{cases} \min(L_a, L_b) & \text{when } u_{\mathrm{sum}} < 1 \\ L_b & \text{when } u_{\mathrm{sum}} = 1 \end{cases} \tag{2.15}$$

Recall from Section 2.4.1 that $d_{i,k}$ denotes the absolute deadline of the kth job of task T_i and it is given by $d_{i,k} = r_{i,k} + D_i$. Zhang and Burns [ZB09] proved the following theorem:

Theorem 2.4.2. *A synchronous periodic taskset T is schedulable if and only if $u_{sum} \leq 1$ and the result of the following algorithm is $demand(T, 0, t) \leq \check{D}$.*

$t \leftarrow \max\{d_{i,k} : d_{i,k} < L^*\}$
while $demand(T, 0, t) \leq t$ ***and*** $demand(T, 0, t) > \check{D}$ ***do***
  if $demand(T, 0, t) < t$ ***then***
    $t \leftarrow demand(T, 0, t)$
  else

$t \leftarrow \max\{d_{i,k} : d_{i,k} < t\}$
end if
end while
if *demand*$(T, 0, t) \leq \check{D}$ ***then***
T is schedulable
else
T is unschedulable
end if

QPA can be used as a *sufficient* test for asynchronous periodic tasksets based on the following lemma from [BG04].

Lemma 2.4.2 ([BG04]). *Let* $T = \{T_1 = (S_1, C_1, P_1, D_1), T_2 = (S_2, C_2, P_2, D_2), \cdots, T_n = (S_n, C_n, P_n, D_n)\}$ *be an asynchronous periodic taskset. T is schedulable if the synchronous periodic taskset* $\tilde{T} = \{\tilde{T}_1 = (0, C_1, P_1, D_1), \tilde{T}_2 = (0, C_2, P_2, D_2), \cdots, \tilde{T}_n = (0, C_n, P_n, D_n)\}$ *is schedulable.*

Fixed-Task Priority

Under fixed-task priority scheduling, Jospeh and Pandya [JP86] derived an exact schedulability test for synchronous periodic tasksets based on *Response Time Analysis (RTA)*. Let hp(T_i) be the set of tasks with priorities higher than the priority of T_i and R_i be the total response time of T_i. Then, the schedulability test is formulated in the following theorem:

Theorem 2.4.3. *A synchronous periodic taskset T is schedulable using fixed-task priority scheduling if and only if*

$$\forall T_i \in T : R_i \leq D_i \tag{2.16}$$

where the total response time R_i *is given by solving the following fixed-point equation:*

$$R_i = C_i + \sum_{\forall T_j \in hp(T_i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \tag{2.17}$$

It is shown in [JP86] that (2.17) has a fixed-point if $u_{\text{sum}}(T) \leq 1$. The test in Theorem 2.4.3 can be used as a *sufficient* test for asynchronous periodic tasksets [Aud91].

An important question in fixed-task priority scheduling is *how to assign the priorities* to the tasks. Let prio$(T_i) \in \mathbb{N}$ denote the priority of task T_i. The lowest priority value is 1 and a higher value of prio(T_i) means a higher priority. Several priority schemes are proposed in the literature such as:

- **Rate Monotonic (RM) [LL73]**: Let T_i and T_j be two tasks. Under RM, if $P_i < P_j$, then $\text{prio}(T_i) > \text{prio}(T_j)$. Liu and Layland in [LL73] derived a sufficient schedulability test for implicit-deadline taskset under using fixed-task priority scheduling when the priorities are assigned according to the RM scheme. The test is:

  Theorem 2.4.4 (From [LL73]). *Let T be an implicit-deadline synchronous taskset where the priorities are assigned according to the RM rule and the tasks are scheduled using a fixed-task priority scheduler. T is schedulable if*

  $$u_{sum}(T) \leq n(2^{1/n} - 1) \tag{2.18}$$

  where $n = |T|$.

  If the size of the taskset grows significantly (i.e., $n \rightarrow \infty$), then we obtain $u_{\text{sum}}(T) = \ln(2) \approxeq 0.693$. This means that any implicit-deadline synchronous taskset with total utilization less than 0.69 is schedulable using fixed-task priority scheduling with RM priority assignment. The RM priority assignment rule is shown to be optimal for synchronous implicit-deadline tasksets when scheduled using a fixed-task priority scheduler [LL73].
- **Deadline Monotonic (DM) [LW82]**: Let T_i and T_j be two tasks. Under DM, if $D_i < D_j$, then $\text{prio}(T_i) > \text{prio}(T_j)$. The exact schedulability test when using the DM policy is the same test presented in Theorem 2.4.3. The DM priority assignment rule is shown to be optimal for synchronous constrained-deadline tasksets when scheduled using a fixed-task priority scheduler [LW82].
- **Optimal Priority Assignment (OPA) [Aud91]**: The optimal priority assignment policy proposed by Audsley in 1991 is an optimal priority assignment for asynchronous periodic tasksets when scheduled using fixed-task priority scheduling. The algorithms for setting the priorities and checking the schedulability can be found in [Aud91].

2.4.4 Multiprocessor Schedulability Analysis

An exact schedulability test for implicit-deadline periodic tasksets on m processors is:

$$u_{\text{sum}}(T) \leq m \tag{2.19}$$

Based on (2.19), one can compute the absolute minimum number of processors needed to schedule an implicit-deadline taskset T as follows:

$$\check{m}_{\text{OPT}} = \lceil u_{\text{sum}}(T) \rceil \tag{2.20}$$

It is important to notice that (2.19) and (2.20) are valid only assuming optimal scheduling algorithms, which are either global or hybrid. However, as mentioned

earlier in Section 2.4.2, we consider in this dissertation partitioned scheduling only. With partitioned scheduling, one must first *allocate* the tasks to processors. Then, the tasks on each processor are scheduled using a uniprocessor scheduling algorithm. Recall from Definition 2.1.1 on page 23 that ${}^{x}T$ denotes an x-partition of a taskset T. The minimum number of processors needed to schedule a taskset T assuming partitioned scheduling is given by:

$$\check{m}_{\text{PAR}} = \min\{x \in \mathbb{N} \mid \exists\, {}^{x}T \text{ and } \forall i : {}^{x}T_i \text{ is schedulable } \} \tag{2.21}$$

Partitioning Schemes

Let V be a set of n items and B be a set of m containers (i.e., bins). Each item $V_i \in V$ has a size, denoted by $\textsc{size}(V_i)$, where $\textsc{size}(V_i) \in [0, 1]$. Similarly, each container B_i has a maximum capacity equal to 1, and a current capacity, denoted by $\textsc{capacity}(B_i)$, which gives the total size of items placed currently onto B_i. We seek for V an m-partition, denoted by ${}^{m}V$, such that the total size of the items in each ${}^{m}V_i$ is less than or equal to the maximum capacity of container B_i. That is

$$\forall i = 1, 2, \cdots, m : \sum_{V_j \in {}^{m}V_i} \textsc{size}(V_j) \le 1 \tag{2.22}$$

Solving the aforementioned partitioning problem requires solving the classical *bin packing* problem which is known to be NP-hard [GJ79]. Therefore, many heuristics exist for partitioning items over bins such as First-Fit, Best-Fit, Worst-Fit, etc. Below, we summarize from [Joh74, CGJ96] some of the most used heuristics.

- **First-Fit** (FF): FF is a straightforward solution to the assignment problem. An item V_i is placed in the first (i.e., lowest indexed) bin B_j that can accommodate V_i. That is
$$j = \min\{k : \textsc{size}(V_i) + \textsc{capacity}(B_k) \le 1\} \tag{2.23}$$
If no such bin exists, then create a new bin and place V_i into it.
- **Best-Fit** (BF): BF places an item V_i into a bin B_j such that B_j will have minimal remaining capacity after placing V_i. That is
$$j = \min\{k : \textsc{size}(V_i) + \textsc{capacity}(B_k) \text{ is closest to, without exceeding, } 1\} \tag{2.24}$$
If no such bin exists, then create a new bin and place V_i into it.
- **Worst-Fit** (WF): WF is the opposite of BF as it tries to place an item V_i into a bin B_j that will have maximal remaining capacity after placing V_i. That is
$$j = \min\{k : \textsc{size}(V_i) + \textsc{capacity}(B_k) \text{ is minimized } \} \tag{2.25}$$
If no such bin exists, then create a new bin and place V_i into it.

Table 2.2: *Approximation rations for known bin packing heuristics*

Heuristic	Approximation ratio	Source
FF	$\mathfrak{R}_{\text{FF}} = 17/10$	[CGJ96]
BF	$\mathfrak{R}_{\text{BF}} = 17/10$	[GJ79]
FFD	$\forall\ V : \text{FFD}(V) \leq 11/9 \cdot \text{OPT}(V) + 1$	[Yue91]

It is also possible to perform a *preprocessing* step before performing the heuristic. This preprocessing step is usually sorting the items based on some criteria, such as their sizes. This leads us to the following heuristics:

- **First-Fit-Decreasing** (FFD): In this heuristic, the items are first sorted, in decreasing (i.e., descending) order, based on their sizes. Then, FF is performed on the sorted items.
- **Best-Fit-Decreasing** (BFD): Similar to FFD, the items are first sorted, in descending order, based on their sizes. Then, BF is performed on the sorted items.

An important metric in evaluating the performance of the aforementioned heuristics is their **approximation ratio**. Let $\text{OPT}(V)$ be the minimum number of bins needed to partition a set of items V using the *optimal* partitioning scheme. Then, for a given heuristic $h(V)$, its approximation ratio, denoted by $\mathfrak{R}_h$, is given by

$$\mathfrak{R}_h = \inf\{r \geq 1 : h(V)/\text{OPT}(V) \leq r \text{ for all } V\} \tag{2.26}$$

Several approximation ratios are available for the heuristics described earlier. These ratios are summarized in Table 2.2.

Chapter 3

Automated Parallelization and Model Construction

A good scientist is a person with original ideas.
A good engineer is a person who makes a design
that works with as few original ideas as possible.

Freeman Dyson

RECALL from Figure 1.7 on page 14 that the first two steps in the proposed design flow are *automated parallelization* and *model construction*. In this chapter, we explain, in detail, these two steps and how they are performed in the proposed design flow. It is important to note that these two steps are not contributions of this dissertation but they are presented here to make the dissertation self-contained.

3.1 Input Programs

We assume that an input program is written in the ANSI 89 C programming language and consists of two parts: a *top-level* part and an *implementation* part. These two parts are explained in detail in the next two sections.

3.1.1 Top-Level Part

This part consists of the `main` function of the program. It defines the *top-level* structure of the algorithm realized by the program and intended for parallelization. We assume that the `main` function is written as a Static Affine Nested Loop Program (SANLP) and does not contain anti-dependencies (see Section 2.2). During parallelization, the automated parallelization tool processes only this part to extract all the dependencies

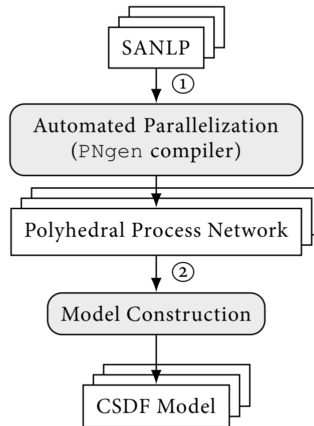


Figure 3.1: *Automated parallelization and model construction*

and generate a parallel version of it. An example of a valid top-level file is shown in Listing 1.

In [VNS07], SANLPs have been defined as *programs that can be represented using the polytope model*. However, we prefer to use a definition that characterizes the semantics of the program. Such a definition is the following one from [Mei10].

Definition 3.1.1 (Static Affine Nested Loop Program (SANLP)). A SANLP is a program where each program statement is enclosed by one or more loops and if-statements, and where:

- C1 loops have a constant step size;
- C2 loops have bounds that are affine expressions of the enclosing loop iterators, static program parameters, and constants;
- C3 if-statements have affine conditions in terms of the loop iterators, static program parameters, and constants;
- C4 index expressions of array references are affine constructs of the enclosing loop iterators, static program parameters, and constants;
- C5 data flow between statements in the loop is explicit, which prohibits that two statements that contain function calls communicate through shared variables outside the `main` function.

3.1.2 Implementation Part

This part consists of all the remaining functions of the program. These functions can contain arbitrary code as long as this code does not break condition C5 from Definition 3.1.1. An example of an implementation part is the implementation of functions `src`, `f1`, `f2`, and `snk` from Listing 1.

3.2 Automated Parallelization

This step is realized using the PNgen compiler [VNS07]. PNgen accepts as input the top-level part written in SANLP form. After that, it performs array dataflow analysis [Fea91] on the SANLP and translates each non-control statement in the top-level part into a separate *process*. A process encapsulates the statement from which it got translated and represents a separately executing entity[1] in the parallel version of the program. The processes communicate among each other using FIFOs in accordance with the semantics of Kahn Process Networks (KPN, [Kah74]). In KPN, the processes are sequential programs that communicate over FIFOs and a process blocks when it attempts to read from an empty FIFO.

Since we consider programs consisting of nested loops, each process (i.e., statement) is executed for a number of iterations. The set of iterations in which a process is executed is called *process domain*. The process domain is represented using the polytope model [Fea96]. A polytope $\mathcal{D}$ is a bounded subset of $\mathbb{Q}^n$ given by

$$\mathcal{D} = \{\vec{x} \in \mathbb{Q}^n \mid F\vec{x} \geq \vec{b}\} \tag{3.1}$$

where $F \in \mathbb{N}^{k \times n}$ and $\vec{b} \in \mathbb{N}^k$. Since a process encapsulates a non-control statement, the process has to *read* the input data needed by the statement, and *write* the output data produced by the statement. Reading and writing data is done through *process input/output ports*. The ports of a process are connected to the FIFO channels interconnecting it with other processes. Similar to the process domain, PNgen constructs *input/output domains* as polytopes. Due to the fact that PNgen represents the execution and communication using the polytope model, the generated process-based representation is called *Polyhedral Process Network (PPN, [VNS07])*. To illustrate the concepts of process, input/output ports, and domain, we give the following example.

Example 3.2.1. Consider the program in Listing 1 on page 6. PNgen processes the program in Listing 1 to produce the parallel program shown in Figure 3.2. The arrows represent the FIFO channels and the black dots represent the ports. READ and WRITE represent the primitives used to read and write from the FIFOs, respectively. The detailed implementation of these primitives is explained later in Chapter 5. For example, process $\mathcal{P}_{\texttt{snk}}$ has three input ports, represented by IP1, IP2, and IP3. Since function snk does not produce any data for other functions, process $\mathcal{P}_{\texttt{snk}}$ does not have any output ports. IP1 provides the data produced by process $\mathcal{P}_{\texttt{f1}}$, IP2 provides the data produced by process $\mathcal{P}_{\texttt{f2}}$, and IP3 provides the data produced by process $\mathcal{P}_{\texttt{src}}$. The process domain of process $\mathcal{P}_{\texttt{snk}}$ is given by:

$$\mathcal{D}_{\texttt{snk}} = \{(w, i, j) \in \mathbb{Z}^3 \mid w > 0 \land 1 \leq i \leq 10 \land 1 \leq j \leq 3\}$$

[1]This entity can be mapped into an OS process or thread.

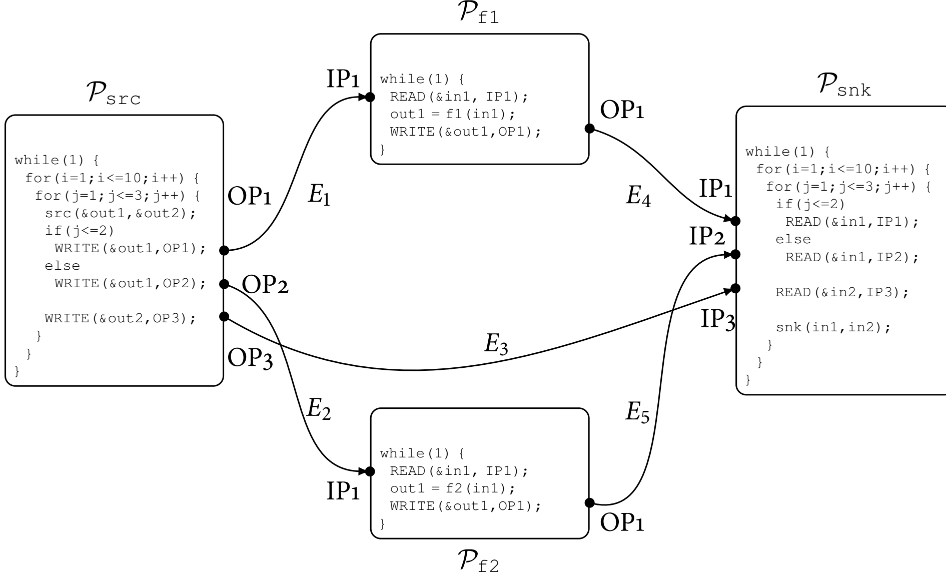


Figure 3.2: *The parallel program corresponding to the SANLP shown in Listing 1. This parallel program is produced using the* `PNgen` *compiler.*

Reading data produced by function `f1` to initialize the first argument `in1` of function `snk` is represented by the following input domain:

$$\mathcal{D}_{\mathrm{IP1}} = \{(w, i, j) \in \mathbb{Z}^3 \mid w > 0 \wedge 1 \leq i \leq 10 \wedge 1 \leq j \leq 2\}$$

In a similar way, reading data produced by function `f2` to initialize the first argument `in1` of function `snk` is represented by the following input domain:

$$\mathcal{D}_{\mathrm{IP2}} = \{(w, i, j) \in \mathbb{Z}^3 \mid w > 0 \wedge 1 \leq i \leq 10 \wedge j = 3\}$$

Finally, reading data produced by function `src` to initialize the second argument `in2` of function `snk` is represented by the following input domain:

$$\mathcal{D}_{\mathrm{IP3}} = \{(w, i, j) \in \mathbb{Z}^3 \mid w > 0 \wedge 1 \leq i \leq 10 \wedge 1 \leq j \leq 3\}$$

These port domains are visualized in Figure 3.3. The dots encapsulated by rectangles represent the port domains, and the arrows show the domain execution (i.e., scanning) order. □

3.3 Model Construction

Model construction is the second step in the proposed design flow shown in Figure 1.7 on page 14. As mentioned in Section 1.3, we adopt the CSDF model as a performance

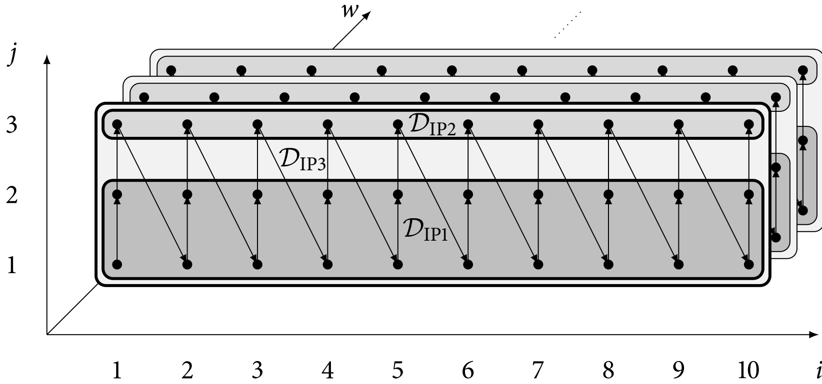


Figure 3.3: *The port domains of* $\mathcal{P}_{snk}$ *shown in Figure 3.2*

analysis model in the proposed design flow. It is straightforward to see that PPNs generated by `PNgen` share the following similarities with CSDF:

- Both of PPN and CSDF are represented as a directed graph, where nodes correspond to either *processes* (in PPN) or *actors* (in CSDF), and edges correspond to FIFOs.
- For a given PPN, the equivalent CSDF graph has the same topology as the PPN.

However, PPN and CSDF differ in the following aspects:

- Communication in PPN is represented using input/output ports domains, while in CSDF, it is represented using sequences of production/consumption rates.
- Synchronization in PPN is implemented by the blocking read/writes, while in CSDF, it is implemented explicitly using a schedule.

Deprettere et al. showed in [DSBS06] that for any non-parameterized PPN, it is possible to derive a functionally equivalent CSDF graph, where the production and consumption rates sequences consist of only 0s and 1s. A '0' in the sequence indicates that a token is not produced/consumed, while a '1' indicates that a token is produced/consumed. The authors in [BZNS12] proposed an algorithm to derive automatically the CSDF graph from a given PPN. We show how this algorithm works through the following example.

Example 3.3.1. Consider the PPN shown in Figure 3.2. The first step in deriving the equivalent CSDF graph is to derive the CSDF topology. The CSDF graph (shown in Figure 2.1 on page 29) has the same topology as the PPN in Figure 3.2. The second step is to derive the access patterns of the processes. For each process in the PPN shown in Figure 3.2, a *process variant* is derived. A process variant v of a process $\mathcal{P}$ is a tuple $(\mathcal{D}_v, ports)$, where $\mathcal{D}_v \subseteq \mathcal{D}_{\mathcal{P}}$ is the variant domain, and *ports* is a set of input/output ports that are accessed in all iterations of $\mathcal{D}_v$. A process variant captures

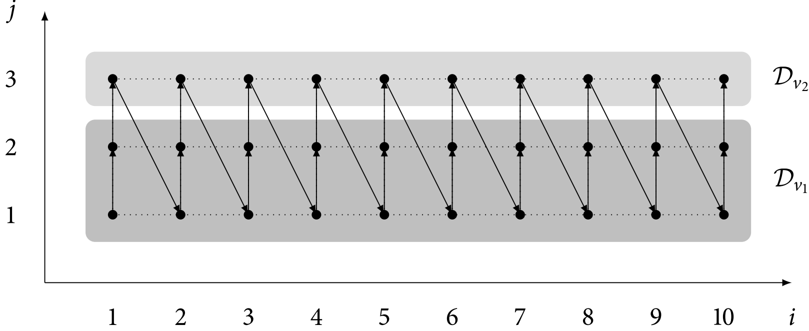


Figure 3.4: *The domains of v_1 and v_2. Dimension w is omitted from the picture.*

the consumption/production behavior of the process. For example, consider process $\mathcal{P}_{\texttt{snk}}$ in Figure 3.2. It has the following process variants:

$$v_1 = (\{(w,i,j) \in \mathbb{Z}^3 \mid w > 0 \wedge 1 \le i \le 10 \wedge 1 \le j \le 2\}, \{\text{IP1}, \text{IP3}\}) \tag{3.2}$$

$$v_2 = (\{(w,i,j) \in \mathbb{Z}^3 \mid w > 0 \wedge 1 \le i \le 10 \wedge j = 3\}, \{\text{IP2}, \text{IP3}\}) \tag{3.3}$$

Process $\mathcal{P}_{\texttt{snk}}$ always reads data from input ports IP1 and IP3 in process variant v_1, while it always reads data from input ports IP2 and IP3 in process variant v_2. The domains of v_1 and v_2 are depicted in Figure 3.4.

The third step is to construct a one dimensional sequence of process variants by scanning the process domain. To do that, we first project out dimension w from all the domains. Then, we build a sequence of the process domain points using their lexicographical order. For example, for process $\mathcal{P}_{\texttt{snk}}$, the process domain can be represented as the following sequence of loop iterator values (i, j) according to the loop execution shown in Figure 3.4 (i.e., lexicographical order):

$$\{(1,1),(1,2),(1,3),\cdots,(2,1),(2,2),\cdots,(10,3)\} \tag{3.4}$$

After that, we replace each iteration with the process variant to which it belongs. This results in the following sequence:

$$\mathcal{S}_{\texttt{snk}} = \{v_1, v_1, v_2, v_1, v_1, v_2, \cdots, v_1, v_1, v_2\} \tag{3.5}$$

In general, sequence $\mathcal{S}_{\mathcal{P}}$ has a length equal to $|\mathcal{D}_{\mathcal{P}}|$, hence, it might be very long. Therefore, it is desirable to express $\mathcal{S}_{\mathcal{P}}$ using the shortest repetitive pattern that covers the whole sequence. This is accomplished in [BZNS12] using a special data structure called *suffix trees* [Ukk95]. For example, for process $\mathcal{P}_{\texttt{snk}}$ and its sequence of variants $\mathcal{S}_{\texttt{snk}}$, the shortest repetitive pattern is $\{v_1, v_1, v_2\}$.

Table 3.1: *Deriving production/consumption rates sequences from the shortest repetitive pattern*

		Pattern		
		v_1	v_1	v_2
Input/output ports	IP1	1	1	0
	IP2	0	0	1
	IP3	1	1	1

The last step in deriving the equivalent CSDF graph is to derive the production and consumption rates sequences from the shortest repetitive pattern. For each port of a CSDF actor, a consumption/production rates sequence is generated. This is done by building a table in which each row corresponds to an input/output port, and each column corresponds to a process variant in the shortest repetitive pattern. If the input/output port is in the set of ports of the process variant, then its entry in the table is '1'. Otherwise, its entry is '0'. Each row in the resulting table represents a consumption/production rates sequence for the corresponding input/output port.

For example, consider process $\mathcal{P}_{\texttt{snk}}$ in Figure 3.2. The consumption and production rates sequences of the corresponding CSDF actor A_4 in Figure 2.1 on page 29 are generated as shown in Table 3.1. We see from Table 3.1 that the consumption/production rates sequences for the ports are the same as the ones shown in Figure 2.1. □

Chapter 4

Scheduling Framework

Everything should be made as simple as possible, but not simpler

Albert Einstein

RECALL from Section 1.1.3 that several algorithms from hard real-time multiprocessor scheduling theory can perform *fast* admission and scheduling decisions for incoming programs while providing hard real-time guarantees. Moreover, these algorithms provide temporal isolation which is, as mentioned in Chapter 1, the ability to start/stop programs, at run-time, without violating the timing requirements of other already running programs. However, most of these algorithms assume *independent* real-time periodic or sporadic tasks [DB11]. Such a simple task model is not directly applicable to modern embedded streaming programs. Modern streaming programs, as shown in Section 2.3, are modeled as dataflow graphs, which means that the graph actors have *data-dependency constraints* and do not necessarily conform to the real-time periodic or sporadic task models. Therefore, we need to *link* the dataflow MoCs used for performance analysis of the programs and the real-time task models used for timing analysis. Such a link is proposed in this chapter. Given a streaming program modeled as an acyclic CSDF graph, we analytically prove that it is possible to execute the graph actors as a set of real-time periodic tasks. We present an analytical scheduling framework for computing the parameters of the periodic tasks corresponding to the graph actors and the minimum buffer sizes of the communication channels such that a valid schedule is guaranteed to exist. The computed parameters of the periodic tasks are the period, deadline, and start time as defined in Section 2.4. These parameters are used, as shown in Figure 4.1 to derive the architecture and mapping specifications needed later in Chapter 5 to perform system-level synthesis. Furthermore, we present several results related to the quality of periodic scheduling of programs modeled as

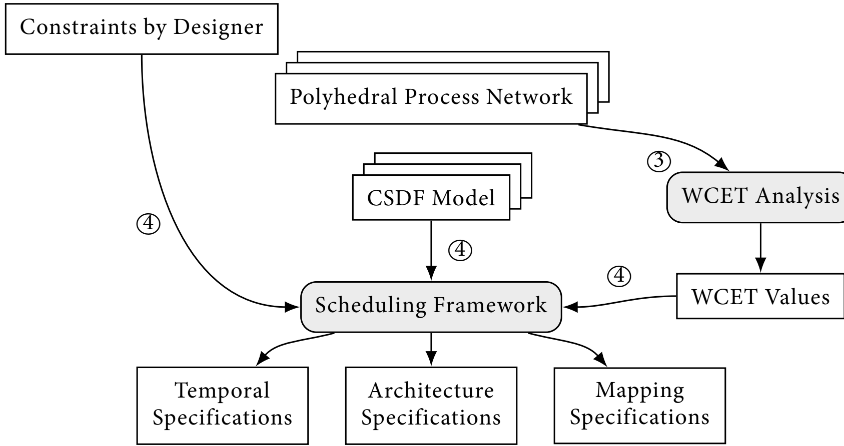


Figure 4.1: *Scheduling framework*

acyclic CSDF graphs. By considering acyclic CSDF graphs, our findings are applicable to most streaming programs since it has been shown recently in [TA10] that around 90% of streaming programs can be modeled as acyclic SDF graphs, which are a subset of CSDF graphs.

4.1 Input Streams

In the remainder of this chapter, a graph G refers to an acyclic consistent CSDF graph. A graph G has a set of periodic input streams connected to its input actors. The set of input streams connected to an actor A_i is disjoint from all other sets of input streams connected to other actors. That is, there are no two actors sharing the same input stream. Let $I_{i,j}$ be the jth input stream received by actor A_i and P_i be the period of A_i. We assume that $I_{i,j}$ satisfies the following: (1) it starts either prior to or together with A_i, (2) it has a constant inter-arrival time equal to P_i, and (3) it has jitter bounds $[J^-_{i,j}, J^+_{i,j}]$. Let t' be the time at which $I_{i,j}$ starts. The interpretation of the jitter bounds is that the kth sample of the stream is expected to arrive in the interval $[t' + kP_i - J^-_{i,j}, t' + kP_i + J^+_{i,j}]$. If a sample arrives in the interval $[t' + kP_i - J^-_{i,j}, t' + kP_i)$, then it is called an *early sample*. On the other hand, if the sample arrives in the interval $(t' + kP_i, t' + kP_i + J^+_{i,j}]$, then it is called a *late sample*.

Suppose that A_i is started at the same time when $I_{i,j}$ starts. It is trivial to show that early samples do not affect the periodicity of the input actor as the samples arrive prior to the actor release time. Late samples, however, pose a problem as they might arrive after an actor is released which will cause the actor to block. To overcome this, it is possible to insert a **de-jitter buffer** before each input actor. The buffer accumulates

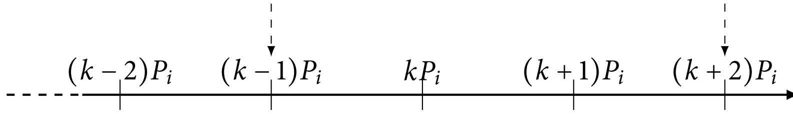


Figure 4.2: *Occurrence of* $t_{MIT}(I_{i,j})$. *Down arrows represent sample arrival.*

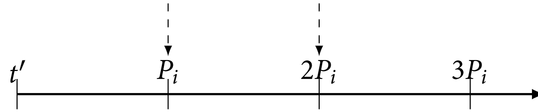


Figure 4.3: *Computing* $t_{buffer}(I_{i,j})$. *Down arrows represent sample arrival.*

a certain amount of incoming samples to the input actor. This accumulation occurs for a certain duration of time, denoted by $t_{\text{buffer}}(I_{i,j})$, before starting the input actor. $t_{\text{buffer}}(I_{i,j})$ has to be computed such that once the input actor is started, it always finds data in the buffer. Suppose that $J^-_{i,j}, J^+_{i,j} \in [0, P_i]$. Then, we can derive the minimum value for $t_{\text{buffer}}(I_{i,j})$ and the minimum de-jitter buffer size. In order to do that, we start with proving the following lemma:

Lemma 4.1.1. *Let $I_{i,j}$ be a jittery input stream with $J^-_{i,j}, J^+_{i,j} \in [0, P_i]$. The maximum inter-arrival time between any two consecutive samples in $I_{i,j}$, denoted by $t_{MIT}(I_{i,j})$, satisfies:*

$$t_{MIT}(I_{i,j}) = 3P_i \tag{4.1}$$

Proof. Based on the jitter model, $t_{\text{MIT}}(I_{i,j})$ occurs when the kth sample is early by the maximum value of jitter (i.e., arrives at time $t = kP_i - P_i$), and the $(k+1)$ sample is late by the maximum value of jitter (i.e., arrives at time $t = (k+1)P_i + P_i$). This is illustrated in Figure 4.2. ∎

Lemma 4.1.2. *An input actor $A_i \in A$ is guaranteed to always find an input sample in each of its input de-jitter buffers if the following holds:*

$$\forall I_{i,j} : t_{buffer}(I_{i,j}) \geq 2P_i \tag{4.2}$$

Proof. During the time interval $(t, t + t_{\text{MIT}}(I_{i,j}))$, A_i can fire at most twice. Therefore, it is necessary to buffer at least two samples in order to guarantee that the input actor A_i can continue firing periodically when the samples are separated by $t_{\text{MIT}}(I_{i,j})$ time-units. Suppose that $I_{i,j}$ is started at time t' and suppose also that the first and second samples are delayed by the maximum jitter value (i.e., P_i) as shown in Figure 4.3. Hence, the duration needed to accumulate two samples before starting A_i (i.e., $t_{\text{buffer}}(I_{i,j})$) has to be equal to or greater than $2P_i$. ∎

Lemma 4.1.3. *Let A_i be an input actor and $I_{i,j}$ be a jittery input stream to A_i. Suppose that $I_{i,j}$ starts at time t' and A_i starts at time $t' + 2P_i$. The de-jitter buffer must be able to hold at least three samples.*

Proof. Suppose that the $(k-1)$ and $(k+1)$ samples arrive late and early, respectively, by the maximum amount of jitter. This means that they arrive at time $t' + kP_i$. Now, suppose that the kth sample arrives with no jitter. This means that at time $t' + kP_i$ there are three samples arriving simultaneously. Hence, the de-jitter buffer must be able to store them. During the interval $[t' + kP_i, t' + (k+1)P_i)$, there are no incoming samples and A_i processes the $(k-1)$ sample. At time $t' + (k+1)P_i$, the $(k+2)$ sample might arrive which means that there are again three samples available to A_i. By the periodicity of A_i and $I_{i,j}$, the previous pattern can repeat. ∎

The main advantage of the de-jitter buffer approach is that the actors are still scheduled as periodic tasks. However, the disadvantages are the extra delay in starting the input actors and the extra memory needed for the buffers. Unless otherwise mentioned, we assume that each input stream is connected to a de-jitter buffer which has a capacity that is at least equal to or greater than the limit given by Lemma 4.1.3.

4.2 Basic Definitions

First, we introduce the following definitions.

Definition 4.2.1 (Execution Time Vector)**.** For a graph $G = (A, E)$, an **execution time vector** $\vec{C}$, where $\vec{C} \in \mathbb{N}^{|A|}$, represents the worst-case execution times, measured in time-units, of the actors in G. The worst-case execution time of an actor $A_j \in A$ is given by

$$C_j = \max_{k=1}^{\mathcal{N}_j} \left(C^R \cdot \sum_{E_l \in \text{inp}(A_j)} y_j^l(k) + C^W \cdot \sum_{E_r \in \text{out}(A_j)} x_j^r(k) + C_j^C(k) \right) \tag{4.3}$$

where $\mathcal{N}_j$ is the length of CSDF firing/production/consumption sequences of actor A_j, C^R is the worst-case time needed to read a single token from an input channel, y_j^l is the consumption sequence of A_j from channel E_l, C^W is the worst-case time needed to write a single token to an output channel, x_j^r is the production sequence of A_j into channel E_r, and $C_j^C(k)$ is the worst-case computation time of A_j in firing k.

The worst-case computation time of an actor A_j is the worst-case time needed to execute the function corresponding to A_j in the PPN. For example, consider the PPN in Figure 3.2 on page 42. The worst-case computation time of process $\mathcal{P}_{\texttt{f2}}$ is

the worst-case time needed to execute function `f2(in1)`. Obtaining the worst-case computation time can be done through two ways [WEE+08]: (1) static analysis of the function source code, or (2) profiling the parallel program on the target platform.

Definition 4.2.2 (Actor Workload)**.** The **workload** of an actor A_i is $W_i = q_i C_i$ and the **maximum actor workload** of the graph is $\hat{W} = \max_{A_i \in A}\{W_i\}$.

Let $\text{lcm}(\vec{q}) = \text{lcm}\{q_1, q_2, \cdots, q_{|A|}\}$. Now, we give the following definition.

Definition 4.2.3 (Matched Input/Output Rates Graph)**.** A graph G is said to be **matched input/output (I/O) rates graph** if and only if

$$\hat{W} \bmod \text{lcm}(\vec{q}) = 0 \tag{4.4}$$

If (4.4) does not hold, then G is said to be *mis-matched I/O rates graph*.

The concept of matched I/O rates applications was first introduced in [TA10] as the applications with *low value of* $\text{lcm}(\vec{q})$. However, the authors in [TA10] did not establish an exact test for determining whether an application has matched I/O rates or not. The test in (4.4) is a novel contribution of this dissertation. If $\hat{W} \bmod \text{lcm}(\vec{q}) = 0$, then there exists at least a single actor in the graph which fully utilizes the processor on which it runs. This, as shown later, allows the graph to achieve optimal throughput. On the other hand, if $\hat{W} \bmod \text{lcm}(\vec{q}) \neq 0$, then there exist idle durations in the period of each actor which results in sub-optimal throughput.

Definition 4.2.4 (Balanced Graph)**.** A graph G is called **balanced** if and only if

$$q_1 C_1 = q_2 C_2 = \cdots = q_{|A|} C_{|A|} \tag{4.5}$$

If (4.5) does not hold, then the graph is called *unbalanced*.

Definition 4.2.5 (Output Path Latency)**.** Let $W_k = \{(A_a, A_b), \cdots, (A_y, A_z)\}$ be an output path in graph G. The **latency** of W_k under periodic input streams, denoted by $\mathcal{L}(W_k)$, is the elapsed time between the start of the first firing of A_a that produces data to channel (A_a, A_b) and the finish of the first firing of A_z that consumes data from channel (A_y, A_z).

Consequently, we define the maximum latency of G as follows:

Definition 4.2.6 (Graph Maximum Latency)**.** For a graph G, the **maximum latency** of G under periodic input streams, denoted by $\mathcal{L}(G)$, is given by:

$$\mathcal{L}(G) = \max_{W_k \in W}\{\mathcal{L}(W_k)\} \tag{4.6}$$

The path with the largest output latency is called the **critical path** of the graph.

Definition 4.2.7 (Self-Timed Schedule)**.** A *Self-Timed Schedule* (STS) is one where all the actors are fired as soon as their input data are available.

A self-timed schedule does not impose any extra latency (e.g., by the scheduler) on the actors. This leads us to the following result proven in [SB09] for HSDF graphs and extended in [Gha08] to SDF graphs.

Theorem 4.2.1 (From [SB09, Gha08])**.** *For a graph G, the minimum achievable latency and the maximum achievable throughput are obtained when the actors of G are scheduled using self-timed scheduling policy.*

Theorem 4.2.1 applies to CSDF graphs since any CSDF graph can be converted to an equivalent (H)SDF graph [BELP96].

Definition 4.2.8 (Worst-Case Self-Timed Schedule)**.** A self-timed schedule in which every actor firing has a duration equal to the value given by (4.3) is called a *worst-case self-timed schedule.*

For acyclic graphs, the **throughput** of an actor A_i, denoted by $\mathcal{R}_{\text{WSTS}}(A_i)$, under a worst-case self-timed schedule is given according to [Gha08] by:

$$\mathcal{R}_{\text{WSTS}}(A_i) = q_i / \hat{W} \tag{4.7}$$

4.3 Deriving Periods

The first step of the proposed scheduling framework is to derive a valid period for each actor in the graph. To do that, we introduce first some definitions.

Definition 4.3.1 (Periodic Actor)**.** Let A_i be a CSDF actor with start time S_i and period P_i. A_i is **periodic** if and only if it can be fired at time instants $S_i + kP_i$ for all $k \in \mathbb{N}_0$.

Definition 4.3.2 (Periodic Schedule)**.** Let G be a CSDF graph. A valid static schedule (see Definition 2.3.1 on page 27) in which all the actors are periodic actors is called a **periodic schedule**.

Definition 4.3.3 (Period Vector)**.** For a CSDF graph G, a **period vector** $\vec{P}$, where $\vec{P} \in \mathbb{N}^{|A|}$, represents the periods, measured in time-units, of the actors in G. $P_j \in \vec{P}$ is the period of actor $A_j \in A$. $\vec{P}$ is given by the solution to both

$$q_1 P_1 = q_2 P_2 = \cdots = q_{n-1} P_{n-1} = q_n P_n \tag{4.8}$$

and

$$\vec{P} - \vec{C} \geq \vec{0}, \tag{4.9}$$

where $q_j \in \vec{q}$ and $n = |A|$.

Definition 4.3.3 implies that all the actors take the same time duration to complete one actor iteration (see Definition 2.3.2 on page 28). This common time duration of the actor iteration is called the **iteration period** and it is denoted by $\alpha = q_i P_i$ for any A_i.

Now, we prove the existence of a periodic schedule when the input streams are strictly periodic. Recall from Section 4.1 that we use de-jitter buffers to hide the effect of jitter and make the input samples delivered to the input actors arrive in a strictly periodic fashion.

Lemma 4.3.1. *For a graph G, the minimum period vector of G, denoted by $\vec{\check{P}}$, is given by*

$$\check{P}_i = \frac{\mathrm{lcm}(\vec{q})}{q_i} \left\lceil \frac{\hat{W}}{\mathrm{lcm}(\vec{q})} \right\rceil \text{for } A_i \in A. \tag{4.10}$$

Proof. (4.8) can be re-written as:

$$\mathbf{Q} \cdot \vec{P} = \vec{0}, \tag{4.11}$$

where $\mathbf{Q} \in \mathbb{Z}^{(|A|-1)\times|A|}$ is given by

$$\mathbf{Q}_{ij} = \begin{cases} q_1 & \text{if } j = 1 \\ -q_j & \text{if } j = i+1 \\ 0 & \text{otherwise} \end{cases} \tag{4.12}$$

Observe that nullity$(\mathbf{Q}) = 1$. Thus, there exists a single vector which forms the basis of the null-space of $\mathbf{Q}$. This vector can be represented by taking any unknown P_k as the free-unknown and expressing the other unknowns in terms of it which results in:

$$\vec{P} = P_k[q_k/q_1, q_k/q_2, \cdots, q_k/q_n]^T$$

The minimum $P_k \in \mathbb{N}$ is

$$P_k = \mathrm{lcm}\{q_1, q_2, \cdots, q_n\}/q_k$$

Thus, the minimum $\vec{P} \in \mathbb{N}$ that solves (4.8) is given by

$$P_i = \mathrm{lcm}(\vec{q})/q_i \text{ for } A_i \in A \tag{4.13}$$

Let $\vec{\tilde{P}}$ be the solution given by (4.13). (4.8) and (4.9) can be re-written as:

$$\mathbf{Q}(c\vec{\tilde{P}}) = \vec{0} \tag{4.14}$$

$$c\tilde{P}_1 \geq C_1, c\tilde{P}_2 \geq C_2, \cdots, c\tilde{P}_n \geq C_n \tag{4.15}$$

where $c \in \mathbb{N}$. (4.15) can be re-written as:

$$c \geq q_1 C_1/\mathrm{lcm}(\vec{q}), c \geq q_2 C_2/\mathrm{lcm}(\vec{q}), \cdots, c \geq q_n C_n/\mathrm{lcm}(\vec{q}) \tag{4.16}$$

It follows that c is greater than or equal to $\max_{A_i \in A}\{C_i q_i\}/\operatorname{lcm}(\vec{q}) = \hat{W}/\operatorname{lcm}(\vec{q})$. However, $\hat{W}/\operatorname{lcm}(\vec{q})$ is not always guaranteed to be an integer. As a result, the value is rounded up by taking its ceiling. It follows that the minimum $\vec{\check{P}}$ which satisfies both of (4.8) and (4.9) is given by

$$\check{P}_i = \operatorname{lcm}(\vec{q})/q_i \lceil \hat{W}/\operatorname{lcm}(\vec{q}) \rceil \text{ for } A_i \in A$$

■

$\check{P}_i$ is the minimum value of the period of an actor A_i according to Lemma 4.3.1. However, in many cases, the designer might not need the minimum period. Therefore, we define **period scaling factor** of a graph G, denoted by $\mu_G \in \mathbb{N}$, which is used to scale all the minimum periods of the actors in a graph G. Thus, the **actual period** of an actor A_i is given by:

$$P_i = \mu_G \check{P}_i \tag{4.17}$$

Observe that, for a graph G, scaling the minimum periods of the actors by μ_G results in periods that satisfy (4.8) and (4.9).

Theorem 4.3.1. *For any graph G, a periodic schedule exists such that every actor $A_i \in A$ is periodic with a constant period P_i given by (4.17) and every communication channel $E_u \in E$ has a bounded buffer capacity.*

Proof. Recall from Section 2.3 that for any input actor A_i, $\operatorname{prec}(A_i) = \emptyset$. By Algorithm 1 on page 27, we obtain that all input actors belong to $^{\mathbb{L}}A_1$ (i.e., level-1 actors). Hence, level-1 actors consist of either input actors or actors that generate data by themselves. Now, recall from Section 4.1 that the input streams to the input actors are periodic with periods equal to the input actors periods. Therefore, it follows that the input actors in level-1 can execute periodically since their input streams are always available when they fire. By Definition 2.3.2 on page 28, level-1 actors will complete one iteration when they fire q_i times, where q_i is the repetition of $A_i \in {}^{\mathbb{L}}A_1$. Assume that level-1 actors start executing at time $t = 0$. Then, by time $t = \alpha$, level-1 actors are guaranteed to finish one iteration. According to Theorem 2.3.1 on page 28, level-1 actors will also generate enough data such that every actor $A_k \in {}^{\mathbb{L}}A_2$ can execute q_k times (i.e., one iteration) with a period P_k. According to (4.8) on page 52, firing A_k for q_k times with a period P_k takes α time-units. Thus, starting level-2 actors at time $t = \alpha$ guarantees that they can execute periodically with their periods given by Definition 4.3.3 for α time-units. Similarly, by time $t = 2\alpha$, level-3 actors will have enough data to execute for one iteration. Thus, starting level-3 actors at time $t = 2\alpha$ guarantees that they can execute periodically for α time-units. By repeating this over all the $\mathbb{L}$ levels, a schedule $\mathbb{S}_1$ (shown in Figure 4.4) is constructed in which all the actors that belong to $^{\mathbb{L}}A_i$ are started at *start time* S_i given by

$$S_i = (i-1)\alpha \tag{4.18}$$

time	$[0,\alpha)$	$[\alpha,2\alpha)$	$[2\alpha,3\alpha)$	$\cdots$	$[(\mathbb{L}-1)\alpha,\mathbb{L}\alpha)$
level	${}^{\mathbb{L}}A_1(1)$	${}^{\mathbb{L}}A_2(1)$	${}^{\mathbb{L}}A_3(1)$	$\cdots$	${}^{\mathbb{L}}A_{\mathbb{L}}(1)$

Figure 4.4: *Schedule* $\mathbb{S}_1$

time	$[0,\alpha)$	$[\alpha,2\alpha)$	$[2\alpha,3\alpha)$	$\cdots$	$[(\mathbb{L}-1)\alpha,\mathbb{L}\alpha)$
level	${}^{\mathbb{L}}A_1(1)$	${}^{\mathbb{L}}A_2(1)$	${}^{\mathbb{L}}A_3(1)$	$\cdots$	${}^{\mathbb{L}}A_{\mathbb{L}}(1)$
		${}^{\mathbb{L}}A_1(2)$	${}^{\mathbb{L}}A_2(2)$	$\cdots$	${}^{\mathbb{L}}A_{\mathbb{L}-1}(2)$

Figure 4.5: *Schedule* $\mathbb{S}_2$

time	$[0,\alpha)$	$[\alpha,2\alpha)$	$[2\alpha,3\alpha)$	$\cdots$	$[(\mathbb{L}-1)\alpha,\mathbb{L}\alpha)$
level	${}^{\mathbb{L}}A_1(1)$	${}^{\mathbb{L}}A_2(1)$	${}^{\mathbb{L}}A_3(1)$	$\cdots$	${}^{\mathbb{L}}A_{\mathbb{L}}(1)$
		${}^{\mathbb{L}}A_1(2)$	${}^{\mathbb{L}}A_2(2)$	$\cdots$	${}^{\mathbb{L}}A_{\mathbb{L}-1}(2)$
			${}^{\mathbb{L}}A_1(3)$	$\cdots$	${}^{\mathbb{L}}A_{\mathbb{L}-2}(3)$
				$\cdots$	${}^{\mathbb{L}}A_{\mathbb{L}-3}(4)$
				$\cdots$	$\cdots$
					${}^{\mathbb{L}}A_1(\mathbb{L})$

Figure 4.6: *Schedule* $\mathbb{S}_{\mathbb{L}}$

${}^{\mathbb{L}}A_j(k)$ denotes level-j actors executing their kth iteration. For example, ${}^{\mathbb{L}}A_2(1)$ denotes level-2 actors executing their first iteration. At time $t = \mathbb{L}\alpha$, G completes one graph iteration. It is trivial to observe from $\mathbb{S}_1$ that as soon as level-1 actors finish one iteration, they can immediately start executing the next iteration since their input streams arrive periodically. If level-1 actors start their second iteration at time $t = \alpha$, their execution will overlap with the execution of the level-2 actors. By doing so, level-2 actors can start immediately their second iteration after finishing their first iteration since they will have all the needed data to execute one iteration periodically at time $t = 2\alpha$. This overlapping can be applied to all the levels to yield the schedule $\mathbb{S}_2$ shown in Figure 4.5. Now, the overlapping can be applied $\mathbb{L}$ times on schedule $\mathbb{S}_1$ to yield a schedule $\mathbb{S}_{\mathbb{L}}$ as shown in Figure 4.6.

Starting from time $t = \mathbb{L}\alpha$, a schedule $\mathbb{S}_\infty$ can be constructed as shown in Figure 4.7. In schedule $\mathbb{S}_\infty$, every actor A_i is fired every P_i time-unit once it starts. The start time defined in (4.18) guarantees that actors in a given level will start only when they have enough data to execute one iteration in a periodic way. The overlapping guarantees that once the actors have started, they will always find enough data for executing the next iteration since their predecessors have already executed one additional iteration. Thus, schedule $\mathbb{S}_\infty$ shows the existence of a periodic schedule of G where every actor $A_j \in A$ is periodic with a period equal to P_j.

time	$[0,\alpha)$	$[\alpha,2\alpha)$	$[2\alpha,3\alpha)$	$\cdots$	$[(\mathbb{L}-1)\alpha,\mathbb{L}\alpha)$	$[\mathbb{L}\alpha,(\mathbb{L}+1)\alpha)$	$\cdots$
level	${}^{\mathbb{L}}A_1(1)$	${}^{\mathbb{L}}A_2(1)$	${}^{\mathbb{L}}A_3(1)$	$\cdots$	${}^{\mathbb{L}}A_{\mathbb{L}}(1)$	${}^{\mathbb{L}}A_{\mathbb{L}}(2)$	$\cdots$
		${}^{\mathbb{L}}A_1(2)$	${}^{\mathbb{L}}A_2(2)$	$\cdots$	${}^{\mathbb{L}}A_{\mathbb{L}-1}(2)$	${}^{\mathbb{L}}A_{\mathbb{L}-1}(3)$	$\cdots$
			${}^{\mathbb{L}}A_1(3)$	$\cdots$	${}^{\mathbb{L}}A_{\mathbb{L}-2}(3)$	${}^{\mathbb{L}}A_{\mathbb{L}-2}(4)$	$\cdots$
				$\cdots$	${}^{\mathbb{L}}A_{\mathbb{L}-3}(4)$	${}^{\mathbb{L}}A_{\mathbb{L}-3}(5)$	$\cdots$
				$\cdots$	$\cdots$	$\cdots$	$\cdots$
					${}^{\mathbb{L}}A_1(\mathbb{L})$	${}^{\mathbb{L}}A_1(\mathbb{L}+1)$	$\cdots$

Figure 4.7: *Schedule* $\mathbb{S}_\infty$

The next step is to prove that $\mathbb{S}_\infty$ executes with bounded memory buffers. In $\mathbb{S}_\infty$, the largest delay in consuming the tokens occurs for a channel $E_u \in E$ connecting a level-1 actor $A_m \in {}^{\mathbb{L}}A_1$ and a level-$\mathbb{L}$ actor. This is illustrated in Figure 4.7 by observing that the data produced by iteration-1 of a level-1 source actor will be consumed by iteration-1 of a level-$\mathbb{L}$ destination actor after $(\mathbb{L}-1)\alpha$ time-units. In this case, E_u must be able to store at least $(\mathbb{L}-1)X_m^u(q_m)$ tokens. However, starting from time $t = \mathbb{L}\alpha$, both of the level-1 and level-$\mathbb{L}$ actors execute in parallel. Thus, we increase the buffer size by $X_m^u(q_m)$ tokens to account for the overlapped execution. Hence, the total buffer size of E_u is $\mathbb{L}X_m^u(q_m)$ tokens. Similarly, if a level-2 actor $A_l \in {}^{\mathbb{L}}A_2$ is connected directly to a level-$\mathbb{L}$ actor via channel E_v, then E_v must be able to store at least $(\mathbb{L}-1)X_l^v(q_l)$ tokens. By repeating this argument over all the different pairs of levels, it follows that each channel $E_u \in E$, connecting a level-i source actor A_k and a level-j destination actor, where $j \geq i$, will store according to schedule $\mathbb{S}_\infty$ at most:

$$b_u = (j - i + 1)X_k^u(q_k) \tag{4.19}$$

tokens, where $q_k \in \vec{q}$. Thus, an upper bound on the FIFO sizes exists. ■

Example 4.3.1. We give now an example on how to compute the periods of the actors for a given CSDF graph. Consider the CSDF graph shown in Figure 2.1 on page 29. Recall from Example 2.3.1 on page 28 that the basic repetition vector of the graph is $\vec{q} = [3,2,1,3]^T$. Suppose that the execution time vector $\vec{C} = [5,8,24,4]^T$. It follows that $\mathrm{lcm}(\vec{q}) = \mathrm{lcm}\{3,2,1\} = 6$ and the maximum actor workload is $\hat{W} = \max\{3\times 5, 2\times 8, 1\times 24, 3\times 4\} = 24$. Based on Lemma 4.3.1, the minimum period vector of the graph is $\vec{\check{P}} = [8,12,24,8]^T$. The periodic schedule resulting from Theorem 4.3.1, assuming $\mu_G = 1$, is depicted in Figure 4.8. The actors in the schedule shown in Figure 4.8 have start times given by (4.18).

□

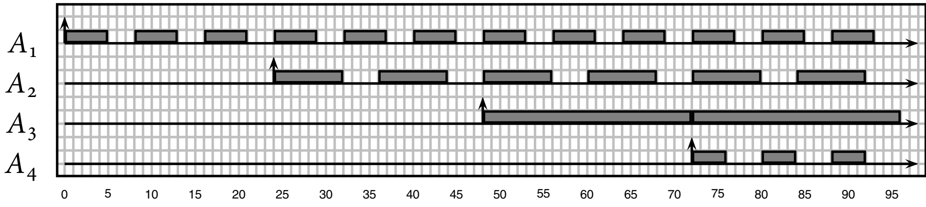


Figure 4.8: *The periodic schedule for the CSDF graph shown in Figure 2.1 constructed using Theorem 4.3.1. The x-axis represents the time and the up-arrows represent the start times of the actors.*

4.4 Deriving Deadlines and Start Times

The second step in the scheduling framework is to derive the deadlines and start times of the actors. In the proof of Theorem 4.3.1, we use the notion of start time S_i to refer to the time at which an actor A_i can start executing. The start time used in Theorem 4.3.1 is *sufficient* but not minimum. Therefore, we are interested in finding the earliest start times of the actors that guarantee the existence of a periodic schedule. Minimizing the start times is crucial since it has a direct impact on the latency of the graph and the buffer sizes of the communication channels.

First, we introduce the **cumulative production** and **cumulative consumption** functions.

Definition 4.4.1 (Cumulative Production Function)**.** The cumulative production function of actor A_i producing into channel E_u during a time interval $[t_s, t_e)$, denoted by $\text{prd}_{[t_s,t_e)}(A_i, E_u)$, is the sum of the number of tokens produced by A_i into E_u during the interval $[t_s, t_e)$.

Definition 4.4.2 (Cumulative Consumption Function)**.** The cumulative consumption function of actor A_i consuming from channel E_u over a time interval $[t_s, t_e)$, denoted by $\text{cns}_{[t_s,t_e)}(A_i, E_u)$, is the sum of the number of tokens consumed by A_i from E_u during the interval $[t_s, t_e)$.

Note that the time interval in Definitions 4.4.1 and 4.4.2 can be either open or closed from the right. Next, we give the following definition:

Definition 4.4.3 (Valid Start Time)**.** Let $E_u = (A_i, A_j)$ be a communication channel in a graph G. Under a periodic schedule, a **valid start time** of A_j, denoted by S_j, guarantees that A_j finds enough data in E_u to fire at time instants $S_j + kP_j$ for all $k \in \mathbb{N}_0$.

Definition 4.4.3 implies that A_j never blocks on reading from E_u when it has a valid start time (i.e., no buffer underflow). We utilize this to determine conditions under which one can compute the earliest start time such that it is always valid regardless of when the actor is actually scheduled to write/read tokens under a periodic schedule.

Lemma 4.4.1. *Let $E_u = (A_i, A_j)$ be a communication channel in a graph G. Suppose that S_i and S_j are valid start times of A_i and A_j, respectively, under a periodic schedule. Suppose also that S_i and S_j are derived when A_i is always scheduled to write data as late as possible and A_j is always scheduled to read data as early as possible. S_i and S_j remain valid when A_i is scheduled to write earlier and/or A_j is scheduled to read later.*

Proof. If A_i is scheduled to write earlier, then this leads to the tokens being available earlier on the channel. Thus, A_j can never have a blocking read regardless of when it is scheduled to read because it has a valid start time derived assuming that it is scheduled to read as early as possible. If A_j is scheduled to read later, then this leads to longer stay of the tokens in the channel. Thus, A_j can never block on reading since the channel has enough tokens because S_j is a valid start time for A_j. ∎

Lemma 4.4.1 states that if we derive the earliest start time assuming that the tokens production happens as late as possible and the tokens consumption happens as early as possible, then the derived earliest start time is valid regardless of when the actor is actually scheduled to execute during its period. Thus, for the purpose of computing the start time, the cumulative production function assumes that the tokens production happens as late as possible. In this case, the cumulative production function is denoted by $\text{prd}^{\text{S}}{}_{[t_s,t_e)}(A_i, E_u)$ and is given by:

$$\underset{[t_s,t_e)}{\text{prd}^{\text{S}}}(A_i, E_u) = \begin{cases} X_i^u(\lfloor \frac{t_e - t_s}{P_i} \rfloor + 1) & \text{if } (t_e - t_s) \bmod P_i \geq D_i \\ X_i^u(\lfloor \frac{t_e - t_s}{P_i} \rfloor) & \text{if } (t_e - t_s) \bmod P_i < D_i \end{cases} \tag{4.20}$$

Similarly, for the purpose of computing the start time, the cumulative consumption function assumes that the data consumption happens as early as possible. The function is denoted by $\text{cns}^{\text{S}}{}_{[t_s,t_e]}(A_i, E_u)$ and is given by

$$\underset{[t_s,t_e]}{\text{cns}^{\text{S}}}(A_i, E_u) = \begin{cases} Y_i^u(\lceil \frac{t_e - t_s}{P_i} \rceil + 1) & \text{if } (t_e - t_s) \bmod P_i = 0 \\ Y_i^u(\lceil \frac{t_e - t_s}{P_i} \rceil) & \text{if } (t_e - t_s) \bmod P_i \neq 0 \end{cases} \tag{4.21}$$

It can be seen from (4.20) that the deadline has an impact on when the tokens are produced. Therefore, we introduce the **deadline factor** of an actor A_i, denoted by η_i, where $\eta_i \in [0, 1]$. Similar to the period scaling factor μ_G, the deadline factor is a parameter controlled by the designer. Given the WCET, period, and deadline factor of an actor A_i, its deadline D_i is given by

$$D_i = \lfloor C_i + \eta_i(P_i - C_i) \rfloor \tag{4.22}$$

Now, we give the following lemma regarding the earliest start time.

Lemma 4.4.2. *For a graph G, the earliest start time of an actor $A_j \in A$, denoted by S_j, under a periodic schedule is given by*

$$S_j = \begin{cases} 0 & \textit{if } prec(A_j) = \varnothing \\ \max_{A_i \in prec(A_j)} \{S_{i \to j}\} & \textit{if } prec(A_j) \neq \varnothing \end{cases} \tag{4.23}$$

where

$$S_{i \to j} = \min_{t \in [0, S_i + \alpha]} \{t \mid \forall k = 0, 1, \cdots, \alpha : \underset{[S_i, \max\{S_i, t\}+k)}{\mathrm{prd}^{\mathrm{S}}}(A_i, E_u) \geq \underset{[t, \max\{S_i, t\}+k]}{\mathrm{cns}^{\mathrm{S}}}(A_j, E_u)\} \tag{4.24}$$

Proof. Theorem 4.3.1 proved that starting a level-k actor A_j at a start time

$$S_j = (k-1)\alpha \tag{4.25}$$

guarantees periodic execution of the actor A_j. Any start time later than that guarantees also periodic execution since A_j will always find enough data to execute in a periodic way.

(4.25) can be re-written as:

$$S_j = \begin{cases} 0 & \text{if prec}(A_j) = \varnothing \\ \max_{A_i \in \text{prec}(A_j)} \{S_i\} + \alpha & \text{if prec}(A_j) \neq \varnothing \end{cases} \tag{4.26}$$

The equivalence follows from observing that a level-k actor, where $k > 1$, has a level-$(k-1)$ predecessor. Hence, applying (4.26) to a level-k actor, where $k > 1$, yields:

$$S_j = \max\{(k-2)\alpha, (k-3)\alpha, \cdots, 0\} + \alpha = (k-1)\alpha$$

Now, we are interested in starting A_j in level-k, where $k > 1$, earlier. That is:

$$S_j \leq \max_{A_i \in \text{prec}(A_j)} \{S_i\} + \alpha \tag{4.27}$$

S_j has also a lower-bound by observing that an actor A_j can not start before the application is started. That is:

$$0 \leq S_j \leq \max_{A_i \in \text{prec}(A_j)} \{S_i\} + \alpha \quad \Rightarrow \quad 0 \leq S_j \leq \max_{A_i \in \text{prec}(A_j)} \{S_i + \alpha\} \tag{4.28}$$

If we select S_j such that

$$S_j = \max_{A_i \in \text{prec}(A_j)} \{S_{i \to j}\}, \text{ where } S_{i \to j} = t', \text{ and } t' \in [0, S_i + \alpha] \tag{4.29}$$

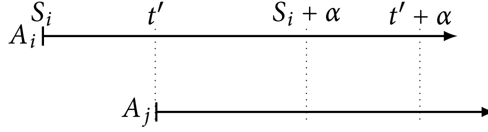


Figure 4.9: *Timeline of A_i and A_j when $t' \geq S_i$*

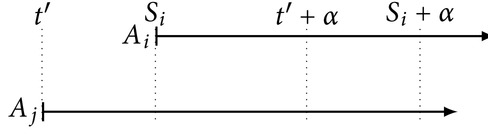


Figure 4.10: *Timeline of A_i and A_j when $t' < S_i$*

then this guarantees that S_j also satisfies (4.28).

In (4.29), a valid start time candidate $S_{i\to j}$ must guarantee that the number of tokens available on channel $E_u = (A_i, A_j)$ at any time instant $t \geq t'$ is greater than or equal to the number of consumed tokens at the same instant. To guarantee that, we consider the following two possible cases:

Case I: $t' \geq S_i$ This case is illustrated in Figure 4.9. In this case, a valid start time candidate t' must satisfy:

$$\forall k = 0, 1, \cdots, \alpha : \underset{[S_i, t'+k)}{\text{prd}^{\text{S}}}(A_i, E_u) \geq \underset{[t', t'+k]}{\text{cns}^{\text{S}}}(A_j, E_u) \tag{4.30}$$

Satisfying (4.30) guarantees that A_j can fire at times $t = t', t' + P_j, \cdots, t' + \alpha$. Thus, a valid value of t' guarantees that once A_j is started, it always finds enough data to fire for one iteration. As a result, A_j executes in a periodic way.

Case II: $t' < S_i$ This case is illustrated in Figure 4.10. A valid start time candidate t' must satisfy:

$$\forall k = 0, 1, \cdots, \alpha : \underset{[S_i, S_i+k)}{\text{prd}^{\text{S}}}(A_i, E_u) \geq \underset{[t', S_i+k]}{\text{cns}^{\text{S}}}(A_j, E_u) \tag{4.31}$$

This case occurs when A_j consumes zeros tokens during the interval $[t', S_i]$. This is a valid behavior since the consumption rates sequence of a CSDF actor can contain zero elements (see for example the CSDF graph on page 29). Since $t' < S_i$, it is sufficient to check the cumulative production and consumption over the interval $[S_i, S_i + \alpha]$ since by time $t = S_i + \alpha$ both A_i and A_j are guaranteed to have finished one iteration. Thus, t' also guarantees that once A_j is started, it always finds enough data to fire. Hence, A_j executes in a periodic way.

Table 4.1: *Computing $\vec{D}$ and $\vec{S}$ for the CSDF graph shown in Figure 2.1 on page 29 under different values of $\vec{\eta}$.*

$\vec{\eta}$	$\vec{D}$	$\vec{S}$
$\vec{1}$	$[8,12,24,8]^T$	$[0,8,24,32]^T$
$\vec{0.5}$	$[6,10,24,6]^T$	$[0,6,22,30]^T$
$\vec{0}$	$[5,8,24,4]^T$	$[0,5,21,29]^T$

Now, we can merge (4.30) and (4.31) which results in:

$$\forall k = 0,1,\cdots,\alpha : \underset{[S_i,\max\{S_i,t'\}+k)}{\text{prd}^{\text{S}}}(A_i, E_u) \geq \underset{[t',\max\{S_i,t'\}+k]}{\text{cns}^{\text{S}}}(A_j, E_u) \tag{4.32}$$

Any value of t' which satisfies (4.32) is a valid start time value that guarantees periodic execution of A_j. Since there might be multiple values of t' that satisfy (4.32), we take the minimum one because it is the earliest start time that guarantees periodic execution of A_j. ■

Example 4.4.1. Now, we give an example on how to compute the earliest start times and deadlines for a given CSDF graph. Consider the CSDF graph shown in Figure 2.1 on page 29. Recall from Example 4.3.1 on page 56 that $\vec{C} = [5,8,24,4]^T$ and $\vec{\check{P}} = [8,12,24,8]^T$. In Table 4.1, we show the values of start times and deadlines under different values of the deadline factors $\vec{\eta}$. These values are computed by applying Lemma 4.4.2 and (4.22) to the graph. □

4.5 Deriving Buffer Sizes

The third step in the scheduling framework is to derive a bounded buffer size of each communication channel in the graph. Similar to the start time, we prove in Theorem 4.3.1 the existence of a periodic schedule when each channel has a bounded buffer size. However, the buffer size used in Theorem 4.3.1 is *sufficient* but not *minimum*. Therefore, we would like to derive the minimum buffer sizes that guarantee periodic execution of all the actors.

Definition 4.5.1 (Valid Buffer Size)**.** Let $E_u = (A_i, A_j)$ be a communication channel in a graph G. Under a periodic schedule, a **valid buffer size** of E_u, denoted by b_u, guarantees that A_i can store tokens to E_u at time instants $S_i + kP_i$ for all $k \in \mathbb{N}_0$.

Definition 4.5.1 implies that a producer never blocks when writing to a communication channel which has a valid buffer size (i.e., no buffer overflow). Similar to

Lemma 4.4.1, we want to derive the conditions under which one can compute the minimum buffer size such that it is always valid regardless of when the actors are actually scheduled to write/read during their periods.

Lemma 4.5.1. *Let $E_u = (A_i, A_j)$ be a communication channel in a graph G. Suppose that E_u has a valid buffer size that is derived when A_i is always scheduled to write as early as possible and A_j is always scheduled to read as late as possible. b_u remains valid when A_i is scheduled to write later and/or A_j is scheduled to read earlier.*

Proof. If A_i is scheduled to write later, then this leads to the tokens being written later on the channel. Thus, A_i can never have a blocking write regardless of when A_j is scheduled to read because b_u is valid and is derived assuming that A_j is always scheduled to read as late as possible. If A_j is scheduled to read earlier, then this leads to shorter stay of the tokens in the channel. Thus, A_i can never block on writing since b_u is valid and the channel has enough free space. ■

Lemma 4.5.1 states that the buffer size derived when the tokens production happens as early as possible and the tokens consumption happens as late as possible is valid regardless of when the actors are actually scheduled to execute during their periods. Thus, for the purpose of computing the buffer size, the cumulative production function assumes that the tokens production happens as early as possible. In this case, the cumulative production function is denoted by $\text{prd}^{\text{B}}{}_{[t_s,t_e]}(A_i, E_u)$ and is given by:

$$\underset{[t_s,t_e]}{\text{prd}^{\text{B}}}(A_i, E_u) = \begin{cases} X_i^u(\lceil \frac{t_e - t_s}{P_i} \rceil + 1) & \text{if } (t_e - t_s) \bmod P_i = 0 \\ X_i^u(\lceil \frac{t_e - t_s}{P_i} \rceil) & \text{if } (t_e - t_s) \bmod P_i \neq 0 \end{cases} \tag{4.33}$$

Similarly, for the purpose of computing the buffer size, the cumulative consumption function assumes that the data consumption happens as late as possible. In this case, the function is denoted by $\text{cns}^{\text{B}}{}_{[t_s,t_e)}(A_i, E_u)$ and is given by

$$\underset{[t_s,t_e)}{\text{cns}^{\text{B}}}(A_i, E_u) = \begin{cases} Y_i^u(\lfloor \frac{t_e - t_s}{P_i} \rfloor + 1) & \text{if } (t_e - t_s) \bmod P_i \geq D_i \\ Y_i^u(\lfloor \frac{t_e - t_s}{P_i} \rfloor) & \text{if } (t_e - t_s) \bmod P_i < D_i \end{cases} \tag{4.34}$$

Lemma 4.5.2. *For a graph G, the minimum bounded buffer size b_u of a communication channel $E_u = (A_i, A_j)$ under a periodic schedule is given by*

$$b_u = \max_{k \in [0,1,\cdots,\alpha]} \left\{ \underset{[S_i, \max\{S_i,S_j\}+k]}{\text{prd}^{\text{B}}}(A_i, E_u) - \underset{[S_j, \max\{S_i,S_j\}+k)}{\text{cns}^{\text{B}}}(A_j, E_u) \right\} \tag{4.35}$$

Proof. (4.35) tracks the maximum cumulative number of unconsumed tokens in E_u during one iteration for A_i and A_j. There are two cases:

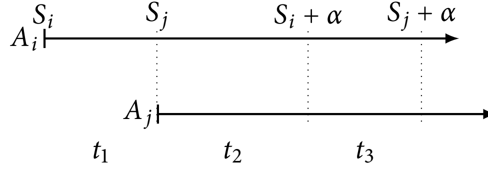


Figure 4.11: *Execution time-lines of A_i and A_j when $S_i \leq S_j$*

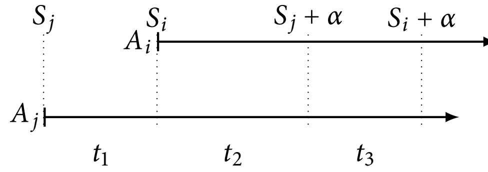


Figure 4.12: *Execution time-lines of A_i and A_j when $S_i > S_j$*

Case I: $S_i \leq S_j$ In this case, (4.35) tracks the maximum cumulative number of unconsumed tokens in E_u during the time interval $[S_i, S_j + \alpha)$. Figure 4.11 illustrates the execution time-lines of A_i and A_j when $S_i \leq S_j$. In interval t_1, A_i is actively producing tokens while A_j has not yet started executing. As a result, it is necessary to buffer all the tokens produced in this interval in order to prevent A_i from blocking on writing. Thus, b_u must be greater than or equal to $\text{prd}^{\text{B}}{}_{[S_i,S_j]}(A_i, E_u)$. Starting from time $t = S_j$, both of A_i and A_j are executing in parallel (i.e., overlapped execution). In the proof of Theorem 4.3.1, an additional $X_i^u(q_i)$ tokens were added to the buffer size of E_u to account for the overlapped execution. However, this value is a "worst-case" value. The minimum number of tokens that needs to be buffered is given by the maximum number of unconsumed tokens in E_u at any time over the time interval $[S_j, S_j + \alpha)$ (i.e., intervals t_2 and t_3 in Figure 4.11). Taking the maximum number of unconsumed tokens guarantees that A_i will always have enough space to write to E_u. Thus, b_u is sufficient and minimum for guaranteeing strictly periodic execution of A_i and A_j in the time interval $[S_i, S_j + \alpha)$. At time $t = S_j + \alpha$, both of A_i and A_j have completed one iteration and the number of tokens in E_u is the same as at time $t = S_j$ (follows from Corollary 2.3.1 on page 28). Due to the periodicity of A_i and A_j, the pattern shown in Figure 4.11 repeats. Thus, b_u is also sufficient and minimum for any $t \geq S_j + \alpha$.

Case II: $S_i > S_j$ Figure 4.12 illustrates this case. According to Lemma 4.4.2, S_j can be smaller than S_i if and only if A_i consumes zero tokens in interval t_1. Therefore, the intervals in which there is actually production/consumption of tokens are t_2 and t_3. During interval t_2, there is overlapped execution and b_u gives the maximum number of unconsumed tokens in E_u during $[S_i, S_j + \alpha)$ which guarantees that A_i always have enough space to write to E_u and A_j has enough data to consume from E_u. At

Table 4.2: *Computing the buffer sizes for the CSDF graph shown in Figure 2.1 on page 29 under different values of $\vec{\eta}$.*

$\vec{\eta}$	$\vec{b}$
$\vec{1}$	$[2,2,5,3,2]^T$
$\vec{0.5}$	$[2,2,5,3,2]^T$
$\vec{0}$	$[2,2,5,3,2]^T$

time $t = S_j + \alpha$, A_j finishes one iteration and interval t_3 starts. During interval t_3, A_i is producing data to E_u while A_j is consuming zero tokens. Therefore, E_u has to accommodate all the tokens produced during interval t_3 and b_u must be greater than or equal to $\text{prd}^{\text{B}}{}_{[S_j+\alpha, S_i+\alpha]}(A_i, E_u)$. As in Case I, b_u is sufficient and minimum for guaranteeing periodic execution of A_i and A_j in the interval $[S_j, S_i + \alpha]$. At time $t = S_i + \alpha$, both of A_i and A_j have completed one iteration and E_u contains a number of tokens equal to the number of tokens at time $t = S_i$. Due to the periodicity of A_i and A_j, their execution pattern repeats. Thus, b_u is also sufficient and minimum for any $t \geq S_i + \alpha$. ■

Example 4.5.1. We give now an example on how to compute the buffer sizes for a given CSDF graph. Consider the CSDF graph shown in Figure 2.1 on page 29. Recall that we computed $\vec{P}$, $\vec{D}$ and $\vec{S}$ in Examples 4.3.1 and 4.4.1. Therefore, we show in Table 4.2 the values $\vec{b}$ under different values of $\vec{\eta}$. Observe in this particular example that the buffer sizes do not change when $\vec{\eta}$ is varied. However, the buffer sizes, in general, are reduced when $\vec{\eta}$ is reduced. □

Now, we present the following theorem which represents the main results in this chapter.

Theorem 4.5.1. *For a graph G, let T be a periodic taskset such that $T_i \in T$ corresponds to $A_i \in A$. T_i is given by:*

$$T_i = (S_i, C_i, D_i, P_i), \tag{4.36}$$

where S_i is the earliest start time of A_i given by (4.23), $C_i \in \vec{C}$ is the WCET given by (4.3), D_i is the deadline given by (4.22), and P_i is the period given by (4.17). T is schedulable on m processors using any hard-real-time scheduling algorithm $\mathcal{A}$ for asynchronous sets of periodic tasks if:

1. *every communication channel $E_u \in E$ has a capacity of at least b_u tokens, where b_u is given by (4.35)*
2. *T satisfies the schedulability test of $\mathcal{A}$ on m processors*

Proof. Follows from Theorem 4.3.1, and Lemmas 4.4.2 and 4.5.2. ■

Theorem 4.5.1 states that for each program modeled as an acyclic CSDF graph, one can schedule the corresponding real-time taskset on any system composed of m processors using a scheduling algorithm $\mathcal{A}$ if (1) the communication channels are sized appropriately, and (2) the taskset satisfies the schedulability test of $\mathcal{A}$ on m processors.

4.6 Throughput Analysis

Now, given a CSDF graph, we analyze the throughput of the graph actors under a periodic schedule and compare it with the throughput under a worst-case self-timed schedule. We start with the following definitions:

Definition 4.6.1 (Actor Throughput)**.** For a graph G, the throughput of actor A_i under a periodic schedule, denoted by $\mathcal{R}_{\text{PS}}(A_i)$, is given by

$$\mathcal{R}_{\text{PS}}(A_i) = 1/P_i \tag{4.37}$$

Definition 4.6.2 (Rate-Optimal Periodic Schedule [MB07])**.** For a graph G, a periodic schedule that delivers the same throughput as a worst-case self-timed schedule for all the actors is called **Rate-Optimal Periodic Schedule (ROPS)**.

Recall matched I/O rates graphs introduced in Definition 4.2.3 on page 51. Now, we give the following result.

Lemma 4.6.1. *For a matched I/O rates graph G, the maximum achievable throughput of the graph actors under a periodic schedule is equal to their throughput under a worst-case self-timed schedule.*

Proof. The maximum achievable throughput under periodic scheduling is the one obtained when $P_i = \check{P}_i$. Recall from (4.10) that

$$\check{P}_i = \frac{\text{lcm}(\vec{q})}{q_i}\left\lceil \frac{\hat{W}}{\text{lcm}(\vec{q})} \right\rceil \tag{4.38}$$

Recall also from Table 2.1 on page 24 that ÷ denotes the integer division operator. Let us re-write $\hat{W}$ as $\hat{W} = p \cdot \text{lcm}(\vec{q}) + r$, where $p = \hat{W} \div \text{lcm}(\vec{q})$, and $r = \hat{W} \bmod \text{lcm}(\vec{q})$. Now, (4.38) can be re-written as

$$\check{P}_i = \begin{cases} \hat{W}/q_i & \text{if } \hat{W} \bmod \text{lcm}(\vec{q}) = 0 \\ (p+1)\,\text{lcm}(\vec{q})/q_i & \text{if } \hat{W} \bmod \text{lcm}(\vec{q}) \neq 0 \end{cases} \tag{4.39}$$

Recall from (4.7) that

$$\mathcal{R}_{\text{WSTS}}(A_i) = q_i/\hat{W} \tag{4.40}$$

Now, recall from Definition 4.2.3 that a matched I/O rates graph satisfies the following condition:

$$\hat{W} \bmod \mathrm{lcm}(\vec{q}) = 0 \tag{4.41}$$

Therefore, the maximum achievable throughput of the actors of a matched I/O rates graph under periodic scheduling is:

$$\mathcal{R}_{\mathrm{PS}}(A_i) = q_i/\hat{W} = \mathcal{R}_{\mathrm{WSTS}}(A_i) \tag{4.42}$$

■

(4.39) shows that the throughput under periodic scheduling depends solely on the relationship between $\mathrm{lcm}(\vec{q})$ and $\hat{W}$. If $\hat{W} \bmod \mathrm{lcm}(\vec{q}) = 0$, then $\mathcal{R}_{\mathrm{PS}}(A_i)$ is exactly the same as $\mathcal{R}_{\mathrm{WSTS}}(A_i)$. If $\hat{W} \bmod \mathrm{lcm}(\vec{q}) \neq 0$, then $\mathcal{R}_{\mathrm{PS}}(A_i)$ is lower than $\mathcal{R}_{\mathrm{WSTS}}(A_i)$.

Now, we prove the following result regarding matched I/O rates programs:

Theorem 4.6.1. *For a matched I/O rates graph G scheduled as a periodic taskset T using its minimum period vector $\vec{\check{P}}$, the maximum utilization factor $\hat{u}(T) = 1$.*

Proof. Recall from Section 2.4.1 that the utilization of a task T_i is defined as $u_i = C_i/P_i$, where $C_i \leq P_i$. Therefore, the maximum possible value for u_i is when $C_i = P_i$ which leads to $u_i = 1$. Now, let A_m be the actor with the maximum workload. It follows that

$$q_m C_m = \max_{A_i \in A}\{q_i C_i\} = \hat{W} \tag{4.43}$$

The period of A_m is P_m given by

$$P_m = \frac{\mathrm{lcm}(\vec{q})}{q_m}\left\lceil \frac{\hat{W}}{\mathrm{lcm}(\vec{q})} \right\rceil \tag{4.44}$$

Now, let us write $\hat{W}$ as $\hat{W} = p \cdot \mathrm{lcm}(\vec{q}) + r$, where $p = \hat{W} \div \mathrm{lcm}(\vec{q})$, and $r = \hat{W} \bmod \mathrm{lcm}(\vec{q})$. Then, we can re-write (4.44) as

$$P_m = \frac{\mathrm{lcm}(\vec{q})}{q_m}\left\lceil p + \frac{r}{\mathrm{lcm}(\vec{q})} \right\rceil \tag{4.45}$$

For matched I/O rates programs, $r = 0$ (see Definition 4.2.3). Therefore, (4.45) can be re-written as

$$P_m = \frac{p \cdot \mathrm{lcm}(\vec{q})}{q_m} \tag{4.46}$$

The utilization of A_m is u_m given by

$$u_m = \frac{C_m}{P_m} = \frac{q_m C_m}{p \cdot \mathrm{lcm}(\vec{q})} \tag{4.47}$$

Since $r = 0$ and $\hat{W} = p \cdot \text{lcm}(\vec{q}) = q_m C_m$, (4.47) becomes

$$u_m = \frac{\hat{W}}{\hat{W}} = 1 \tag{4.48}$$

■

4.7 Latency Analysis

Given the start times and deadlines of the actors as computed using Lemmas 4.4.2 and 4.5.2, we can compute the latency of an output path in the graph. Let W_k be an output path in a graph G, where E_r is the first channel and E_u is the last channel. We define K_j^u and K_i^r as two constants given by:

$$K_i^r = \min\{k \in \mathbb{N} : x_i^r(k) > 0\} - 1 \tag{4.49}$$

$$K_j^u = \min\{k \in \mathbb{N} : y_j^u(k) > 0\} - 1 \tag{4.50}$$

Then, according to Definitions 4.2.5 and 4.2.6 in Section 4.2 on page 51, the graph latency $\mathcal{L}(G)$ is given by:

$$\mathcal{L}(G) = \max_{W_k \in W} \{S_j + K_j^u P_j + D_j - (S_i + K_i^r P_i)\} \tag{4.51}$$

where S_j and S_i are the earliest start times of the output actor A_j and the input actor A_i, respectively, P_j and P_i are the periods of A_j and A_i, respectively, and D_j is the deadline of A_j.

Example 4.7.1. We give an example to illustrate how to compute the graph latency. Consider the CSDF graph shown in Figure 2.1 on page 29. Recall from Example 2.3.1 on page 28 that the graph has three output paths given by $W = \{W_1 = \{(A_1, A_2), (A_2, A_4)\}, W_2 = \{(A_1, A_3), (A_3, A_4)\}, W_3 = \{(A_1, A_4)\}\}$. Recall also from Example 4.3.1 on page 56 that the minimum period vector of the graph is $\vec{\check{P}} = [8, 12, 24, 8]^T$. First, we list the values of K_i^u and K_j^u defined in (4.49) and (4.50) in Table 4.3. Then, in Table 4.4, we show the output paths latencies and the graph maximum latency under different values of $\vec{\eta}$. Observe in this example that the different output paths latencies (i.e., $\mathcal{L}(W_1)$, $\mathcal{L}(W_2)$, and $\mathcal{L}(W_3)$) are equal since the source and sink actors (i.e., A_1 and A_4) are the same for all paths and the values of K_i^u and K_j^u are equal for each output path. □

Recall from (4.22) that the designer controls the deadline values using the deadline factors $\vec{\eta}$. Such control allows the designer to reduce the latency by reducing the values of the deadlines as illustrated in Example 4.7.1. However, it is not necessary to reduce *all* the deadlines in order to reduce the latency. Recall from Definition 4.2.6 on page 51, that the graph latency is dictated by the critical path which is the output

Table 4.3: *Values of K_i^u and K_j^u defined in (4.49) and (4.50) for the CSDF graph shown in Figure 2.1.*

Path	K_i^u	K_j^u
W_1	0	0
W_2	2	2
W_3	0	0

Table 4.4: *The output paths latencies and graph maximum latency of the CSDF graph shown in Figure 2.1 on page 29 under different values of $\vec{\eta}$.*

$\vec{\eta}$	$\mathcal{L}(W_1)$	$\mathcal{L}(W_2)$	$\mathcal{L}(W_3)$	$\mathcal{L}(G)$
$\vec{1}$	40	40	40	40
$\vec{0.5}$	36	36	36	36
$\vec{0}$	33	33	33	33

path with the largest latency. Therefore, it is sufficient to reduce the deadlines of the actors belonging to the critical path while leaving the deadlines of the other actors set to their periods. This is useful since lower deadlines translate to constrained-deadline tasks which translates, in general, into higher number of processors that are needed to schedule the actors. Reducing the deadlines of the critical path is accomplished using Algorithm 2. Algorithm 2 starts by initializing the deadlines of all actors to their periods. After that, it iteratively reduces the deadline of actor A_m that induces the largest start time on a given destination actor A_j. This reduction in deadline results in reducing $S_{m \to j}$. However, reducing $S_{m \to j}$ might cause another actor (e.g., A_k) to have a larger $S_{k \to j}$ than $S_{m \to j}$. Thus, the deadline reduction is repeated until no other actor A_k with larger $S_{k \to j}$ is found.

Now, we show that for two sub-classes of CSDF graphs, one can derive a simpler expression for the latency. These two classes are: (1) balanced graphs (see Definition 4.2.4 on page 51), and (2) graphs where $\vec{q} = \vec{1}$.

Theorem 4.7.1. *The minimum achievable latency of a balanced graph G when its actors are scheduled as implicit-deadline periodic tasks is equal to its latency under a worst-case self-timed schedule.*

Proof. By Definitions 4.2.4 on page 51 and 4.3.3 on page 52, a balanced graph has a period vector in which each actor has a period equal to its execution time (i.e., $P_i = C_i$). Therefore, each actor requires execution on a dedicated processor (since $u_i = C_i/P_i = 1$). In this case, the actor utilizes fully the processor on which it executes. Hence, there is no idle time or latency imposed by the scheduler and the actors execute in a way

Algorithm 2 SET-DEADLINE-AND-START-TIME$(G, \vec{\eta})$

Require: A graph G
Require: Deadline factors $\vec{\eta}$
1: **for all** $A_j \in A$ **do**
2: **if** $\text{prec}(A_j) = \varnothing$ **then**
3: $S_j \leftarrow 0$
4: **else**
5: Initialize the deadline of each actor $A_i \in \text{prec}(A_j)$ to its period (i.e., $D_i = P_i$)
6: Find $A_m \in \text{prec}(A_j)$ such that $S_{m\to j} = \max_{A_i \in \text{prec}(A_j)}\{S_{i\to j}\}$
7: Set the deadline of A_m to $D_m = C_m + \eta_m(P_m - C_m)$
8: Find $S_j = \max_{A_i \in \text{prec}(A_j)}\{S_{i\to j}\}$ with the new deadline of A_m
9: Find $A_k \in \text{prec}(A_j)$ such that $S_{k\to j} = \max_{A_i \in \text{prec}(A_j)}\{S_{i\to j}\}$
10: **if** $A_k \neq A_m$ **then**
11: Repeat lines 6 to 9 {A new actor A_k is the bottleneck}
12: **else**
13: $S_j \leftarrow \max_{A_i \in \text{prec}(A_j)}\{S_{i\to j}\}$ {A_m remains the bottleneck even after reducing its deadline}
14: **end if**
15: **end if**
16: **end for**
17: Set the deadline of each output actor A_o to $D_o = C_o + \eta_o(P_o - C_o)$
18: **return** $\vec{D}$, where $D_i \in \vec{D}$ is the deadline of actor A_i, and $\vec{S}$, where $S_i \in \vec{S}$ is the start time of actor A_i

that emulates self-timed execution. As a result, the graph has the same latency as if it is executed in a self-timed way. ■

Theorem 4.7.1 implies that a balanced graph scheduled to achieve the minimum achievable latency requires a number of processors $m = n$, where n is the number of actors in the graph. In such a case, the system has "one-to-one" mapping (i.e., one task per processor). Hence, the type of scheduler is irrelevant since each task requires a dedicated processor, which is fully utilized, in order to achieve the minimum achievable latency.

Theorem 4.7.2. *The minimum achievable latency of a graph G with basic repetition vector $\vec{q} = \vec{1}$, when its actors are scheduled as implicit-deadline periodic tasks, is $\mathcal{L}_{PS}(G) = \mathbb{L}\max_{A_i \in A}\{C_i\}$.*

Proof. Based on Lemma 4.3.1 on page 53, a CSDF graph with basic repetition vector $\vec{q} = \vec{1}$ will have a period vector in which all the periods are the same and equal to $\max_{A_i \in A}\{C_i\}$. Thus, the latency of the graph is the sum of the periods along the longest output path. By Algorithm 1 on page 27, the longest output path has $\mathbb{L}$ actors in it. Thus, the graph latency is $\mathcal{L}(G) = \mathbb{L}\max_{A_i \in A}\{C_i\}$. ■

Theorem 4.7.3. *The minimum achievable latency of a graph G with basic repetition vector $\vec{q} = \vec{1}$, when the actors of G are scheduled as constrained-deadline periodic tasks with $\vec{D} = \vec{C}$, is equal to its latency under a worst-case self-timed schedule.*

Proof. When the actors of G are scheduled with $\vec{D} = \vec{C}$, an actor $A_i \in A$ is started immediately after all its predecessors have finished at least one firing. Hence, the latency encountered by the first sample is equal to the sum of actors' execution times along the output path with the largest sum of actors' execution times. This is equivalent to the latency under a worst-case self-timed schedule. ∎

We summarize the results presented so far in the form of the **decision tree** shown in Figure 4.13. This decision tree can be used by the designer to determine the quality of periodic schedules for a given graph G. If G is a matched I/O rates graph (see Definition 4.2.3 on page 51), then the graph will achieve its optimal throughput when it is scheduled using its minimum periodic vector. If a matched I/O rates graph G is a balanced one (see Definition 4.2.4 on page 51) or has a unity basic repetition vector (i.e., $\vec{q} = \vec{1}$), then it has also an optimal latency when it is scheduled using its minimum period vector. Otherwise, G might have a sub-optimal latency under a periodic schedule. Finally, mis-matched I/O rates graphs have sub-optimal throughput and latency under periodic schedules.

4.8 Deriving Architecture and Mapping Specifications

So far, we have shown how to derive the parameters of the taskset corresponding to a given CSDF graph. Recall from Figure 1.7 on page 14 that the scheduling framework produces two outputs that are used as inputs to the system-level synthesis step. These outputs are the architecture and mapping specifications. The architecture specification refers to the number of processors needed to schedule the taskset, while the mapping specification refers to the allocation of tasks to processors. Suppose that we want to run a set of programs modeled by a set of graphs $\mathbf{G} = \{G_1, G_2, \cdots, G_N\}$ on a platform. Let $T(G_i)$ denote the taskset corresponding to the actors of G_i. Then, we define $\mathbf{T}$ to be a *super* taskset given by

$$\mathbf{T} = \bigcup_{G_i \in \mathbf{G}} T(G_i) \tag{4.52}$$

After that, we apply one of the partitioning schemes described in Section 2.4.4 on $\mathbf{T}$. This results in a m-partition of $\mathbf{T}$, denoted by ${}^{m}\mathbf{T}$. Then, the architecture specification states that the system consists of m processors, while the mapping specification states that the allocation of tasks to processors is given by ${}^{m}\mathbf{T}$. To illustrate these steps, we give the following example.

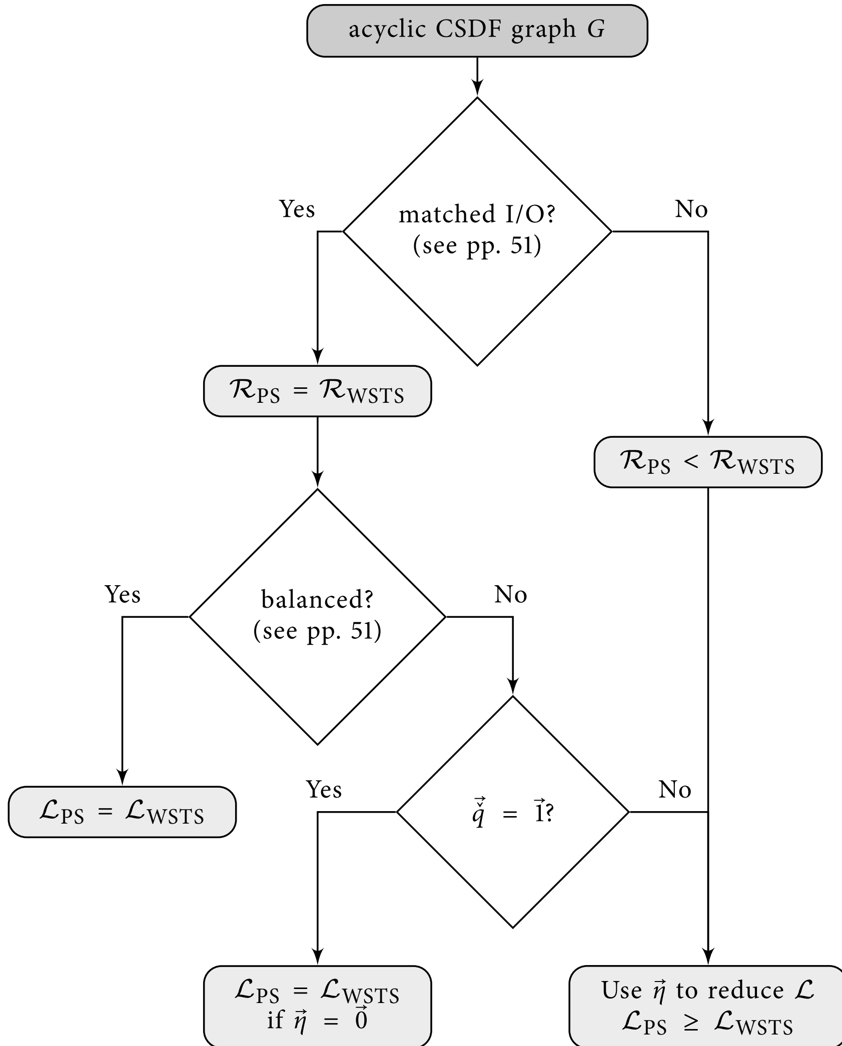


Figure 4.13: *Decision tree for scheduling CSDF actors as real-time periodic tasks. $\mathcal{R}_{PS}$ and $\mathcal{L}_{PS}$ refer to the throughput and latency, respectively, when $\vec{P} = \vec{\check{P}}$.*

Example 4.8.1. Consider the programs shown in Listing 1 (on page 6) and Listing 2. Applying the automated parallelization and model construction steps explained in Chapter 3 to both programs results in the CSDF graphs shown in Figures 2.1 and 4.14. We denote the graph shown in Figure 2.1 by G_1 and the one shown in Figure 4.14 by G_2. Recall from Example 4.3.1 that the execution time vector of G_1 is $\vec{C} = [5, 8, 24, 4]^T$. For G_2, the execution times are given between the parentheses in Figure 4.14 and

Listing 2 Another example of a SANLP in C

```
int main()
{
  for (i = 0; i < 100; i++) {
    for (j = 0; j < 100; j++) {
      in(&x);
      g1(x, &y);
      g2(y, &z);
      out(z);
    }
  }
  return 0;
}
```

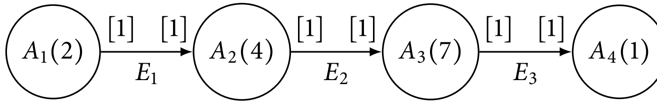


Figure 4.14: *The CSDF graph corresponding to the SANLP program in Listing 2. A_1 corresponds to* `in`*, A_2 corresponds to* `g1`*, A_3 corresponds to* `g2`*, and A_4 corresponds to* `out`*. The numbers between the parentheses are the WCET of the actors. For example, the WCET of actor A_2 is 4.*

Graph	$\vec{C}$	$\vec{P}$	$\vec{S}$	$\vec{b}$	$u_{\text{sum}}(T)$
G_1	$[5,8,24,4]^T$	$[8,12,24,8]^T$	$[0,8,24,32]^T$	$[2,2,5,3,2]^T$	$2.791\overline{6}$
G_2	$[2,4,7,1]^T$	$[7,7,7,7]^T$	$[0,7,14,21]^T$	$[2,2,2]^T$	2.0

Table 4.5: *The taskset parameters for G_1 and G_2 assuming $\mu_G = 1$ and $\vec{\eta} = \vec{1}$ for both graphs. $\vec{D}$ is omitted since $\vec{D} = \vec{P}$.*

$\vec{C} = [2,4,7,1]^T$. Assume that the period scaling factor for both graphs is 1 (see (4.17) on page 54) and the deadline factor for both graphs is 1.0 (see (4.22) on page 58). Then, we compute the periods, start times, and buffer sizes of both graphs as shown in Table 4.5. We see from Table 4.5 that the total utilization of $T(G_1)$ and $T(G_2)$ is equal to $4.791\overline{6}$. Thus, the absolute minimum number of processors needed to schedule G_1 and G_2 on a system, assuming an optimal scheduling algorithm, is given by (2.20) on page 35 and is equal to $\check{m}_{\text{OPT}} = \lceil 4.791\overline{6} \rceil = 5$. Under partitioned scheduling, the minimum number of processors needed to schedule G_1 and G_2 on a system is given by (2.21) on page 36.

Suppose that EDF is used as a scheduling algorithm and FFD (see Section 2.4.4) is used as an allocation algorithm. Then, the resulting mapping is shown in Figure 4.15. In this case, the minimum number of processors needed to schedule G_1 and G_2 is

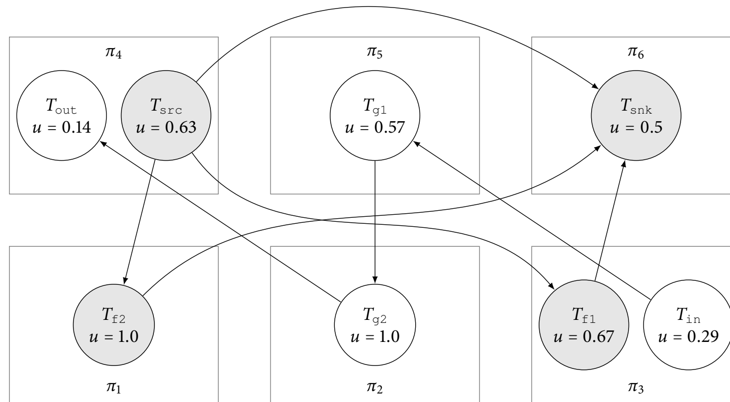


Figure 4.15: *Mapping of G_1 and G_2 onto 6 processors assuming EDF and FFD. π_i denotes the ith processor and u denotes the total utilization of a task. Each task corresponds to a function in either Listing 1 or 2. For example, T_{g1} corresponds to function `g1` in Listing 2. The arrows represent the communication between the tasks.*

equal to 6. Figure 4.15 contains both the architecture and mapping specifications needed for performing the system-level synthesis in Chapter 5. It states how many processors are needed (i.e., 6) and the mapping of each task to processors together with the communication pattern among the processors. □

Chapter 5

System-Level Synthesis

Build a system that even a fool can use,
and only a fool will want to use it.

George Bernard Shaw

SYSTEM-level synthesis represents the fifth step in the proposed design flow. The inputs to this step are the program, architecture, and mapping specifications. The program specification consists of the PPNs derived in Chapter 3 together with the tasks' parameters and buffer sizes derived in Chapter 4. The architecture specification describes the number of processors as derived in Section 4.8. Finally, the mapping specification, derived in Section 4.8, associates each task with the processor on which it runs. All these specifications are used, as shown in Figure 5.1, to generate the MPSoC platform which consists of the hardware part together with the software running on that hardware. The whole system-level synthesis procedure is performed using ESPAM [NSD08]. ESPAM is a system-level synthesis tool for streaming systems with support for model-based design. We extended ESPAM in order to: (1) generate the hardware architecture explained in the following section, (2) generate the software with the proper scheduling and communication infrastructures explained later in Section 5.2, and (3) add support for Xilinx ML605 and Zynq boards that are used later in Chapter 6 for prototyping the synthesized systems.

5.1 Hardware

In this dissertation, we consider a hardware platform consisting of a tiled distributed memory MPSoC as shown in Figure 5.2. The on-chip interconnect is assumed to be a *predictable* on-chip interconnect. A predictable on-chip interconnect is one that

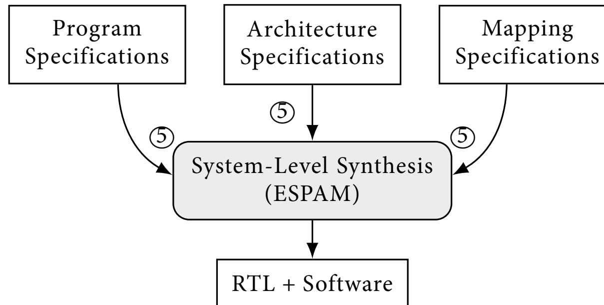


Figure 5.1: *Electronic System-Level Synthesis*

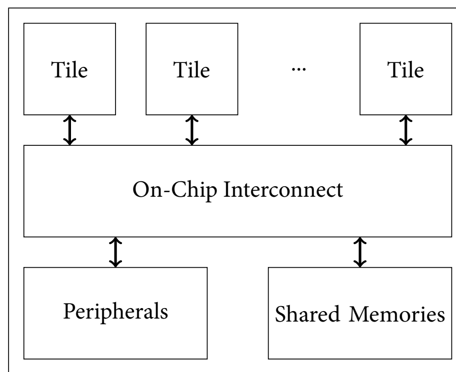


Figure 5.2: *Top-level block diagram of the hardware platform considered in this dissertation*

provides bounded worst-case latency on read/write operations between any communicating source and destination pair in the SoC. An example of such interconnect is the Æthereal network-on-chip [GDR05]. The aforementioned assumption is necessary in order to compute in Section 4.3 a safe upper bound on the worst-case execution time of each actor.

Each tile consists, as shown in Figure 5.3, of a processor, several memories, and a timer. Each tile contains three dedicated memories:

- Program Memory (PM) to store the programs' binaries
- Data Memory (DM) to store the data segments, heap and stack
- Communication Memory (CM) to store the data sent to other processors

Each processor writes the processed data to its local communication memory. After that, remote consumer processors read this data from the communication memory of the producer processor. This means that data writes are always local, and data reads are either local or remote depending on the actual mapping of tasks to processors. All the memories are implemented as dual-port memories which means that the commu-

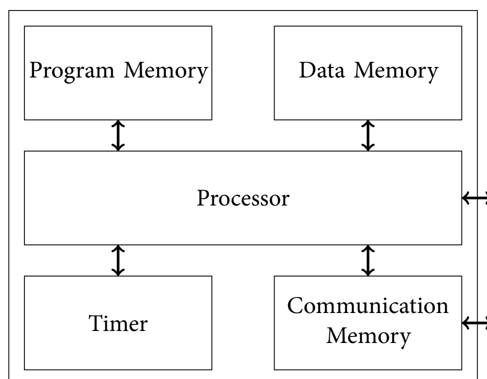


Figure 5.3: *Tile organization*

nication memory can be accessed by its owner processor and a remote processor at the same time.

A complete detailed picture of the SoC architecture integrated into ESPAM is shown in Figure 5.4. The on-chip interconnect in the SoC is a general-purpose, high performance AXI-4 [ARM10] crossbar switch which is provided by a commercial IP vendor. The crossbar features a Shared-Address, Multiple-Data (SAMD) topology as shown in Figure 5.5. It has two arbiters: one for read transactions and one for write transactions. Both arbiters are independent and can be active at the same time. The arbitration policy can be configured to be round-robin or priority-based. Parallel write and read data pathways connect each master to all the slaves that it can access according to a sparse connectivity map. When more than one source has data to send to different destinations, data transfers can occur independently and in parallel. We configure the crossbar to use the round-robin arbitration policy which enables us to derive a safe upper bound on the latencies of the communication operations.

In order to perform hardware generation, we store the hardware platform shown in Figure 5.4 as a parametrized template in ESPAM. Then, we use ESPAM to generate the actual platform in Xilinx Platform Studio (XPS) Microprocessor Hardware Specifications (MHS, [Xil11]) format. This allows importing the hardware project directly into the Xilinx XPS tool and performing FPGA synthesis.

5.2 Software

Recall from Chapter 3 that the code of the parallelized programs is generated by the `PNgen` compiler. The generated code has a form as the one shown in the example in Figure 3.2 on page 42. Thus, the remaining components that we have to generate are: (1) the scheduling infrastructure, and (2) the communication infrastructure implementing

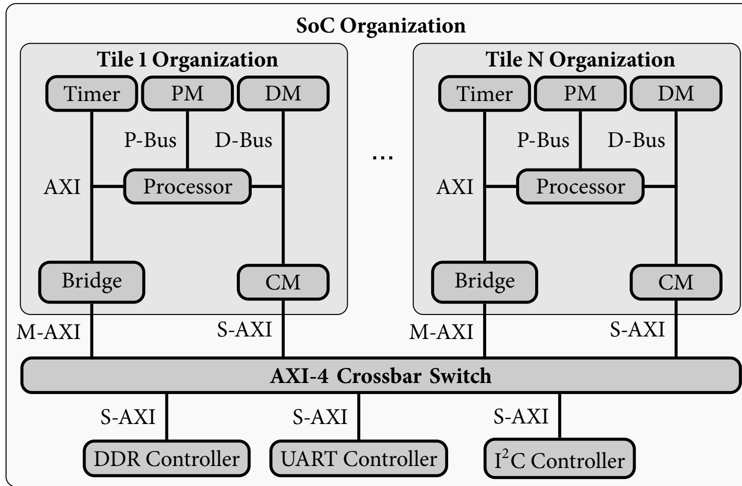


Figure 5.4: *Complete MPSoC architecture. P-Bus and D-Bus stand for program and data buses, respectively. M-AXI and S-AXI stand for AXI master and slave, respectively.*

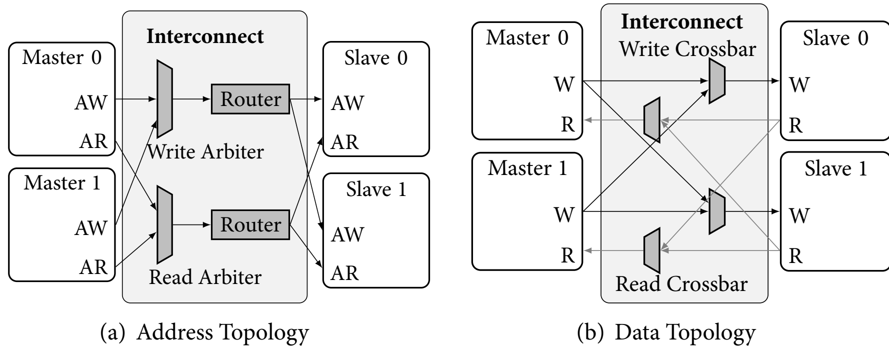
the FIFO reads/writes.

5.2.1 Scheduling Infrastructure

Recall from Chapter 4 that we schedule the tasks as periodic tasks. For the scheduler, we consider fixed task priority scheduling with Deadline Monotonic priority assignment. This choice is driven by the wide availability of real-time operating systems supporting fixed task priority scheduling. Nevertheless, it is important to note that different scheduling algorithms (e.g., EDF) can be used. We chose to implement the scheduling infrastructure using FreeRTOS [Reab]. FreeRTOS is an open source RTOS that implements fixed task priority scheduling and supports Xilinx FPGAs which are used later in Chapter 6 for evaluating the synthesized systems.

Tick-Based Implementation

FreeRTOS, as many real-time operating systems, relies on using **hardware timers** to keep track of time. Such timers generate periodic interrupts and these interrupts cause the OS to invoke the scheduler. A single interrupt and the associated scheduling event are called **OS clock tick**. The OS clock tick defines the shortest time granularity visible to the OS. OS clock tick is different from the processor (or CPU) clock tick. A processor clock tick refers to the duration of a single clock cycle of the clock signal used to operate the processor. Most of the WCET analysis tools measure the WCET of a task in terms of processor clock cycles. However, the RTOS can keep track only of time durations that



(a) Address Topology (b) Data Topology

Figure 5.5: *Crossbar Topology. AW stands for AXI write address channel, AR for AXI read address channel, W for write data channel, and R for read data channel.*

are multiple of the OS clock tick duration. Therefore, it is necessary during the system synthesis phase to ensure that all the timing parameters are converted to the appropriate OS clock tick values. This can be done, for example, by rounding the parameters up to the nearest multiple of the OS clock tick duration. To illustrate the previous concepts, we provide the following example.

Example 5.2.1. Suppose that we have a system comprised of a processor with a clock frequency equal to 1 GHz (i.e., processor clock cycle is 1 ns). Suppose that we want to run a task T_1 with the following parameters (all in processor clock cycles) $T_1 = (C_1 = 1.5 \times 10^6, P_1 = 2.5 \times 10^6, D_1 = 2.5 \times 10^6, S_1 = 0)$. Now, suppose that the OS clock tick frequency is 1000 Hz. This means that the OS performs scheduling events every 1 ms, which is equivalent to 10^6 processor clock cycles. We see that the period and deadline of T_1 are not multiples of the OS clock tick duration. Therefore, P_1 and D_1 must be rounded up to the nearest multiple of the OS clock tick duration which is 3.0×10^6. Such rounding might of course violate the timing requirements dictated by the designer. Therefore, it is important to keep in mind the effect of such rounding while specifying the system and program timing requirements. □

One effect of the tick-based implementation that must be taken into account is the ratio between the tasks' WCET and the OS clock tick duration. If the WCET is a fraction of the OS clock tick, then the resulting schedule has sub-optimal throughput with under-utilized processors and the overhead of the RTOS is not amortized. On the other hand, if the WCET is larger than the OS clock tick duration (preferably multiples of the OS clock tick), then the RTOS overhead is amortized. Therefore, it is important to consider this relation between the tasks' WCET and the OS clock tick duration when the timing parameters are converted from CPU clock cycles into OS clock ticks.

A periodic task T_i can be implemented in FreeRTOS as shown in Listing 3. Variable

Listing 3 Implementing a periodic task in FreeRTOS

```
1  void task(void *arg) {
2    portTickType LastReleaseTime;
3    const portTickType Period = 5;
4    LastReleaseTime = xTaskGetTickCount();
5
6    for (;;) {
7      function();
8      vTaskDelayUntil( &LastReleaseTime, Period );
9    }
10 }
```

`LastReleaseTime` records, as its name implies, the last release time of T_i in OS clock ticks. This variable is initialized when the task starts. Constant `Period` represents the period of the task in terms of OS clock ticks. For example, in Listing 3, the period is 5 OS clock ticks. Inside the `for`-loop, the task function (i.e., `function()`) is executed infinitely. Upon each execution, function `vTaskDelayUntil`, which is part of the FreeRTOS API, is called. The detailed description of `vTaskDelayUntil` is shown in Figure 5.6. The function takes two parameters: `LastReleaseTime` and `Period`. Upon calling it, it puts the task in the sleep state and schedules it for reactivation at time t = `LastReleaseTime` + `Period`. It also updates the value of `LastReleaseTime` accordingly.

Another effect of the tick-based implementation that must be also taken into account is the need to synchronize the time returned by `xTaskGetTickCount()` among the different processors. In an MPSoC, the clock signals of the different processors are usually generated from a single "reference" clock signal produced by an oscillator. Therefore, the clock signals used by the processors can be kept in phase. The moment, at which the OS clock tick count returned by `xTaskGetTickCount()` is initialized, can be synchronized through the use of a global barrier in the initialization code of each processor. For example, see line 14 in Listing 5.

Example 5.2.2. Consider the PPN shown in Figure 3.2 on page 42. Process $\mathcal{P}_{\texttt{snk}}$ is realized under FreeRTOS as shown in Listing 4. Note that the `while`-loop shown in Figure 3.2 is replaced with `for(;;)` in Listing 4. Note also that `vTaskDelayUntil` is placed such that after each invocation of function `snk`, the task postpones its next execution to the next release time in accordance with the real-time periodic task model as defined in Section 2.4.1. The `READ` primitive together with its counterpart `WRITE` are used to read/write from/to the FIFOs, respectively. These two primitives are explained later in Section 5.2.2. □

Function Prototype:

```
void vTaskDelayUntil( portTickType *LastReleaseTime,
                      portTickType Period );
```

Description:
Delay a task until a specified time. This function can be used by cyclical tasks to ensure a constant execution frequency.
This function differs from `vTaskDelay()` in one important aspect: `vTaskDelay()` specifies a time at which the task wishes to unblock relative to the time at which `vTaskDelay()` is called, whereas `vTaskDelayUntil()` specifies an absolute time at which the task wishes to unblock.
`vTaskDelay()` will cause a task to block for the specified number of ticks from the time `vTaskDelay()` is called. It is therefore difficult to use `vTaskDelay()` by itself to generate a fixed execution frequency as the time between a task unblocking following a call to `vTaskDelay()` and that task next calling `vTaskDelay()` may not be fixed (the task may take a different path though the code between calls, or may get interrupted or preempted a different number of times each time it executes).
Whereas `vTaskDelay()` specifies a wake time relative to the time at which the function is called, `vTaskDelayUntil()` specifies the absolute (exact) time at which it wishes to unblock.
It should be noted that `vTaskDelayUntil()` will return immediately (without blocking) if it is used to specify a wake time that is already in the past. Therefore a task using `vTaskDelayUntil()` to execute periodically will have to re-calculate its required wake time if the periodic execution is halted for any reason (for example, the task is temporarily placed into the Suspended state) causing the task to miss one or more periodic executions. This can be detected by checking the variable passed by reference as the `LastReleaseTime` parameter against the current tick count. This is however not necessary under most usage scenarios.
This function must not be called while the RTOS scheduler has been suspended by a call to `vTaskSuspendAll()`.

Parameters:
`LastReleaseTime`: Pointer to a variable that holds the time at which the task was last unblocked. The variable must be initialized with the current time prior to its first use (see the example below). Following this the variable is automatically updated within `vTaskDelayUntil()`.
`Period`: The cycle time period. The task will be unblocked at time `(*LastReleaseTime + Period)`. Calling `vTaskDelayUntil` with the same `Period` parameter value will cause the task to execute with a fixed interval period.

Figure 5.6: *Detailed description of function* `vTaskDelayUntil`*. Source: [Reaa].*

Enforcing Start Times

In many commercial real-time operating systems, the API provided by the RTOS does not allow the programmer to specify explicitly the start time of a task when the task is

Listing 4 Implementing process $\mathcal{P}_{\texttt{snk}}$ in Figure 3.2 as a periodic task under FreeRTOS

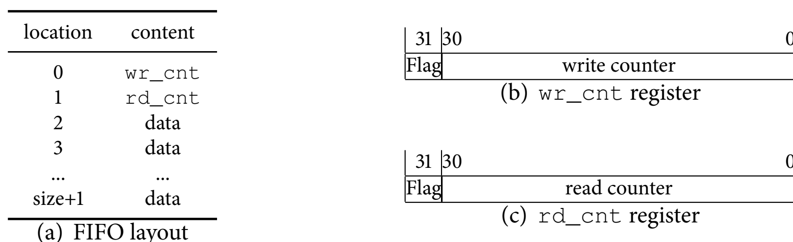
```
void task_snk(void *arg) {
  portTickType LastReleaseTime;
  const portTickType Period = 5;
  LastReleaseTime = xTaskGetTickCount();

  for (;;) {
    for(i=1;i<=10;i++) {
      for(j=1;j<=3;j++) {
        if(j<=2)
          READ(&in1,IP1,SIZE_OF_in1,SIZE_OF_FIFO_E4);
        else
          READ(&in1,IP2,SIZE_OF_in1,SIZE_OF_FIFO_E5);

        READ(&in2,IP3,SIZE_OF_in2,SIZE_OF_FIFO_E3);

        snk(in1,in2);
        vTaskDelayUntil( &LastReleaseTime, Period );
      }
    }
  }
}
```

created. Therefore, it is the programmer's responsibility to implement a mechanism which ensures that a task starts on its specified start time as derived in Section 4.4. The start time of a task may be realized under such an RTOS using several mechanisms. We list here two possible mechanisms:

1. The first mechanism is to use a *master* task that releases the programs' tasks at the time when they are supposed to start. This master task releases each task at the OS clock tick on which the task should begin its execution. After starting all the tasks, the master task can be terminated or put into a permanent sleep state.
2. The second mechanism is to release all the tasks simultaneously. After that, each task is put into sleep state from the moment of simultaneous release till the moment at which it should start.

The first mechanism provides tight control on when to start the tasks. However, a disadvantage of this mechanism is the extra overhead introduced by the master task. The utilization of the master task must be taken into account while deriving the architecture and mapping specifications (see Section 4.8) in order to avoid any deadline misses.

The second mechanism does not have the utilization overhead of the master task mechanism. This mechanism keeps the execution of the task conforming to the periodic



location	content
0	wr_cnt
1	rd_cnt
2	data
3	data
...	...
size+1	data

(a) FIFO layout

(b) `wr_cnt` register

(c) `rd_cnt` register

Figure 5.7: *FIFO layout in memory and the read/write registers*

task model as it does not cause the task to block the processor from other tasks.

Example 5.2.3. Consider task `snk` shown in Listing 4. The implementation of `snk` assuming the second mechanism (i.e., simultaneous release) is shown in Listing 5. Variable `SimultaneousReleaseTime` is passed from the `main` function that releases all the program's tasks. This variable is used to put the task in sleep state until its start time. After that, the task executes as a periodic task starting from the time assigned to `LastReleaseTime` at line 29. □

5.2.2 Communication Infrastructure

The communication infrastructure deals with the implementation of the `READ` and `WRITE` primitives shown for example in Figure 3.2 on page 42 and Listing 4 on page 82. These primitives provide the actors with the ability to communicate among each other. Recall from Section 5.1 that each actor produces data to its local communication memory, and reads data from its communication memory and/or remote communication memories. The FIFOs are implemented as circular buffers, and they are stored in the communication memories of the processors (see Figure 5.4 on page 78). The size of a single data word in the FIFOs is 32 bits. Each FIFO contains two special data words called `wr_cnt` and `rd_cnt` as shown in Figure 5.7(a). These data words store two pieces of information as shown in Figure 5.7(b) and 5.7(c): (1) the write and read counters of the FIFO, and (2) a special bit called "Flag" which is used for detecting counter overflows. Whenever the read/write counter exceeds the FIFO size, the flag bit is toggled. Storing the counter and flag in one data word enables updating the FIFO state in a producer/consumer task using a single atomic operation.

A detailed implementation of the read/write operations is depicted in Listings 6 and 7. The read/write operations accept four input parameters: (1) a pointer to the value read/written (`val`), (2) a pointer to the FIFO (`pos`), (3) the amount of data, in 32-bit words, being read/written during an invocation of the read/write operation (`len`), and (4) the size of the FIFO in 32-bit words (`size`). The implementation shown in Listings 6 and 7 assumes that the amount of data written/read by the producer/consumer,

Listing 5 Implementing the simultaneous release mechanism under FreeRTOS. The listing shows only the relevant code to the simultaneous release mechanism and other non-relevant details are omitted.

```
int main() {
  static portTickType SimultaneousReleaseTime;
  /* xTaskCreate() (part of FreeRTOS API) creates new tasks */
  xTaskCreate( task_snk, "snk", SNK_STACK_SIZE, \
               &SimultaneousReleaseTime, SNK_PRIORITY, NULL);
  /* Other xTaskCreate() invocations go here */

  /* xTaskGetTickCount() (part of FreeRTOS API) returns the count
     of OS clock ticks since vTaskStartScheduler() was called.
     If vTaskStartScheduler() was not called, it returns 0 */
  SimultaneousReleaseTime = xTaskGetTickCount();

  /* Set up a global barrier to synchronize the processors */
  waitForGlobalStartSignal(); /*  */

  /* vTaskStartScheduler() (part of FreeRTOS API) invokes the
     scheduler for the first time. It also resets the count of
     OS clock ticks returned by xTaskGetTickCount() to 0 */
  vTaskStartScheduler();
}
void task_snk(void *arg) {
  portTickType LastReleaseTime, SimultaneousReleaseTime;
  const portTickType Period = 5;
  const portTickType StartTime = 20;
  /* SimultaneousReleaseTime is set in main() */
  SimultaneousReleaseTime = *((portTickType *) arg);
  vTaskDelayUntil( &SimultaneousReleaseTime, StartTime );
  /* Set LastReleaseTime to the actual start time */
  LastReleaseTime = xTaskGetTickCount(); /*  */

  for (;;) {
    for(i=1;i<=10;i++) {
      for(j=1;j<=3;j++) {
        if(j<=2) READ(&in1,IP1,SIZE_OF_in1,SIZE_OF_FIFO_E4);
        else     READ(&in1,IP2,SIZE_OF_in1,SIZE_OF_FIFO_E5);
        READ(&in2,IP3,SIZE_OF_in2,SIZE_OF_FIFO_E3);
        snk(in1,in2);
        vTaskDelayUntil( &LastReleaseTime, Period );
} } } }
```

Listing 6 An example implementation of the read macro under FreeRTOS

```
1  READ(void *val, void *pos, int len, int size){
2    volatile int *fifo=(int *)pos;
3    int r_cnt = fifo[1];
4    int w_cnt = fifo[0];
5    int i = 0;
6    while(w_cnt == r_cnt){
7      taskDISABLE_INTERRUPTS();
8      xil_printf("PANIC! Buffer Underflow\n");
9      for (;;);
10   }
11   for(i = 0; i < len; i++){
12     ((volatile int *)val)[i]= fifo[(r_cnt & 0x7FFFFFFF)+2+i];
13   }
14   r_cnt += len;
15   if((r_cnt & 0x7FFFFFFF) == size){
16     r_cnt &= 0x80000000;
17     r_cnt ^= 0x80000000;
18   }
19   fifo[1] = r_cnt;
20 }
```

respectively, is always the same. That is, for a given communication channel, the value of `len` used in `WRITE` by the producer and the value of `len` used in `READ` by the consumer are the same. When a task T_i reads `len` words from the FIFO into a buffer `val`, the read macro performs the following steps:

1. The read and write counters are copied into local variables (lines 3 and 4)
2. If a buffer underflow occurs (i.e., FIFO is empty), then the interrupts are disabled and a "panic" message is printed to the user to indicate that a buffer underflow has occurred (lines 6-10). It is important to note that this situation should not occur under normal operating conditions since the start times and buffer sizes derived in Sections 4.4 and 4.5 are valid. Recall from Section 1.2 that normal operating conditions mean that both system hardware and software function properly without faults.
3. The `for`-loop copies the data from the communication memory into `val` and the read counter is incremented (lines 11-14).
4. After that, the read counter is checked for overflow condition and Flag is toggled accordingly (lines 15-18).
5. Finally, the macro updates `rd_cnt` register in the FIFO with the new value of the read counter by doing a single atomic assignment (line 19).

Analogously, when a task T_i writes `len` words to the FIFO from a buffer `val`, the

Listing 7 An example implementation of the write macro under FreeRTOS

```
WRITE(void *val, void *pos, int len, int size){
  volatile int *fifo=(int *)pos;
  int w_cnt = fifo[0];
  int r_cnt = fifo[1];
  int i = 0;
  while(r_cnt == (w_cnt ^ 0x80000000)){
    taskDISABLE_INTERRUPTS();
    xil_printf("PANIC! Buffer overflow\n");
    for (;;);
  }
  for(i = 0; i < len; i++) {
    fifo[(w_cnt & 0x7FFFFFFF)+2+i] = ((volatile int *)val)[i];
  }
  w_cnt += len;
  if((w_cnt & 0x7FFFFFFF) == size){
    w_cnt &= 0x80000000;
    w_cnt ^= 0x80000000;
  }
  fifo[0] = w_cnt;
}
```

write macro performs the following steps:

1. The read and write counters are copied into local variables (lines 3 and 4)
2. If a buffer overflow occurs (i.e., FIFO is full), then, similar to `READ`, the interrupts are disabled and a "panic" message is printed to the user (lines 6-10). Note again that this situation should not occur under normal operating circumstances since the start times and buffer sizes derived in Sections 4.4 and 4.5 are valid.
3. The `for`-loop copies the data from `val` into the communication memory and the write counter is incremented (lines 11-14).
4. After that, the write counter is checked for overflow condition and Flag is toggled accordingly (lines 15-18).
5. Finally, the macro updates `wr_cnt` register in the FIFO with the new value of the write counter by doing a single atomic assignment (line 19).

Chapter 6

Evaluation and Results

In theory, there is no difference between theory and practice. But, in practice, there is.

Jan L. A. van de Snepscheut

IN this chapter, we evaluate the proposed scheduling framework and design flow by performing a set of experiments. The first experiment evaluates the first two phases of the proposed design flow (i.e., automated parallelization and model construction) by measuring the time needed to perform them on a set of real life programs. The second experiment evaluates the scheduling framework proposed in Chapter 4. Namely, it evaluates the following performance and resource usage metrics for streaming programs under periodic scheduling: (1) throughput, (2) latency, (3) processor requirements, and (4) memory requirements. It also compares these metrics to their counterparts obtained under self-timed scheduling. Finally, the third experiment validates the synthesized systems by running them on actual hardware and checking the timing behavior during system run-time against the timing specifications reported by the scheduling framework during the design process.

Unless mentioned otherwise, all the experiments were performed on a Lenovo ThinkPad T500 laptop which has the specifications outlined in Table 6.1.

Table 6.1: *Specifications of the machine on which the experiments were performed*

Property	Value
Processor	Intel Core2 Duo T9400 CPU at 2.53GHz
RAM	4 GB
Operating system	Ubuntu 12.04 LTS (64-bit)

Table 6.2: *Time needed to parallelize and derive the CSDF model for the benchmark programs*

Program	No. of actors	No. of edges	Lines of code	Time (s)
Filter-bank	69	89	367	1.60
Alternating direction implicit solver	28	167	209	7.26
FM radio	28	39	195	0.66
2D finite difference time domain kernel	17	71	144	0.89
2D gauss blur filter for image processing	11	26	75	7.82
Gram-Schmidt	9	20	48	1.85
Regularity detector	8	11	54	2.86

6.1 Experiment I: Evaluating Automated Parallelization and Model Construction

In this experiment, we evaluate the first two phases in the proposed design flow. These two phases are automated parallelization and model construction. We do that by parallelizing a set of real-life programs and deriving their CSDF models. The used programs are from the PolyBench benchmark [Pou]. The programs are specified as sequential programs in C and vary in their size and complexity. The list of programs together with the time needed to parallelize them and derive their CSDF models is shown in Table 6.2.

The time reported in Table 6.2 includes: (1) the time needed by the `PNgen` compiler to parse the C program and generate the PPN, (2) the time needed to derive the CSDF model as described in Chapter 3. We see clearly that the first two phases of the proposed flow (i.e., automated parallelization and model construction) are very fast. The fast derivation of the PPN and CSDF models relieves the designer from the burden of writing the parallel specifications manually. Moreover, this allows the designer to explore a large number of alternative program specifications in a short period of time [ZNS13, ZBS13].

6.2 Experiment II: Evaluating Performance and Resource Usage Metrics under Periodic Scheduling

In this experiment, given a streaming program executed under a periodic schedule, we evaluate the following performance and resource usage metrics: (1) throughput, (2) latency, (3) processor requirements, and (4) memory requirements. Then, we compare these metrics with those obtained under a self-timed schedule. Recall from Theorem 4.2.1 on page 52 that the maximum achievable throughput and minimum achievable latency of a streaming program modeled as a CSDF graph are the ones achieved under self-timed scheduling. For brevity, we refer in the remainder of this section to periodic scheduling/schedule as PS and the self-timed scheduling/schedule

as STS. In this experiment, we report the throughput for the output actors (i.e., the actors producing the output streams of the program, see Section 2.3). For latency, we report the graph maximum latency according to Definition 4.2.6 on page 51. Under periodic scheduling, we use the minimum period vector given by Lemma 4.3.1 on page 53. The self-timed schedule parameters are computed using the SDF^3 tool-set [SGB06]. SDF^3 is a powerful analysis tool-set which is capable of analyzing CSDF and SDF graphs to check for consistency errors, compute the repetition vector, compute the maximum achievable throughput and latency, etc. SDF^3 defines $\mathcal{R}_{\text{STS}}(G)$ as the graph throughput under self-timed scheduling, and $\mathcal{R}_{\text{STS}}(A_i) = q_i \mathcal{R}_{\text{STS}}(G)$ as the actor throughput. Similarly, $\mathcal{L}_{\text{STS}}(G)$ denotes the graph latency under self-timed scheduling. We use the `sdf3analysis` tool from SDF^3 to compute the throughput and latency for the self-timed schedule assuming unbounded FIFO channel sizes. We also use the same tool to compute the minimum buffer sizes required to achieve the maximum achievable throughput under a self-timed schedule. We compute the throughput using the `throughput` option, the latency using the `latency(min_st)` option, and the buffer sizes using the `buffersize` option.

This experiment is performed on a set of real-life streaming programs. These programs come from different domains (e.g., signal processing, communication, multimedia, etc.). The benchmark programs are described in detail in the following section.

6.2.1 Benchmarks

We collected the benchmarks from several sources. The first source is the StreamIt benchmark [TA10] which contributes 11 streaming programs. The second source is the SDF^3 benchmark [SGB06] which contributes five streaming programs. The third source is individual research articles which contain real-life CSDF graphs such as [MBvdBvM08, OH04, PMN+09]. In total, 19 programs are considered as shown in Table 6.3. These programs are modeled using a mixture of CSDF and SDF graphs. For StreamIt benchmarks, the actors' execution times are specified in CPU clock cycles measured on MIT RAW architecture [TKM+02], while for SDF^3 benchmarks, the actors' execution times are specified in CPU clock cycles on the ARM architecture. For the graphs from [OH04, PMN+09], the authors do not mention explicitly the actors' execution times. As a result, we make assumptions regarding the execution times which are reported below Table 6.3.

6.2.2 Throughput Evaluation

Table 6.4 shows the results of comparing the throughput of the output actor for every program under both self-timed and periodic schedules. The most important column in the table is the last column which shows the *ratio* of the PS schedule throughput to

Table 6.3: *Benchmarks used for evaluating the periodic scheduling framework proposed in Chapter 4. $|A|$ denotes the number of actors in the graph, while $|E|$ denotes the number of communication channels.*

Domain	No.	Program	$\|A\|$	$\|E\|$	Source
Signal Processing	1	Multi-channel beamformer	57	70	[TA10]
	2	Discrete cosine transform (DCT)	8	7	-
	3	Fast Fourier transform (FFT) kernel	17	16	-
	4	Filterbank for multirate signal processing	85	99	-
	5	Time delay equalization (TDE)	29	28	-
Cryptography	6	Data Encryption Standard (DES)	53	60	-
	7	Serpent	120	128	-
Sorting	8	Bitonic Parallel Sorting	40	46	-
Video processing	9	MPEG2 video	23	26	-
	10	H.263 video decoder	4	3	[SGB06]
Audio processing	11	MP3 audio decoder	14	18	-
	12	CD-to-DAT rate converter (SDF)[1]	6	5	[OH04]
	13	CD-to-DAT rate converter (CSDF)	6	5	-
	14	Vocoder	114	147	[TA10]
Communication	15	Software FM radio with equalizer	43	53	-
	16	Data modem	6	5	[SGB06]
	17	Satellite receiver	22	26	-
	18	Digital Radio Mondiale receiver	4	3	[MBvdBvM08]
Medical	19	Heart pacemaker[2]	4	3	[PMN+09]

[1] We use two implementations for CD-to-DAT: SDF and CSDF and we refer to them as CD2DAT-S and CD2DAT-C, respectively. The assumed WCET are $\vec{C} = [5, 2, 3, 1, 4, 6]^T$ μs.
[2] We assume the following WCET: Motion Est.: 4 μs, Rate Adapt.: 3 μs, Pacer: 5 μs, and EKG: 2 μs.

the STS schedule throughput ($\mathcal{R}_{\text{PS}}(A_{\text{out}})/\mathcal{R}_{\text{STS}}(A_{\text{out}})$), where A_{out} denotes the output actor. We clearly see that periodic scheduling delivers the same throughput as self-timed scheduling for 16 out of 19 programs. All these 16 programs are matched I/O rates programs that have $\mathcal{R}_{\text{WSTS}}$ equal to $\mathcal{R}_{\text{STS}}$. Only three programs (CD2DAT-(S,C) and Satellite) are mis-matched and have lower throughput under periodic scheduling. Table 6.4 confirms also the observation made by the authors in [TA10] who reported an interesting finding: *"Neighboring actors often have matched I/O rates. This reduces the opportunity and impact of advanced scheduling strategies proposed in the literature"*. According to [TA10], the advanced scheduling strategies proposed in the literature (e.g., [SB09]) are suitable for *mis-matched* I/O rates programs. Looking into the results in Table 6.4, we see that periodic scheduling performs very-well for matched I/O programs.

6.2.3 Latency Evaluation

Figure 6.1 shows the ratios of the latency under periodic scheduling to the latency under self-timed scheduling. Recall from Section 4.7 that the latency can be controlled using the deadline factors $\vec{\eta}$. For all the programs scheduled using $\vec{\eta} = \vec{1}$, the average latency

Table 6.4: *Results of Throughput Comparison. A_{out} denotes the output actor.*

Program	$\check{q}_{\text{out}}$	$\mathcal{R}_{\text{STS}}(A_{\text{out}})$	$\hat{W}$	$\text{lcm}(\vec{q})$	$\mathcal{R}_{\text{PS}}(A_{\text{out}})$	$\frac{\mathcal{R}_{\text{PS}}(A_{\text{out}})}{\mathcal{R}_{\text{STS}}(A_{\text{out}})}$
Beamformer	1	1.97×10^{-4}	5076	1	1/5076	1.0
DCT	1	2.1×10^{-5}	47616	1	1/47616	1.0
FFT	1	8.31×10^{-5}	12032	1	1/12032	1.0
Filterbank	1	8.84×10^{-5}	11312	1	1/11312	1.0
TDE	1	2.71×10^{-5}	36960	1	1/36960	1.0
DES	1	9.765×10^{-4}	1024	1	1/1024	1.0
Serpent	1	2.99×10^{-4}	3336	1	1/3336	1.0
Bitonic	1	1.05×10^{-2}	95	1	1/95	1.0
MPEG2	1	1.30×10^{-4}	7680	1	1/7680	1.0
H.263	1	3.01×10^{-6}	332046	594	1/332046	1.0
MP3	2	5.36×10^{-7}	3732276	2	1/1866138	1.0
CD2DAT-S	160	1.667×10^{-1}	960	23520	1/147	0.04
CD2DAT-C	160	1.361×10^{-1}	1176	23520	1/147	0.05
Vocoder	1	1.1×10^{-4}	9105	1	1/9105	1.0
FM	1	6.97×10^{-4}	1434	1	1/1434	1.0
Modem	1	6.25×10^{-2}	16	16	1/16	1.0
Satellite	240	2.27×10^{-1}	1056	5280	1/22	0.2
Receiver	288000	4.76×10^{-2}	6048000	288000	1/21	1.0
Pacemaker	64	2.0×10^{-1}	320	320	1/5	1.0

under periodic scheduling is five times the latency under self-timed scheduling. We also see that the mis-matched programs have large latency due to their sub-optimal throughput. If we exclude the mis-matched programs, then the average latency is four times the latency under self-timed scheduling. For latency-insensitive programs, this is acceptable as long as they can be scheduled using the periodic task model to achieve the maximum achievable throughput. For latency-sensitive programs, reducing the latency can be done by using smaller values of the deadline factors $\vec{\eta}$ as explained in Section 4.7. For example, the Vocoder program has a ratio $\mathcal{L}_{\text{PS}}(G)/\mathcal{L}_{\text{STS}}(G) \approxeq 13.5$ when $\vec{\eta} = \vec{1}$. This ratio is reduced to 1.0 when $\vec{\eta} = \vec{0}$. If $\vec{\eta}$ is set to $\vec{0}$ for all the programs, then we see that 14 out of 19 programs achieve optimal latency. Two matched I/O rates programs (Receiver and Pacemaker) have sub-optimal latency under periodic scheduling when $\vec{\eta} = \vec{0}$. This is due to the following reasons. First, the execution times in these two programs have large variations between firings. Such variation is captured under self-timed scheduling, while the scheduling framework proposed in Chapter 4 assumes always the WCET. Second, we report the maximum latency while SDF^3 reports the actual latency under a self-timed schedule.

6.2.4 Processor Requirements Evaluation

For processor requirements, we compute the minimum number of processors needed to schedule the program under optimal and partitioned schedulers. We choose to

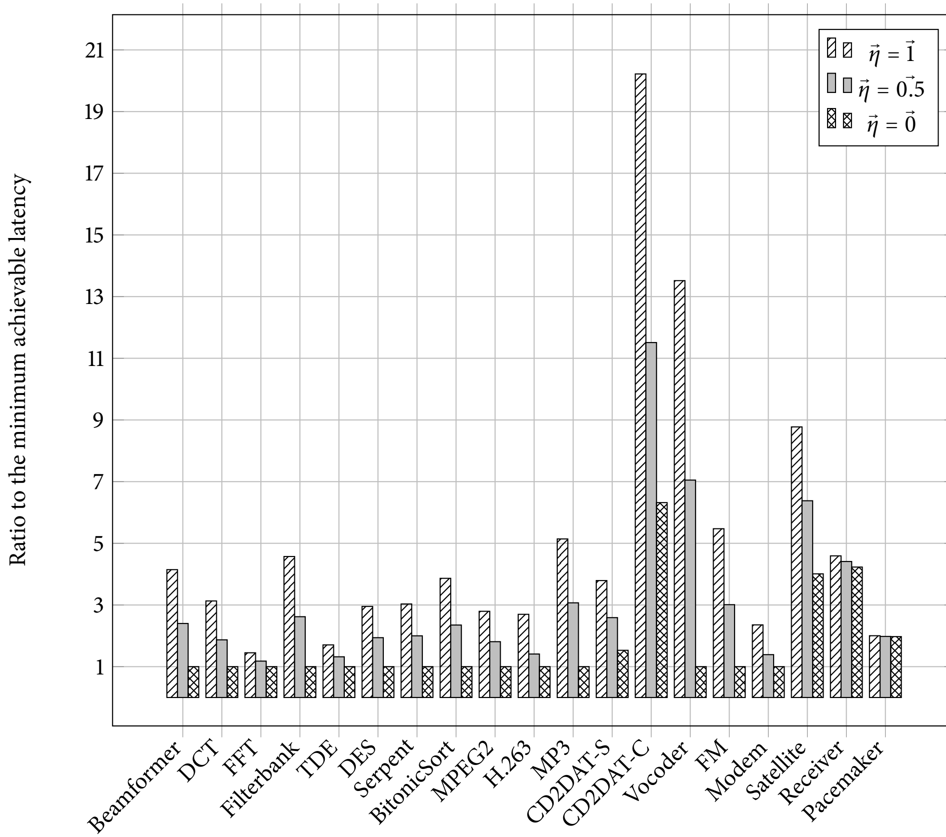


Figure 6.1: *Results of the latency evaluation. The latency is computed by setting $\vec{\eta}$ and applying Algorithm 2 to compute the start times and deadlines.*

compute the minimum required number of processors under these two classes of schedulers because it is possible to do so in an analytical and easy way using (2.20) and (2.21) on page 35. Unfortunately, such easy computation of the minimum number of processors is not possible under self-timed scheduling. This is because the minimum number of processors required by self-timed scheduling, denoted by $\check{m}_{\text{STS}}$, can not be easily computed with equations such as (2.20) and (2.21). Finding $\check{m}_{\text{STS}}$ in practice requires design space exploration procedures to find the best allocation which delivers the required throughput and latency. SDF^3 tool-set used to compute the self-timed scheduling parameters does not support such design space exploration for self-timed scheduling. Therefore, we choose to set the number of processors required by self-timed scheduling to its upper bound which is the number of actors in the graph.

Figure 6.2 shows the minimum number of processors required to schedule the

matched I/O rates programs from Table 6.3 under optimal and partitioned schedulers. The number of processors is computed assuming: (1) EDF algorithm with QPA schedulability test (see Section 2.4.3), (2) FFD allocation (see Section 2.4.4), and (3) period scaling factor $\mu_G = 1$ for all the programs. When $\vec{\eta} = \vec{1}$, we see that nine out of 16 programs require the same number of processors under both optimal and partitioned schedulers. The remaining seven programs require on average 14% more processors under partitioned schedulers. As $\vec{\eta}$ is decreased, we observe the following trends. First, some programs tend to require the same or slightly higher number of processors compared to the case when $\vec{\eta} = \vec{1}$ (e.g., Filterbank and FMRadio). Second, some other programs tend to have an increase in the number of processors that is proportional to the increase in $\vec{\eta}$ (e.g., Beamformer and Serpent). Finally, for all programs, we observe a large "jump" in the number of processors when $\vec{\eta} = \vec{0}$.

6.2.5 Memory Requirements Evaluation

Given a streaming program, we compute the total amount of memory needed to realize the buffers in the communication channels under periodic and self-timed schedules. We compute the total amount of memory assuming period scaling factor $\mu_G = 1$ and deadline factors $\vec{\eta} = \vec{0}$. This part of Experiment II is conducted on a Dell PowerEdge T710 server running Ubuntu 11.04 (64-bit) Server OS. Table 6.5 shows, for the matched I/O rates programs in Table 6.3, the total amount of memory required under a periodic schedule, denoted by M_{PS}, and total amount of memory required under a self-timed schedule, denoted by M_{STS}. We also report the time needed to compute the buffer sizes under both schedules (i.e., t_{PS} and t_{STS}). We see that seven out of 16 programs have identical memory requirements under both periodic and self-timed schedules. MPEG, Vocoder, and Modem have increased memory requirements under periodic schedules, nevertheless, the increase remains below 12%. Only one program (Pacemaker) has +144% increase in memory requirements under periodic schedules. The reason for this huge increase is related to the reason for its sub-optimal latency as explained in Section 6.2.3. Pacemaker has large variations in its execution time which are taken into account under self-timed scheduling. These variations, however, are not taken into account under the scheduling framework proposed in Chapter 4 which assumes always the WCET.

6.2.6 Summary of Experiment II

In Sections 6.2.2-6.2.5, we provided a detailed comparison between periodic and self-timed scheduling for a set of real streaming programs. We compared the following metrics: (1) throughput, (2) latency, (3) processors usage, and (4) memory requirements. It is shown that, for more than 70% of the benchmarks, periodic scheduling results

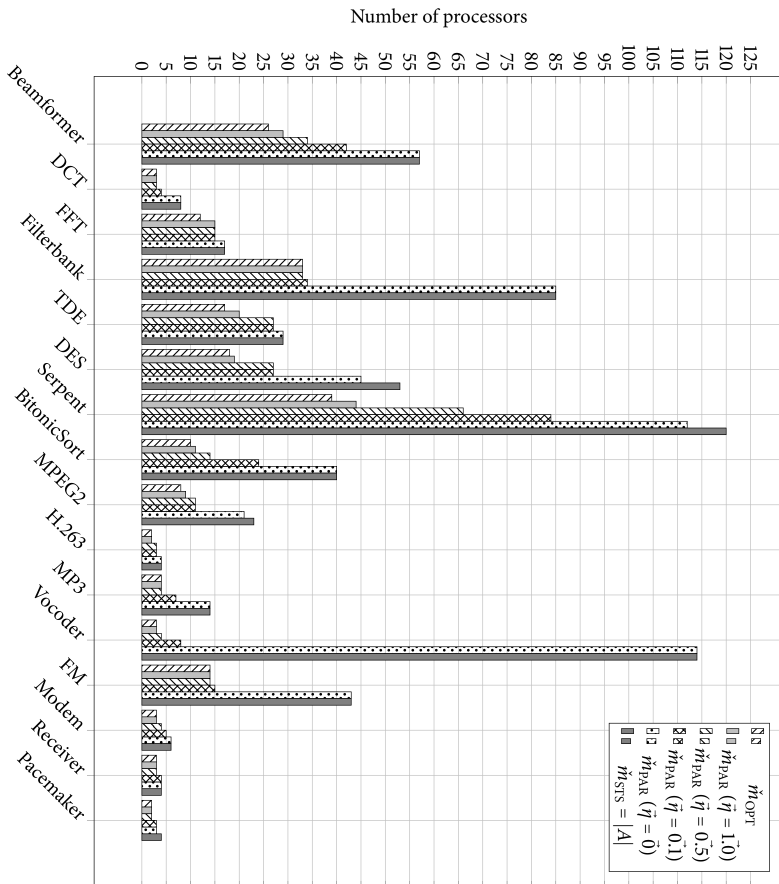


Figure 6.2: *Minimum number of processors required by optimal and partitioned schedulers. We set $\vec{\eta}$ to the value shown in the legend and then we apply Algorithm 2 to compute the deadlines. After that, we apply QPA and FFD to find $\check{m}_{\mathrm{PAR}}$.*

Table 6.5: *The total amount of memory needed to realize the buffers in the communication channels under periodic and self-timed schedules. Fields marked with "N/A" indicate that SDF[3], which is used to compute the self-timed schedule parameters, could not compute a solution within 30 days.*

Program	M_{PS}	t_{PS} (seconds)	M_{STS}	t_{STS} (seconds)	M_{PS}/M_{STS}
Beamformer	366	3.6	366	859	1.0
DCT	2816	3.0	2816	0.15	1.0
FFT	15872	0.42	15872	1.1	1.0
Filterbank	1128	13.72	N/A	-	-
TDE	77280	1.41	77280	2.96	1.0
DES	4420	1.0	3986	6400×60	1.11
Serpent	19241	12.4	N/A	-	-
BitonicSort	175	0.23	175	9.96	1.0
MPEG2	9753	2.1	9393	40.73	1.038
H.263	1257	131.4	1257	81.88	1.0
MP3	22	652	22	0.48	1.0
Vocoder	700	31.7	699	1.55	1.001
FM	63	1.13	63	0.1	1.0
Modem	20	0.1	18	0.02	1.111
Receiver	3475	19431	N/A	-	-
Pacemaker	105	0.2	43	2.03	2.442

in optimal throughput and latency. It is also shown that the memory requirements under a periodic schedule, compared to a self-timed one, are the same or slightly higher (at most +11%) for 75% of the benchmarks. In the cases when periodic scheduling is optimal (for example in terms of throughput and latency), we argue that it provides additional benefits over self-timed scheduling. These benefits are:

1. Ability to modify the set of running programs easily. Under periodic scheduling, one can use efficient schedulability tests and partitioning schemes explained in Chapter 2 to perform online admission control of new programs. In contrast, self-timed scheduling requires either re-performing the design space exploration or using heuristics to devise the new allocation of the programs' tasks.
2. Ability to use a wide variety of scheduling algorithms for real-time periodic tasks. This variety gives the designer extra flexibility in choosing the most suitable algorithm for a certain platform.

6.3 Experiment III: Validating Synthesized Systems

In this experiment, we synthesize a set of MPSoC systems, where each system runs a set of streaming programs. After that, the synthesized systems are validated by instru-

Listing 8 Task implementation with code for detecting deadline misses

```
1  void task(void *arg) {
2    portTickType LastReleaseTime, SimultaneousReleaseTime;
3    portTickType ticks;
4    const portTickType Period = 5;
5    const portTickType StartTime = 20;
6
7    SimultaneousReleaseTime = *((portTickType *) arg);
8    vTaskDelayUntil( &SimultaneousReleaseTime, StartTime );
9    LastReleaseTime = xTaskGetTickCount();
10
11   for (;;) {
12     function();
13     ticks = xTaskGetTickCount();
14     vTaskDelayUntil( &LastReleaseTime, Period );
15     if (ticks > *LastReleaseTime) {
16       taskDISABLE_INTERRUPTS();
17       xil_printf("PANIC! Deadline miss\n");
18       for (;;);
19     }
20   }
21 }
```

menting the generated code to detect deadline misses and buffer underflows/overflows. Deadline misses, for implicit-deadline tasks, can be detected in FreeRTOS using the following observation from Figure 5.6: *"It should be noted that `vTaskDelayUntil()` will return immediately (without blocking) if it is used to specify a wake time that is already in the past"*. Therefore, the task implementation shown in Listing 3 is updated to detect deadline misses which results in Listing 8. We see in Listing 8 that the function invocation is followed by an `if` statement that checks whether `LastReleaseTime` has elapsed or not. If a deadline miss is detected, then the following actions are performed: (1) all interrupts are disabled, (2) a message is printed to the user, and (3) the system freezes its execution by entering into an infinite loop.

This experiment is conducted using the programs outlined in Table 6.6. They include a mixture of real and synthetic programs. The programs shown in Table 6.6 are validated on two types of hardware platforms. These hardware platforms are listed in Table 6.7. When prototyping the systems on ML605 board, the synthesized systems have an architecture as the one shown in Figure 5.4. In contrast, the Zedboard is based on Xilinx Zynq-7000 SoC which is a dual ARM system with largely fixed functionality as shown in Figure 6.3.

The set of synthesized systems is outlined in Table 6.8. The synthesis time includes the time needed to perform all the steps shown in Figure 1.7 except for the WCET

Table 6.6: *Programs used in Experiment III*

Program	Description	No. of tasks
JPEG-E	Image encoder from raw format to JPEG format	6
JPEG-D	Image decoder from JPEG format to raw format	2
Sobel	Sobel edge-detector filter	5
G_1	The program shown in Listing 1	4
G_2	The program shown in Listing 2	4

Table 6.7: *Hardware platforms used in Experiment III*

Platform	Description
ML605	Xilinx ML605 Virtex-6 FPGA board with SoC architecture shown in Figure 5.4
Zedboard	Avnet ZedBoard board with Xilinx Zynq-7000 SoC shown in Figure 6.3

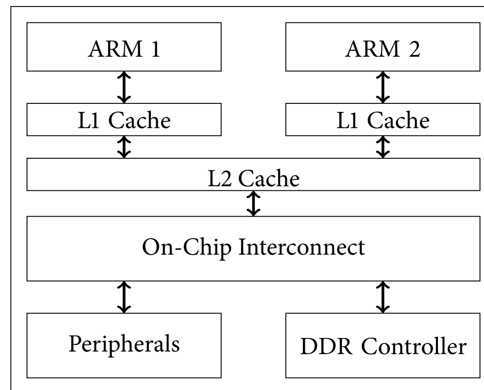


Figure 6.3: *Zynq-7000 SoC architecture*

analysis and low-level synthesis and compilation. The short synthesis times in Table 6.8 demonstrate clearly the speed of the proposed design flow. The throughput constraints correspond to the periods that are requested by the designer from the scheduling framework. This is done by choosing different values of the period scaling factor μ_G. By using μ_G, it is possible to reduce the programs throughput, and accordingly, reduce the required number of processors. This reduction is necessary, for example, on the Zynq platform in order to ensure that the programs can be scheduled on two processors. For the deadline factors $\vec{\eta}$, we always use $\vec{\eta} = \vec{1}$. Each synthesized system is validated by running it with real input data for a duration between 1 and 12 hours. For all the synthesized systems shown in Table 6.8, no deadline misses or buffer underflows/overflows are detected during the whole validation phase.

Table 6.8: *The set of synthesized systems*

System	Board	Programs	Synthesis time (seconds)	Throughput constraints
Sys1	ML605	JPEG-D | Sobel	11.3	1 fps | 4.7 fps
Sys2	Zedboard	JPEG-D | Sobel	8.6	2 fps | 5 fps
Sys3	ML605	G_1 | G_2	8.2	3.125 fps | 3.571 fps

In order to "double-check" the previous finding, we measured also the throughput of the actors in the generated systems. The throughput is measured by measuring the periods of the output actors using custom hardware counters. For example, for Sys3 in Table 6.8, we measured the the periods of the output actors when the system is ran on ML605 board for a time duration equal to 1 hour. The periods that were requested from the scheduling framework are 280 ms for G_2 and 320 ms for G_1 and the OS clock tick was set to 10 ms. The average measured periods were 279915940 ns for G_2 and 319719020 ns for G_1. The deviations from these averages were always less than 1 ms, which is much below the time granularity visible to the RTOS. Thus, these measurements reconfirm the correctness of the systems generated by our proposed design flow.

Chapter 7

Summary and Future Work

Show me a hard real-time system, and I will show you a hammer that will cause it to miss its deadlines.

Paul E. McKenney

THIS dissertation addressed the problem of designing hard real-time streaming systems running a set of parallel streaming programs in an automated way such that the programs provably meet their timing requirements. Such systems are usually realized nowadays as MPSoCs. Model-based design and electronic system-level synthesis have emerged as de facto solutions to the problems of designing parallel software for MPSoCs and generating the complete MPSoC, respectively. However, no such de facto solution exists yet for the problem of scheduling parallel streaming programs on MPSoCs. Scheduling has a direct influence on the architecture and mapping specifications needed to perform electronic system-level synthesis. One possible and attractive solution is to use classical hard real-time scheduling algorithms. However, most hard real-time scheduling algorithms assume independent periodic or sporadic tasks, while modern streaming programs are often modeled as directed graphs in which the actors (i.e., tasks) have data dependency constraints and do not necessarily conform to the real-time periodic task model. In this dissertation, a scheduling framework is proposed with which it is analytically proven that any streaming program, modeled as an acyclic CSDF graph, can be executed as a set of real-time periodic tasks. The proposed framework computes the parameters of the periodic tasks corresponding to the graph actors and the minimum buffer sizes of the communication channels such that a valid periodic schedule is guaranteed to exist. The proposed framework shows that the use of both models is possible and that they complement each other; CSDF captures the functional aspects of the program, while the real-time periodic task model captures the timing aspects. Using both models, as demonstrated by the proposed framework,

enables the designer to: (1) schedule the tasks to meet certain performance metrics (i.e., throughput and latency), (2) derive analytically the scheduling parameters that guarantee the required performance, and (3) compute analytically the minimum number of processors that guarantee the required performance together with the mapping of tasks to processors. Additionally, the scheduling framework (explained in Chapter 4) establishes the following results:

- Matched I/O rates graphs (which correspond to roughly 90% of streaming programs) have a throughput under periodic schedules that is equal to their throughput under worst-case self-timed schedules.
- For certain classes of CSDF graphs, it is possible to achieve throughput and latency under periodic schedules that are equal to the throughput and latency under worst-case self-timed schedules. It is also shown that, for CSDF graphs in general, the latency can be reduced via reducing the deadlines of the actors along the critical paths.

In order to demonstrate the effectiveness and efficiency of the proposed scheduling framework, a system-level design flow that incorporates the scheduling framework is proposed. This design flow accepts, as input, algorithmic sequential specifications of streaming programs, and then applies a set of systematic and fully automated steps that produce, as output, a complete system implementation which provably meets the timing requirements of the programs. The system implementation consists of the parallelized versions of the input streaming programs together with the hardware needed to run them. The proposed design flow consists of the following key steps: (1) automated parallelization and model construction, (2) scheduling framework (as proposed in Chapter 4), and (3) electronic system-level synthesis. A complete implementation of the proposed design flow is available for download, as an open source framework called Daedalus$^{\mathrm{RT}}$, from http://daedalus.liacs.nl/.

The proposed scheduling framework and design flow are evaluated through a set of experiments. The first experiment (Section 6.1) shows that automated parallelization and model construction are very fast for many real life programs. The second experiment (Section 6.2) demonstrates the quality of periodic scheduling of streaming programs in terms of (1) throughput, (2) latency, (3) processors usage, and (4) memory requirements. It shows that, for more than 70% of the benchmarks, periodic scheduling results in optimal throughput and latency. It also shows that the memory requirements under a periodic schedule, compared to a self-timed one, are the same or slightly higher (at most +11%) for 75% of the benchmarks. In the cases where periodic scheduling is optimal, it can be argued that it provides additional benefits over self-timed scheduling. These benefits are:

1. Ability to modify the set of running programs easily. Under periodic scheduling, one can use efficient schedulability tests and partitioning schemes explained

in Chapter 2 to perform online admission control of new programs. In contrast, self-timed scheduling requires either re-performing the design space exploration or using heuristics to devise the new allocation of the programs' tasks.

2. Ability to use a wide variety of scheduling algorithms for real-time periodic tasks. This variety gives the designer extra flexibility in choosing the most suitable algorithm for a certain platform.

Finally, the third experiment (Section 6.3) validated the correctness of the synthesized systems using the proposed design flow. The validation was done by running the synthesized systems on FPGA boards with real input data for long durations. During the whole validation phase, no deadline misses or buffer underflows/overflows were observed.

7.1 Suggestions for Future Work

In this section, we provide a summary of the issues that deserve further investigation in the future.

Support for More Expressive MoCs

A more expressive MoC allows a more accurate analysis of the modeled program. A first step towards this goal is the work in [BTV12]. The authors in [BTV12], as mentioned in Section 1.4, presented a scheduling framework similar to ours with support for a MoC called Affine Data-Flow (ADF), which extends the CSDF model. Another option is to consider support for dynamic MoCs which model programs that change their behavior during run-time (e.g., Boolean Data-Flow [BL93]).

Improving the WCET by Considering the Effect of Mapping

In Chapter 4, we assume that the WCET of an actor is computed assuming the worst-case latency of communication operations. This worst-case latency occurs when the underlying interconnect is fully congested. However, such assumption overestimates the WCET value. In a real system, many communication streams are isolated from the others (see for example Figure 4.15). Therefore, communication operations occur without congestion and they do not take their worst-case latency. Therefore, it is possible to reduce the WCET values if the mapping is taken into account. A first step towards "communication-aware" allocation in hard real-time systems realized on MPSoCs is the work presented in [ZM12]. Zimmer and Mueller in [ZM12] presented a framework for deriving low-contention mapping of real-time programs mapped onto NoC-based MPSoCs. They devised two solvers: one based on exhaustive search and another based

on a heuristic. The resulting mapping tries to reduce the communication contention and, hence, reduce the communication latency. This, in turn, leads to a tighter WCET estimates of the tasks.

Support for Hierarchical Scheduling

The architecture and mapping derivation explained in Section 4.8 does not support hierarchical scheduling. Hierarchical scheduling is becoming more popular in modern hard real-time systems since it allows different programs to be scheduled using different scheduling policies. Furthermore, in some application domains such as avionics, it is mandatory to use two-levels of scheduling in order to provide complete partitioning in time and space as mandated by industry standards (such as ARINC 653 Specification [ARI]). Therefore, it is interesting to investigate how such hierarchical scheduling schemes affect the derivation of the architecture and mapping specifications.

Bibliography

[AB98] L. Abeni and G. Buttazzo. Integrating multimedia applications in hard real-time systems. In *Proceedings of the 19th IEEE Real-Time Systems Symposium*, RTSS '98, pages 4–13, 1998. doi:10.1109/REAL.1998.739726.

[AJ02] Björn Andersson and Jan Jonsson. Preemptive multiprocessor scheduling anomalies. In *Proceedings of the International Parallel and Distributed Processing Symposium*, IPDPS 2002, Los Alamitos, CA, USA, 2002. IEEE Computer Society. doi:10.1109/IPDPS.2002.1015483.

[ALSU86] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, Boston, MA, U.S.A, 2nd edition, 1986.

[Amd67] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the Spring Joint Computer Conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM. doi:10.1145/1465482.1465560.

[ARI] ARINC Incorporated. 653P1-3 Avionics Application Software Standard Interface, Part 1, Required Services. URL: http://www.arinc.com/ [cited September 13, 2013].

[ARM10] ARM Ltd. *AMBA® AXI Protocol - Version: 2.0 : Specification*, 2010. URL: http://www.arm.com/ [cited June 19, 2012].

[Aud91] Neil C. Audsley. Optimal priority assignment and feasibility of static priority tasks with arbitrary start times. Technical Report YCS 164, University of York, 1991.

[Bar03] Sanjoy K. Baruah. Dynamic- and Static-priority Scheduling of Recurring Real-time Tasks. *Real-Time Systems*, 24(1):93–128, 2003. doi:10.1023/A:1021711220939.

[Bar10] Sanjoy Baruah. The Non-cyclic Recurring Real-Time Task Model. In *Proceedings of the IEEE 31st Real-Time Systems Symposium*, RTSS '10, pages 173–182, Los Alamitos, CA, USA, 2010. IEEE Computer Society. doi:10.1109/RTSS.2010.19.

[Bas04] Cédric Bastoul. *Improving Data Locality in Static Control Programs*. PhD thesis, Université Pierre-et-Marie-Curie (Paris VI), France, 2004.

[BCGM99] Sanjoy Baruah, Deji Chen, Sergey Gorinsky, and Aloysius Mok. Generalized Multiframe Tasks. *Real-Time Systems*, 17:5–22, 1999. doi:10.1023/A:1008030427220.

[BCPV96] S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel. Proportionate progress: A notion of fairness in resource allocation. *Algorithmica*, 15(6):600–625, 1996. doi:10.1007/BF01940883.

[BELP96] Greet Bilsen, Marc Engels, Rudy Lauwereins, and Jean Peperstraete. Cyclo-static dataflow. *IEEE Transactions on Signal Processing*, 44(2):397–408, 1996. doi:10.1109/78.485935.

[Ber66] A. J. Bernstein. Analysis of Programs for Parallel Processing. *IEEE Transactions on Electronic Computers*, EC-15(5):757–763, 1966. doi:10.1109/PGEC.1966.264565.

[BG04] Sanjoy Baruah and Joël Goossens. Scheduling Real-Time Tasks: Algorithms and Complexity. In Joseph Y.-T. Leung, editor, *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*. CRC Press, Boca Raton, FL, U.S.A, 2004. doi:10.1201/9780203489802.ch28.

[BHM+05] Marco Bekooij, Rob Hoes, Orlando Moreira, Peter Poplavko, Milan Pastrnak, Bart Mesman, Jan Mol, Sander Stuijk, Valentin Gheorghita, and Jef Meerbergen. Dataflow Analysis for Real-Time Embedded Multiprocessor System Design. In Peter van der Stok, editor, *Dynamic and Robust Streaming in and between Connected Consumer-Electronic Devices*, volume 3, pages 81–108. Springer Netherlands, 2005. doi:10.1007/1-4020-3454-7_4.

[BL93] Joseph T. Buck and Edward A. Lee. Scheduling dynamic dataflow graphs with bounded memory using the token flow model. In *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing*, volume 1 of *ICASSP '93*, pages 429–432, 1993. doi:10.1109/ICASSP.1993.319147.

[BMKdD12] B. Bodin, A. Munier-Kordon, and B.D. de Dinechin. K-Periodic schedules for evaluating the maximum throughput of a Synchronous Dataflow graph. In *Proceedings of the International Conference on Embedded Computer Systems*, SAMOS '12, pages 152–159, 2012. doi:10.1109/SAMOS.2012.6404169.

[BMKdD13] Bruno Bodin, Alix Munier-Kordon, and Benoît Dupont de Dinechin. Periodic Schedules for Cyclo-Static Dataflow. In *Proceedings of the 11th IEEE Symposium on Embedded Systems for Real-Time Multimedia*, ESTIMedia 2013, pages 105–114, 2013. doi:10.1109/ESTIMedia.2013.6704509.

[BNHMMK12] Abir Benabid-Najjar, Claire Hanen, Olivier Marchetti, and Alix Munier-Kordon. Periodic Schedules for Bounded Timed Weighted Event Graphs. *IEEE Transactions on Automatic Control*, 57(5):1222–1232, 2012. doi:10.1109/TAC.2012.2191871.

[Bra11] Björn B. Brandenburg. *Scheduling and Locking in Multiprocessor Real-Time Operating Systems*. PhD thesis, University of North Carolina at Chapel Hill, U.S.A., 2011.

[BRH90] Sanjoy K. Baruah, Louis E. Rosier, and Rodney R. Howell. Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor. *Real-Time Systems*, 2:301–324, 1990. doi:10.1007/BF01995675.

[BS11] Mohamed Bamakhrama and Todor Stefanov. Hard-real-time scheduling of data-dependent tasks in embedded streaming applications. In *Proceedings of the ninth ACM International Conference on Embedded Software*, EMSOFT '11, pages 195–204, New York, NY, USA, 2011. ACM. doi:10.1145/2038642.2038672.

[BS12] Mohamed A. Bamakhrama and Todor Stefanov. Managing latency in embedded streaming applications under hard-real-time scheduling. In *Proceedings of the eighth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS

'12, pages 83–92, New York, NY, USA, 2012. ACM. doi:10.1145/2380445.2380464.

[BT13] Adnan Bouakaz and Jean-Pierre Talpin. Buffer minimization in earliest-deadline first scheduling of dataflow graphs. In *Proceedings of the 14th ACM SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems*, LCTES '13, pages 133–142, New York, NY, USA, 2013. ACM. doi:10.1145/2465554.2465558.

[BTV12] Adnan Bouakaz, Jean-Pierre Talpin, and Jan Vitek. Affine Data-Flow Graphs for the Synthesis of Hard Real-Time Applications. In *Proceedings of the 12th International Conference on Application of Concurrency to System Design*, ACSD '12, pages 183–192, Los Alamitos, CA, USA, 2012. IEEE Computer Society. doi:10.1109/ACSD.2012.16.

[But11] Giorgio C. Buttazzo. *Hard Real-Time Computing Systems*. Springer US, 3rd edition, 2011. doi:10.1007/978-1-4614-0676-1.

[BZNS12] Mohamed A. Bamakhrama, Jiali Teddy Zhai, Hristo Nikolov, and Todor Stefanov. A methodology for automated design of hard-real-time embedded streaming systems. In *Proceedings of the 15th Design, Automation Test in Europe Conference and Exhibition*, DATE 2012, pages 941–946, 2012. doi:10.1109/DATE.2012.6176632.

[Cas13] Jeronimo Castrillon. *Programming Heterogeneous MPSoCs: Tool Flows to Close the Software Productivity Gap*. PhD thesis, Rheinisch-Westfälische Technische Hochschule Aachen, Germany, 2013.

[CFH+04] John Carpenter, Shelby Funk, Philip Holman, Anand Srinivasan, James Anderson, and Sanjoy Baruah. A Categorization of Real-time Multiprocessor Scheduling Problems and Algorithms. In Joseph Y.-T. Leung, editor, *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*. CRC Press, Boca Raton, FL, U.S.A, 2004. doi:10.1201/9780203489802.ch30.

[CGJ96] E. G. Coffman, Jr., M. R. Garey, and D. S. Johnson. Approximation algorithms for bin packing: A survey. In Dorit S. Hochbaum, editor, *Approximation algorithms for NP-hard problems*, pages 46–93. PWS Publishing Co., Boston, MA, USA, 1996.

[CKT03] Samarjit Chakraborty, Simon Künzli, and Lothar Thiele. A general framework for analysing system properties in platform-based embedded system designs. In *Proceedings of the Design, Automation and Test*

in Europe Conference and Exhibition, DATE '03, pages 190–195, 2003. doi:10.1109/DATE.2003.1253607.

[CMQV89] Guy Cohen, Pierre Moller, Jean-Pierre Quadrat, and Michel Viot. Algebraic tools for the performance evaluation of discrete event systems. *Proceedings of the IEEE*, 77(1):39–85, 1989. doi:10.1109/5.21069.

[CRJ10] Hyeonjoong Cho, Binoy Ravindran, and E. Douglas Jensen. T-L plane-based real-time scheduling for homogeneous multiprocessors. *Journal of Parallel and Distributed Computing*, 70(3):225–236, 2010. doi:DOI:10.1016/j.jpdc.2009.12.003.

[DB11] Robert I. Davis and Alan Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM Computing Surveys*, 43(4):35:1–35:44, 2011. doi:10.1145/1978802.1978814.

[DSBS06] Ed F. Deprettere, Todor Stefanov, Shuvra S. Bhattacharyya, and Mainak Sen. Affine Nested Loop Programs and their Binary Parameterized Dataflow Graph Counterparts. In *Proceedings of the International Conference on Application-specific Systems, Architectures and Processors*, ASAP 2006, pages 186–190, 2006. doi:10.1109/ASAP.2006.7.

[EJ09] Christof Ebert and Capers Jones. Embedded Software: Facts, Figures, and Future. *IEEE Computer*, 42(4):42–52, 2009. doi:10.1109/MC.2009.118.

[Fea91] Paul Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20(1):23–53, 1991. doi:10.1007/BF01407931.

[Fea96] Paul Feautrier. Automatic Parallelization in the Polytope Model. In Guy-René Perrin and Alain Darte, editors, *The Data Parallel Programming Model: Foundations, HPF Realization, and Scientific Applications*, volume 1132, pages 79–103. Springer Berlin Heidelberg, 1996. doi:10.1007/3-540-61736-1_44.

[FGB10] Nathan Fisher, Joël Goossens, and Sanjoy Baruah. Optimal online multiprocessor scheduling of sporadic real-time tasks is impossible. *Real-Time Systems*, 45(1-2):26–71, 2010. doi:10.1007/s11241-010-9092-7.

[FKPY07] Elena Fersman, Pavel Krcal, Paul Pettersson, and Wang Yi. Task automata: Schedulability, decidability and undecidability. *Information and Computation*, 205(8):1149–1172, 2007. doi:10.1016/j.ic.2007.01.009.

[FLS+11] Shelby Funk, Greg Levin, Caitlin Sadowski, Ian Pye, and Scott Brandt. DP-Fair: a unifying theory for optimal hard real-time multiprocessor scheduling. *Real-Time Systems*, 47(5):389–429, 2011. doi:10.1007/s11241-011-9130-0.

[Fra10] Björn Franke. C Compilers and Code Optimization for DSPs. In Shuvra S. Bhattacharyya, Ed F. Deprettere, Rainer Leupers, and Jarmo Takala, editors, *Handbook of Signal Processing Systems*, pages 575–601. Springer US, 2010. doi:10.1007/978-1-4419-6345-1_21.

[FSH+13] Shakith Fernando, Firew Siyoum, Yifan He, Akash Kumar, and Henk Corporaal. MAMPSx: A Design Framework for Rapid Synthesis of Predictable Heterogeneous MPSoCs. In *Proceedings of IEEE International Symposium on Rapid System Prototyping*, RSP '13, pages 136–142, 2013. doi:10.1109/RSP.2013.6683970.

[GAC+13] Kees Goossens, Arnaldo Azevedo, Karthik Chandrasekar, Manil Dev Gomony, Sven Goossens, Martijn Koedam, Yonghui Li, Davit Mirzoyan, Anca Molnos, Ashkan Beyranvand Nejad, Andrew Nelson, and Shubhendu Sinha. Virtual Platforms for Mixed-Time-Criticality Applications: The CompSOC Architecture and Design Flow. *ACM SIGBED Review*, 10(3):23–34, 2013. doi:10.1145/2544350.2544353.

[GB04] Marc Geilen and Twan Basten. Reactive process networks. In *Proceedings of the 4th ACM International Conference on Embedded Software*, EMSOFT '04, pages 137–146, New York, NY, USA, 2004. ACM. doi:10.1145/1017753.1017778.

[GDR05] Kees Goossens, John Dielissen, and Andrei Rădulescu. Æthereal Network on Chip: Concepts, Architectures, and Implementations. *IEEE Design and Test of Computers*, 22(5):414–421, 2005. doi:10.1109/MDT.2005.99.

[Gha08] Amir Hossein Ghamarian. *Timing Analysis of Synchronous Data Flow Graphs*. PhD thesis, Technische Universiteit Eindhoven, Netherlands, 2008.

[GHB13] Stefan J. Geuns, Joost P.H.M. Hausmans, and Marco J.G. Bekooij. Automatic dataflow model extraction from modal real-time stream processing applications. In *Proceedings of the 14th ACM SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems*, LCTES '13, pages 143–152, New York, NY, USA, 2013. ACM. doi:10.1145/2465554.2465561.

[GHP+09] Andreas Gerstlauer, Christian Haubelt, Andy D. Pimentel, Todor P. Stefanov, Daniel D. Gajski, and Jürgen Teich. Electronic System-Level Synthesis Methodologies. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 28(10):1517–1530, 2009. doi:10.1109/TCAD.2009.2026356.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. WH Freeman & Co., New York, NY, USA, 1979.

[God98] Steve Goddard. *On the Management of Latency in the Synthesis of Real-Time Signal Processing Systems from Processing Graphs*. PhD thesis, University of North Carolina at Chapel Hill, U.S.A., 1998.

[Gra69] R. L. Graham. Bounds on Multiprocessing Timing Anomalies. *SIAM Journal on Applied Mathematics*, 17(2):416–429, 1969. doi:10.1137/0117039.

[Hen03] Jörg Henkel. Closing the SoC design gap. *IEEE Computer*, 36(9):119–121, 2003. doi:10.1109/MC.2003.1231200.

[HGBH09] Andreas Hansson, Kees Goossens, Marco Bekooij, and Jos Huisken. CoMPSoC: A template for composable and predictable multi-processor system on chips. *ACM Transactions on Design Automation of Electronic Systems*, 14(1):2:1–2:24, 2009. doi:10.1145/1455229.1455231.

[HGWB13] Joost P. H. M. Hausmans, Stefan J. Geuns, Maarten H. Wiggers, and Marco J.G. Bekooij. Two parameter workload characterization for improved dataflow analysis accuracy. In *Proceedings of the IEEE 19th Real-Time and Embedded Technology and Applications Symposium*, RTAS '13, pages 117–126, Los Alamitos, CA, USA, 2013. IEEE Computer Society. doi:10.1109/RTAS.2013.6531085.

[HHBT09] Wolfgang Haid, Kai Huang, Iuliana Bacivarov, and Lothar Thiele. Multiprocessor SoC software design flows. *IEEE Transactions on Signal Processing*, 26(6):64–71, 2009. doi:10.1109/MSP.2009.934111.

[HKL+08] Soonhoi Ha, Sungchan Kim, Choonseung Lee, Youngmin Yi, Seongnam Kwon, and Young-Pyo Joo. PeaCE: A hardware-software codesign environment for multimedia embedded systems. *ACM Transactions on Design Automation of Electronic Systems*, 12(3):24:1–24:25, 2008. doi:10.1145/1255456.1255461.

[HNO97] Lance Hammond, Basem A. Nayfeh, and Kunle Olukotun. A single-chip multiprocessor. *IEEE Computer*, 30(9):79–85, 1997. doi:10.1109/2.612253.

[HO10] Soonhoi Ha and Hyunok Oh. Decidable Signal Processing Dataflow Graphs: Synchronous and Cyclo-Static Dataflow Graphs. In Shuvra S. Bhattacharyya, Ed F. Deprettere, Rainer Leupers, and Jarmo Takala, editors, *Handbook of Signal Processing Systems*, pages 851–874. Springer US, 2010. doi:10.1007/978-1-4419-6345-1_30.

[HWGB13] Joost P. H. M. Hausmans, Maarten H. Wiggers, Stefan J. Geuns, and Marco J. G. Bekooij. Dataflow analysis for multiprocessor systems with non-starvation-free schedulers. In *Proceedings of the 16th International Workshop on Software and Compilers for Embedded Systems*, M-SCOPES '13, pages 13–22, New York, NY, USA, 2013. ACM. doi:10.1145/2463596.2463603.

[Int11] International Technology Roadmap for Semiconductors. 2011 Edition: Executive Summary, 2011. URL: http://www.itrs.net/ [cited August 19, 2013].

[Joh74] David S. Johnson. Fast algorithms for bin packing. *Journal of Computer and System Sciences*, 8(3):272–314, 1974. doi:10.1016/S0022-0000(74)80026-7.

[JP86] M. Joseph and P. Pandya. Finding Response Times in a Real-Time System. *The Computer Journal*, 29(5):390–395, 1986. doi:10.1093/comjnl/29.5.390.

[JS05] A. Jantsch and I. Sander. Models of computation and languages for embedded system design. *IEE Proceedings-Computers and Digital Techniques*, 152(2):114–129, 2005. doi:10.1049/ip-cdt:20045098.

[JSM91] Kevin Jeffay, Donald F. Stanat, and Charles U. Martel. On non-preemptive scheduling of period and sporadic tasks. In *Proceedings of the 12th Real-Time Systems Symposium*, RTSS '91, pages 129–139, 1991. doi:10.1109/REAL.1991.160366.

[JTW05] Ahmed Jerraya, Hannu Tenhunen, and Wayne Wolf. Multiprocessor Systems-on-Chips. *IEEE Computer*, 38(7):36–40, 2005. doi:10.1109/MC.2005.231.

[Kah74] Gilles Kahn. The Semantics of Simple Language for Parallel Programming. In *Proceedings of the IFIP Congress*, pages 471–475. North-Holland Publishing Company, 1974.

[KKLR13] Junsung Kim, Hyoseung Kim, Karthik Lakshmanan, and Ragunathan (Raj) Rajkumar. Parallel scheduling for cyber-physical systems: analysis and case study on a self-driving car. In *Proceedings of the ACM/IEEE 4th International Conference on Cyber-Physical Systems*, ICCPS '13, pages 31–40, New York, NY, USA, 2013. ACM. doi:10.1145/2502524.2502530.

[KM66] Richard M. Karp and Raymond E. Miller. Properties of a Model for Parallel Computations: Determinacy, Termination, Queueing. *SIAM Journal on Applied Mathematics*, 14(6):1390–1411, 1966. doi:10.1137/0114108.

[KM+00] Kurt Keutzer, , Sharad Malik, A. Richard Newton, Jan M. Rabaey, and A. Sangiovanni-Vincentelli. System-level design: orthogonalization of concerns and platform-based design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19(12):1523–1543, 2000. doi:10.1109/43.898830.

[Lee06] Edward A. Lee. The Problem with Threads. *IEEE Computer*, 39(5):33–42, 2006. doi:10.1109/MC.2006.180.

[LH89] Edward Ashford Lee and Soonhoi Ha. Scheduling strategies for multiprocessor real-time DSP. In *Proceedings of the IEEE Global Telecommunications Conference and Exhibition: Communications Technology for the 1990s and Beyond*, volume 2 of *GLOBECOM 1989*, pages 1279–1283, 1989. doi:10.1109/GLOCOM.1989.64160.

[Liu12] Isaac Liu. *Precision Timed Machines*. PhD thesis, University of California, Berkeley, U.S.A., 2012.

[LL73] C. L. Liu and James W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the ACM*, 20(1):46–61, 1973. doi:10.1145/321738.321743.

[LM87] Edward A. Lee and David G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, 1987. doi:10.1109/PROC.1987.13876.

[LN05] E. A. Lee and S. Neuendorffer. Concurrent models of computation for embedded software. *IEE Proceedings-Computers and Digital Techniques*, 152(2):239–250, 2005. doi:10.1049/ip-cdt:20045065.

[LW82] Joseph Y.-T. Leung and Jennifer Whitehead. On the complexity of fixed-priority scheduling of periodic, real-time tasks. *Performance Evaluation*, 2(4):237–250, 1982. doi:10.1016/0166-5316(82)90024-4.

[MB07] Orlando M. Moreira and Marco J. G. Bekooij. Self-Timed Scheduling Analysis for Real-Time Applications. *EURASIP Journal on Advances in Signal Processing*, 2007(1), 2007. doi:10.1155/2007/83710.

[MBvdBvM08] Arno Moonen, Marco Bekooij, René van den Berg, and Jef van Meerbergen. Cache aware mapping of streaming applications on a multiprocessor system-on-chip. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '08, pages 300–305, New York, NY, USA, 2008. ACM. doi:10.1145/1403375.1403448.

[Mei10] Sjoerd Meijer. *Transformations for Polyhedral Process Networks*. PhD thesis, Universiteit Leiden, Netherlands, 2010.

[Moo65] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8):114–117, 1965.

[Mor12] Orlando Moreira. *Temporal Analysis and Scheduling of Hard Real-Time Radios running on a Multi-Processor*. PhD thesis, Technische Universiteit Eindhoven, Netherlands, 2012.

[MRP+11] Mario Morales, Shane Rau, Michael J. Palma, Mali Venkatesan, Flint Pulskamp, and Abhi Dugar. *Worldwide Intelligent Systems 2011–2015 Forecast: The Next Big Opportunity*. International Data Corporation, 5 Speen Street, Framingham, MA 01701, U.S.A., 2011.

[NSD08] Hristo Nikolov, Todor Stefanov, and Ed Deprettere. Systematic and Automated Multiprocessor System Design, Programming, and Implementation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(3):542–555, 2008. doi:10.1109/TCAD.2007.911337.

[NTS+08] H. Nikolov, M. Thompson, T. Stefanov, A. Pimentel, S. Polstra, R. Bose, C. Zissulescu, and E. Deprettere. Daedalus: toward composable multimedia MP-SoC design. In *Proceedings of the 45th annual Design Automation Conference*, DAC '08, pages 574–579, New York, NY, USA, 2008. ACM. doi:10.1145/1391469.1391615.

[NVC10] Vincent Nollet, Diederik Verkest, and Henk Corporaal. A Safari Through the MPSoC Run-Time Management Jungle. *Journal of Signal Processing Systems*, 60:251–268, 2010. doi:10.1007/s11265-008-0305-4.

[OH04] Hyunok Oh and Soonhoi Ha. Fractional Rate Dataflow Model for Efficient Code Synthesis. *The Journal of VLSI Signal Processing*, 37:41–51, 2004. doi:10.1023/B:VLSI.0000017002.91721.0e.

[PC13] Keshab K. Parhi and Yanni Chen. Signal Flow Graphs and Data Flow Graphs. In Shuvra S. Bhattacharyya, Ed F. Deprettere, Rainer Leupers, and Jarmo Takala, editors, *Handbook of Signal Processing Systems*, pages 1277–1302. Springer New York, 2013. doi:10.1007/978-1-4614-6859-2_39.

[PL95] Thomas M. Parks and Edward A. Lee. Non-preemptive real-time scheduling of dataflow systems. In *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, volume 5 of *ICASSP-95*, pages 3235–3238, 1995. doi:10.1109/ICASSP.1995.479574.

[PMN+09] Rodolfo Pellizzoni, Patrick Meredith, Min-Young Nam, Mu Sun, Marco Caccamo, and Lui Sha. Handling mixed-criticality in SoC-based real-time embedded systems. In *Proceedings of the 7th ACM International Conference on Embedded Software*, EMSOFT '09, pages 235–244, New York, NY, USA, 2009. ACM. doi:10.1145/1629335.1629367.

[Pou] Louis-Noël Pouchet. PolyBench/C: the Polyhedral Benchmark suite. URL: http://www.cs.ucla.edu/~pouchet/software/polybench/ [cited July 22, 2013].

[Reaa] Real Time Engineers Ltd. Task Control: `vTaskDelayUntil`. URL: `http://www.freertos.org/vtaskdelayuntil.html` [cited July 29, 2013].

[Reab] Real Time Engineers Ltd. The FreeRTOS Project. URL: `http://www.freertos.org/` [cited June 19, 2012].

[RLM+12] Paul Regnier, George Lima, Ernesto Massa, Greg Levin, and Scott Brandt. Multiprocessor scheduling by reduction to uniprocessor: an original optimal approach. *Real-Time Systems*, pages 1–39, 2012. `doi:10.1007/s11241-012-9165-x`.

[SB09] Sundararajan Sriram and Shuvra S. Bhattacharyya. *Embedded Multiprocessors: Scheduling and Synchronization*. CRC Press, Boca Raton, FL, USA, 2nd edition, 2009.

[SBW09] Marcel Steine, Marco Bekooij, and Maarten Wiggers. A Priority-Based Budget Scheduler with Conservative Dataflow Model. In *Proceedings of the 12th Euromicro Conference on Digital System Design, Architectures, Methods and Tools*, DSD '09, pages 37–44, 2009. `doi:10.1109/DSD.2009.148`.

[Sch09] Martin Schoeberl. Time-predictable computer architecture. *EURASIP Journal on Embedded Systems*, 2009:2:1–2:17, 2009. `doi:10.1155/2009/758480`.

[SEGY11a] Martin Stigge, Pontus Ekberg, Nan Guan, and Wang Yi. On the Tractability of Digraph-Based Task Models. In *Proceedings of the 23rd Euromicro Conference on Real-Time Systems*, ECRTS '11, pages 162–171, Los Alamitos, CA, USA, 2011. IEEE Computer Society. `doi:10.1109/ECRTS.2011.23`.

[SEGY11b] Martin Stigge, Pontus Ekberg, Nan Guan, and Wang Yi. The Digraph Real-Time Task Model. In *Proceedings of the 17th IEEE Real-Time and Embedded Technology and Applications Symposium*, RTAS '11, pages 71–80, Los Alamitos, CA, USA, 2011. IEEE Computer Society. `doi:10.1109/RTAS.2011.15`.

[SG13] IEEE Computer Society and The Open Group. Standard for Information Technology Portable Operating System Interface (POSIX®): Base Specifications, Issue 7. *IEEE Std 1003.1, 2013 Edition*, pages 1–3906, 2013. `doi:10.1109/IEEESTD.2013.6506091`.

[SGB06] Sander Stuijk, Marc Geilen, and Twan Basten. SDF^3: SDF For Free. In *Proceedings of the 10th International Conference on Application of Concurrency to System Design*, ACSD '06, pages 276–278, Los Alamitos, CA, USA, 2006. IEEE Computer Society. doi:10.1109/ACSD.2006.23.

[SKS+10] A. Shabbir, A. Kumar, S. Stuijk, B. Mesman, and H. Corporaal. CA-MPSoC: An automated design flow for predictable multi-processor architectures for multiple applications. *Journal of Systems Architecture*, 56(7):265–277, 2010. doi:10.1016/j.sysarc.2010.03.007.

[TA10] William Thies and Saman Amarasinghe. An empirical characterization of stream programs and its implications for language and compiler design. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, pages 365–376, New York, NY, USA, 2010. ACM. doi:10.1145/1854273.1854319.

[TBHH07] Lothar Thiele, Iuliana Bacivarov, Wolfgang Haid, and Kai Huang. Mapping Applications to Tiled Multiprocessor Embedded Systems. In *Proceedings of the Seventh International Conference on Application of Concurrency to System Design*, ACSD '07, pages 29–40, 2007. doi:10.1109/ACSD.2007.53.

[Tei12] Jürgen Teich. Hardware/Software Codesign: The Past, the Present, and Predicting the Future. *Proceedings of the IEEE*, 100(Special Centennial Issue):1411–1430, 2012. doi:10.1109/JPROC.2011.2182009.

[Thr10] Sebastian Thrun. Toward robotic cars. *Communications of the ACM*, 53(4):99–106, 2010. doi:10.1145/1721654.1721679.

[TKM+02] Michael Bedford Taylor, Jason Kim, Jason Miller, David Wentzlaff, Fae Ghodrat, Ben Greenwald, Henry Hoffman, Paul Johnson, Jae-Wook Lee, Walter Lee, Albert Ma, Arvind Saraf, Mark Seneski, Nathan Shnidman, Volker Strumpen, Matt Frank, Saman Amarasinghe, and Anant Agarwal. The Raw microprocessor: a computational fabric for software circuits and general-purpose programs. *IEEE Micro*, 22(2):25–35, 2002. doi:10.1109/MM.2002.997877.

[TNS+07] Mark Thompson, Hristo Nikolov, Todor Stefanov, Andy D. Pimentel, Cagkan Erbas, Simon Polstra, and Ed F. Deprettere. A framework

for rapid system-level exploration, synthesis, and programming of multimedia MP-SoCs. In *Proceedings of the 5th IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS '07, pages 9–14, New York, NY, USA, 2007. ACM. doi:10.1145/1289816.1289823.

[TS09] Lothar Thiele and Nikolay Stoimenov. Modular performance analysis of cyclic dataflow graphs. In *Proceedings of the seventh ACM International Conference on Embedded Software*, EMSOFT '09, pages 127–136, New York, NY, USA, 2009. ACM. doi:10.1145/1629335.1629353.

[UCS+10] Theo Ungerer, Francisco J. Cazorla, Pascal Sainrat, Guillem Bernat, Zlatko Petrov, Hugues Cassé, Christine Rochange, Eduardo Quiãones, Sascha Uhrig, Mike Gerdes, Irakli Guliashvili, Michael Houston, Florian Kluge, Stefan Metzlaff, Jörg Mische, Marco Paolieri, and Julian Wolf. Merasa: Multicore Execution of Hard Real-Time Applications Supporting Analyzability. *IEEE Micro*, 30(5):66–75, 2010. doi:10.1109/MM.2010.78.

[Ukk95] Esko Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14:249–260, 1995. doi:10.1007/BF01206331.

[VAvGL01] Wim F. J. Verhaegh, Emile H. L. Aarts, Paul C. N. van Gorp, and Paul E. R. Lippens. A two-stage solution approach to multidimensional periodic scheduling. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 20(10):1185–1199, 2001. doi:10.1109/43.952736.

[VLA+96] W.F.J. Verhaegh, P.E.R. Lippens, E.H.L. Aarts, J.L. Meerbergen, and A. Werf. Multidimensional periodic scheduling model and complexity. In Luc Bougé, Pierre Fraigniaud, Anne Mignotte, and Yves Robert, editors, *Euro-Par'96 Parallel Processing*, volume 1124 of *Lecture Notes in Computer Science*, pages 226–235. Springer Berlin Heidelberg, 1996. doi:10.1007/BFb0024706.

[VNS07] Sven Verdoolaege, Hristo Nikolov, and Todor Stefanov. pn: a tool for improved derivation of process networks. *EURASIP Journal on Embedded Systems*, 2007(1):19–19, 2007. doi:10.1155/2007/75947.

[WBS09] Maarten H. Wiggers, Marco J.G. Bekooij, and Gerard J.M. Smit. Monotonicity and run-time scheduling. In *Proceedings of the seventh*

ACM International Conference on Embedded Software, EMSOFT '09, pages 177–186, New York, NY, USA, 2009. ACM. doi:10.1145/1629335.1629359.

[WEE+08] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem–overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems*, 7(3):36:1–36:53, 2008. doi:10.1145/1347375.1347389.

[WGR+09] Reinhard Wilhelm, Daniel Grund, Jan Reineke, Marc Schlickling, Markus Pister, and Christian Ferdinand. Memory Hierarchies, Pipelines, and Buses for Future Architectures in Time-Critical Embedded Systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 28(7):966–978, 2009. doi:10.1109/TCAD.2009.2013287.

[Woo] Victoria Woollaston. New images show how Google's self-driving cars see the world. URL: http://www.dailymail.co.uk/sciencetech/article-2317594/ [cited September 9, 2013].

[Xil11] Xilinx, Inc. *Platform Specification Format Reference Manual: Embedded Development Kit (EDK) 13.2*, July 2011. URL: http://www.xilinx.com/ [cited June 19, 2012].

[Yue91] Minyi Yue. A simple proof of the inequality $\mathrm{FFD}(L) \leq 11/9\,\mathrm{OPT}(L) + 1, \forall L$ for the FFD bin-packing algorithm. *Acta Mathematicae Applicatae Sinica*, 7:321–331, 1991. doi:10.1007/BF02009683.

[ZB09] Fengxiang Zhang and Alan Burns. Schedulability Analysis for Real-Time Systems with EDF Scheduling. *IEEE Transactions on Computers*, 58(9):1250–1258, 2009. doi:10.1109/TC.2009.58.

[ZBS13] Jiali Teddy Zhai, Mohamed A. Bamakhrama, and Todor Stefanov. Exploiting just-enough parallelism when mapping streaming applications in hard real-time systems. In *Proceedings of the 50th Annual Design Automation Conference*, DAC '13, pages 170:1–170:8, New York, NY, USA, 2013. ACM. doi:10.1145/2463209.2488944.

[ZM12] Christopher Zimmer and Frank Mueller. Low Contention Mapping of Real-Time Tasks onto TilePro 64 Core Processors. In *Proceedings of the IEEE 19th Real-Time and Embedded Technology and Applications Symposium*, RTAS '12, pages 131–140, Los Alamitos, CA, USA, 2012. IEEE Computer Society. doi:10.1109/RTAS.2012.36.

[ZNS13] Jiali Teddy Zhai, Hristo Nikolov, and Todor Stefanov. Mapping of streaming applications considering alternative application specifications. *ACM Transactions on Embedded Computing Systems*, 12(1s):34:1–34:21, 2013. doi:10.1145/2435227.2435230.

[ZUE00] Dirk Ziegenbein, Jan Uerpmann, and Rolf Ernst. Dynamic response time optimization for SDF graphs. In *Proceedings of the 2000 IEEE/ACM International Conference on Computer-Aided Design*, ICCAD '00, pages 135–141, Piscataway, NJ, USA, 2000. IEEE Press. doi:10.1109/ICCAD.2000.896463.

Curriculum Vitae

Mohamed A. Bamakhrama was born on July 11, 1983 in Dubai, United Arab Emirates. In 2001, he obtained his high school diploma from Muaath Bin Jabal secondary school in Sharjah, United Arab Emirates and was ranked among the top 10 students in United Arab Emirates. In 2005, he obtained a B.Sc. (honors) in Computer Engineering from University of Sharjah and graduated with the highest GPA in the whole university. In 2007, he obtained a M.Sc. (honors) from the Institute of Informatics in the Technical University of Munich in Germany. From May 2008 till September 2009, he worked as a research scientist at the Research department of NXP Semiconductors in Eindhoven, Netherlands. Between October 2009 and October 2013, he has been working, towards his Ph.D. degree, as a research assistant at Leiden University in Leiden, Netherlands. Since October 2013, he is working as a postdoctoral researcher at the same university. His research interests include real-time embedded systems design and programming, hardware/software co-design, computer architecture, computer communication and protocols design.

List of Publications

The work described in this dissertation has resulted in the following publications:

Journal Articles

- **Mohamed A. Bamakhrama** and Todor P. Stefanov. On the hard-real-time scheduling of embedded streaming applications. *Design Automation for Embedded Systems*, 2012. `doi: 10.1007/s10617-012-9086-x`.

Referred, Peer-Reviewed Conference Proceedings

- Emanuele Cannella, **Mohamed A. Bamakhrama**, and Todor Stefanov, "System-level Scheduling of Real-time Streaming Applications using a Semi-partitioned Approach," Accepted for publication in *Proceedings of the 17th Design, Automation, and Test in Europe Conference and Exhibition*, DATE '14, 2014.
- Jiali Teddy Zhai, **Mohamed A. Bamakhrama**, and Todor Stefanov, "Exploiting just-enough parallelism when mapping streaming applications in hard real-time systems," in *Proceedings of the 50th Annual Design Automation Conference*, DAC '13, (New York, NY, USA), pp. 170:1–170:8, ACM, 2013. `doi: 10.1145/2463209.2488944`. **Winner of HiPEAC 2013 Paper Award.**
- **Mohamed A. Bamakhrama** and Todor Stefanov, "Managing latency in embedded streaming applications under hard-real-time scheduling," in *Proceedings of the eighth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS '12, (New York, NY, USA), pp. 83–92, ACM, 2012. `doi: 10.1145/2380445.2380464`.
- **Mohamed A. Bamakhrama**, Jiali Teddy Zhai, Hristo Nikolov, and Todor Stefanov, "A methodology for automated design of hard-real-time embedded streaming systems," in *Proceedings of the 15th Design, Automation, and Test in Europe conference*, DATE '12, pp. 941–946, 2012. `doi: 10.1109/DATE.2012.6176632`.
- **Mohamed Bamakhrama** and Todor Stefanov, "Hard-real-time scheduling of data-dependent tasks in embedded streaming applications," in *Proceedings of the ninth ACM International Conference on Embedded Software*, EMSOFT '11, (New York, NY, USA), pp. 195–204, ACM, 2011. `doi: 10.1145/2038642.2038672`. **Best Paper Candidate**.

Samenvatting

Dit proefschrift onderbouwt modellen en methoden voor het (semi-) automatisch ontwerpen van ingebedde rekensystemen die *één of meer* datastromen op een zodanige manier verwerken dat zij aantoonbaar aan hun tijdsrestricties voldoen. De rekensystemen die in het proefschrift de voorkeur genieten zijn de zogenoemde meervoudige processoren op een enkele silicon drager, *MPSoCs* in vakjargon. Het ontwerpen van software en hardware die op een natuurlijke manier gelijktijdigheid tot uitdrukking brengt is gebleken successvol te zijn uitgaande van *abstracte modellen en systeem-niveau synthese methoden* Gelijktijdigheid in software vergt een tijdsordening in parallele hardware. Tijdsordeningsmethoden voor de implementatie van multi-datastroom toepassingen met harde tijdsrestricties zijn schaars en niet echt bevredigend. De klassieke tijdsordeningsalgorithmes veronderstellen onafhankelijke periodieke taken en zijn dus niet van toepassing op datastroom applicaties die worden gemodelleerd als grafen waarin de taken onderling afhankelijk zijn afhankelijk zijn.

Dit proefschrift geeft een tijdsordening van taken (of actoren) in niet-cyclische datastroom grafen die bewezen in tijd geordend kunnen worden als een verzameling van onafhankelijke periodieke taken. Daarmee wordt het domein van dataflow grafen toehankelijk voor de klassieke literatuur betreffende de ordening van onafhankelijke periodieke taken die aan strikte tijdscondities moeten voldoen. De niet-cyclische datastroom grafen die in dit proefschrift aan bod komen zijn de zogenoemde *synchrone dataflow grafen* (SDF), en *cyclische synchrone dataflow grafen* (CSDF). Het woord cyclisch hier slaat niet op de grafen maar op de evaluatiecyclus van de taken.

Het in dit proefschrift voorgestelde raamwerk berekent parameters van de periodieke taken die overeenstemmen met de functionele graafactoren, evanals de minimum capaciteit van de buffers op de intertaak communicatie kanalen, zodanig dat het bestaan van een valide periodieke tijdsordening is gegarandeerd. De doorbraak hier is dat het gebruik van twee modellen - het datastroom graaf model en het aan tijdsrestricties ondergevige periodieke taak model niet alleen mogelijk is maar ook complementair zijn: De SDF en CSDF grafen modelleren het functioneel gedrag van van de toepaasing, terwijl het periodieke taak model met tijdsrestricites het tijdsgedrag modelleert. Met behulp van deze twee modellen kan een ontwerper aan de slag. Zij kan de taken in tijd ordenen zodanig dat prestatie maten (opstarttijd, en data doorvoor snelheid) in acht worden genomen. Zij kan tijdsordening parameters die de prestaties garanderen analytisch afleiden. En zij kan het minimum aantal processoren en de minimale capaciteit van communicatiekanalen die nodig zijn om aan de prestatievoorwaarden te voldoen analytisch bepalen. Tenslotte kan zij de toekening van taken aan processoren eenduidig bepalen.

De voorgestelde modellen en methoden leiden bovendien tot de volgende resultaten. De zogenoemde *matched I/O rates graphs* die ongeveer 80% van de datastroom

grafen omvatten, hebben een optimale doorstroomtijd onder een periodieke tijdsordening. Voor een zekere verzameling van CSDF grafen kan een optimale opstarttijd en doorstroomtijd bereikt worden onder een periodieke tijdsordening. Verder kan de opstarttijd van CDSF grafen in het algemeen worden gereduceerd door de tijdlimiet van de actoren in het critische pad te verkleinen.

Voor zover een gedegen theoretische onderbouwing nog vraagt om een experimentele bevestiging, is toch een systeem-niveau ontwerp methodologie opgezet waarin de tijdsordeningsmodellen en -methoden die in dit proefschrift zijn ontwikkeld zijn opgenomen. Het ontwerpschema gaat uit van een sequentiele specificatie (of programma) van een datastroom toepassing. Na een serie van systematische en automatische transformaties wordt een parallel implementatie aangeboden die onvoorwaardelijk aan de gestelde tijdsrestricties voldoet. De implementatie is een afbeelding van een niet-sequentiele versie van het oorspronkelijk sequebtiele programma op een multi-processor executie platform op een enkele silicon drager. De belangrijste transformaties zijn: Automatische afleiding van een niet-sequentiele variant van het ingangsprogramma en van het bijbehorende model; de tijdsordeningsmethodologie zoals hierboven beschreven; een systeem-niveau synthese. De gehele procedure en bijbehorende software implementatie is beschikbaar in het publieke domein. Zie `http://daedalus.liacs.nl/`

Tenslotte biedt het proefschrift een aantal validatie experimenten die, met dank aan het publiek beschikbare raamwerk, door iedereen kan worden overgedaan. Wat blijkt? De eerste van de genoemde transformaties vergt weinig tijd, althans voor vrij veel zinvolle programmas. Dit wordt bevestigd door experimenten. Andere experimenten bevestigen de quaiteit van periodieke tijdsordeningen van datastroom toepassingen, in termen van doorstroomsnelheid, opstarttijd, processor activiteit, and geheugen vereisten. Voor meer dan 70% van de doorgerekende toepassingen blijkt dat periodieke tijdsordening leidt tot een optimale opstarttijd en doorstroomtijd. Ook wanneer een periodieke tijdsordening even goed is (in termen van opstarttijd en doorvoertijd) dan vrije tijdsordening, kan worden beargumenteerd dat periodieke tijdsordening voordelen heeft boven vrije tijdsordening. Die voordelen zijn: De mogelijkheid om de verzameling van actieve programmas eenvoudig te wijzigen, en de mogelijkheid om een veelheid van tijdsordeningsalgorithmen voor reele-tijd periodieke taken aan te wenden. De gesynthetiseerde datastroom toepassingen zijn tenslotte beoordeeld op hun werkelijke prestaties vergeleken met hun verwachte prestaties. Daartoe zijn de gesynthetiseerde implementaties geexecuteerd op FPGA hardware versies. Geen van alle experimentele voorbeelden vertoonde een tijdslimiet overschrijding, noch een foutmelding met betrekking tot de berekende capaciteit van communicatie buffers.

Acknowledgments

First and foremost, all the thanks and praises are to God, the lord of the worlds, for all what he gave to me. Over the past four years, many people and organizations have contributed to the successful completion of this dissertation. On the scientific and professional level, I would like to thank the following researchers for their support and collaboration: Bill Thies from Microsoft Research, Sander Stuijk from Technische Universiteit Eindhoven, Orlando Moreira from Ericsson, and Adnan Bouakaz from Université de Rennes 1. I would like also to thank Xilinx, Inc. for their generous donations, in the form of hardware development boards and software licenses, which enabled conducting my PhD research. Another thank you goes to the European CATRENE program, which funded my research partially through its Tera-Scale Multicore Architectures (TSAR) project. I would like to thank all my current and ex-colleagues at Leiden Embedded Research Center (LERC) who formed a true team. I enjoyed every single moment of my work with you. A very BIG thank you goes to Jiali Teddy Zhai, Hristo Nikolov, Sven van Haastregt, Emanuele Cannella, Mohammed Al-Hissi, Di Liu, Jelena Spasic, Sjoerd Meijer, and Dmitry Nadezhkin.

On the social level, I made several friends during my stay in the Netherlands who were truly helpful and supportive and gave me the feeling of being at home. They all deserve my thanks and gratitude. From Leiden, I would like to thank the following friends and their families: Moosa Elayah, Saleh (Samir) Naser Al-Deen, Taleb Alkurdi, Bilal Karasneh, Abdeljalil El Boujadayni, Khaled Younes, Mohamed Thabit, Mohamed Ghaly, Attiya Abdelbaki, Umar Ryad, Hafiz Osman, and Mohamd Al-Sulami. From Eindhoven, I would like to thank the following friends and their families: Hishem Ali, Nabil Eissa, Mohammed Ezz, Ammar Osaiweran, Younes El-Waffaoui, Mohamed Azimane, and Sabri Boughorbel.

On the family level, I would like to start by expressing my thanks and gratitude to my whole family, and in particular, my mother, my sister Khadeja, my brothers Abdulrahman and Khalid, and my nephew Ahmed. Thank you all for your continuous support, prayers, and encouragement. Special thanks go to my kids, Ahmed and Khadeja, who always drew smile on my face and brought hope to me. A special thank you goes to my wife's family, especially my parents-in-law, for their understanding, encouragement, and patience during the last four years. I would like to conclude this acknowledgement with its "dessert". The biggest thank you goes to my other half, Bushra, who shared the PhD journey with me from Day 1 and was instrumental to the completion of this work. She was always there when I needed her and always encouraged me to continue the PhD journey. This dissertation would have not been possible without her sacrifices, patience, and continuous support. I can not express my gratitude in words, so I will just say: May God reward you Bushra for all what you did!