Improved hard real-time scheduling and transformations for embedded Streaming Applications

Spasic, J.

Improved Hard Real-Time Scheduling and Transformations for Embedded Streaming Applications

Jelena Spasić

Improved Hard Real-Time Scheduling and Transformations for Embedded Streaming Applications

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 dinsdag 14 november 2017
klokke 13:45 uur

door

Jelena Spasić
geboren te Trgovište, Servië
in 1984

Promotor:	Prof. dr. Joost N. Kok	Universiteit Leiden
Co-Promotor:	Dr. Todor P. Stefanov	Universiteit Leiden
Promotion Committee:	Prof. dr. Alix Munier Kordon	Université de Paris - LIP6
	Prof. dr. Petru Eles	Linköpings Universitet
	Dr. Andy Pimentel	Universiteit van Amsterdam
	Prof. dr. Aske Plaat	Universiteit Leiden
	Prof. dr. Jaap van den Herik	Universiteit Leiden
	Prof. dr. Harry Wijshoff	Universiteit Leiden

Improved Hard Real-Time Scheduling and Transformations
for Embedded Streaming Applications
Jelena Spasić. -
Dissertation Universiteit Leiden. - With ref. - With summary in Dutch.

Cover designed by Miloš Ačanski.

This dissertation was typeset using LaTeX.

ISBN 978-94-6299-783-7
Printed by Ridderprint, Ridderkerk, The Netherlands.

Mojoj porodici
To my family

Contents

List of Figures

List of Tables

List of Abbreviations

ADF	Affine Dataflow
BF	Best-Fit
BFD	Best-Fit Decreasing
CDP	Constrained-Deadline Periodic
CPU	Central Processing Unit
CSDF	Cyclo-Static Dataflow
DCT	Discrete Cosine Transform
DLP	Data-Level Parallelism
DM	Deadline Monotonic
DPEM	Data Parallel Energy Minimization
DSE	Design Space Exploration
EDF	Earliest Deadline First
EE	Energy Efficient
ESL	Electronic System-Level
FF	First-Fit
FFD	First-Fit Decreasing
FFID	First-Fit Increasing Deadlines
FIFO	First-In First-Out

FPGA	Field-Programmable Gate Array
GPU	Graphics Processing Unit
HSDF	Homogeneous SDF
ICP	Integer Convex Programming
IDP	Implicit-Deadline Periodic
ILP	Integer Linear Programming
ISA	Instruction-Set Architecture
ISPS	Improved Strictly Periodic Scheduling
ISS	Instruction Set Simulators
ITRS	International Technology Roadmap for Semiconductors
KPN	Kahn Process Network
LLF	Least Laxity First
LP	Linear Programming
LTE	Long-Term Evolution
MIDCP	Mixed Integer Disciplined Convex Programming
MJPEG	Motion JPEG
MoC	Model of Computation
MPSoC	Multi-Processor System-on-Chip
NoC	Network-on-Chip
NP	Non-deterministic Polynomial-time
PE	Performance Efficient
PLP	Pipeline-Level Parallelism
PM	Power Management
PPN	Polyhedral Process Network

PS	Periodic Scheduling
RM	Rate Monotonic
RSD	Reed Solomon Decoder
RTA	Response Time Analysis
RTL	Register-Transfer-Level
SDF	Synchronous Data Flow
SPS	Strictly Periodic Scheduling
STS	Self-timed Scheduling
TLP	Task-Level Parallelism
VFS	Voltage-Frequency Scaling
VLE	Variable Length Encoder
WCET	Worst-Case Execution Time
WF	Worst-Fit
WFD	Worst-Fit Decreasing

Chapter 1

Introduction

IN the modern-day world, electronics is not only a tool for survival but an integral part of almost every aspect of human lives. Everything from our home appliances, cars, tablets to our cell-phones uses electronics or electronic components in some way. Constantly improving, the electronics technology is making life faster, easier and more convenient for people. Modern electronics technology is rapidly changing the way people communicate and transmit data and information. Thus, it is possible and common today to execute work related tasks remotely. Health-care systems have also benefited a lot from electronics technology. There, electronics technology is helping doctors accurately diagnose and treat illnesses in a timely manner. For example, in Philips Healthcare, live image guided intervention has been used in treatment of structural heart diseases. Using electronics in home automation received popularity in the past decades. People have the capability to control almost everything in their "smart homes" from heating, air conditioning, and lighting, to kitchen appliances and security systems.

Even though electronics technology has been used in all of the above cases, in each case it has its dedicated purpose within a larger system it has been embedded into, hence the name "embedded electronics". Embedded electronics, that is, **embedded systems**, are tightly coupled to the environment in which they operate. They collect information about the environment through sensors and control that environment through actuators, hence embedded systems must provide real-time guarantees, that is, a correct on-time output [Mar06]. Given that embedded systems are dedicated towards a certain application, they are designed to implement well-defined set of functionalities. In addition, having that many embedded systems are battery-operated they have to be efficient in terms of energy consumption and resource usage.

An important class of embedded systems are **embedded streaming systems**. Embedded streaming systems process a long, potentially infinite, stream of input data coming from the environment. Each data item is processed for a limited time. The processing operations on different data items are self-contained and there is little control flow between the operations. The result of the processing is a long, potentially infinite, stream of output data fed into the environment. Usually, streaming applications must process a large amount of data within short periods of time. Thus, efficiency, in terms of both *throughput* and *latency*, is of primary concern in the design of embedded streaming systems. The throughput represents the rate at which output data items are produced, while the latency represents the time interval between the arrival of a data item to the application input and the production of the corresponding data item at the application output. Examples of streaming applications include audio beamforming, video encoding and decoding, image and signal processing, network protocol processing, navigation, computer vision, and others.

One of the key properties of embedded streaming systems is that their correct functionality depends not only on the correct result but also on the time at which the result is produced. Such systems, where the timing is critical to the correct functionality, are called **real-time systems**. Real-time systems can be classified into **hard** and **soft** systems. A hard real-time system is one where not meeting the timing requirements leads to a system failure, which, in life-critical systems, may have catastrophic consequences. In contrast, in soft real-time systems, not meeting the timing requirements does not lead to a failure but to degraded system performance that can be tolerated given that the timing miss rates are below a certain threshold. Classifying a system into hard or soft real-time depends usually on the overall system requirements and the environment where the system is deployed.

As examples of real-time embedded streaming systems, today we have increasing number of various autonomous mobile systems that need to interact and respond to their dynamic environment extremely fast. These include very complex systems such as self-driving cars and planes, but also modern "toys" such as drones. In recent years, drones have been used extensively as data collectors in many areas. For example, drones have been used in law enforcement for surveillance, tracking and rescue operations. They have been used for monitoring purposes in agriculture and farming, archaeological and land surveying, for delivery purposes in healthcare, crowd monitoring and control, and other cases of monitoring and control. They can carry various types of equipment including live-feed video cameras, infrared cameras, in-

ertial, position and heat sensors. The large amount of data which should be collected and (pre-)processed, the battery-powered operation nature, and the need to react in a short time create demand for designing a high-performance energy-efficient real-time embedded streaming systems. In the next section, we discuss the current trends in designing such systems.

1.1 Trends in the Design of Embedded Streaming Systems

As introduced earlier, there is a demand in modern embedded streaming systems for high performance, in terms of high application throughput and short application latency, and a demand for real-time and energy-efficient execution. In addition, the complexity of applications running on embedded platforms increases [EJ09]. Therefore, we discuss below the trends in designing such complex embedded streaming systems to meet all the demands.

1.1.1 Platform Trend: Multi-Processor System-on-Chip (MPSoC)

Following the trend in general purpose systems, embedded streaming systems designers have relied for a long time on improvement of the computational power of uniprocessors to meet the high-performance requirements of streaming applications. The improvements of the computational power were driven by the increase in the clock frequency, advances in the semiconductor technology, that is, technology scaling, and innovations in the architecture (pipelining, out-of-order execution, branch prediction, and others.) [HP06]. However, the monotonically increasing performance curve with the successive generations of uniprocessors flattened in the early 2000s [PDG06]. The reasons for the curve flattening were increased dynamic power consumption and design complexity with the frequency increase and architecture innovations as well as increased static power and power density with the technology scaling [PDG06]. To increase system performance further, such that high-performance requirements of running applications are met, designers went for **multi-processor platforms** as the natural next evolutionary step in staying on the increasing performance curve [HP06]. By using multiple processors, the issue of increased power consumption is partially addressed by lower operating voltage and frequency, thereby decreasing the power consumption while maintaining high system performance through parallel execution. Moreover, nowadays, embedded systems designers integrate multiple processors, memories, interconnections, and peripherals into a **Multi-Processor System-on-Chip (MPSoC)** [JTW05].

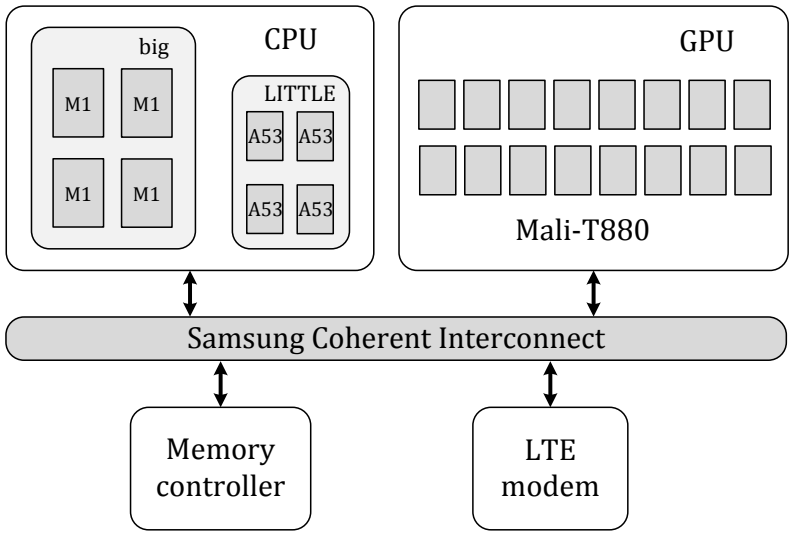


Figure 1.1: *An MPSoC platform example.*

Usually, an MPSoC contains different kinds of processors dedicated to certain functionalities: Central Processing Unit (CPU) for general purpose processing, Graphics Processing Unit (GPU) for graphical processing, a dedicated processor for wireless communication, and others. The processors communicate with each other through an on-chip communication infrastructure. To enable efficient communication, designers proposed and developed high-performance buses, such as ARM AMBA communication infrastructures [ARM], and Network-on-Chip (NoC) [BDM02] infrastructures, such as Xpipes [BB04] and Æthereal [GDR05]. Figure 1.1 gives an example of an MPSoC, the Exynos 8 Octa 8890 [Sama], which can be found in the Samsung S7 mobile phones. The Exynos 8 Octa has eight CPUs in a big.LITTLE architecture [Gre11]. That is, by integrating CPUs with different power-performance characteristics, namely, performance-efficient Exynos M1 cores (big cores) and energy-efficient Cortex A53 cores (LITTLE cores), this MPSoC provides more than 30% improvement in performance and 10% improvement in power efficiency compared to its predecessor [Sama]. The MPSoC also contains a 16-core GPU for 2D/3D graphical processing. The on-chip LTE modem is used for high-speed wireless data communication. All the processors are connected through a high-performance cache coherent interconnect. In this thesis, we consider such type of MPSoC platforms, efficiently utilizing their CPU part by mapping streaming applications on the CPUs.

1.1.2 Design Trend: Model-based Design Methodology

Driven by constant improvements in the semiconductor technology, MPSoC platforms integrate more and more processing elements on a chip. On the other hand, the complexity of embedded software also increases [EJ09]. In order to design such a complex embedded streaming system in an efficient manner in terms of system quality, design effort and time, designers had to raise the level of design abstraction from *Register-Transfer-Level (RTL)* to *system-level* [KMN+00], [NSD08]. At the system-level, a hardware **platform** is modeled as a set of primitive blocks describing, at a high-level of abstraction, processing elements, memories and interconnects. An **application** is modeled as a set of tasks which can be allocated to hardware resources in many different ways, which means that there are many possible mappings of tasks to platform resources. Once it is determined how the application tasks are going to be allocated to the hardware resources such that all design *constraints* are met, that is, once we have a **mapping** specification, an Electronic System-Level (ESL) synthesis tool [GHP+09] generates in an automated way the hardware description at a lower level of abstraction and the software for each processor in a platform.

In order to achieve the desired performance, the applications which are going to execute on the MPSoC platform have to be specified in a way which utilizes the *parallel* processing elements in the platform. In general, identifying parallelism in an application is a difficult step. In addition, designers should determine the mapping and execution order, that is, scheduling, of application tasks to a platform, and code should be generated for each used processor in the platform. In order to perform all these design steps in an efficient way, designers raise the level of abstraction, as introduced earlier, by building high-level models of applications. Then, the designer can use these models to analyze the performance of different applications-to-platform mappings. Such design approach is called **Model-based design** and the models used in such an approach are called **Models of Computation (MoCs)**. A MoC describes in a formal way how an application works. In this thesis, we consider only *parallel* MoCs because they are suitable for expressing parallelism in an application which is going to be executed on an MPSoC platform. In a parallel MoC, an application is decomposed into tasks which can be executed in parallel. The parallel MoC defines how tasks communicate and synchronize with each other.

Streaming applications have ample amount of parallelism which should be exploited efficiently to satisfy the performance requirements. Researchers have identified three types of parallelism:

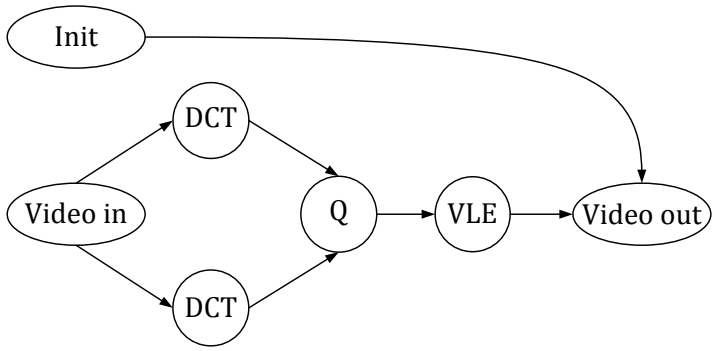


Figure 1.2: *Motion JPEG encoder application.*

1. Task-Level Parallelism (TLP): an application is split into set of tasks which can execute concurrently;
2. Data-Level Parallelism (DLP): a task of an application executes in parallel on multiple processing elements where each copy of the task processes its own data stream;
3. Pipeline-Level Parallelism (PLP): different iterations of a pair of data producer and consumer tasks execute in parallel.

Usually, an application contains more than one type of parallelism. Task-level parallelism is typically considered first when specifying an application as a set of concurrent tasks. Data-level parallelism is usually used by replicating tasks in an application in order to process more data in parallel and, hence, increase the application performance. However, if consecutive executions of a task depend on each other, data-level parallelism cannot be exploited and pipeline parallelism comes as an important form of parallelism to exploit. Figure 1.2 shows a Motion JPEG (MJPEG) encoder application represented as a set of communicating tasks which can execute concurrently. Here, we can identify examples of all three types of parallelism introduced above: 1) TLP between *Video in* and *Init*; 2) DLP between the two *DCT* tasks; and 3) PLP between different iterations of *VLE* and *Video out*.

It has been identified that *dataflow* MoCs are the most suitable parallel MoCs to express parallelism in streaming applications [TA10]. In a dataflow MoC, an application is represented as a *directed* graph, with graph nodes representing the application tasks and graph edges representing data dependences among the tasks. Thus, the parallelism is explicitly specified in the model. Dataflow MoCs differ among each other in their *expressiveness* and *decidability*. The expressiveness of a model indicates which type of applications can be modeled by the model and how compact the model is [SGTB11]. The

decidability of a model represents the extent to which designers can analyze liveness and performance of an application at compile-time. In general, expressiveness and decidability of MoCs are inversely related, meaning that more expressive MoCs are less decidable, and the opposite; hence the choice of a suitable MoC depends on the problem being addressed. For example, within the Daedalus^{RT} [BZNS12] design methodology, the Cyclo-Static Dataflow MoC [BELP96] is used as an analysis model, to analyze design non-functional properties such as throughput, latency, hard real-time behavior, while the Polyhedral Process Network MoC [VNS07] is used as an implementation model. The MoCs considered in this thesis to represent streaming applications are **Synchronous Data Flow (SDF)** [LM87], **Cyclo-Static Dataflow (CSDF)** [BELP96] and **Polyhedral Process Network (PPN)** [VNS07], given in the order of increased expressiveness, hence decreased analyzability. Because of their very good analyzability, we use the SDF and CSDF MoCs to analyze the application throughput, latency, hard real-time behavior and calculate the required size of buffers used to implement inter-task communication. On the other hand, we use the PPN MoC to generate efficient code for processors in an MPSoC and build highly accurate energy model to analyze the energy consumption. A more detailed and complete comparison of different dataflow MoCs is given in [SGTB11].

1.2 Design Requirements and Basic Approaches to Meet the Requirements

In Section 1.1.2, it has been explained that model-based design methodologies have been used to design embedded streaming MPSoCs to provide the desired system performance. In this section, we introduce the requirements which are usually put on embedded streaming MPSoCs and the basic approaches proposed by research communities to meet these requirements.

1.2.1 Timing Requirements

As mentioned earlier, the performance of a streaming application running on an MPSoC is represented with two metrics: throughput and latency. Usually, embedded streaming MPSoCs execute simultaneously multiple applications and for each application throughput and latency requirements have to be met. In addition, these multiple applications should be **temporally isolated** between each other. This means that an application can be started or stopped at run-time without violating the timing requirements of other running ap-

plications. Beside performance requirements, many embedded streaming systems have to process data within a certain time interval, that is, before a *deadline*, meaning that they have hard real-time requirements.

In general, to provide timing guarantees for streaming applications, researchers proposed either analysis approaches on the dataflow MoCs, or they specified applications as periodic real-time tasks, or devised techniques which are mixture of the previous two. In the first case, techniques are devised to provide timing guarantees for streaming applications by performing analysis on a dataflow MoC, for example, techniques proposed in [GGS+06], [SGB08], [MB07] and [BMKdD13]. The approach in [GGS+06], [SGB08] analyzes an application modeled using the (C)SDF MoC [LM87] by performing state-space exploration of the (C)SDF graph in order to find the application throughput and latency. On the other hand, the approach in [MB07] converts an initial SDF application specification into an equivalent homogeneous SDF (HSDF) specification, and does the performance analysis on the HSDF. However, the state-space of an SDF graph is exponential in the worst case, and the conversion from an SDF into an equivalent HSDF results in an application graph which size grows exponentially in the worst case, hence the analysis approaches in [GGS+06] and [MB07] have high time complexity. The approach in [BMKdD13] does application performance analysis on a CSDF graph by formulating the problem of finding performance guarantees as an Integer Linear Programming (ILP) problem. Thus, that approach has high time complexity given that ILP-based approaches suffer from severe scalability issues. All the approaches [GGS+06], [MB07] and [BMKdD13], do not provide temporal isolation among applications and need complex design space exploration to find the minimum number of processors in a platform required to provide timing guarantees.

Another way of providing timing guarantees is by specifying applications as *classical real-time tasks* [DB11]. The classical real-time task model [LL73] specifies applications as *independent tasks*. The invocations of tasks are periodic, with constant execution time for each invocation and constant interval between invocations. By using the hard real-time schedulability theories [DB11], the minimum number of processors needed to schedule applications while providing timing guarantees, and temporal isolation between the applications can be determined in a fast analytical way. However, this classical real-time task model does not model data dependencies among tasks usually found in streaming applications.

Recently, several approaches, such as [BS11], [BS13] and [BTV12], have been proposed which combine advantages of the previously mentioned ap-

proaches by converting an application specified using a dataflow MoC, hence modeling data dependencies, to real-time tasks, thus enabling temporal isolation and fast calculation of the minimum number of processors to provide timing guarantees. Therefore, in this thesis, we utilize benefits of both dataflow MoCs and real-time task models to further improve the system timing guarantees and the utilization of hardware resources.

1.2.2 Energy Requirements

As indicated in Section 1.1.1, one of the main reasons for the flattening of the performance curve across different generations of uniprocessors was the increased power consumption due to the increased clock frequency to boost performance, and the technology scaling. The idea of using MPSoC platforms partially solved the power consumption issue by allowing performance boost through parallel execution while running processors at a lower frequency. Given that the technology scaling is still one ongoing process which provides more parallel processing resources but also results in larger power dissipation, it has been identified by the International Technology Roadmap for Semiconductors (ITRS) [fSI] that the power and energy consumption are the main problems in the system design. This results in a need for design techniques which target more performance and functionality at constant power density, constrained by thermal issues, and constant energy consumption, constrained by the battery capacity. The inability to manage power dissipation limits the amount of switched-on logic content in a SoC, known as the "dark silicon" issue [EBSA+11].

Widely used techniques to reduce the power/energy consumption are Voltage-Frequency Scaling (VFS) and Power Management (PM). VFS reduces the power consumption by adjusting the voltage and operating frequency of processors while PM exploits idle times of processors by putting them to a very low-power sleep mode. In addition, according to ITRS reports, heterogeneous MPSoCs were identified as a promising solution in terms of energy-efficiency [Mit15]. Heterogeneous MPSoCs [Mit15] have been also considered as a promising solution to the dark silicon problem. Especially, the asymmetric multi-core architecture, also known as a single-ISA heterogeneous architecture, was recognized as a good trade-off in terms of energy-efficiency and programming effort [Mit15]. A single-ISA heterogeneous MPSoC consists of cores with different power-performance characteristics but with the same instruction-set architecture (ISA). Apart from containing cores with different power-performance characteristics, such heterogeneous MPSoCs cover large set of power-performance design points through voltage-frequency scaling of

the cores [Mit15]. However, with the advent of many-core systems, per-core VFS becomes impractical due to the high hardware cost and area requirement [HM07]. Therefore, to balance the energy saving and the hardware cost, cores are grouped into clusters and cores in each cluster run at the same voltage and frequency level. In addition, it has been recognized by ITRS that the accuracy of power modeling and estimation has to be improved in order to manage the power consumption to extreme limits [Kah13].

1.3 Problem Statement

After introducing the trends and requirements in the design of embedded streaming systems in Section 1.1 and Section 1.2, in this section, we formulate the problems addressed in this thesis concerning the design of embedded streaming systems.

1.3.1 Problem 1

Meeting the timing requirements is one of the most important design objectives when designing embedded streaming MPSoCs. As explained in Section 1.2.1, there are several research approaches on how to guarantee the timing behavior of streaming applications. Among them, the most appropriate one is the research approach which combines the benefits of dataflow MoC-based analysis and hard real-time analysis. The existing works [BS11], [BS13], [BTV12], following this approach, assume that each execution of an application task takes the same amount of time. However, a common behavior in streaming applications is that different executions of the same application task differ in execution time. When such changing execution nature of an application is hidden by considering one and the same value for the execution time of an application task, the application throughput is underestimated, the application latency overestimated, while the processors in an MPSoC platform are underutilized. Thus, the first problem addressed in this thesis is:
Problem 1: Can we apply the hard real-time scheduling theory for real-time periodic tasks to streaming applications while considering different execution times among different executions of an application task to obtain tighter bounds on throughput and latency and better utilize processors?

1.3.2 Problem 2

As introduced in Section 1.1.2, streaming applications contain ample amount of parallelism and can be efficiently represented by using parallel MoCs. How-

ever, the initial parallel application specification often is not the most suitable one for a given MPSoC platform. This is because application developers mainly focus on realizing certain application behavior while the computational capacity and power consumption profile of the MPSoC platform is often not fully taken into account. That is, the initial parallel specification does not expose enough parallelism, particularly in the form of DLP, to better exploit the platform to satisfy timing and energy requirements. To better utilize the underlying MPSoC platform, the initial specification of an application, that is, the initial task graph, should be transformed to an alternative one that exposes more DLP while preserving the same application behavior. This can be achieved through an unfolding transformation where the tasks from the initial graph are replicated, in an equivalent graph, a certain number of times. Special care should be taken during the unfolding transformation to avoid all unnecessary overheads caused by data management among replicas. Moreover, having more tasks' replicas than necessary results in an inefficient system due to overheads in code and data memory, scheduling and inter-tasks communication. Thus, the right amount of DLP, depending on the underlying MPSoC platform, should be determined in a parallel application specification to achieve maximum performance and timing guarantees. Therefore, the second problem we address in this thesis consists of two sub-problems. The first sub-problem is:

Problem 2a: How to convert an initial application graph into input-output equivalent graph while avoiding unnecessary overheads caused by data management among task replicas?

The second sub-problem follows as:

Problem 2b: How many times to replicate each task in the initial application graph, such that the obtained equivalent graph exposes the right amount of parallelism that maximizes the utilization of the available processors in an MPSoC platform while meeting all timing requirements?

1.3.3 Problem 3

Apart from timing requirements, energy consumption requirements are very important requirements to be met for proper functioning of embedded systems. As introduced in Section 1.2.2, the ITRS proposed heterogeneous parallel processing and frequency islands as design innovations to address the power/energy consumption requirements. In particular, the asymmetric multi-core architecture was recognized by both academia and industry as a good platform for design of energy-efficient embedded systems. Some examples of commercial asymmetric cluster MPSoCs are Samsung Exynos 5 Octa

SoC [Samb], nVidia Tegra X1 [NVI15], which include ARM big.LITTLE [Gre11] integrating high-performance cores into big clusters and low-power cores into LITTLE clusters. As mentioned in Section 1.3.2, when developing an application, application designers often do not have the timing and energy behavior of a platform in mind. Hence, it may happen that an application consists of highly imbalanced tasks in terms of the task workload, that is, task utilization. Especially, in cluster heterogeneous MPSoCs, when several tasks are mapped onto the same cluster, the task with the heaviest utilization will determine the required voltage and frequency of the whole cluster and will significantly increase the energy consumption of the other tasks mapped on the same cluster. When task replication, that is, DLP, is applied to application tasks with heavy utilization, their utilization can be decreased while still providing the same application performance. Thus, the third problem, we address in this thesis is:
Problem 3: How to map embedded streaming applications under timing requirements by utilizing per-cluster VFS and task replication to reduce the energy consumption of a system?

1.3.4 Problem 4

It was pointed out by ITRS that the accuracy of power/energy modeling is very important for efficient power/energy management [Kah13]. Model accuracy is usually traded-off for modeling and evaluation effort. Energy models used with more analyzable functional models, that is, MoCs, are usually more abstract in order to be more efficient in terms of modeling and evaluation effort and time, hence they try to capture the worst-case energy consumption. Such a model of the energy consumption results in safe but not very accurate estimates when compared with the actual energy consumption measurements on real implementations. Hence, an energy model should be more closely related to the actual running system yet be enough efficient in terms of modeling and evaluation effort and time. More expressive MoCs, as the PPN MoC for example, can give better insight of the final implementation of application tasks on a platform, hence they are often used as implementation models. However, providing timing guarantees by doing analysis on the PPN MoC is rather difficult, if not impossible [Zha15], hence the PPN MoC is used in systems where the timing requirements are not necessarily specified but it is required that the system runs at the best of its capacity (best-effort systems). Therefore, as the forth problem, we investigate:
Problem 4: How to model as accurately as possible the energy consumption of a mapping of an application onto an MPSoC platform while such a system runs at the best of its capacity?

1.4 Research Contributions

By addressing the research problems outlined in Section 1.3, in this section, we summarize the research contributions of this thesis.

Contribution 1: Proposing a scheduling approach which converts dataflow MoCs to real-time periodic tasks while considering different execution times for different executions of an application task.
To address the first problem, namely, Problem 1 in Section 1.3.1, we propose a scheduling approach, published in [SLCS15] and [SLCS16], and presented in Chapter 3, to schedule streaming applications modeled as acyclic CSDF graphs on an MPSoC platform. The proposed approach converts each task in a CSDF graph to a set of real-time periodic tasks by deriving task parameters (periods, start times, and deadlines) while considering different execution times for different executions of each task in the CSDF graph. The conversion enables application of many hard real-time scheduling algorithms which offer fast calculation of the required number of processors for scheduling the tasks with a certain guaranteed throughput and latency. In addition, the proposed approach calculates the minimum buffer sizes of the communication channels between the tasks in the CSDF graph such that the converted tasks can be scheduled as periodic real-time tasks. As part of our scheduling approach, we propose a method to reduce the graph latency by carefully selecting the deadlines of the converted real-time periodic tasks. We show, on a set of real-life streaming applications, that our approach leads to equal or higher application throughput and shorter application latency while reducing the number of processors required to schedule a given application, compared to a related approach which does not consider different execution times for different executions of CSDF tasks. However, our approach results in increased memory requirements to implement the communication among the tasks.

Contribution 2: Proposing a graph transformation and an approach that uses the transformation to exploit the right amount of parallelism that maximizes the utilization of the available processors in an MPSoC platform while meeting all timing requirements.
We address the second problem, Problem 2a-2b stated in Section 1.3.2, by proposing an unfolding graph transformation for SDF graphs and an algorithm that adapts the exploited parallelism in an application modeled using the SDF MoC according to the resources in an MPSoC by using the unfolding transformation and the hard-real time scheduling approach of CSDF graphs devised within Contribution 1 such that the application performance is maximized and hard real-time behavior guaranteed. Our contribution has been published in [SLS16b] and explained in Chapter 4. In particular, our unfolding

graph transformation carefully distributes data among task replicas, enabling more parallel execution of tasks, and our algorithm determines simultaneously which SDF tasks and how many times to replicate them, and the allocation of tasks to processors in the MPSoC. We show, on a set of real-life streaming applications, that our unfolding graph transformation for SDF graphs results in graphs with the same application throughput, shorter application latency and smaller communication memory compared to related approaches. In addition, we show that our algorithm delivers, in 98% of the conducted experiments, a solution with a shorter latency, smaller communication memory and smaller values for task replication factors compared to a related approach while the same performance and timing requirements are satisfied.

Contribution 3: Proposing an approach that exploits the right amount of parallelism in an application and per-cluster VFS to map the application onto a cluster heterogeneous MPSoC such that the energy consumption is minimized and all timing requirements met.
In our third contribution which addresses Problem 3 in Section 1.3.3, we propose a novel algorithm [SLS16a], presented in Chapter 5, to efficiently map real-time streaming applications onto cluster heterogeneous MPSoCs, which are subject to throughput constraints, such that the energy consumption of the cluster heterogeneous MPSoC is reduced by using task replication and per-cluster VFS. By using the hard real-time scheduling approach of CSDF graphs, we devised within Contribution 1, we propose an efficcient way to determine a suitable processor type for each task in an (C)SDF graph such that the energy consumption is minimized and the throughput constraint is met. Then, by using our unfolding graph transformation, devised within Contribution 2, we propose a method to determine a replication factor for each task in an SDF graph such that the distribution of the workload on the same type of processors is balanced, which enables processors to run at a lower frequency, hence reducing the energy consumption. We show, on a set of real-life streaming applications, that our proposed energy minimization approach outperforms related approaches in terms of energy consumption while meeting the same throughput constraints.

Contribution 4: Proposing an accurate energy model for best-effort streaming applications mapped onto heterogeneous MPSoC platforms.
To address the problem of accurate power/energy modeling, namely, Problem 4 in Section 1.3.4, we devise an accurate energy model [SS13] for streaming applications modeled by the PPN MoC and mapped onto heterogeneous MPSoC platforms. The energy model is based on the well-defined properties of the PPN application model. To guarantee the accuracy of the energy model,

values of important model parameters are obtained by real measurements. In addition, our energy model can model different types of communication infrastructures: with and without contention. The accuracy of the proposed energy model is evaluated on FPGA-based MPSoC platforms running two real-life streaming applications against real measurements of the energy consumption from the FPGA. The model and its accuracy and efficiency is presented in Chapter 6.

1.5 Thesis Outline

Below we give an outline of this thesis, summarizing the contents of the following chapters.

Chapter 2 gives an overview of the dataflow MoCs considered in this thesis, and some techniques from hard-real time scheduling theories relevant for this thesis.

Chapters 3 to 6 contain the contributions of this thesis. Each chapter is organized in a self-contained way, meaning that each chapter contains more specific introduction to the problem addressed, related work, the proposed solution approach, experimental evaluation, and concluding discussion.

Chapter 3 presents our hard real-time scheduling approach for streaming applications modeled as acyclic CSDF graphs.

Chapter 4 describes our unfolding graph transformation for SDF graphs and our algorithm for finding proper replication factors for each task in an SDF graph, which uses our scheduling framework described in Chapter 3, such that the processing resources are utilized as best as possible, while providing hard real-time guarantees.

Chapter 5 presents our energy-minimization approach which uses our scheduling framework described in Chapter 3 and our unfolding transformation described in Chapter 4 to find task-to-processor type assignment and task replication factors such that the energy consumption is minimized.

Chapter 6 presents our accurate energy model for streaming applications modeled using the PPN MoC and mapped onto MPSoC platforms.

Chapter 7 ends this thesis by providing conclusions regarding the work done within this thesis and discussions on potential future work.

Chapter 2

Background

THIS chapter introduces the background necessary to understand the contribution of this thesis presented in the following chapters. First, we give in Table 2.1 a summary of the mathematical notations used throughout the thesis. Then, we present the dataflow models considered in this thesis in Section 2.1, while some results from the hard real-time scheduling theory relevant for this thesis are presented in Section 2.2.

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
$\lvert x \rvert$	The cardinality (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
mod	The integer modulo operator

2.1 Dataflow Models-of-Computations (MoCs)

As mentioned earlier in Section 1.1.2, dataflow MoCs have been used to efficiently express parallelism in streaming applications. In this section, we present the dataflow MoCs considered in this thesis, that is, the CSDF and SDF MoCs are given in Section 2.1.1, and the PPN MoC is given in Section 2.1.2. The CSDF MoC is used to specify streaming applications within the hard

real-time scheduling framework proposed in Chapter 3. The SDF MoC is used to specify the input streaming applications in the techniques which exploit parallelism in streaming applications to maximize the resource utilization and minimize the energy consumption, presented in Chapters 4 and 5, respectively. The PPN MoC is used to specify streaming applications within the solution for highly accurate modeling of energy consumption, presented in Chapter 6.

2.1.1 Cyclo-Static Dataflow (CSDF)

An application modeled as a CSDF [BELP96] is a directed graph $G = (\mathcal{V}, \mathcal{E})$ that consists of a set of actors $\mathcal{V}$ which communicate with each other through a set of communication channels $\mathcal{E}$. Actors represent a certain functionality of the application, while communication channels are first-in first-out (FIFO) buffers representing data dependencies and transferring *data tokens* between the actors. A data token is an atomic data object belonging to a stream of data transferred between the actors. We can associate each actor $v_i \in \mathcal{V}$ in a graph with two sets of actors, the *predecessors set*, denoted by prec(v_i), and the *successors set*, denoted by succ(v_i). These sets are given by:

$$\text{prec}(v_i) = \{v_j \in \mathcal{V} : \exists e_u = (v_j, v_i) \in \mathcal{E}\} \tag{2.1}$$

$$\text{succ}(v_i) = \{v_j \in \mathcal{V} : \exists e_u = (v_i, v_j) \in \mathcal{E}\} \tag{2.2}$$

In addition, we can define for each actor $v_i \in \mathcal{V}$ in a graph two sets of communication channels, the *input set*, denoted by inp(v_i), and the *output set*, denoted by out(v_i). The *input set* contains all the input channels to v_i, while the *output set* contains all the output channels from v_i. If an actor v_i receives an input data stream from the environment then v_i is called *input actor*, and v_i does not have input channels, that is, inp(v_i)= $\emptyset$. Similarly, if an actor v_i produces an output data stream for the environment then v_i is called *output actor*, and v_i does not have output channels, that is, out(v_i)= $\emptyset$. A *path* $w_{i \to j}$ between actors v_i and v_j is an ordered sequence of channels connecting v_i and v_j denoted as $w_{i \to j} = \{(v_i, v_k), (v_k, v_l), \cdots, (v_m, v_j)\}$.

Every actor $v_i \in \mathcal{V}$ has an *execution sequence* $[F_i(1),\ F_i(2),\ \cdots,\ F_i(\phi_i)]$ of length ϕ_i, that is, it has ϕ_i phases. The kth time that actor v_i is fired, it executes the function $F_i(((k-1) \bmod \phi_i) + 1)$. As a consequence, the execution time of actor v_i is also a sequence $[C_i^C(1),\ C_i^C(2),\ \cdots,\ C_i^C(\phi_i)]$ consisting of the worst-case computation time values for each phase. Similarly, every output channel e_u of an actor v_i has a predefined token *production sequence* $[x_i^u(1), x_i^u(2), \cdots, x_i^u(\phi_i)]$ of length ϕ_i. Analogously, token consumption on every input channel e_u of an actor v_i is a predefined se-

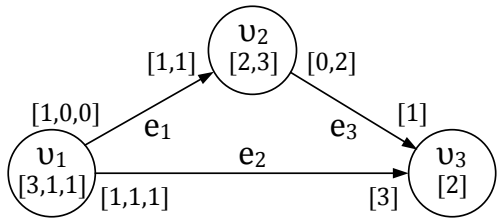


Figure 2.1: *A CSDF graph G.*

quence $[y_i^u(1), y_i^u(2), \cdots, y_i^u(\phi_i)]$, called *consumption sequence*. The total number of tokens on a channel e_u produced by v_i during its first n invocations and the total number of tokens consumed on the same channel by v_j during its first n invocations are $X_i^u(n) = \sum_{l=1}^{n} x_i^u(((l-1) \bmod \phi_i) + 1)$ and $Y_j^u(n) = \sum_{l=1}^{n} y_j^u(((l-1) \bmod \phi_j) + 1)$, respectively.

Figure 2.1 shows an example of a CSDF graph. For instance, actor v_1 has 3 phases, that is, $\phi_1 = 3$, its execution time sequence (in time units) is $[C_1^C(1), C_1^C(2), C_1^C(3)] = [3, 1, 1]$ and its token production sequence on channel e_1 is $[1, 0, 0]$.

An acyclic CSDF graph can be partitioned into a number of *levels*, denoted by L, in a way similar to topological sort. In that way, all input actors belong to level-1, the actors from level-2 have all immediate predecessors in level-1, the actors from level-3 have immediate predecessors in level-2 and can also have immediate predecessors in level-1, and so on.

An important property of the CSDF model is the ability to derive, at design time, a schedule for the actors. In order to derive a valid static schedule for a CSDF graph at design time, it has to be consistent and live.

Theorem 2.1.1 (From [BELP96])**.** *In a CSDF graph G, a repetition vector* $\vec{q} = [q_1, q_2, \cdots, q_N]^T$ *is given by*

$$\vec{q} = \mathbf{\Phi} \cdot \vec{r}, \qquad with \qquad \Phi_{jk} = \begin{cases} \phi_j & if\ j = k \\ 0 & otherwise \end{cases} \tag{2.3}$$

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

$$\mathbf{\Gamma} \cdot \vec{r} = \vec{0} \tag{2.4}$$

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

$$\Gamma_{uj} = \begin{cases} X_j^u(\phi_j) & if\ actor\ v_j\ produces\ on\ channel\ e_u \\ -Y_j^u(\phi_j) & if\ actor\ v_j\ consumes\ from\ channel\ e_u \\ 0 & otherwise. \end{cases} \tag{2.5}$$

A CSDF graph G is said to be consistent if a positive integer solution $\vec{r} = [r_1, r_2, \cdots, r_N]^T$ exists for the balance equation, Equation (2.4). We call $\vec{r}$ *aggregated repetition vector*. The smallest non-trivial aggregated repetition vector $\vec{r}$ is called *basic aggregated repetition vector* $\vec{r}$. Its corresponding repetition vector $\vec{q}$ is called *basic repetition vector* $\vec{q}$. If a deadlock-free schedule can be found, G is said to be live.

Definition 2.1.1. For a consistent and live CSDF graph G, an **actor iteration** is the invocation of an actor $v_i \in \mathcal{V}$ for q_i times, a **phase iteration** is the invocation of one phase of an actor $v_i \in \mathcal{V}$ for r_i times, and a **graph iteration** is the invocation of *every actor* $v_i \in \mathcal{V}$ for q_i times, where $q_i \in \vec{q}$, and *every phase* of *every actor* $v_i \in \mathcal{V}$ for r_i times, where $r_i \in \vec{r}$.

For the example CSDF graph G shown in Figure 2.1, we can compute the basic repetition vectors $\vec{r}$ and $\vec{q}$ by using the equations in Theorem 2.1.1, namely, Equations (2.3), (2.4) and (2.5), as follows:

$$\mathbf{\Gamma} = \begin{bmatrix} 1 & -2 & 0 \\ 3 & 0 & -3 \\ 0 & 2 & -1 \end{bmatrix}, \vec{r} = \begin{bmatrix} 2 \\ 1 \\ 2 \end{bmatrix}, \mathbf{\Phi} = \begin{bmatrix} 3 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \text{and } \vec{q} = \begin{bmatrix} 6 \\ 2 \\ 2 \end{bmatrix}.$$

Two important subsets of the CSDF MoC are the Synchronous Data Flow (SDF) MoC [LM87] and the Homogeneous Synchronous Data Flow (HSDF) MoC [LM87]. All actors in an SDF graph $G = (\mathcal{V}, \mathcal{E})$ have only one phase, that is, for each $v_i \in \mathcal{V}, \phi_i = 1$. In an HSDF graph $G = (\mathcal{V}, \mathcal{E})$, in addition to $\forall v_i \in \mathcal{V}, \phi_i = 1$, all channels have production and consumption sequences equal to 1, that is, for each $e_u = (v_i, v_j) \in \mathcal{E}, x_i^u = [x_i^u(1)] = 1, y_j^u = [y_j^u(1)] = 1$.

2.1.2 Polyhedral Process Network (PPN)

An application modeled as a PPN [VNS07] is a directed graph $G = (\mathcal{P}, \mathcal{C})$ that consists of a set of processes $\mathcal{P}$, which communicate with each other via a set of communication channels $\mathcal{C}$. Processes in $\mathcal{P}$ represent tasks of an application. Channels in $\mathcal{C}$ are bounded FIFOs and represent one direction of data communication between two processes, that is, a channel $CH_l = (P_i, P_j)$ represents a data dependency between processes P_i and P_j, where P_i is the producer and P_j is the consumer process. An example of a PPN consisting of 4 processes which communicate with each other through 5 channels is given in Figure 2.2(a). Each PPN process has a set of *input ports* it reads from and a set of *output ports* it writes to. Process P_3 in the PPN example in Figure 2.2(a) has 3 input ports IP_1, IP_2, and IP_3, and 2 output ports OP_1 and OP_2. Channels of a

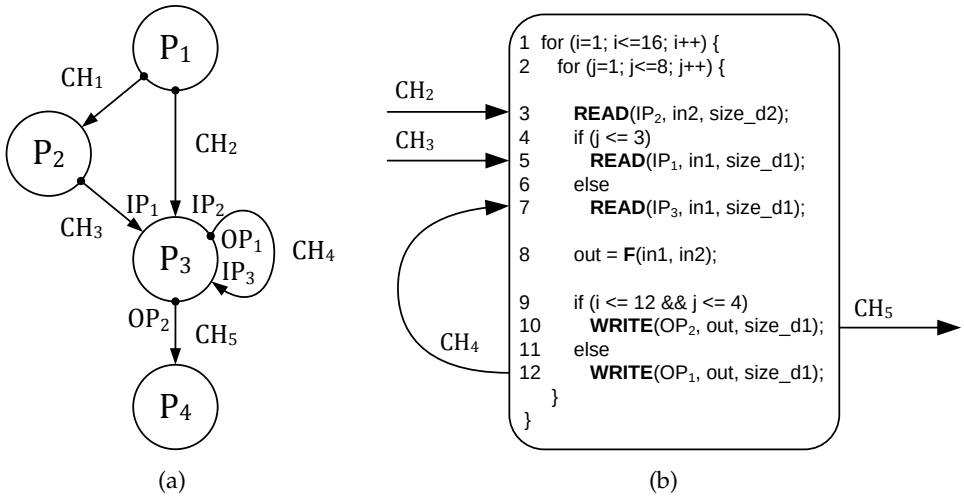


Figure 2.2: *Example of a PPN (a) and the structure of process P3 (b).*

process P_i connected to its input ports are *input channels* of P_i, while channels connected to the output ports of P_i are *output channels* of P_i.

The synchronization mechanism between the processes in the PPN MoC is *blocking read* from an empty FIFO and *blocking write* to a full FIFO. The execution of a PPN process is defined by using nested *for loops*, that is, the process execution is a set of iterations, called *process domain*. The process domain is represented using the polytope model [Fea96a]. Each PPN process has a precisely defined structure: the process reads data from a subset of its input ports depending on the values of loop iterators; then, it performs a computation on input data that generates output data; and finally, the process writes the output data through a subset of its output ports depending on the values of loop iterators.

Figure 2.2(b) shows the structure of process P_3 in the PPN example given in Figure 2.2(a). Process P_3 reads data from and writes data to channels through read and write primitives **READ**$(\cdots)$ and **WRITE**$(\cdots)$, respectively. The computation behavior of process P_3 is represented by a function **F**$(\cdots)$ in Line 8 in Figure 2.2(b). The process domain of process P_3 is given as the polytope $D_{P_3} = \{(i,j) \in \mathbb{Z}^2 \mid 1 \leq i \leq 16 \wedge 1 \leq j \leq 8\}$. Accessing an input port of the PPN process is represented as a subset of the process domain, called *input port domain*. Similarly, accessing an output port of the PPN process is represented through *output port domain*. Process P_3 in Figure 2.2 reads data from input ports IP_1, IP_2 and IP_3. The input port domain of input port IP_2 is equal

to process domain D_{P_3}, while the input port domain of port IP_1 is given as $D_{IP_1} = \{(i,j) \in \mathbb{Z}^2 \mid 1 \leq i \leq 16 \wedge 1 \leq j \leq 3\}$. Process P_3 writes data to output ports OP_1 and OP_2. Domain $D_{OP_2} = \{(i,j) \in \mathbb{Z}^2 \mid 1 \leq i \leq 12 \wedge 1 \leq j \leq 4\}$ is the output port domain of port OP_2.

2.2 Real-Time Scheduling Theory

In this section, we introduce the real-time periodic task model [DB11] and some important real-time scheduling concepts [DB11] instrumental to the approaches we present in Chapters 3, 4 and 5 of this thesis.

2.2.1 Task Model

The majority of the research on real-time scheduling considers a simple model to represent applications running on a hardware platform. In this simple model applications are modeled as a task set $\mathcal{T} = \{\tau_1, \tau_2, \cdots, \tau_n\}$ of n periodic tasks, which can be preempted at any time. A periodic task $\tau_i \in \mathcal{T}$ is defined by the 4-tuple $\tau_i = (S_i, C_i, D_i, T_i)$, where S_i is the start time of τ_i in absolute time units, C_i is the worst-case execution time (WCET), D_i is the deadline of τ_i in relative time units, and T_i is the task period in relative time units, where $C_i \leq D_i \leq T_i$. Each task τ_i executes periodically through a sequence of task invocations, that is, *job releases*, at $s_{i,k} = S_i + kT_i$, $k \in \mathbb{N}_0$. Once released, each job $\tau_{i,k}$, $k \in \mathbb{N}_0$, of a task τ_i must execute C_i time units before $s_{i,k} + D_i$, that is, the job must finish its execution before its deadline D_i. If $D_i = T_i$, then τ_i is said to have an *implicit-deadline*. Otherwise, if $D_i < T_i$, then τ_i is said to have a *constrained-deadline*. If all the tasks in a task set $\mathcal{T}$ are implicit-deadline periodic tasks, then task set $\mathcal{T}$ is an *implicit-deadline periodic (IDP) task set*. Otherwise, task set $\mathcal{T}$ is a *constrained-deadline periodic (CDP) task set*. Similarly, if all the tasks in a task set $\mathcal{T}$ have the same start time, then task set $\mathcal{T}$ is *synchronous*. Otherwise, task set $\mathcal{T}$ is *asynchronous*. In this thesis, we consider asynchronous task sets.

The **utilization** of task τ_i, denoted as u_i, where $u_i \in (0,1]$, is defined as $u_i = C_i/T_i$. For a task set $\mathcal{T}$, $u_\mathcal{T}$ is the total utilization of $\mathcal{T}$ given by $u_\mathcal{T} = \sum_{\tau_i \in \mathcal{T}} u_i$. Similarly, the **density** of task τ_i is $\delta_i = C_i/D_i$ and the total density of $\mathcal{T}$ is $\delta_\mathcal{T} = \sum_{\tau_i \in \mathcal{T}} \delta_i$. The worst-case response time of task τ_i, denoted as R_i, is defined as the longest time interval from the arrival of a job of task τ_i to the completion of job's execution.

The *processor demand bound function* of a task set $\mathcal{T}$ over a time interval $[t_1, t_2]$ represents the maximum amount of task execution that can be released

and completed in the time interval $[t_1, t_2]$, and is given by [BRH90]:

$$dbf(\mathcal{T}, t_1, t_2) = \sum_{\tau_i \in \mathcal{T}} \max\{0, \left\lfloor \frac{t_2 - S_i - D_i}{T_i} \right\rfloor - \max\{0, \left\lceil \frac{t_1 - S_i}{T_i} \right\rceil\} + 1\} \cdot C_i. \tag{2.6}$$

2.2.2 System Model

To present the important results from the real-time scheduling theory relevant for this thesis, we consider a system composed of a set $\Pi = \{\pi_1, \pi_2, \cdots, \pi_m\}$ of m identical processors. However, our contribution approaches, presented in this thesis, are applicable to both homogeneous and heterogeneous MPSoCs, because the processor heterogeneity is captured within the WCET of a task, which will be explained in more detail in Chapter 5. Thus, the results presented in the following section, Section 2.2.3, are applicable to heterogeneous MPSoCs as well.

2.2.3 Real-Time Scheduling Algorithms

In this section, we present some important scheduling concepts for scheduling applications modeled as real-time periodic tasks, introduced in Section 2.2.1, on a system modeled as described in Section 2.2.2.

Real-time scheduling algorithms for multiprocessors try to solve two problems [DB11]:

- The *allocation problem*, that is, on which processor a task should execute.
- The *priority problem*, that is, when and in which order each job of a task should execute with respect to jobs of other tasks.

Depending on how they solve the *allocation problem*, scheduling algorithms are classified into:

- *No migration*. Each task is allocated to one processor and no migration is allowed.
- *Task-level migration*. The jobs of a task can execute on different processors. However, each job can execute only on one processor.
- *Job-level migration*. A job can migrate and execute on different processors. However, parallel execution of a job on processors is not allowed.

Scheduling algorithms that allow any job to migrate are called **global** algorithms. On the other hand, algorithms which do not allow migration are called **partitioned** algorithms. Finally, scheduling algorithms that allow migration of jobs released by a subset of tasks among a subset of processors are called **hybrid** algorithms.

Depending on how they solve the *priority problem*, scheduling algorithms are classified into:

- *Fixed task priority*. Each task has a single fixed priority shared by all its jobs. Examples of this class are the Rate Monotonic (RM) [LL73] and the Deadline Monotonic (DM) [LW82] scheduling algorithms.
- *Fixed job priority*. The jobs of a task may have different priorities, but each job has a single static priority. An example of this class is the Earliest Deadline First (EDF) scheduling algorithm [LL73].
- *Dynamic priority*. A single job may have different priorities during its execution. An example of this class is the Least Laxity First (LLF) scheduling algorithm [Leu89], [DK89].

A task set $\mathcal{T}$ is *feasible* on a system Π if there exist a scheduling algorithm that can construct a schedule of tasks such that all task deadlines are met. A task $\tau_i \in \mathcal{T}$ is *schedulable* on Π by using a scheduling algorithm $\mathcal{A}$ if its worst-case response time R_i under $\mathcal{A}$ is less than or equal to its deadline D_i. If all tasks in $\mathcal{T}$ are schedulable on Π under $\mathcal{A}$, then task set $\mathcal{T}$ is *schedulable* on Π under $\mathcal{A}$. Finally, a scheduling algorithm $\mathcal{A}$ is *optimal* with respect to a task model and a system if it can schedule all task sets that comply with the task model and are feasible on the system.

The real-time scheduling theory provides various *schedulability tests* to check a schedulability of a task set on a system under a given scheduling algorithm. A schedulability test is termed *sufficient* if all of the task sets that are deemed schedulable according to the test are in fact schedulable [DB11]. A schedulability test is termed *necessary* if all of the task sets that are deemed unschedulable according to the test are in fact unschedulable [DB11]. Finally, a schedulability test that is both sufficient and necessary is an *exact* schedulability test.

2.2.4 Uniprocessor Schedulability Analysis

In this section, we will present the most used scheduling algorithms and their schedulability tests for real-time periodic tasks on uniprocessors. These scheduling algorithms are the Earliest Deadline First (EDF), Rate Monotonic (RM) and Deadline Monotonic (DM) scheduling algorithm.

Earliest Deadline First (EDF)

The EDF algorithm is a scheduling algorithm that schedules tasks' jobs according to their deadlines. The earlier deadline a task's job has, the higher execution priority is given to it. The schedulability of an implicit-deadline

periodic task set on a uniprocessor under EDF can be verified through the processor utilization. In particular, the following theorem gives a schedulability test for an implicit-deadline periodic task set on a uniprocessor under EDF [LL73]:

Theorem 2.2.1. *A set of periodic tasks $\mathcal{T}$ with implicit deadlines is schedulable under EDF if and only if*

$$\sum_{\tau_i \in \mathcal{T}} u_i \leq 1. \tag{2.7}$$

This schedulability test on uniprocessors under EDF is *exact*. In addition, the EDF scheduling algorithm is an *optimal* scheduling algorithm for periodic tasks on uniprocessors.

The *exact* schedulability test for constrained-deadline periodic tasks on uniprocessors under EDF is given by the following lemma [BRH90]:

Lemma 2.2.1. *A periodic task set $\mathcal{T}$ is feasible on one processor if and only if*

1. $\sum_{\tau_i \in \mathcal{T}} u_i \leq 1$, *and*
2. $dbf(\mathcal{T}, t_1, t_2) \leq (t_2 - t_1)$ *for all* $0 \leq t_1 < t_2 < \hat{S} + 2H$,

where $\hat{S} = \max\{S_1, \cdots, S_n\}$ *and* $H = \text{lcm}\{T_1, \cdots, T_n\}$.

However, this schedulability test is known to be co-NP-hard in the strong sense [BRH90], hence performing the schedulability test is very time consuming. To improve, that is, reduce, the schedulability test time, researchers proposed several *sufficient* schedulability tests for (asynchronous) constrained-deadline periodic tasks on uniprocessors under EDF, such as [ZB09], [AS04] and [BF05]. These algorithms either check smaller number of time points to determine the schedulability of a task set [ZB09] or approximate the processor demand bound function to simplify the computation when checking the schedulability of a task set [AS04], [BF05].

Rate Monotonic (RM)

The RM algorithm is a scheduling algorithm that assigns priorities to tasks according to their rates, that is, periods. The higher rates a task has (that is the shorter period), the higher execution priority is given to the task. Given that the period of a periodic task is constant, RM is the fixed-priority algorithm. The *sufficient* schedulability test for an implicit-deadline periodic task set on uniprocessor under RM is given by the following theorem [LL73]:

Theorem 2.2.2. *A set of periodic tasks $\mathcal{T} = \{\tau_1, \tau_2, \cdots, \tau_n\}$ with implicit deadlines is schedulable under RM if*

$$\sum_{\tau_i \in \mathcal{T}} u_i \leq n(2^{1/n} - 1). \tag{2.8}$$

When the size of the task set is significantly big ($n \to \infty$), then $\sum_{\tau_i \in \mathcal{T}} u_i = ln(2) \approxeq 0.693$. That means that any implicit-deadline task set with total utilization less than 0.69 is schedulable using RM scheduling algorithm. It has been shown in [LL73] that RM is optimal among all fixed-priority assignments in the sense that no other fixed-priority algorithms can schedule an implicit-deadline task set that cannot be scheduled by RM. However, RM is in general not optimal on uniprocessors for real-time periodic task sets.

Deadline Monotonic (DM)

The DM algorithm [LW82] extends the RM algorithm by considering tasks with deadlines less than or equal to their period, that is,constrained deadlines. According to the DM algorithm, higher priorities are given to tasks with shorter relative deadlines. The schedulability of a task set with constrained deadlines can be checked by using the utilization based test given by Relation (2.8), where instead of putting the sum of task utilizations on the left-hand side, we put the sum of task densities. However, such a test would be quite pessimistic, because the workload on the processor would be overestimated. A less pessimistic schedulability test has been proposed in [ABRW91], [ABR$^+$93] based on *Response Time Analysis (RTA)*. That test is formulated in the following theorem:

Theorem 2.2.3. *A periodic taskset $\mathcal{T}$ is schedulable using DM priority scheduling if and only if*

$$\forall \tau_i \in \mathcal{T} : R_i \leq D_i \tag{2.9}$$

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

$$R_i = C_i + \sum_{\forall \tau_j \in \mathcal{T}_{hp}(\tau_i)} \left\lceil \frac{R_i}{\tau_j} \right\rceil C_j \tag{2.10}$$

and $\mathcal{T}_{hp}(\tau_i)$ represents the set of tasks with priorities higher than the priority of τ_i.

The test in Theorem 2.2.3 can be used as a *sufficient* test for asynchronous periodic tasks.

2.2.5 Multiprocessor Schedulability Analysis

Given a system consisting of m homogeneous processors, and a task set consisting of n periodic tasks, multiprocessor schedulability analysis should determine whether the tasks can be scheduled on the processors. In the following subsections we will present some scheduling algorithms on multiprocessors with regard to how they solve the allocation problem, as introduced earlier in Section 2.2.3.

Global Scheduling Algorithms

Global scheduling algorithms schedule tasks on processors while allowing task migration. Some of these algorithms are *optimal* for implicit-deadline periodic tasks, such as Pfair [BCPV96], LLREF [CRJ06], and SA [KS97]. In the case when these algorithms are used, an *exact* schedulability test for a set $\mathcal{T}$ of implicit-deadline periodic tasks on m processors is:

$$\sum_{\tau_i \in \mathcal{T}} u_i \leq m. \tag{2.11}$$

From Equation (2.11) the absolute minimum number of processors needed to schedule a set $\mathcal{T}$ of implicit-deadline periodic tasks can be computed as:

$$m_{\text{OPT}} = \lceil \sum_{\tau_i \in \mathcal{T}} u_i \rceil. \tag{2.12}$$

In the case of constrained-deadline periodic tasks, there are no optimal online (nonclairvoyant) algorithms for the preemptive scheduling of these tasks on multiprocessors [Fis07].

Partitioned Scheduling Algorithms

Although global scheduling algorithms can be optimal for implicit-deadline periodic tasks, they introduce high migration and preemption overhead. To avoid these overheads, researchers proposed another class of scheduling algorithms, namely, *partitioned* algorithms. As soon as a set of tasks has been *partitioned* into subsets that will be executed on individual processors, the uniprocessor real-time scheduling and analysis techniques can be applied to each processor, which is the main advantage of using partitioning approaches to multiprocessor scheduling.

Given a system with m processors, and a task set of n periodic tasks, a partitioned scheduling algorithm should find a *schedulable x-partition* of the

tasks, with $x \leq m$. The schedulable x-partition is a partition: 1) which subsets contain different tasks among each other, that is, a task is allocated to only one processor, 2) where all tasks are partitioned into subsets of tasks, and 3) that guarantees that each subset of the partition is schedulable on one processor under the considered uniprocessor scheduling algorithm.

The task allocation problem in a partitioned multiprocessor scheduling approach is analogous to the *bin packing* problem [DB11], where items corresponds to tasks and bins corresponds to processors. The bin packing problem is known to be NP-hard [GJ79]. Therefore, many heuristics have been proposed to approximately solve the bin packing problem [Joh74], [CGJ96]. Below, we present the most used bin packing heuristics.

The capacity of each bin, that is, processor in a system, is equal to the maximum possible processor utilization, that is, 1. The size of each item, that is, task τ_i, is equal to task utilization u_i. Let $\mathcal{T}_k$ denote the set of tasks currently assigned to processor π_k and $u_{\pi_k} = \sum_{\tau_i \in \mathcal{T}_k} u_i$ denote the total utilization currently assigned to processor π_k. At the beginning of the task partitioning no task is assigned to a processor, that is, $\mathcal{T}_k = \emptyset$ and $u_{\pi_k} = 0$ for all π_k. The heuristics assign each task $\tau_i \in \mathcal{T}$ to a certain processor $\pi_k \in \Pi$ following a certain, heuristic-specific, rule till the task is schedulable on a processor by a certain (selected) scheduling algorithm $\mathcal{A}$, considering one task at a time.

- **First-Fit (FF).** A task τ_i is assigned to the lowest-indexed processor π_k that can contain the task such that all the tasks assigned so far to π_k are schedulable. That is

$$k = \min_{j \in [1 \cdots m]} \{j : u_i + u_{\pi_j} \leq 1 \wedge \mathcal{T}_j \text{ schedulable by } \mathcal{A} \text{ on } \pi_j\}. \quad (2.13)$$

  If the condition is not satisfied by any processor used so far, task τ_i is assigned to an unused processor in the platform. If no such processor exists, task set $\mathcal{T}$ is not schedulable on system Π.
- **Best-Fit (BF).** A task τ_i is assigned to a processor π_k such that π_k has the minimal remaining utilization after the task assignment and all the tasks assigned so far to π_k are schedulable. That is

$$k = \min_{j \in [1 \cdots m]} \{j : u_i + u_{\pi_j} \text{ is closest to, without exceeding } 1 \wedge \mathcal{T}_j \text{ schedulable by } \mathcal{A} \text{ on } \pi_j\}. \quad (2.14)$$

  If the condition is not satisfied by any processor used so far, task τ_i is assigned to an unused processor in the platform. If no such processor exists, task set $\mathcal{T}$ is not schedulable on system Π.

- **Worst-Fit (WF).** A task τ_i is assigned to a processor π_k such that π_k has the maximal remaining utilization after the task assignment and all the tasks assigned so far to π_k are schedulable. That is

$$k = \min_{j \in [1 \cdots m]} \{j : u_i + u_{\pi_j} \text{ is minimal} \wedge \mathcal{T}_j \text{ schedulable by } \mathcal{A} \text{ on } \pi_j\}. \tag{2.15}$$

  If the condition is not satisfied by any processor used so far, task τ_i is assigned to an unused processor in the platform. If no such processor exists, task set $\mathcal{T}$ is not schedulable on system Π.

Often a preprocessing step is performed on tasks before performing a heuristic to improve the performance of the heuristic. The preprocessing step represents task sorting according to certain criteria, such us, increasing or decreasing task utilization, increasing or decreasing task density, and so on. Usually, the tasks are sorted in decreasing order of their utilization. When adding this preprocessing step to the previously presented partitioning heuristics, we obtain the **First-Fit Decreasing (FFD), Best-Fit Decreasing (BFD), and Worst-Fit Decreasing (WFD)** heuristics.

The partitioning heuristics can be compared among each other by using their *approximation ratio* metric. The approximation ratio of a heuristic says how much more processors are required to schedule a set of tasks when tasks are partitioned by using the corresponding partitioning heuristic in comparison to an optimal partitioning algorithm. For example, the approximation ratios for FF and BF are 17/10 [CGJ96], [GJ79], while approximation ratio for FFD is 11/9 [Yue91].

The minimum number of processors needed to schedule a task-set $\mathcal{T}$ by a partitioned scheduling algorithm is given by:

$$m_{\text{PAR}} = \min_{x \in \mathbb{N}} \{x | \exists x\text{-partition of } \mathcal{T} \wedge \forall k \in [1, x] : \mathcal{T}_k \text{ is schedulable on } \pi_k\}. \tag{2.16}$$

Note that m_{OPT} is the lower bound on the number of processors m_{PAR} needed by partitioned scheduling algorithms.

Hybrid Scheduling Algorithms

Although partitioned scheduling approaches have low preemption overheads and do not introduce any migration overhead, they may not, in general, fully utilize available processing resources in a system, that is, they may introduce processing capacity loss. To utilize the benefits of global and partitioned

scheduling approaches, researchers proposed hybrid approaches which combine elements of both. Hybrid scheduling approaches can be divided into semi-partitioned and clustering (hierarchical) approaches [DB11].

In the semi-partitioned scheduling approaches, a small number of tasks is allowed to migrate between certain processors to utilize better the processing resources. Examples of semi-partitioned scheduling algorithms which split some tasks into two components that execute at different times on different processors are EKG [AT06], Ehd2-SIP [KY07] and EDF-fm [ABD08].

In hierarchical scheduling approaches [LB03], [HA06], a set of tasks are grouped together and scheduled as a single entity, called server task or supertask. When the entity is scheduled, one of its tasks is selected to execute according to an internal scheduling policy. Hence, the supertasks/servers are scheduled globally, while the scheduling of the tasks within a supertask/server is done locally, that is, it is analogous to scheduling on a uniprocessor. In another case of hierarchical scheduling approaches, for example, [SEL08], tasks are allocated to (virtual) clusters of processors and scheduled according to a global scheduling algorithm on processors within their cluster. In this way, processing capacity loss is less than in fully partitioned approaches, while the small number of processors in each cluster may reduce migration overheads, depending on the particular hardware architecture.

Chapter 3

Hard Real-Time Scheduling Framework

Jelena Spasic, Di Liu, Emanuele Cannella, Todor Stefanov, "On the Improved Hard Real-Time Scheduling of Cyclo-Static Dataflow", *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 15, Issue 4, Article 68, August 2016.

Jelena Spasic, Di Liu, Emanuele Cannella, Todor Stefanov, "Improved Hard Real-Time Scheduling of CSDF-modeled Streaming Applications", *In Proceedings of the IEEE/ACM/IFIP International Conference on HW/SW Codesign and System Synthesis (CODES+ISSS'15)*, pp. 65–74, Amsterdam, The Netherlands, October 4-9, 2015.

IN this chapter, we present a scheduling approach to provide timing guarantees for streaming applications mapped on MPSoCs. In particular, we describe in more detail our solution approach, introduced in Section 1.4, to the research problem, **Problem 1**, described in Section 1.3.

The remainder of this chapter is organized as follows. Section 3.1 continues the introduction by describing in more detail the addressed research problem. It is followed by Section 3.2, which gives a summary of the contributions presented in this chapter. An overview of the related work is given in Section 3.3. Then, we give an example in Section 3.4 to motivate the need for our scheduling approach. The proposed scheduling approach is described in Section 3.5. The experimental evaluation of our proposed scheduling approach is presented in Section 3.6. The concluding discussion is given in Section 3.7.

3.1 Problem Statement

Recently, the authors in [BS13] proposed a framework to schedule streaming applications modeled as acyclic CSDF graphs as a set of real-time periodic tasks on an MPSoC platform. They also derive the minimum number of processors needed to schedule the applications on a platform. However, in that framework, the authors use one and the same worst-case execution time (WCET) value for all execution phases of a task in the CSDF graph, although a task in the CSDF graph may have a different WCET value for every phase. The authors simply take and use the maximum WCET value among the WCET values for all phases of a task. By doing this, the cyclically changing execution nature of an application modeled by the CSDF model is hidden, which leads to underestimation of the throughput, overestimation of the latency, and underutilization of processors. In another recent work [BMKdD13], the authors proposed a framework to evaluate a lower bound of the maximum throughput of a periodically scheduled CSDF-modeled application. However, the authors do not provide a method to determine the number of processors required for scheduling the application. Moreover, their approach does not ensure temporal isolation among applications, that is, the schedule of applications has to be recalculated once a new application comes in the system and hence it may be possible that the previously calculated throughput of an application can no longer be reached. Thus, in this chapter, we investigate the possibility to **schedule streaming applications modeled as acyclic CSDF graphs as real-time periodic tasks** on an MPSoC platform while **considering different WCET values for task's phases** in an acyclic CSDF graph and providing temporal isolation of applications and hard real-time guarantees.

3.2 Contributions

In order to address the problem described in Section 3.1, we propose a scheduling approach which contributions are summarized as follows:

- We prove that considering a different WCET value for each execution phase of a task we can convert the execution phases of each task in an acyclic CSDF graph to strictly periodic real-time tasks. This enables the use of many hard real-time scheduling algorithms to schedule such tasks with a certain guaranteed throughput and latency. (Theorem 3.5.2)
- We prove that our scheduling approach gives equal or higher throughput than the existing hard real-time scheduling approach for acyclic CSDF graphs. (Theorem 3.5.3)

- We propose a method for reducing the latency of an acyclic CSDF graph scheduled as a set of strictly periodic real-time tasks. (Section 3.5.5)
- We show, on a set of real-life streaming applications, that scheduling each execution phase of a CSDF task as a strictly periodic task and considering different WCET per phase lead not only to tighter guarantee on the throughput of an application but also to better utilization of processor resources. (Section 3.6.1)
- We demonstrate, on a set of real-life streaming applications, that the total time required by our approach to derive the schedule of the tasks, to calculate the minimum number of processors needed to schedule the tasks, and to calculate the size of communication buffers between tasks is comparable to the time required by the existing hard real-time scheduling approach for CSDF graphs. In addition, we show that the total time needed by our approach is much shorter in comparison to the existing periodic scheduling and self-timed scheduling approaches for CSDF graphs. (Section 3.6.2)
- We show, on a set of real-life streaming applications, that the latency of the applications scheduled by our scheduling approach can be reduced by our proposed latency reduction method in most cases to the desirable latency values while keeping higher or equal application throughput and requiring equal or smaller number of processors in comparison to the existing scheduling approaches. (Section 3.6.3)

Note that by considering acyclic CSDF graphs, our solution approach is applicable to many streaming applications as it has been shown in [TA10] that around 90% of streaming applications can be modeled as acyclic SDF graphs.

3.3 Related Work

Research on scheduling of streaming applications modeled by parallel MoCs has been active for a long period of time. Below, we compare our approach with some of the existing hard real-time scheduling approaches for streaming applications and with the scheduling approaches which do not provide hard real-time guarantees but are similar to our approach.

[HGWB13] proposes a two parameter (σ, ρ) workload characterization to reduce the difference between the worst-case throughput, determined by the analysis, and the actual throughput of the application. They consider different execution times for task's phases and then the average worst-case execution time is used to improve the minimum guaranteed throughput/latency. Similar to them, we consider different execution times for task's phases in a CSDF

graph. But in contrast to them, we convert task's phases to classical periodic hard real-time tasks, which allows us to calculate the minimum number of processors required to guarantee certain throughput and latency in a fast and analytical way for global scheduling and in a polynomial time for partitioned scheduling by using our algorithm given in Section 3.5.6.

In [BTV12], the authors propose an analysis framework for hard real-time applications modeled as Affine Dataflow (ADF) graphs. The actors in an ADF graph are scheduled as periodic tasks. The ADF model proposed in [BTV12] extends the CSDF model and hence, is more expressive than the CSDF. However, in their approach only one value is considered as the WCET value of a task, while we consider a different WCET value per each phase of a task, thereby efficiently exploiting the cyclic nature of the CSDF model and providing a tighter throughput guarantee.

[BMKdD13] proposes a framework to derive the maximum throughput of a CSDF graph under a periodic schedule and to calculate the buffer sizes in the graph with a throughput constraint. Both problems are represented as Linear Programming (LP) problems and solved approximately. Similar to our work, their work considers different execution times for each phase of a task. However, it is not explicitly given in [BMKdD13] how to compute the number of processors needed to schedule the graph according to the derived schedule. One possible way is to look at the derived schedules and find the maximum number of active tasks at any given point in time. However, this procedure has an exponential time complexity in the worst case. In contrast, in our case the conversion of CSDF task's phases to classical periodic hard real-time tasks enables fast and analytical calculation of the minimum number of processors for global scheduling of the tasks, and a polynomial time derivation of the number of processors for partitioned scheduling by using our algorithm given in Section 3.5.6.

The closest to our work, in terms of scope of work and methods proposed to schedule streaming applications modeled as acyclic CSDF graphs, is the work in [BS13]. The authors in [BS13] convert each task in a CSDF graph to a periodic task by deriving parameters such as period and start time. Then they use hard real-time schedulability analysis to determine the minimum number of processors required to execute the derived task-set. Our approach differs from [BS13] in the following: we use different WCET values for each execution phase of a task and each phase is converted to a periodic task, while in [BS13], only one WCET value is used for a task and every execution of a task is periodic with a calculated period. By considering different WCET values for each task phase and converting each phase to a periodic task, we

can guarantee tighter throughput and better utilization of processor resources.

3.4 Motivational Example

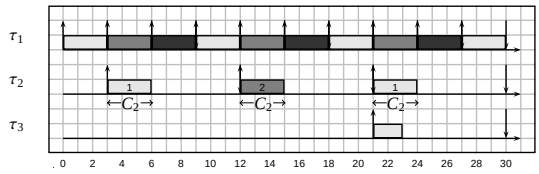
The goal of this section is to show that the real-time strictly periodic scheduling (SPS) approach [BS13] is not efficient in terms of throughput, latency and utilization of processor resources. In the framework proposed in [BS13], every actor v_i in a CSDF graph G is converted to a real-time periodic task τ_i by computing the task parameters S_i, D_i, T_i and C_i, where C_i is computed as the maximum WCET value of actor v_i, that is, $C_i = \max_{1 \leq \varphi \leq \phi_i}\{C_i(\varphi)\}$, where $C_i(\varphi)$ contains the worst-case computation, the worst-case data read and the worst-case data write times of a phase φ of actor v_i. To execute graph G strictly periodically, period T_i for each actor v_i, that is, each corresponding task τ_i, is computed as:

$$T_i = \frac{\text{lcm}(\vec{q})}{q_i} \left\lceil \frac{\max_{v_j \in V}\{C_j \cdot q_j\}}{\text{lcm}(\vec{q})} \right\rceil, \forall v_i \in \mathcal{V}, \tag{3.1}$$

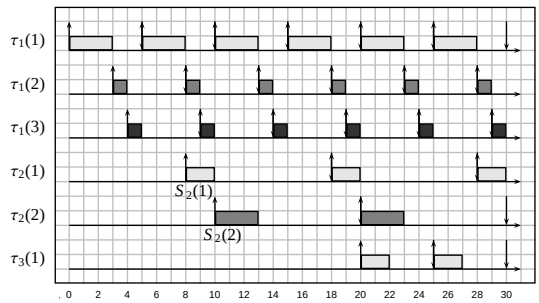
where $\text{lcm}(\vec{q})$ is the least common multiple of all repetition entries in $\vec{q}$. The strictly periodic schedule of all actors in G, given in Figure 2.1, is shown in Figure 3.1(a), under the assumption that data read and write times are 0 (for the sake of simplicity). For example, actor v_2 (task τ_2 in Figure 3.1(a)) executes periodically with the calculated period $T_2 = 9$. Note that for every actor's phase one and the same WCET value is considered, that is, for actor v_2 we have two phases 1 and 2 and the considered WCET value C_2 for each phase is $C_2 = \max\{C_2(1), C_2(2)\} = \max\{C_2^C(1), C_2^C(2)\} = \max\{2, 3\} = 3$.

To demonstrate the need of considering different WCET values of actor's phases and the drawback of strictly periodic schedule between actor phases, we analyze two different schedules of the CSDF graph G in Figure 2.1. The first schedule we consider is SPS, visualized in Figure 3.1(a). The throughput $\mathcal{R}$, latency $\mathcal{L}$ of G and the required number of processors m are given in Table 3.1 under SPS.

However, by taking one and the same value as the WCET for all execution phases of an actor, the cyclic behavior of the CSDF actors is hidden. Assume that we convert each actor v_i in G to a set of ϕ_i IDP tasks $\tau_i(1), \tau_i(2) \cdots \tau_i(\phi_i)$ considering different WCET values for each execution phase and execute them as periodic tasks. The execution schedule of such task-set is given in Figure 3.1(b). Again, here we assume that data read and write times are 0. For example, actor v_2 is converted to 2 IDP tasks $\tau_2(1)$ and $\tau_2(2)$ where each task is executed periodically with a period equal to 10. Moreover, the WCET values



(a)



(b)

Figure 3.1: *(a) The SPS and (b) ISPS of graph G in Figure 2.1.*

Table 3.1: *Throughput, latency and number of processors for G under different scheduling schemes.*

SPS			ISPS		
$\mathcal{R}$	$\mathcal{L}$	m	$\mathcal{R}$	$\mathcal{L}$	m
1/18	30	2	1/10	25	2

of the tasks $\tau_2(1)$ and $\tau_2(2)$ are not the same but $\tau_2(1)$ has WCET $C_2(1) = 2$ and $\tau_2(2)$ has WCET $C_2(2) = 3$, as the original specification in Figure 2.1.

We can see from Table 3.1 under ISPS that by scheduling G in such a way we can obtain almost 2 times higher graph throughput and shorter graph latency while resources in terms of the required number of processors are the same compared with SPS and thus, the processor resources are better utilized in the case of ISPS. This is especially important in case of a timing constraint because it may happen that the graph cannot meet the constraint when scheduled under SPS. Here the throughput and latency under ISPS are calculated by using our approach described in Section 3.5. The required number of processors for both SPS and ISPS is calculated by Equation (2.12). Moreover, the number of processors needed for partitioned scheduling in both cases is the same as the number needed for global scheduling given by Equation (2.12). We can see from the motivational example that the SPS approach from [BS13] yields to lower throughput and larger latency of a graph by using one and the same value for the WCET of each phase of an actor and by strictly periodic scheduling of all executions of the actor. Thus, different WCET values for actor phases should be considered and the constraint on strictly periodic scheduling between the actor phases should be removed.

3.5 Improved Hard Real-Time Scheduling of CSDF

In this section, we present our scheduling framework, called *improved strictly periodic scheduling* (ISPS), which enables a conversion of every actor of an acyclic CSDF graph to a set of periodic tasks. Each set of periodic tasks corresponding to an actor has as many elements as the number of phases of that actor. By taking into account the WCET value of each phase of an actor in a graph, the proposed approach computes the parameters S_i and T_i of tasks corresponding to the actor and the minimum buffer sizes of the communication channels such that ISPS is guaranteed to exist.

The proposed conversion procedure is given in Algorithm 1. First, the periods of tasks corresponding to actors are calculated in lines 1-2, explained in Section 3.5.1. Then, relative deadlines D_i of the tasks corresponding to an actor v_i are selected from the range $D_i \in [\max_{1\leq\varphi\leq\phi_i}\{C_i(\varphi)\}, \check{T}_i]$, lines 3-6. For example, if one wants to minimize the number of processors needed to schedule the converted tasks, he/she should select relative deadlines of the tasks to be equal to the corresponding task periods, that is, $D_i = \check{T}_i$. On the other hand, if one wants to reduce the graph latency, he/she should use our latency reduction method proposed in Section 3.5.5. The start times for each task-set corresponding to an actor are computed in lines 7-12, for details see Section 3.5.2. Finally, the buffer sizes of the communication channels are derived in lines 13-14, for details see Section 3.5.3.

3.5.1 Deriving Periods of Tasks

The first step in constructing the ISPS of a CSDF graph is to derive the valid period for each periodic task corresponding to a phase of an actor in the graph. To calculate the periods, we introduce the following definitions:

Definition 3.5.1. For each actor v_i in an acyclic CSDF graph G, the **WCET sequence** $C_i = [C_i(1), C_i(2), \cdots, C_i(\phi_i)]$, represents the sequence of the WCET values, measured in time units, for each execution phase of v_i. The WCET value $C_i(\varphi)$ for a phase φ is given by:

$$C_i(\varphi) = \Big(C^R \cdot \sum_{e_r \in \text{in}(v_i)} y_i^r(\varphi)\Big) + C_i^C(\varphi) + \Big(C^W \cdot \sum_{e_w \in \text{out}(v_i)} x_i^w(\varphi)\Big), \qquad (3.2)$$

where C^R represents the platform-dependent worst-case time needed to read a single token from an input channel e_r from the set of input channels in(v_i) of actor v_i; analogously, C^W is the worst-case time needed to write a single token to an output channel e_w from the set of output channels out(v_i) of v_i;

Algorithm 1: Procedure to convert a CSDF graph to a set of periodic tasks.

Input: A CSDF graph $G = (\mathcal{V}, \mathcal{E})$.
Output: For each actor $v_i \in \mathcal{V}$, a set of periodic tasks $\mathcal{T}_{v_i} = \{\tau_i(1), \cdots, \tau_i(\phi_i)\}$, and for each channel $e_u \in \mathcal{E}$, the size of the buffer b_u.

for *actor* $\tau_i \in \mathcal{V}$ **do**
  Compute the minimum common period $\check{T}_i$ by using Equation (3.5);
for *actor* $v_i \in \mathcal{V}$ **do**
  Select deadline D_i, where $D_i \in [\max_{1 \leq \varphi \leq \phi_i}\{C_i(\varphi)\}, \check{T}_i]$;
  for *phase* φ *of* v_i, $1 \leq \varphi \leq \phi_i$ **do**
    $\tau_i(\varphi) = (0, C_i(\varphi), D_i, \check{T}_i)$;
for *actor* $v_i \in \mathcal{V}$ **do**
  Compute the start time of the first phase $S_i(1)$ by using Equation (3.11);
  $\tau_i(1) = (S_i(1), C_i(1), D_i, \check{T}_i)$;
  for *phase* φ *of* v_i, $2 \leq \varphi \leq \phi_i$ **do**
    Compute the start time of the φth phase $S_i(\varphi)$ by using Equation (3.9);
    $\tau_i(\varphi) = (S_i(\varphi), C_i(\varphi), D_i, \check{T}_i)$;
for *communication channel* $e_u \in \mathcal{E}$ **do**
  Compute the buffer size b_u by using Equation (3.17);

$y_i^r(\varphi)$ and $x_i^w(\varphi)$ is the number of tokens read from e_r and written to e_w by v_i, respectively, during its execution phase φ; and $C_i^C(\varphi)$ is the worst-case computation time of v_i in its phase φ.

Definition 3.5.2. For each actor v_i in an acyclic CSDF graph G, the **maximum WCET value** MC_i is given by $MC_i = \max_{1 \leq \varphi \leq \phi_i}\{C_i(\varphi)\}$.

Definition 3.5.3. For an acyclic CSDF graph G, an **aggregated execution vector** $\vec{AC}$, where $\vec{AC} \in \mathbb{N}^N$, represents the aggregated WCET values of the actors in G and its elements are given by $AC_i = \sum_{\varphi=1}^{\phi_i} C_i(\varphi)$, where $C_i(\varphi)$ is the WCET value of v_i's phase φ.

Each actor $v_i \in V$ in graph G is converted to a periodic task set $\mathcal{T}_{v_i} = \{\tau_i(1), \cdots, \tau_i(\phi_i)\}$.

Definition 3.5.4. A task $\tau_i(\varphi)$ corresponding to a phase φ of an actor v_i, where $1 \leq \varphi \leq \phi_i$, in an acyclic CSDF graph G is a **strictly periodic task** iff the time period between any two consecutive firings of that task is constant.

All tasks belonging to a periodic task set $\mathcal{T}_{v_i}$ corresponding to an actor v_i have the same period T_i, which we call *common period*.

Definition 3.5.5. For an acyclic CSDF graph G, a **common period vector** $\vec{T}$, where $\vec{T} \in \mathbb{N}^N$, represents the periods, measured in time units, of periodic task-sets corresponding to actors in G. $T_i \in \vec{T}$ is common period of periodic task-set corresponding to actor $v_i \in V$. $\vec{T}$ is given by the solution to both

$$r_1 T_1 = r_2 T_2 = \cdots = r_{N-1} T_{N-1} = r_N T_N \tag{3.3}$$

and

$$\vec{T} - \vec{AC} \geq \vec{0}, \tag{3.4}$$

where $r_i \in \vec{r}$, and $\vec{r}$ is the aggregated repetition vector introduced in Section 2.1.1.

Lemma 3.5.1. *For an acyclic CSDF graph G, the minimum common period vector $\vec{\check{T}}$ is given by:*

$$\check{T}_i = \frac{\text{lcm}(\vec{r})}{r_i} \left\lceil \frac{\max_{v_j \in V}\{AC_j \cdot r_j\}}{\text{lcm}(\vec{r})} \right\rceil, \forall v_i \in V, \tag{3.5}$$

where $lcm(\vec{r})$ is the least common multiple of all phase repetition entries in $\vec{r}$.

Proof. The minimum common period vector $\vec{\check{T}}$ that solves Equation (3.3) is given by:

$$\check{T}_i = \text{lcm}\{r_1, r_2, \cdots, r_N\}/r_i, \forall v_i \in V.$$

Inequality (3.4) can be re-written as:

$$c\check{T}_1 \geq AC_1, c\check{T}_2 \geq AC_2, \cdots, c\check{T}_N \geq AC_N, c \in \mathbb{N}. \tag{3.6}$$

Further, Inequality (3.6) can be re-written as:

$$c \geq AC_1 r_1 / \text{lcm}(\vec{r}), \cdots, c \geq AC_N r_N / \text{lcm}(\vec{r}). \tag{3.7}$$

From Inequality (3.7), it follows that c is greater than or equal to $\max_{v_j \in V}\{AC_j r_j\} / \text{lcm}(\vec{r})$. However, $\max_{v_j \in V}\{AC_j r_j\} / \text{lcm}(\vec{r})$ is not always guaranteed to be an integer. Because of that, the value is rounded up by taking its ceiling. Thus, the minimum common period vector which satisfies both Equation (3.3) and Inequality (3.4) is given by Equation (3.5). ■

For the CSDF graph in Figure 2.1, the derived minimum common periods in time units are $[\check{T}_1, \check{T}_2, \check{T}_3] = [5, 10, 5]$.

Theorem 3.5.1. *For any acyclic CSDF graph G, where G has L topological sort levels, a periodic schedule exists with start times $S_i(\varphi)$, $\varphi \in [1, \phi_i]$, for each level-k actor $v_i \in V$ given by:*

$$S_i(1) = (k-1) \cdot 2\alpha \tag{3.8}$$

and

$$S_i(\varphi) = S_i(\varphi - 1) + C_i(\varphi - 1),\ \forall \varphi \in [2, \phi_i], \tag{3.9}$$

such that every phase of an actor $v_i \in V$ is strictly periodic with a constant period $T_i \in \vec{T}$ and every communication channel $e_u = (v_i, v_j) \in E$ has a bounded buffer capacity, given by:

$$b_u = (l - k + 1) \cdot 2X_i^u(\phi_i r_i), \tag{3.10}$$

where $\alpha = r_1 T_1 = \cdots = r_N T_N$ is the iteration period of G, v_i is level-k actor and v_j is level-l actor, $l \geq k$.

Proof. Let us assume that graph G is partitioned into L levels in a way similar to topological sort. In that way, all input actors belong to level-1, the actors from level-2 have all immediate predecessors in level-1, the actors from level-3 have immediate predecessors in level-2 and can also have immediate predecessors in level-1, and so on. The graph iteration period is $\alpha = r_1 T_1 = \cdots = r_N T_N$. During the iteration period each phase of v_i is executed r_i times. Assume that the first phase of level-1 actors starts at time $t = 0$. Other phases of an actor are scheduled to be fired as soon as the WCET of the previous phase elapses. Recall that every actor v_i in graph G is converted to a set of strictly periodic tasks where a task corresponds to a phase of the actor. Consider now an actor from level-1, denoted as v_1. By time $t = \alpha + S_1(\phi_1)$, the last phase of v_1 will finish its r_1th execution, where $S_1(\phi_1)$ is the start time of the last phase of v_1. Level-1 actors will complete a whole iteration by time $t_1 = \alpha + \max_{v_i \in \text{level-1}}\{S_i(\phi_i)\}$ and will continue executing their second iteration. According to Equation (2.4), level-1 actors will produce enough data on all channels to level-2 actors by time t_1 such that level-2 actors can execute a whole iteration if their first phases are started at t_1, at the earliest. Let us start the first phases of level-2 actors at time $t = 2\alpha$ and all the other phases of a level-2 actor one after the other. Similarly, by time $t_2 = 3\alpha + \max_{v_i \in \text{level-2}}\{S_i(\phi_i) - S_i(1)\}$, level-3 actors will have enough data to execute one iteration. Thus, starting the first phases of level-3 actors at time $t = 4\alpha$ guarantees that the actors can execute a whole iteration. By repeating the same procedure to the actors of the last level, level L, (by starting their first phases at $t = (L - 1) \cdot 2\alpha$ and all the other phases as soon as the WCET of previous phase elapses), we obtain an overlapping schedule σ where all actors execute their corresponding iterations. In the constructed schedule, the first phase of an actor v_j corresponding to a level-i will start execution at time $t = (i - 1) \cdot 2\alpha$ and once it starts it will be fired every T_j time units. The other phases start their executions one after the other and all within period T_j. Once started, each phase is re-executed every T_j time units.

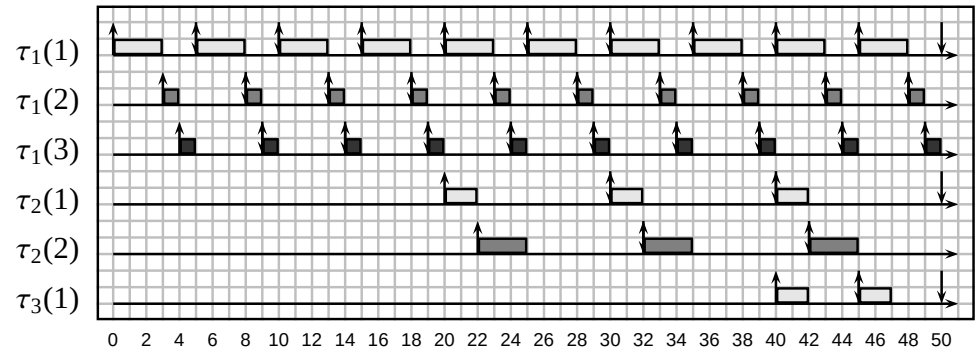


Figure 3.2: *The periodic schedule σ for the CSDF graph G shown in Figure 2.1.*

Now, we will prove that the constructed schedule executes with bounded buffers. The longest delay which may happen between production and consumption of data tokens is in case when there is a dependency e_u between the first iteration of a level-1 actor and the first iteration of a level-L actor. In this case the delay is equal to $(L-1)\cdot 2\alpha$ and during that period the level-1 actor will produce on channel e_u at most $(L-1)\cdot 2X_1^u(\phi_1 r_1)$ data tokens, where $X_1^u(\phi_1 r_1)$ is the number of tokens produced during $\phi_1 r_1$ executions of the level-1 actor. However, starting from $L\cdot 2\alpha$ level-1 and level-L execute in parallel, so we should increase the buffer size by $2X_1^u(\phi_1 r_1)$ which then becomes $L\cdot 2X_1^u(\phi_1 r_1)$. We can now use the methodology described above to determine the buffer size of each communication channel in a graph: each channel $e_u \in E$, connecting a level-i source actor v_k and a level-j destination actor ($j \geq i$) will store according to schedule σ at most:

$$b_u = (j-i+1)\cdot 2X_k^u(\phi_k r_k)$$

tokens. Thus, an upper bound on the buffer sizes exists. ■

For the example graph G given in Figure 2.1, actors in G are grouped into 3 levels such that v_1 is level-1 actor, v_2 level-2 and v_3 is level-3 actor. The calculated graph iteration period α is equal to 10. The periodic schedule resulting from Theorem 3.5.1, namely schedule σ, is depicted in Figure 3.2.

3.5.2 Deriving the Earliest Start Time of Actor's First Phase

In order to represent an actor of a CSDF graph as a set of strictly periodic tasks, in Theorem 3.5.1 we already introduced the start times of phases of the actors corresponding to different levels. However, although start times given

by Equation (3.9) are minimal relative to the start time of the corresponding first phase $S_i(1)$, start times $S_i(1)$ given by Equation (3.8) are not minimal. Minimizing the start times is very important because it has a direct impact on the latency of the graph and the buffer sizes of the communication channels. Therefore, the earliest (minimal) start times of actor's first phase $S_i(1)$ are derived below.

We derive the earliest start times assuming that the token production happens as late as possible (at the deadlines) and the tokens consumption happens as early as possible (at the beginning of execution of each phase).

Lemma 3.5.2. *For an acyclic CSDF graph G, the earliest start time of the first phase of an actor $v_j \in V$, denoted $S_j(1)$, under ISPS is given by:*

$$S_j(1) = \begin{cases} 0 & \text{if } prec(v_j) = \emptyset \\ \max_{v_i \in prec(v_j)}\{S_{i\to j}(1)\} & \text{if } prec(v_j) \neq \emptyset \end{cases} \tag{3.11}$$

where $prec(v_j)$ is the set of predecessors of v_j, and $S_{i\to j}(1)$ is given by:

$$\begin{aligned} S_{i\to j}(1) = \min_{t\in[0,S_i(1)+\alpha+\Delta_i(\phi_i)]}\{t : & \operatorname*{prd^S}_{[S_i(1),\max\{S_i(1),t\}+k)}(v_i,e_u) \\ & \geq \operatorname*{cns^S}_{[t,\max\{S_i(1),t\}+k]}(v_j,e_u),\ \forall k \in [0,\alpha+\Delta_i(\phi_i)]\}, \end{aligned} \tag{3.12}$$

where $S_i(1)$ is the earliest start time of the first phase of a predecessor actor v_i, $\alpha = r_i T_i = r_j T_j$, $\Delta_i(\phi_i) = S_i(\phi_i) - S_i(1)$, $\mathsf{prd}^{\mathsf{S}}_{[t_s,t_e)}(v_i,e_u)$ is the number of tokens produced by v_i into channel e_u during the time interval $[t_s,t_e)$, and $\mathsf{cns}^{\mathsf{S}}_{[t_s,t_e]}(v_j,e_u)$ is the number of tokens consumed by v_j from channel e_u during the time interval $[t_s,t_e]$.

Proof. In Theorem 3.5.1, we have proved the existence of ISPS when the first phase of level-k actors was started at time $(k-1)\cdot 2\alpha$. According to the schedule σ, level-$(k-1)$ predecessor v_i will start the execution of its first phase at $S_i(1) = (k-2)\cdot 2\alpha$. Level-$k$ actor v_j can then start the execution of its first phase at:

$$S_j(1) = (k-1)\cdot 2\alpha = (k-2)\cdot 2\alpha + 2\alpha = S_i(1) + 2\alpha.$$

Observe now that in the proof of Theorem 3.5.1, instead of 2α we could more precisely take $\alpha + S_i(\phi_i) - S_i(1)$, because the last production of an iteration of an actor v_i will happen $\alpha + \Delta_i(\phi_i)$ time units after the start of its first phase, at the latest. Given this and taking into account all predecessors of v_j, we can write:

$$S_j(1) = \max_{v_i \in prec(v_j)}\{S_i(1) + \alpha + \Delta_i(\phi_i)\}.$$

We are now interested in starting the first phase of v_j earlier, which means we search for $S_j(1) \leq \max_{v_i \in prec(v_j)}\{S_i(1) + \alpha + \Delta_i(\phi_i)\}$, and the earliest possible $S_j(1)$ can be at the time when the application starts, which is $t = 0$. This can be written as:

$$S_j(1) = \max_{v_i \in prec(v_j)} \{S_{i \to j}(1)\} \text{ where } S_{i \to j}(1) = t', t' \in [0, S_i(1) + \alpha + \Delta_i(\phi_i)].$$

A valid start time candidate $S_{i \to j}(1)$ must guarantee that the number of tokens available on channel $e_u = (v_i, v_j)$ at any time instant $t \geq t'$ is greater than or equal to the number of consumed tokens at the same instant such that v_j can be executed as a set of strictly periodic tasks. Here, we have two cases:

Case 1: $t' \geq S_i(1)$: In order to guarantee that v_j can fire its first phase at times $t = t', t' + T_j, \cdots, t' + \alpha$ and each other phase φ as early as possible at times $t = t' + \Delta_j(\varphi), t' + \Delta_j(\varphi) + T_j, \cdots, t' + \Delta_j(\varphi) + \alpha - T_j$, where $\Delta_j(\varphi) = \sum_{l=1}^{\varphi-1} C_j(l)$, t' must satisfy:

$$\forall k \in [0, \alpha + \Delta_i(\phi_i)] : \underset{[S_i(1), t'+k)}{\mathsf{prd}^{\mathsf{S}}}(v_i, e_u) \geq \underset{[t', t'+k]}{\mathsf{cns}^{\mathsf{S}}}(v_j, e_u). \tag{3.13}$$

Thus, a valid value of t' guarantees that once v_j starts, it always finds enough data to fire for one iteration.

Case 2: $t' < S_i(1)$: This case happens when v_j consumes zero tokens in the interval $[S_i(1), t']$ or there are initial tokens on the channel. It is sufficient to check the cumulative production and consumption over the interval $[S_i(1), S_i(1) + \alpha + \Delta_i(\phi_i)]$ because by time $t = S_i(1) + \alpha + \Delta_i(\phi_i)$ both v_i and v_j are guaranteed to have finished one iteration:

$$\forall k \in [0, \alpha + \Delta_i(\phi_i)] : \underset{[S_i(1), S_i(1)+k)}{\mathsf{prd}^{\mathsf{S}}}(v_i, e_u) \geq \underset{[t', S_i(1)+k]}{\mathsf{cns}^{\mathsf{S}}}(v_j, e_u). \tag{3.14}$$

By merging Equation (3.13) and Equation (3.14) and then selecting among valid start times t' the minimum one, we obtain Equation (3.12). Start times for the tasks corresponding to the actor phases other than the first phase are obtained by adding the WCET value of the previous phase to the derived start time of the previous phase, which is given by Equation (3.9). The start times derived in such a way enable the serialized execution of tasks corresponding to actor phases, when it is needed, by careful allocation and certain scheduling algorithms, which will be explained in more detail in Section 3.5.4. ■

Note that we derive by Lemma 3.5.2 the earliest start times assuming that the tokens production happens at the deadlines and the tokens consumption

happens at the beginning of the execution of each phase. In this case, the cumulative production and the cumulative consumption functions can be computed efficiently by:

$$\underset{[t_s,t_e)}{\mathsf{prd}^{\mathsf{S}}}(v_i,e_u)=\begin{cases}X_i^u\left(\left(\left\lfloor\frac{t_e-t_s}{T_i}\right\rfloor-1+\left\lfloor\frac{\Delta}{T_i}\right\rfloor\right)\cdot\phi_i+k_1\right) & \text{if } t_e-t_s\geq T_i\\ X_i^u(k_2) & \text{if } D_i\leq t_e-t_s\leq T_i\\ 0 & \text{if } t_e-t_s<D_i\end{cases} \tag{3.15}$$

with $\Delta=(t_e-t_s) \bmod T_i+T_i-D_i$, $k_1=\max_{l\in[1,\phi_i]}\{l:\Delta \bmod T_i\geq\sum_{\varphi=0}^{l-1}C_i(\varphi)\}$, $k_2=\max_{l\in[1,\phi_i]}\{l:t_e-t_s-D_i\geq\sum_{\varphi=0}^{l-1}C_i(\varphi)\}$, and $C_i(0)=0$.

$$\underset{[t_s,t_e]}{\mathsf{cns}^{\mathsf{S}}}(v_i,e_u)=\begin{cases}Y_i^u\left(\left\lfloor\frac{t_e-t_s}{T_i}\right\rfloor+k\right) & \text{if } t_e\geq t_s\\ 0 & \text{if } t_e<t_s\end{cases} \tag{3.16}$$

with $k=\max_{l\in[1,\phi_i]}\{l:(t_e-t_s) \bmod T_i\geq\sum_{\varphi=0}^{l-1}C_i(\varphi)\}$, and $C_i(0)=0$.

For example, the derived earliest start times for phases of actor v_2 in G, shown in Figure 2.1, are $S_2(1)=8$ and $S_2(2)=S_2(1)+C_2(1)=10$, as illustrated in Figure 3.1(b).

3.5.3 Deriving Channel Buffer Sizes

Equation (3.10) in Theorem 3.5.1 shows that ISPS has bounded buffer sizes b_u. These buffer sizes b_u are sufficient but not minimal. Therefore, we want to derive the minimum buffer sizes that guarantee periodic execution of tasks corresponding to actor phases.

We want to derive the minimum buffer size such that the derived buffer size is always valid regardless of when the actor phases are actually scheduled to produce/consume during its common period. Hence, we assume that the token production happens as early as possible (at the beginning of execution of each phase) and the token consumption happens as late as possible (at the deadlines).

Lemma 3.5.3. *For an acyclic CSDF graph G, the minimum buffer size b_u of a communication channel $e_u=(v_i,v_j)$ under ISPS is given by:*

$$b_u=\max_{k\in[0,\alpha+\Delta_j(\phi_j)]}\Big\{\underset{[S_i(1),\max\{S_i(1),S_j(1)\}+k]}{\mathsf{prd}^{\mathsf{B}}}(v_i,e_u)-\underset{[S_j(1),\max\{S_i(1),S_j(1)\}+k)}{\mathsf{cns}^{\mathsf{B}}}(v_j,e_u)\Big\}, \tag{3.17}$$

where $S_i(1)$ is the earliest start time of the first phase of a predecessor actor v_i, $\alpha = r_i T_i = r_j T_j$, $\Delta_j(\phi_j) = S_j(\phi_j) - S_j(1)$, $\mathsf{prd}^{\mathsf{B}}{}_{[t_s,t_e]}(v_i, e_u)$ is the number of tokens produced by v_i into channel e_u during the time interval $[t_s, t_e]$, and $\mathsf{cns}^{\mathsf{B}}{}_{[t_s,t_e)}(v_j, e_u)$ is the number of tokens consumed by v_j from channel e_u during the time interval $[t_s, t_e)$.

Proof. Equation (3.17) tracks the maximum cumulative number of unconsumed tokens on channel e_u during one iteration of v_i and v_j. We have two cases:

Case 1: $S_j(1) \geq S_i(1)$: Here we have two intervals $[S_i(1), S_j(1))$ and $[S_j(1), S_j(1) + \alpha + \Delta_j(\phi_j)]$. During the first interval only phases of actor v_i are executing, so tokens are only produced and buffer size should be large enough to accommodate all produced tokens in that interval. During the second interval phases of both actors execute in parallel. Thus, the minimum number of tokens that needs to be stored is given by the maximum number of unconsumed tokens on e_u at any time over this interval. At time $t = S_j(1) + \alpha + \Delta_j(\phi_j)$, both v_i and v_j have completed one iteration and the number of tokens on e_u is the same as at time $t = S_j(1) + \Delta_j(\phi_j)$ [BELP96]. Due to the periodicity of v_i and v_j, their execution pattern repeats. Thus, b_u given by Equation (3.17) is the minimum buffer size which guarantees periodic execution of v_i and v_j.

Case 2: $S_j(1) < S_i(1)$: Here we have three intervals $[S_j(1), S_i(1))$, $[S_i(1), S_j(1) + \alpha + \Delta_j(\phi_j)]$ and $(S_j(1) + \alpha + \Delta_j(\phi_j), S_i(1) + \alpha + \Delta_j(\phi_j)]$. During the first interval there is no production nor consumption or there are initial tokens on the channel, and hence b_u during that interval is equal to the number of initial tokens. During the second interval phases of both actors execute in parallel and b_u gives the maximum number of unconsumed tokens on e_u. During the third interval phases of actor v_j executes their second iteration, again either there is no consumption, which means that e_u has to accommodate all the tokens produced during this interval or there is consumption and b_u gives the maximum number of unconsumed tokens on e_u. At time $t = S_i(1) + \alpha + \Delta_j(\phi_j)$, both v_i and v_j have completed one iteration and the number of tokens on e_u is the same as at time $t = S_i(1) + \Delta_j(\phi_j)$ [BELP96]. Due to the periodicity of v_i and v_j, their execution pattern repeats. Thus, b_u given by Equation (3.17) is the minimum buffer size which guarantees periodic execution of v_i and v_j. ■

The cumulative production and consumption functions used for the calculation of buffer sizes under the assumption of the earliest token production

and the latest token consumption can be computed efficiently by:

$$\underset{[t_s,t_e)}{\mathsf{prd}^{\mathsf{B}}}(v_i,e_u)=\begin{cases}X_i^u\left(\left\lfloor\frac{t_e-t_s}{T_i}\right\rfloor+k\right) & \text{if } t_e\geq t_s\\ 0 & \text{if } t_e<t_s\end{cases} \tag{3.18}$$

with $k=\max_{l\in[1,\phi_i]}\{l:(t_e-t_s) \bmod T_i\geq\sum_{\varphi=0}^{l-1}C_i(\varphi)\}$, and $C_i(0)=0$.

$$\underset{[t_s,t_e)}{\mathsf{cns}^{\mathsf{B}}}(v_i,e_u)=\begin{cases}Y_i^u\left(\left(\left\lfloor\frac{t_e-t_s}{T_i}\right\rfloor-1+\left\lfloor\frac{\Delta}{T_i}\right\rfloor\right)\cdot\phi_i+k_1\right) & \text{if } t_e-t_s\geq T_i\\ Y_i^u(k_2) & \text{if } D_i\leq t_e-t_s\leq T_i\\ 0 & \text{if } t_e-t_s<D_i\end{cases} \tag{3.19}$$

with $\Delta=(t_e-t_s) \bmod T_i+T_i-D_i$, $k_1=\max_{l\in[1,\phi_i]}\{l:\Delta \bmod T_i\geq\sum_{\varphi=0}^{l-1}C_i(\varphi)\}$, $k_2=\max_{l\in[1,\phi_i]}\{l:t_e-t_s-D_i\geq\sum_{\varphi=0}^{l-1}C_i(\varphi)\}$, and $C_i(0)=0$.

For the example graph G given in Figure 2.1, the calculated buffer sizes in tokens are $[b_1,b_2,b_3]=[4,15,4]$.

3.5.4 Hard Real-Time Schedulability

We give now a theorem which summarizes the presented results for our improved strictly periodic scheduling (ISPS):

Theorem 3.5.2. *For an acyclic CSDF graph G, let $\mathcal{T}_G$ be a set of periodic task sets $\mathcal{T}_{v_i}$ such that $\mathcal{T}_{v_i}$ corresponds to $v_i\in\mathcal{V}$. $\mathcal{T}_{v_i}$ consists of ϕ_i periodic tasks given by:*

$$\tau_i(\varphi)=(S_i(\varphi),C_i(\varphi),D_i,T_i),\ 1\leq\varphi\leq\phi_i, \tag{3.20}$$

where $S_i(\varphi)$ is the earliest start time of a phase φ of actor v_i given by Equation (3.11) and Equation (3.9), $C_i(\varphi)$ is the WCET value of a phase φ given by Equation (3.2), D_i is the relative deadline, $\max_{1\leq\varphi\leq\phi_i}\{C_i(\varphi)\}\leq D_i\leq T_i$, and T_i is the period of $\mathcal{T}_{v_i}$ given by Equation (3.5). $\mathcal{T}_G$ is schedulable on m processors using a hard real-time scheduling algorithm A for periodic tasks if:

1. *A is partitioned Earliest Deadline First, partitioned Rate Monotonic, partitioned Deadline Monotonic or hierarchical global hard real-time scheduling algorithm,*
2. *$\mathcal{T}_G$ satisfies the schedulability test of A on m processors,*
3. *every communication channel $e_u\in E$ has a capacity of at least b_u tokens, where b_u is given by Equation (3.17).*

Proof. According to Theorem 3.5.1, the graph is converted into strictly periodic tasks. The task set $\mathcal{T}_{v_i}$ corresponding to an actor v_i should be scheduled in a way which preserves the dependency between the actor phases. The hard real-time scheduling algorithms which can do this are partitioned Earliest Deadline First (EDF), Rate Monotonic (RM) [LL73] and Deadline Monotonic (DM) [LW82], or hierarchical [HA06], [LB03]. In case of the partitioned algorithms, tasks which correspond to phases of an actor should be allocated to the same processor and scheduled by EDF or DM because the deadlines of the phases are in the same order as the phases themselves thereby, preserving the data-dependencies between the phases, or by RM fixed priority scheduler where ties should be broken in favor of jobs arrived earlier in a system. In hierarchical scheduling a set of tasks are grouped together and scheduled as a single entity, called server task or supertask. When the entity is scheduled, one of its tasks is selected to execute according to an internal scheduling policy. Hence, the supertasks/servers are scheduled globally, while the scheduling of the tasks within a supertask/server is done locally, that is, it is analogous to scheduling on uniprocessor. By grouping the tasks which correspond to phases of an actor with data-dependent phases into a supertask/server and scheduling them by a scheduler which preserves their order (for example, EDF) the synchronization problem of such dependent tasks is solved. ■

3.5.5 Performance Analysis

Once an acyclic CSDF graph has been converted to a set of strictly periodic tasks, the calculated task parameters S_i, C_i, D_i, and T_i, where S_i is the start time of τ_i, C_i is the WCET, D_i is the deadline of τ_i, and T_i is the task period, are used for performance analysis of the graph, that is, for analysis of the graph's throughput and latency.

Throughput Analysis under ISPS

The throughput of a graph G scheduled by ISPS is given by:

$$\mathcal{R}(G) = \frac{1}{\alpha} = \frac{1}{r_i \breve{T}_i}, v_i \in \mathcal{V}, \tag{3.21}$$

where $\breve{T}_i$ is calculated by Equation (3.5). Given that during one graph iteration every actor $v_i \in \mathcal{V}$ is executed q_i times, the throughput of each actor is calculated as:

$$\mathcal{R}_i = \frac{q_i}{\alpha} = \frac{\phi_i}{\breve{T}_i}, v_i \in \mathcal{V}. \tag{3.22}$$

Theorem 3.5.3. *For any acyclic CSDF graph G scheduled by ISPS, the throughput of the graph is never less than the graph throughput when G is scheduled by SPS.*

Proof. The throughput of a graph scheduled under SPS [BS13] is $1/\alpha^{\text{SPS}} = 1/(q_i T_i^{\text{SPS}})$, $v_i \in V$. If the same graph is scheduled under our ISPS, then its throughput is $1/\alpha^{\text{ISPS}} = 1/(r_i T_i^{\text{ISPS}})$, $v_i \in V$. By using Equation (3.1) and Equation (3.5) and denoting $u = \max_{v_j \in V}\{r_j \sum_{\varphi=1}^{\phi_j} C_j(\varphi)\}$ and $w = \max_{v_j \in V}\{q_j \max_{1 \le \varphi \le \phi_j}\{C_j(\varphi)\}\}$, we can write the relation which we want to prove, $\alpha^{\text{ISPS}} \le \alpha^{\text{SPS}}$, as follows:

$$\text{lcm}(\vec{r}) \left\lceil \frac{u}{\text{lcm}(\vec{r})} \right\rceil \le \text{lcm}(\vec{q}) \left\lceil \frac{w}{\text{lcm}(\vec{q})} \right\rceil. \tag{3.23}$$

We have that $u \le w$. Given that the least common multiple of positive integer numbers can be found using prime factorization, and the relation between vectors $\vec{r} = [r_1, \cdots, r_N]^T$ and $\vec{q} = \mathbf{\Phi} \cdot \vec{r} = [\phi_1 r_1, \cdots, \phi_N r_N]^T$, we have that $\text{lcm}(\vec{q})$ is divisible by $\text{lcm}(\vec{r})$.

Finally, to prove relation (3.23) we consider the following cases (with regard to divisibility by the corresponding *lcm* term):

Case 1: workloads u and w on both sides of Inequality (3.23) are divisible by the corresponding *lcm* terms. Then by removing the ceiling operation we obtain inequality $u \le w$, which always holds.

Case 2: u is divisible by $\text{lcm}(\vec{r})$, w is not divisible by $\text{lcm}(\vec{q})$. We can represent the ceiling operation on the right-hand side as $(w + \text{lcm}(\vec{q}) - w \bmod (\text{lcm}(\vec{q})))/\,\text{lcm}(\vec{q})$. In the worst case $w \bmod (\text{lcm}(\vec{q}))$ is equal to $\text{lcm}(\vec{q}) - 1$. By putting this into Inequality (3.23) we obtain $u \le w + 1$, which holds.

Case 3: u is not divisible by $\text{lcm}(\vec{r})$, w is divisible by $\text{lcm}(\vec{q})$ (also divisible by $\text{lcm}(\vec{r})$). We can represent u and w as $k_u \,\text{lcm}(\vec{r}) + u \bmod (\text{lcm}(\vec{r}))$ and $k_w \,\text{lcm}(\vec{r})$, respectively, for some integer constants k_u and k_w, $k_u < k_w$. We represent the ceiling operation as in *Case 2*, so Inequality (3.23) becomes $u + \text{lcm}(\vec{r}) - u \bmod (\text{lcm}(\vec{r})) \le w$. Now, by putting the k_u-representation of u and k_w-representation of w, the inequality becomes $k_u + 1 \le k_w$, which is true and thus, Inequality (3.23) holds.

Case 4: workloads on both sides of Inequality (3.23) are not divisible by the corresponding *lcm* terms. Similarly to *Case 2* and *Case 3*, we can represent the ceiling operation through the modulo operation. In the worst case, we have on the right-hand side the smallest possible value for the ceiling operation which is $(w+1)/\,\text{lcm}(\vec{q})$ and this value is divisible by both $\text{lcm}(\vec{q})$ and $\text{lcm}(\vec{r})$. In the worst case we have $u = w$, which means that u also needs only 1 unit to be rounded up to a value divisible by $\text{lcm}(\vec{r})$. Thus, Inequality (3.23) becomes $w + 1 \le w + 1$, which holds. ■

Latency Analysis under ISPS

The latency of G scheduled by ISPS is given by:

$$\mathcal{L}(G) = \max_{w_{\text{in}\to\text{out}} \in \mathcal{W}} \{S_{\text{out}}(g^C_{\text{out}}) + D_{\text{out}} - S_{\text{in}}(g^P_{\text{in}})\}, \tag{3.24}$$

where $\mathcal{W}$ is the set of all paths from any input actor v_{in} to any output actor v_{out}, and $w_{\text{in}\to\text{out}}$ is one path of the set. $S_{\text{out}}(g^C_{\text{out}})$ and $S_{\text{in}}(g^P_{\text{in}})$ are the earliest start times of the first phase of τ_{out} with non-zero token consumption (phase g^C_{out}) and the first phase of v_{in} with non-zero token production (phase g^P_{in}) on a path $w_{\text{in}\to\text{out}} \in \mathcal{W}$, respectively. D_{out} is the relative deadline of v_{out}.

From Equation (3.24) we can see that the latency of a graph depends on start times and deadlines of the graph's actors. Given that actor start times are dependent on deadlines (see Section 3.5.2), in order to reduce the latency we should reduce actor deadlines, that is, we should change the token production times. However, given that reducing the deadlines increases the number of processors required to schedule the graph, we are interested in selecting the deadlines which lead to required graph latency while the number of processors needed to obtain that latency is minimized. To select deadlines properly, we devise the solution approach presented in this section that formulates the problem of selecting task deadlines under a given latency constraint while the number of processors is minimized when a CSDF graph is converted to real-time periodic tasks by using our ISPS approach as a mathematical programming problem. In order to formulate our problem as a mathematical programming problem, we need to rewrite the start time computation in a proper form.

Lemma 3.5.4. *For an acyclic CSDF graph G, the earliest start time of the first phase of an actor $\tau_j \in V$, denoted $S_j(1)$, under ISPS is given by:*

$$S_j(1) = \begin{cases} 0 & prec(v_j) = \emptyset \\ \max_{v_i \in prec(v_j)} \{S_i(1) + (S^{min}_{i\to j}(1) - S^{min}_i(1) - MC_i) + D_i\} & prec(v_j) \neq \emptyset \end{cases} \tag{3.25}$$

where $prec(v_j)$ is the set of predecessors of v_j, $S_i(1)$, MC_i, and D_i are the earliest start time of the first phase, the maximum WCET (Definition 3.5.2), and deadline of the predecessor actor v_i, respectively. $S^{min}_i(1)$ is the earliest start time of the first phase of v_i given by Equation (3.11) when $D_k = MC_k, \forall v_k \in V$, and $S^{min}_{i\to j}(1)$ is given by Equation (3.12) when $D_k = MC_k, \forall v_k \in V$.

Proof. Let us consider an arbitrary channel $e_u = (v_i, v_j)$ in a CSDF graph $G = (\mathcal{V}, \mathcal{E})$. Actor v_j starts execution of its first phase after v_i has started and

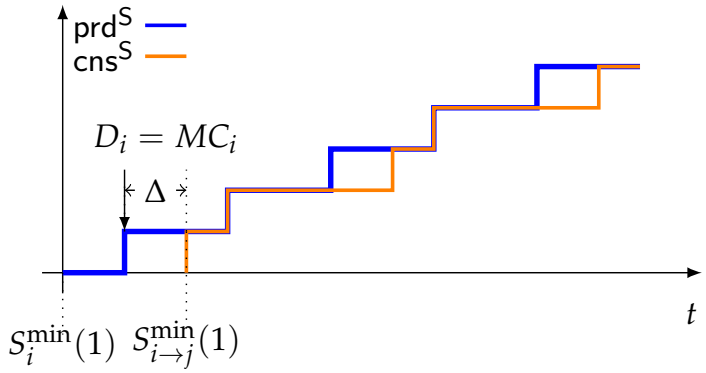


Figure 3.3: *Production and consumption curves on edge* $e_u = (v_i, v_j)$.

fired a certain number of times. This number of firings is independent from the execution speed of the actors and depends only on the production and consumption rates of v_i and v_j on e_u, where cumulative production and cumulative consumption functions are given by Equation (3.15) and Equation (3.16). Suppose that $D_k = MC_k, \forall v_k \in V$. The production ($\mathsf{prd}^{\mathsf{S}}$) and consumption ($\mathsf{cns}^{\mathsf{S}}$) curves of v_i and v_j are shown in Figure 3.3. Interval Δ in Figure 3.3 can be calculated as:

$$\Delta = S_{i\to j}^{\min}(1) - S_i^{\min}(1) - MC_i. \tag{3.26}$$

Now, suppose that $D_k > MC_k, \forall v_k \in V$. The production curve will move to the right for certain time units, and the new start time of the first phase of v_i is $S_i(1)$. If the consumption curve does not move, the relation between the production and consumption given by Equation (3.12) will be violated, that is, it will happen in some point in time that the cumulative consumption is greater than the cumulative production. This means that we have to move the consumption curve to the right by the same number of time units such that the new start time $S_{i\to j}(1)$ satisfies Equation (3.12). Hence, interval Δ will stay the same, and it is given by:

$$\Delta = S_{i\to j}(1) - S_i(1) - D_i. \tag{3.27}$$

By rewriting Equation (3.26) and Equation (3.27), we obtain:

$$S_{i\to j}(1) = S_i(1) + (S_{i\to j}^{\min}(1) - S_i^{\min}(1) - MC_i) + D_i. \tag{3.28}$$

■

We can derive from Equation (3.25) the following set of linear inequality constraints, where the number of the linear inequality constraints is equal to

the number of edges in the CSDF:

$$S_i(1) + (S^{\min}_{i\to j}(1) - S^{\min}_i(1) - MC_i) + D_i \leq S_j(1), \forall e_u \in \mathcal{E}. \tag{3.29}$$

In addition, we can rewrite Equation (3.24) as follows:

$$\mathcal{L}(G) = \max_{w_{\text{in}\to\text{out}}\in\mathcal{W}} \{S_{\text{out}}(1) + \sum_{k=1}^{g^C_{\text{out}}-1} C_{\text{out}}(k) + D_{\text{out}} - S_{\text{in}}(1) - \sum_{k=1}^{g^P_{\text{in}}-1} C_{\text{in}}(k)\}. \tag{3.30}$$

Since the number of processors needed to schedule constrained-deadline periodic (CDP) tasks depends on the total density δ_{sum} of the tasks [DB11], our objective is to minimize δ_{sum} in order to minimize the number of processors. Therefore, we formulate our optimization problem as follows:

$$\text{Minimize} \quad \delta_{\text{sum}} = \sum_{v_k\in V} \frac{AC_k}{D_k} \tag{3.31a}$$

$$\text{subject to:} \quad S_{\text{out}}(1) + D_{\text{out}} - S_{\text{in}}(1) \leq \mathcal{L} - \sum_{k=1}^{g^C_{\text{out}}-1} C_{\text{out}}(k) + \sum_{k=1}^{g^P_{\text{in}}-1} C_{\text{in}}(k), \quad \forall w_{\text{in}\to\text{out}} \in \mathcal{W} \tag{3.31b}$$

$$S_i(1) + D_i - S_j(1) \leq -(S^{\min}_{i\to j}(1) - S^{\min}_i(1) - MC_i), \quad \forall e_u \in \mathcal{E} \tag{3.31c}$$

$$-D_k \leq -MC_k, D_k \leq T_k, \quad \forall v_k \in \mathcal{V} \tag{3.31d}$$

where (3.31a) is the objective function and D_k is an optimization variable. The objective function (3.31a) has $|V|$ optimization variables and is subject to a latency constraint $\mathcal{L}$. Therefore, (3.31b) comes from (3.30). For each channel in a graph we have Equation (3.29), which can be rewritten as (3.31c). In addition, (3.31d) bounds all optimization variables in the objective function. $S_i(1)$ and $S_j(1)$ (including $S_{\text{in}}(1)$, $S_{\text{out}}(1)$) are implicit variables which are not in the objective function (3.31a), but still need to be considered in the optimization procedure. $\mathcal{L}$, g^P_{in}, g^C_{out}, $S^{\min}_{i\to j}(1)$, $S^{\min}_i(1)$, MC_k, and T_k are constants. Given that all variables are integers and both the objective function and the constraints are convex, problem (3.31) is an integer convex programming (ICP) problem [LSZ+14] which can be solved by using existing convex programming solvers, such as CVX solver [GB14].

Algorithm 2: Procedure to derive the number of processors.

Input: A CSDF graph $G = (\mathcal{V}, \mathcal{E})$, a partitioned scheduling algorithm A, an allocation heuristic $\mathcal{H}$.
Output: Number of processors m_{PAR}, task allocation *alloc*.

1 **for** *actor* v_i *in* $\mathcal{V}$ **do**
2 Compute the minimum common period $\check{T}_i$ by using Equation (3.5);
3 $u_{\text{total}} = 0$;
4 $U \leftarrow \emptyset$; (the set of allocation units, initially empty)
5 **for** *actor* $v_i \in \mathcal{V}$ **do**
6 $u_i = 0$;
7 **for** *phase* φ *of* v_i, $1 \leq \varphi \leq \phi_i$ **do**
8 $u_i(\varphi) = \frac{C_i(\varphi)}{\check{T}_i}$;
9 $u_i = u_i + u_i(\varphi)$;
10 $u_{\text{total}} = u_{\text{total}} + u_i(\varphi)$;
11 $U = U \cup u_i$;
12 $m_{\text{PAR}} = m_{\text{OPT}} = \lceil u_{\text{total}} \rceil$;
13 Reorder elements of U if required by an allocation heuristic $\mathcal{H}$;
14 **for** $u \in U$ **do**
15 $\Pi = \{\pi_1, \pi_2, \cdots, \pi_{m_{\text{PAR}}}\}$;
16 Apply bin-packing allocation heuristic $\mathcal{H}$ to u on $\pi_j \in \Pi$ and check the schedulability test of algorithm A on π_j;
17 **if** u *is not allocated to any* $\pi_j \in \Pi$ **then**
18 Allocate u on a new processor $\pi_{m_{\text{PAR}}+1}$;
19 $m_{\text{PAR}} = m_{\text{PAR}} + 1$;
20 **return** m_{PAR}, *alloc*;

3.5.6 Deriving the Number of Processors

As introduced in Section 2.2.5, by using Equation (2.12) one can compute the absolute minimum number of processors m_{OPT} needed to schedule the tasks with deadlines equal to the periods. The tasks can be scheduled on m_{OPT} if an optimal scheduling algorithm is used. The optimal scheduling algorithms are either global or hybrid, and hence, they require task migration. On the other hand, the partitioned scheduling algorithms do not require task migration. In that case the tasks are first allocated to the processors, for example by using a task partitioning heuristic, as described in Section 2.2.5, and then the tasks on each processor are scheduled using a uniprocessor scheduling algorithm.

The procedure to calculate the number of processors required for the partitioned scheduling of the task set obtained by the conversion procedure described in Section 3.5 (see Algorithm 1) is given in Algorithm 2. Algorithm 2 takes as inputs a CSDF graph G, a partitioned scheduling algorithm A

and an allocation heuristic $\mathcal{H}$. The minimum common period for each actor is calculated in lines 1-2 of the algorithm. Once the periods are calculated, then the total utilization of the converted task set and the utilization per task set corresponding to an actor are calculated in lines 3-10. Line 11 in Algorithm 2 ensures that the task set corresponding to an actor is considered as one scheduling entity, that is, one allocation unit. The absolute minimum number of processors m_{OPT} for scheduling the tasks is computed in line 12. Some allocation heuristics require a preprocessing step to be performed on the tasks before applying the heuristic. This preprocessing step is usually sorting the tasks based on some criteria, such as their utilization. That step is done in Algorithm 2 in line 13. The following lines find the number of processors and the allocation of tasks to processors. Given that m_{OPT} is the lower bound on the number of processors m_{PAR} needed by partitioned scheduling algorithms, Algorithm 2 starts with the task partitioning on m_{OPT} processors. If the tasks pass the schedulability test on all m_{PAR} processors, for example, in the case of IDP tasks and EDF scheduler the utilization of the tasks allocated to a processor is not greater than 1, then the algorithm returns m_{PAR} and the corresponding allocation of the tasks to the processors *alloc*.

Let us now analyze the time complexity of Algorithm 2 in the worst case. The first **for loop** in lines 1-2 takes linear time to calculate the minimum common period of each actor, that is, its time complexity is $O(|\mathcal{V}|)$. The second **for loop** in lines 5-11 has a nested for loop and hence, its time complexity in the worst case is given by $O(|\mathcal{V}|\phi)$, where ϕ is the maximum number of execution phases per actor, $\phi = \max_{v_i \in \mathcal{V}}\{\phi_i\}$. If the task sorting in line 13 should be performed prior to performing the task allocation, it will have $O(|\mathcal{V}|\phi \log(|\mathcal{V}|\phi))$ time complexity given that the maximum number of tasks is $|\mathcal{V}|\phi$. The **for loop** in lines 14-19 implements the allocation of the tasks to the processors by applying certain allocation heuristic and scheduling algorithm. Given that the maximum number of tasks is $|\mathcal{V}|\phi$ and the maximum number of processors needed to allocate and schedule an CSDF graph is equal to the number of actors in the graph $|\mathcal{V}|$, the time complexity of finding the number of processors m_{PAR} and the feasible task allocation is $O(|\mathcal{V}|\phi \log |\mathcal{V}|)$ [PZMA04], [BF05]. Thus, we can conclude that the running time of Algorithm 2 is polynomial and its complexity is $O(|\mathcal{V}|\phi \log |\mathcal{V}|)$ or $O(|\mathcal{V}|\phi \log(|\mathcal{V}|\phi))$ if the preprocessing step is performed.

Table 3.2: *Benchmarks used for evaluation.*

Domain	Benchmark	$\lvert\mathcal{V}\rvert$	$\lvert\mathcal{E}\rvert$	$\lvert\mathcal{T}\rvert$	Source
Medical	Heart pacemaker	4	3	67	[PMN+09]
Communication	Reed Solomon Decoder (RSD)	6	6	904	[BMMKM10]
Financial	BlackScholes	41	40	261	[BMKdD13]
Computer Vision	Disparity map	5	6	11	[ZK00]
	Pdetect	58	76	4045	[BMKdD13]
Audio processing	CELP algorithm	9	10	167	[BELP96]
	CD2DAT rate converter	6	5	22	[OH04]
	MP3 Playback	4	3	8	[WBJS07]
Image processing	JPEG2000	240	703	639	[BMKdD13]

3.6 Evaluation

We evaluate our approach in terms of its performance and time complexity by performing experiments on the benchmarks given in Table 5.2. Columns 3, 4 and 5 in Table 5.2 give for each benchmark the number of actors $|\mathcal{V}|$, the number of channels $|\mathcal{E}|$ in the corresponding CSDF graph of a benchmark, and the number of periodic tasks $|\mathcal{T}|$ obtained after converting the actors of the CSDF graph by our approach to a set of periodic tasks $\mathcal{T}$. The WCETs of actors in the benchmarks are given in clock cycles [BMKdD13] or in time units [BMMKM10], [WBJS07]. If the execution times of a benchmark are not given [BELP96], [PMN+09], [OH04], certain values based on a static analysis are assumed. The execution times of benchmark [ZK00] are obtained from the measurements of the benchmark running on a MicroBlaze processor.

Our approach is evaluated by comparison to 3 related scheduling approaches - *strictly periodic scheduling*, SPS, proposed in [BS13], *periodic scheduling*, PS, presented in [BMKdD13], and *self-timed scheduling*, STS, given in [SGB08]. We implemented our approach in Python. The SPS approach was implemented in Python within the *darts* tool-set [Bam12]. The approach in [SGB08] was implemented in C++ within the SDF^3 tool-set [SGB06]. In addition, we implemented the approach in [BMKdD13] in Python as well. We formulated both LP problems [BMKdD13] for finding the period of a graph, and for finding the start times and the buffer sizes as integer linear programming (ILP) problems, and we added the constraint that the periods of all actors in a graph have to be integers. We used CPLEX Optimization Studio [IBM12] to solve the ILP problems and mixed integer disciplined convex programming (MIDCP) in CVX [GB14] to solve our latency reduction problem. We have run all the experiments on a Dell PowerEdge T710 server running Ubuntu 11.04 (64-bit) Server OS.

3.6.1 Performance of the ISPS Approach

The main objective of the evaluation is to compare the throughput of streaming applications and the required number of processors to guarantee the throughput when scheduled by our ISPS with the throughput and the number of processors under SPS [BS13], PS [BMKdD13] and STS [SGB08]. In addition, we compare our ISPS and the other scheduling approaches in terms of application latency and memory resources needed to implement the communication channels.

We used the `sdf3analysis-csdf` tool from SDF^3 [SGB06] to obtain the maximum achievable throughput of a graph, which is the throughput under STS, and to compute the minimum buffer sizes required to achieve that throughput. Unfortunately, the `sdf3analysis-csdf` tool does not support the latency calculation and the calculation of the number of processors. Thus, we were not able to compare them with our approach. We were also not able to obtain the number of processors for a graph scheduled under PS, because the calculation of the number of processors was not considered in [BMKdD13].

Results of the performance evaluation are given in Table 3.3. We report the throughput of the output actors under ISPS, calculated by Equation (3.22), in the second column of Table 3.3. Here t.u. denotes the corresponding time unit of a benchmark. Columns 7, 12 and 15 show the ratio between the throughput of the output actors under our ISPS and SPS, PS and STS, respectively. Given that the main objective of this experiment is to evaluate the throughput of the benchmarks scheduled under ISPS and the minimum number of processors needed to obtain that throughput, our ISPS approach converts the CSDF graphs of the benchmarks to IDP tasks, which minimizes the number of processors required to schedule the benchmarks. For processor requirements in case of ISPS and SPS, we compute the minimum number of processors for IDP tasks under optimal and partitioned First-Fit Decreasing (Utilization) EDF (FFD-EDF) schedulers by using Equation (2.12) and Algorithm 2 for ISPS, and Equation (2.12) and Equation (2.16) for SPS - see columns 4, 5, 9 and 10. By comparing the throughputs under ISPS and SPS, we can see that for the majority of the benchmarks the throughput under our ISPS is higher than the corresponding throughput under SPS. Only in two cases the throughputs are the same for both schedules. The first case is MP3 Playback, which bottleneck actor (the actor with the biggest workload over one iteration period) is the same under both SPS and ISPS, and that actor has only one phase, so the influence of different WCET for actor phases on throughput cannot be seen. However, the influence can be seen from the required number of processors needed for scheduling of MP3 Playback by optimal schedulers,

which is smaller in the case of our ISPS. The second case is CD2DAT. For this benchmark lcm($\vec{q}$) and lcm($\vec{r}$) are equal and much higher than the maximum workload of actors over an iteration period for both SPS and ISPS, which leads to the same iteration period for both schedules. However, the WCET-awareness of ISPS leads to smaller number of processors. Note that if we want to schedule a task-set on smaller number of processors than the one calculated by Equation (2.12) or Equation (2.12)/Algorithm 2, we should scale up the computed actor periods by the same scaling factor [ZBS13]. Hence, to schedule CD2DAT by SPS on the same number of processors required by ISPS, we need to scale up actor periods by 2, which will lead to decrease in throughput by 2. Thus, ISPS outperforms SPS in terms of throughput when CD2DAT is scheduled on 1 processor. Benchmarks JPEG2000 and RSD can achieve much better throughput when scheduled under ISPS, but in that case they require larger number of processors to be scheduled. Note that the throughputs of these two benchmarks cannot be increased under SPS even when the number of processors is increased. If we apply the period scaling technique [ZBS13] for these two benchmarks to schedule them under ISPS on the same number of processors as required under SPS the throughput values for JPEG2000 and RSD under our ISPS are 3.93 and 11.2 times higher, as given in column 7 in parenthesis, than the corresponding values under SPS. Therefore, we can conclude that in all cases the minimum number of processors required to guarantee certain throughput under our ISPS is smaller than or equal to the minimum number of processors under SPS while the throughput under ISPS is increased in most cases, thus, processors are better utilized.

Column 12 in Table 3.3 shows the ratio of the maximum throughput of the output actors achieved by our ISPS to the maximum throughput of the output actors achieved by PS. We can see that both approaches give the same throughput for all benchmarks, which is expected given that PS schedules phases of an actor in a CSDF graph statically within a period of the actor, hence the scheduling granularity is similar between these two approaches.

Table 3.3 shows in column 15 the ratio of the maximum throughput of the output actors achieved by our approach to the absolute maximum throughput of the output actors achieved by self-timed scheduling of actor firings, which is the optimal scheduling in terms of throughput. We can see that the throughput under ISPS is equal or very close to the throughput under STS for the majority of the benchmarks. Differences in the throughput appear as a result of the ceiling operation during the calculation of actor common periods in Equation (3.5). The biggest difference is in the case of the CD2DAT benchmark. For this benchmark lcm($\vec{r}$) is much higher than the maximum

workload of actors over an iteration period, and thus, the calculated actor periods are underutilized, which leads to lower throughput. The throughput value N/A for JPEG2000 indicates that the SDF^3 tool-set [SGB06] returned an infeasible throughput (most likely related to an integer overflow).

Let us now analyze the latency and the memory resources needed to implement the communication channels of the benchmarks. The graph latency under our ISPS is calculated by Equation (3.24) for IDP tasks and shown in column 3 of Table 3.3. Column 8 shows the ratio between the graph latency under our ISPS and SPS. As we can see from columns 4, 5, 7-10 in Table 3.3: for 4 benchmarks (highlighted in the table) under ISPS we obtain higher throughput and smaller latency than under SPS without increasing the number of processors (with JPEG2000 and RSD scheduled on the same number of processors as in case of the SPS); for the other 3 benchmarks (BlackScholes, Disp. map, Pdetect) the obtained increase in throughput is less than the increase in latency on a platform with the same (or 1 less for BlackScholes under ISPS, partitioned scheduling) number of processors; for the rest 2 benchmarks we obtained the same throughput with the increase in latency, but also with the decrease in the number of processors. For the tested benchmarks, the calculated buffer sizes under ISPS are never smaller than the buffer sizes under SPS, see column 11 in Table 3.3. The highest ratio in buffer sizes between ISPS and SPS is obtained for BlackScholes and CD2DAT. However, the actual increase in communication memory resources is 215 KB and less than 1 KB, respectively, which is acceptable given the size of the memory available in modern embedded systems. Note that both latency and buffer sizes under our ISPS can be reduced by carefully selecting deadlines for individual actors (actors phases). This will be shown later in Section 3.6.3.

Column 13 gives the ratio of the maximum latency of benchmarks under our ISPS to the latency of benchmarks under PS. Although [BMKdD13] does not provide the latency calculation for their PS, we were able to extract the latency information from the start times obtained by solving the ILP problem. However, for benchmarks JPEG2000 and Pdetect we could not get a solution from the ILP solver after more than 1 day, so we could not calculate the latency for these two benchmarks. As we can see, the latency of benchmarks under ISPS is always larger than the latency under PS. As mentioned above, reducing the latency under ISPS can be done by carefully selecting deadlines for individual actors (actors phases), as shown in Section 3.6.3. Moreover, ISPS reports the maximum latency while PS reports the actual latency under a certain schedule. The ratio of the calculated buffer sizes under ISPS to the calculated buffer sizes under PS and STS is given in columns 14 and 16, respectively.

Table 3.3: *Comparison of different scheduling approaches.*

Benchmark	ISPS					SPS					PS			STS	
	$\mathcal{R}_{out}^{ISPS}[\frac{1}{t.u.}]$	$\mathcal{L}^{ISPS}$[t.u.]	m_{OPT}^{ISPS}	m_{PAR}^{ISPS}	M^{ISPS}[B]	$\frac{\mathcal{R}_{out}^{ISPS}}{\mathcal{R}_{out}^{SPS}}$	$\frac{\mathcal{L}^{ISPS}}{\mathcal{L}^{SPS}}$	m_{OPT}^{SPS}	m_{PAR}^{SPS}	$\frac{M^{ISPS}}{M^{SPS}}$	$\frac{\mathcal{R}_{out}^{ISPS}}{\mathcal{R}_{out}^{PS}}$	$\frac{\mathcal{L}^{ISPS}}{\mathcal{L}^{PS}}$	$\frac{M^{ISPS}}{M^{PS}}$	$\frac{\mathcal{R}_{out}^{ISPS}}{\mathcal{R}_{out}^{STS}}$	$\frac{M^{ISPS}}{M^{STS}}$
Pacemaker	1/10	1920	2	2	436	1.5	0.99	2	2	1.47	1	2.93	4.95	0.91	5.07
RSD	1/1080 (1/2160)	6295 (11695)	2 (1)	2 (1)	5205 (5460)	22.4 (11.2)	0.05 (0.097)	1	1	1.56 (1.63)	1	2.8	3.23	0.83	–
BlackScholes	1/3234876	24764218	16	16	260284	1.33	1.58	16	17	6.41	1	5.31	11.57	1	–
Disp. map	1/65326	382593	2	2	995520	1.03	1.13	2	2	1	1	3.18	2	1	2
Pdetect	1/2033760	36608557	11	13	13464910	1.0002	1.12	11	13	1.26	1	–	–	1	–
CELP	1/2	964	6	6	1780	1.5	0.99	6	6	1.68	1	2.24	2.38	1	–
CD2DAT	1/147	2637	1	1	116	1	3.18	2	2	4.83	1	8.88	11.6	0.17	5.09
MP3 Playback	1/25	46355	3	4	3860	1	1.84	4	4	1.48	1	2.02	1.76	0.91	1.66
JPEG2000	1/811008 (1/14598144)	27255343 (497471535)	18 (1)	18 (1)	9625878 (10006530)	70.65 (3.93)	0.02 (0.3)	1	1	1.17 (1.21)	1	–	–	N/A	N/A

Table 3.4: *Time complexity (in seconds) of different scheduling approaches.*

Benchmark	ISPS		SPS		PS		STS	
	$t_{\mathcal{R}}^{ISPS}$	$t_{S\&B}^{ISPS}$	$t_{\mathcal{R}}^{SPS}$	$t_{S\&B}^{SPS}$	$t_{\mathcal{R}}^{PS}$	$t_{S\&B}^{PS}$	$t_{\mathcal{R}}^{STS}$	$t_{S\&B}^{STS}$
Pacemaker	1.24e-05	0.056	1.31e-05	0.007	0.19	0.34	0.004	1.52
RSD	1.62e-05	4	1.74e-05	3.3	115.11	146.66	0.06	>1 day
BlackScholes	9.7e-05	1.13	9.46e-05	0.43	0.28	1.22	0.05	>1 day
Disp. map	1.36e-05	0.0014	1.69e-05	0.00087	0.027	0.055	0.004	0.01
Pdetect	0.00014	3.52	0.00013	0.65	83.64	>1 day	0.33	>1 day
CELP	2.26e-05	0.097	2.43e-05	0.029	0.56	0.95	0.01	>1 day
CD2DAT	1.67e-05	0.59	1.76e-05	0.66	0.061	0.17	0.004	108.56
MP3 Playback	1.41e-05	59.07	1.37e-05	55.87	0.021	0.034	0.004	3236.31
JPEG2000	0.00053	27.22	0.00053	3.55	0.51	>1 day	N/A	N/A

Table 3.5: *Time complexity (in seconds) for the calculation of number of processors.*

Benchmark	$t_{m_{OPT}}^{ISPS}$	$t_{m_{PAR}}^{ISPS}$
Pacemaker	4.51e-05	0.00095
RSD	0.00049	0.012
BlackScholes	0.00017	0.0077
Disp. map	1.19e-05	0.00037
Pdetect	0.0028	0.2
CELP	0.0001	0.0029
CD2DAT	1.72e-05	0.0021
MP3 Playback	9.06e-06	0.00039
JPEG2000	0.00048	0.42

Again, for benchmarks JPEG2000 and Pdetect under PS we could not get a solution from the ILP solver after more than 1 day. Similarly, for benchmarks RSD, BlackScholes, Pdetect and CELP under STS we could not get a solution for longer than 1 day. As mentioned before, value N/A for JPEG2000 indicates that SDF^3 tool-set returned an infeasible throughput, and hence the buffer sizes were not calculated. As we can see, the buffer sizes under PS and STS are always smaller than the buffer sizes under ISPS. The highest ratio in buffer sizes between ISPS and PS is obtained for BlackScholes and CD2DAT, with the actual increase in communication memory resources of 232 KB and less than 1 KB, respectively. The highest increase in buffer sizes under ISPS when compared to STS is less than 1 KB. The reason for the difference in the buffer sizes is that in both PS and STS approaches it is assumed that the production of tokens happens at the end of the actor firing, while the consumption happens at the start of the firing, while in our case (and in SPS case) the worst-case scenario is considered, that is, the production of tokens happens at the earliest possible start of the actor firing (at start times), while the consumption happens at the latest possible end of actor firing (at deadlines). Note that in an implementation of a dataflow application, data may be consumed from input channels and produced to output channels at arbitrary points in time during an actor firing. To guarantee that buffer overflow/underflow does not occur, buffer sizes have to be sufficiently large. Thus, the assumption in PS and STS limits the actual implementation of reading and writing of tokens, while the buffers calculated in our case are valid regardless of the actual point in time where reading and writing of tokens happens and thus, our approach does not limit the implementation of the reading and writing of tokens. Moreover, the buffer sizes calculated in PS and STS are valid for that specific schedule and the specific production/consumption pattern, while in the case of our ISPS the computed buffer sizes are valid for any schedule of actor firings during its period and for any production/consumption pattern during its firing.

3.6.2 Time Complexity of the ISPS Approach

In this section, we evaluate the efficiency of our ISPS approach in terms of the execution time of our algorithms to calculate the throughput of an application, and to find a schedule and buffer sizes of communication channels. The execution times are given in Table 3.4. We compare these execution times with the corresponding execution times of related approaches – SPS, PS and STS.

Let us first analyze the time needed to calculate the throughput of an application. The execution times needed to find the application throughput under ISPS, SPS, PS and STS are given in columns 2, 4, 6 and 8, respectively.

As we can see, the times spent on calculating the throughput of an application under ISPS and SPS are similar and much shorter than the time needed for solving the ILP problem to find the application throughput under PS and the time spent on finding the maximum achievable throughput of the application, that is, the throughput under STS. Thus, our approach outperforms PS and STS in terms of time required to calculate the throughput of an application. Given that in most cases ISPS gives higher throughput of an application than SPS within almost the same time, we can say that ISPS outperforms SPS as well.

Next, we compare the time needed to derive the start times of actor firings, that is, the schedule, and the buffer sizes of communication channels. These times are given in columns 3, 5, 7 and 9, for ISPS, SPS, PS and STS, respectively. By comparing the times under ISPS and SPS, we can see that both approaches find the start times and the buffer sizes within less than 4 seconds in most cases, and within a minute in two cases. Then, we compare ISPS with PS. In all but two cases ISPS is faster than PS. For those two cases (CD2DAT and MP3 Playback), the ILP problems for PS are not complex and hence they can be solved very fast. As shown in Table 3.4, ISPS gives a solution for those two cases within a second, and within a minute. On the other hand, for benchmarks Pdetect and JPEG2000 we could not get a solution from the ILP solver for PS after more than a day, while our ISPS produced the results in a couple of seconds and within a minute. By comparing to STS, our ISPS approach is always much faster. Moreover, for 4 benchmarks, we were not able to get the solution for the buffer sizing problem under STS after more than a day.

We report in Table 3.5 the execution time for calculating the minimum number of processors needed to temporally schedule the tasks, obtained by the conversion of an application by using our ISPS approach, under global optimal and partitioned FFD-EDF schedulers. In the case of global optimal scheduling, the minimum number of processors is calculated by Equation (2.12), while the calculation procedure for FFD-EDF partitioned scheduling is presented in Algorithm 2 in Section 3.5.6. As we can see, the number of processors in the case of optimal scheduling can be calculated within a millisecond for most of the benchmarks, while in the case of partitioned scheduling the calculation is done within less than 12 milliseconds for most cases and within less than 420 milliseconds in two cases. Thus, the calculation of the number of processors required to schedule an application under our ISPS is very efficient. We obtained similar times for the calculation of the number of processors under SPS and global and partitioned FFD-EDF schedulers. We could not numer-

ically compare the time complexity of our approach with regard to the PS approach because the calculation of the number of processors was not considered in [BMKdD13]. As mentioned already in Section 3.3, one possible way to find the minimum number of processors under PS is to trace the schedules but that procedure has an exponential time complexity in the worst case, whereas our Algorithm 2 for finding the minimum number of processors under ISPS has a polynomial time complexity, see Section 3.5.6. Finding the minimum number of processors under STS requires complex Design Space Exploration (DSE) procedures, with an exponential time complexity in the worst case, to find the best allocation which delivers the maximum achievable throughput. The SDF^3 tool-set used to compute the self-timed scheduling parameters does not support such design space exploration for self-timed scheduling. Thus, we could not numerically compare the time complexity of ISPS with the time complexity of STS. However, given that ISPS finds the minimum number of processors for scheduling an application in polynomial time in the worst case, as shown in Section 3.5.6, we can conclude that our ISPS is faster than STS.

3.6.3 Reducing Latency under ISPS

We have shown in the previous experiments that when compared to the SPS approach our ISPS delivers in 5 out of 9 cases larger graph latency. When compared to the PS approach, our ISPS approach always results in a graph schedule with larger graph latency. If we want to reduce graph latency under ISPS we could use the latency reduction method presented in Section 3.5.5. We would like to see how close we are in graph latency in comparison to the SPS and PS approaches after applying our latency reduction method. Therefore, in this section, we present results obtained after applying our latency reduction method introduced in Section 3.5.5 on the benchmarks given in Table 5.2. The results are given in Table 3.6. In order to apply our latency reduction method, we should set a latency constraint. To compare our ISPS approach to the SPS approach, we set the latency constraint to be equal to the graph latency obtained under SPS, $\mathcal{L}^{\mathsf{SPS}}$, and we apply our method for latency reduction. We can see from column 3 in Table 3.6 that we significantly reduce latency for the benchmarks that had higher latency under ISPS than SPS, see column 8 in Table 3.3, and that we were able to meet the latency constraint $\mathcal{L}^{\mathsf{SPS}}$ for all the benchmarks. Moreover, we see that reduction in graph latency does not influence the graph throughput, that is, the ratio of the graph throughput under ISPS to the graph throughput under SPS in column 2 is the same as the corresponding ratio given in column 7 in Table 3.3 with the period scaling technique applied for benchmarks RSD and JPEG2000 under ISPS. Columns

4 to 6 give the results on resources in terms of the number of processors required by ISPS and SPS, and the ratio between ISPS and SPS approach in buffer sizes needed to implement communication channels in a graph. We find the minimum number of processors under partitioned First-Fit Increasing Deadlines EDF (FFID-EDF) [BF05] scheduler by using Algorithm 2 for ISPS, and Equation (2.16) for SPS and FFD-EDF scheduler. We can see from columns 2 to 5 that our ISPS approach with our latency reduction method is able to schedule almost all benchmarks on the same number of processors as the SPS approach, while obtaining better graph throughput and shorter graph latency. Only in one case, for benchmark BlackScholes, our approach needs one processor more than the SPS approach. However, our approach delivers better throughput for benchmark BlackScholes than the SPS. Although the ratio between the buffer sizes under ISPS and the buffer sizes under SPS, given in column 6 in Table 3.6, is smaller than the corresponding ratio in Table 3.3, column 11, the buffer sizes under ISPS are still always bigger than the corresponding buffer sizes under SPS.

Columns 7 to 10 give the results when our latency reduction method is applied with the latency constraint dictated by the PS approach, $\mathcal{L}^{\mathsf{PS}}$. Since we could not obtain the solution from the ILP solver in the case of the PS approach after 1 day for Pdetect and JPEG2000 benchmarks – see Table 3.3, we could not provide latency and buffer sizes ratios for these two benchmarks. We can see from column 8 in Table 3.6 that we significantly reduce the latency for all benchmarks, see column 13 in Table 3.3. However, in four cases, for benchmarks BlackScholes, CELP, CD2DAT, and MP3 playback, our latency reduction method was not able to meet the latency constraint $\mathcal{L}^{\mathsf{PS}}$. The reason is that the PS approach gives the actual latency under a static schedule while our ISPS approach calculates the maximum latency for a CSDF graph converted into real-time periodic tasks. For these 4 benchmarks, column 8 gives the shortest achievable latency under ISPS obtained by applying our latency reduction method. The ratio of the graph throughput under ISPS to the graph throughput under PS is given in column 7 and it is the same as the corresponding ratio given in column 12 in Table 3.3. We report in column 9 the minimum number of processors under ISPS and FFID-EDF found by Algorithm 2. We can see that the number of processors needed by all the benchmarks with reduced latency under ISPS is higher than the corresponding number of processors given in Table 3.3, column 5, which is expected. The number of processors for a graph scheduled under PS is not given because the calculation of the number of processors was not considered in [BMKdD13]. Although the ratio between the buffer sizes under ISPS and

the buffer sizes under PS, given in column 10 in Table 3.6, is smaller than the corresponding ratio in Table 3.3, column 14, the buffer sizes under ISPS are still always bigger than the buffer sizes under PS. As explained previously, the reason for the difference in the buffer sizes is that the PS approach considers specific schedule and the specific production/consumption pattern, while in the case of our ISPS the computed buffer sizes are valid for any schedule of actor firings during its periods and for any production/consumption pattern during its firing.

We also measured the execution times of our ISPS approach enhanced with the latency reduction method to find tasks' deadlines and a schedule, that is, tasks' start times, such that the latency constraint is satisfied. In most cases our latency reduction method needed less than a second, and in three cases less than a minute, to find tasks' deadlines and a schedule which meets the latency constraint.

Table 3.6: *Performance of the ISPS approach under different latency constraints.*

Benchmark	$\mathcal{L}_{\text{constraint}} = \mathcal{L}^{\text{SPS}}$					$\mathcal{L}_{\text{constraint}} = \mathcal{L}^{\text{PS}}$			
	$\frac{\mathcal{R}_{\text{out}}^{\text{ISPS}}}{\mathcal{R}_{\text{out}}^{\text{SPS}}}$	$\frac{\mathcal{L}^{\text{ISPS}}}{\mathcal{L}^{\text{SPS}}}$	$m_{\text{PAR}}^{\text{ISPS}}$	$m_{\text{PAR}}^{\text{SPS}}$	$\frac{\text{M}^{\text{ISPS}}}{\text{M}^{\text{SPS}}}$	$\frac{\mathcal{R}_{\text{out}}^{\text{ISPS}}}{\mathcal{R}_{\text{out}}^{\text{PS}}}$	$\frac{\mathcal{L}^{\text{ISPS}}}{\mathcal{L}^{\text{PS}}}$	$m_{\text{PAR}}^{\text{ISPS}}$	$\frac{\text{M}^{\text{ISPS}}}{\text{M}^{\text{PS}}}$
Pacemaker	1.5	0.99	2	2	1.47	1	1	4	2.64
RSD	11.2	0.097	1	1	1.56	1	1	3	1.15
BlackScholes	1.33	1	18	17	5.7	1	1.16	41	6.86
Disp. map	1.03	0.95	2	2	1	1	1	5	1.33
Pdetect	1.0002	0.9	13	13	1.09	1	–	54	–
CELP	1.5	0.99	6	6	1.68	1	1.1	9	1.6
CD2DAT	1	1	2	2	4.75	1	3.35	6	8.8
MP3 Playback	1	1	4	4	1.13	1	1.1	4	1.26
JPEG2000	3.93	0.3	1	1	1.21	1	–	230	–

3.7 Discussion

The theoretical analysis presented in Section 3.5 proves that streaming applications, modeled as acyclic CSDF graphs, can be converted to real-time periodic tasks by using our scheduling approach which converts each actor in a CSDF graph, by considering different WCET value for each actor phase, to a set of strictly periodic tasks. As a result, a variety of hard real-time scheduling algorithms can be applied to temporally schedule the graph on a platform with calculated number of processors with a certain guaranteed throughput and latency. Additionally, the latency reduction method presented in Section 3.5 can be used to reduce the graph latency when the converted tasks are scheduled as real-time periodic tasks. The experiments on a set of real-life applications

showed that our ISPS approach gives tighter guarantee on the throughput and better processor utilization with acceptable increase in terms of communication memory requirements when compared with the SPS hard real-time scheduling approach. By applying our proposed latency reduction method, the ISPS delivers shorter graph latency while providing better throughput and processor utilization than the SPS approach. When compared with the PS approach, our proposed approach gives the same throughput with increased communication memory but takes much shorter time for deriving the schedule and for calculating the minimum number of processors and the size of communication buffers. Finally, our approach gives throughput that is equal or very close to the absolute maximum throughput achieved by the self-timed scheduling (STS) of actor firings but requires much shorter time to derive the schedule.

Chapter 4

Exploiting Parallelism in Hard Real-Time Systems to Maximize Performance

Jelena Spasic, Di Liu, Todor Stefanov, "Exploiting Resource-constrained Parallelism in Hard Real-Time Streaming Applications", *In Proceedings of the International Conference on Design, Automation and Test in Europe (DATE'16)*, pp. 954–959, Dresden, Germany, March 14-18, 2016.

THIS chapter presents our solution to the problem of exploiting the right amount of parallelism in a streaming application, **Problem 2** given in Section 1.3, according to an MPSoC platform such that performance is maximized and the timing guarantees are provided. That is, the chapter describes our solution approach consisting of an unfolding graph transformation and an algorithm that adapts the parallelism in the application according to the resources in an MPSoC by using the unfolding transformation.

The remainder of this chapter continues with the problem description in Section 4.1 and summarizes our contributions in Section 4.2. Then, we give an overview of the related work in Section 4.3. A motivational example is given in Section 4.4. It is followed by the description of our proposed solution approach given in Sections 4.5 and 4.6. The experimental evaluation of our proposed approach is presented in Section 4.7. The concluding discussion is given in Section 4.8.

4.1 Problem Statement

To meet the computational demands and timing requirements of modern streaming applications, the parallel processing power of MPSoC platforms has to be exploited efficiently. Exploiting the available parallelism in an MPSoC platform to guarantee performance and timing constraints is a challenging task. This is because it requires the designer to expose the right amount of parallelism available in the application and to decide how to allocate and schedule the tasks of the application on the available processing elements such that the platform is utilized efficiently and the timing constraints are met. However, as introduced in Section 1.3, the given initial parallel application specification often is not the most suitable one for the given MPSoC platform. To better utilize the underlying MPSoC platform, the initial specification of an application, that is, the initial task graph, should be transformed by an *unfolding graph transformation* to an alternative one that exposes more parallelism while preserving the same application behavior. The unfolding graph transformations proposed so far: 1) introduce additional tasks for managing data among tasks' replicas [KM08], [FKBS11], which introduces communication and scheduling overhead; 2) do data reordering or increase rates of data production/consumption on channels [ZBS13], [SLA12], which causes an increase of buffer sizes of data communication channels between the tasks and an increase of the application latency. Thus, special care should be taken during the unfolding transformation to avoid all the unnecessary overheads. Moreover, having more tasks' replicas than necessary results in an inefficient system due to overheads in code and data memory, scheduling and inter-tasks communication [FKBS11], [ZBS13]. Thus, the right amount of parallelism (tasks' replicas), that is, the proper values of unfolding factors, depending on the underlying MPSoC platform, should be determined in a parallel application specification to achieve maximum performance and timing guarantees.

Therefore, in this chapter, we investigate the following sub-problems: (1) How to efficiently **unfold a given initial acyclic SDF graph** of an application to avoid unnecessary communication/scheduling overheads and unnecessary increases in buffer sizes and the application latency?, and (2) How to **find a proper unfolding factor of each task in the initial graph**, such that the obtained alternative graph exposes the right amount of parallelism **that maximizes the utilization of the available processors in an MPSoC platform under hard real-time scheduling**?

4.2 Contributions

Our contributions to the solution of the research problem described in Section 4.1 are summarized as follows:

- We propose a new unfolding graph transformation for SDF graphs which results in graphs with shorter application latency and smaller buffer sizes compared to the related approaches [KM08], [FKBS11], [SLA12], [ZBS13], as shown in Section 4.7.
- We propose a new algorithm for finding a proper value for the unfolding factor of each task in a graph when mapping the graph on a platform such that the platform is utilized as much as possible under hard real-time scheduling.
- We show, on a set of real-life streaming applications, that in more than 98% of the experiments, our unfolding graph transformation and algorithm result in a solution with a shorter latency, smaller buffer sizes and smaller values for unfolding factors compared to the solution obtained from [ZBS13] while the same performance and timing requirements are satisfied.

Scope of work. We assume that a given SDF graph is acyclic. This limitation comes from the hard real-time scheduling framework, presented in Chapter 3, we use to schedule an SDF graph. However, as already mentioned earlier in Chapter 3, even with this limitation our approach is still applicable to many real-life streaming applications because a recent work [TA10] has shown that around 90% of streaming applications can be modeled as acyclic SDF graphs. In addition, our approach does not unfold *stateful* tasks and input/output tasks. A *stateful* task is a task which current execution depends on its previous execution, thus those executions cannot be run in parallel. Input and output tasks are the tasks connected to the environment, hence they are not unfolded.

4.3 Related Work

[KM08] proposes an Integer Linear Programming (ILP) based approach for maximizing the throughput of an application modeled as an SDF graph by exploiting data parallelism when mapping the application on a platform with fixed number of processors. However, an ILP-based approach suffers from an exponential worst-case time complexity. To overcome the time complexity issue of the approach in [KM08], [FKBS11] separates the task replication and the allocation of replicas. However, decomposing the problem into two strongly related problems and solving them separately has a negative impact on the

solution quality. In addition, the maximum data-level parallelism is revealed in the application without considering the platform constraints. In contrast, in our approach, we solve the problem of task replication and the mapping of replicas simultaneously while taking into account the platform constraints. Both approaches [KM08] and [FKBS11] use *splitter* (S) and *joiner* (J) tasks to distribute and merge data streams processed by replicas, see Figure 4.2(a). Those tasks introduce additional communication overhead as data streams have to be sent to them and to the replicas. Moreover, the splitter/joiner tasks have to be considered in the process of mapping and scheduling of tasks. In contrast, in our approach, we do not introduce additional tasks for data management, but we propose a new transformation on an SDF graph in Section 4.5 where the data is sent by replicas of the original tasks only to replicas which need the data for computation. Thus, we avoid the overhead of scheduling splitter/joiner tasks and duplicated data transfers, as shown in Section 4.7.

[SLA12] proposes a throughput driven transformation of an application modeled as an SDF graph for mapping the application on a platform. The graph transformation method in [SLA12] increases the rates of data production/consumption and hence increases the buffer capacities needed to store the data, see Figure 4.2(b). In addition, to enable unfolding of tasks, multiple firings of a certain task in the initial graph are combined into one firing of the corresponding task in the transformed graph, see the increased execution times of tasks in Figure 4.2(b), which leads to an increase in latency. In contrast, our transformation technique does not increase the rates of data production/consumption on communication channels and does not combine multiple task firings into one firing which in turn leads to shorter application latency and smaller buffer sizes of the communication channels, as shown in Section 4.7.

The closest to our work, in terms of scope and methods proposed to efficiently utilize the parallelism of an application mapped onto resource-constrained platform, is the work in [ZBS13]. The authors in [ZBS13] propose an approach for exploiting *just-enough* parallelism when mapping a streaming application modeled as an SDF graph on a platform with fixed number of processing elements. The graph transformation method in [ZBS13] transforms an initial SDF graph to functionally equivalent CSDF graph while keeping the same rates of data production/consumption on communication channels, see Figure 4.2(c). However, the transformation approach in [ZBS13] is not efficient in terms of application latency and buffer sizes of the communication channels, as shown in Section 4.7. Moreover, the proposed algorithm in [ZBS13] for finding the values of unfolding factors and the mapping of task replicas does

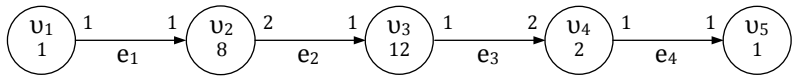


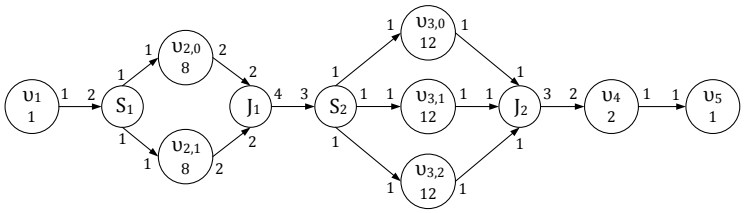
Figure 4.1: *An SDF graph G.*

not reveal the right amount of parallelism, but it reveals more parallelism than needed and hence the platform is unnecessarily overloaded, as shown in Section 4.7. In contrast, the approach we propose unfolds a graph by doing more aggressive token-flow analysis leading to shorter application latency and smaller buffer sizes. In addition, our approach finds smaller unfolding factors for tasks which leads to less memory needed to store the code of replicas and less memory to implement communication channels between the replicas.

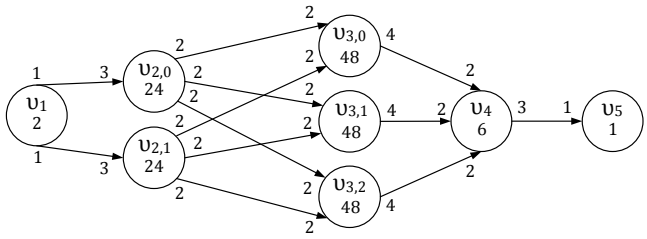
4.4 Motivational Example

In the first part of this section, we motivate the need for our new unfolding graph transformation. The throughput of graph G given in Figure 4.1 when scheduled under our ISPS presented in Chapter 3 is the same as the throughput obtained under *self-timed* scheduling [SGB08] and it is equal to $\frac{1}{24}$. Note that an unfolding graph transformation is used to increase the application throughput if it is allowed by the hardware platform on which the application is executed. Let us assume that actors v_2 and v_3 of graph G in Figure 4.1 are unfolded by factors 2 and 3, respectively, in order to increase the throughput of G. Figure 4.2 shows four functionally equivalent graphs obtained after applying the unfolding transformations proposed by [KM08], [FKBS11] – see Figure 4.2(a), by [SLA12] – see Figure 4.2(b), and by the transformation in [ZBS13] – see Figure 4.2(c), while the graph given in Figure 4.2(d) is obtained by applying our transformation described in Section 4.5. Our transformation method unfolds an SDF graph by doing more aggressive data token flow analysis with the aim to spread equally the workload of an actor during the hyperperiod and run in parallel as much replicas of the actor as possible.

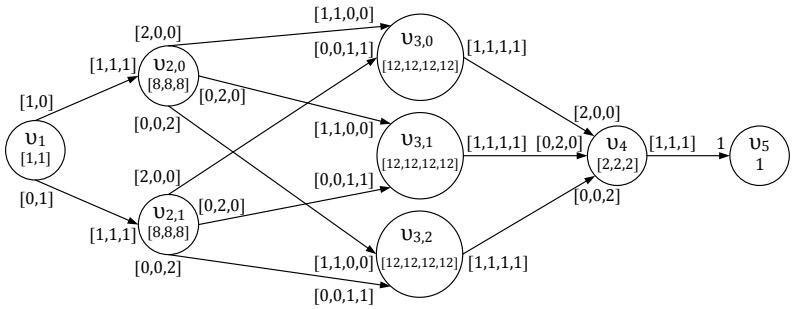
Table 4.1 gives for all four equivalent graphs of G the throughput $\mathcal{R}_{\text{out}}$ of the output actor, actor v_5, the maximum latency $\mathcal{L}_{\text{in}\rightarrow\text{out}}$ on an input-output path, the total size M, of the communication buffers, the total code size CS, and the total number of processors m needed to schedule the graphs under ISPS and the self-timed scheduling while achieving the same throughput $\mathcal{R}_{\text{out}}$. We can see from the table that by applying our unfolding transformation we can obtain, under ISPS, 2.29, 3.14, and 1.43 times shorter latency and 2.08, 2.75, and 1.33 times smaller buffers than the unfolding methods in [KM08]



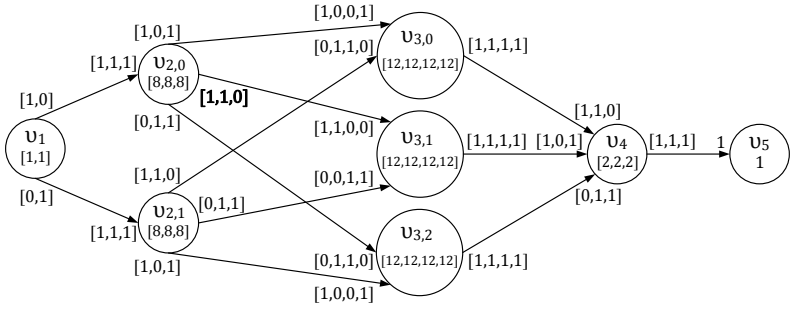
(a) Equivalent of G in Figure 4.1 after the transformation in [KM08], [FKBS11]



(b) Equivalent of G in Figure 4.1 after the transformation in [SLA12]



(c) Equivalent of G in Figure 4.1 after the transformation in [ZBS13]



(d) Equivalent of G in Figure 4.1 after our transformation

Figure 4.2: *Equivalent graphs of the SDF graph in Figure 4.1 by unfolding actor v_2 by factor 2 and v_3 by factor 3.*

Table 4.1: *Results for G transformed by different transformation approaches.*

Approach	ISPS					[SGB08]				
	$\mathcal{R}_{\text{out}}[\frac{1}{\mu s}]$	$\mathcal{L}_{\text{in}\to\text{out}}[\mu s]$	$M[B]$	$CS[kB]$	m	$\mathcal{R}_{\text{out}}[\frac{1}{\mu s}]$	$\mathcal{L}_{\text{in}\to\text{out}}[\mu s]$	$M[B]$	$CS[kB]$	m
[KM08], [FKBS11]	1/8	128	50	40	5	1/8	67	31	40	12
[SLA12]	1/8	176	66	36	5	1/8	93	57	36	8
[ZBS13]	1/8	80	32	36	5	1/8	76	24	36	8
our	1/8	56	24	36	5	1/8	62	21	36	8

Table 4.2: *Results for G transformed and mapped on 2 processors by different approaches.*

Approach	$\mathcal{R}_{\text{out}}[\frac{1}{\mu s}]$	$\mathcal{L}_{\text{in}\to\text{out}}[\mu s]$	$M[B]$	$CS[kB]$	m
[ZBS13]	1/18	180	31	44	2
our	1/18	108	16	32	2

and [FKBS11], [SLA12], and [ZBS13], respectively. The number of processors needed to schedule the graph obtained after the transformation under ISPS is equal for all the transformation methods. Under self-timed scheduling [SGB08] we obtain 1.08, 1.5, and 1.23 times shorter latency, while buffers are smaller 1.47, 2.71, and 1.14 times compared to the related approaches. Assuming one-to-one mapping for the self-timed scheduling, we need the same number of processors to schedule the unfolded graph obtained by the methods in [SLA12] and [ZBS13], and 1.5 times less processors than the unfolding methods in [KM08] and [FKBS11]. For both scheduling algorithms we obtain equal code size as the unfolding methods in [SLA12] and [ZBS13], and 1.11 times smaller code size than the methods in [KM08] and [FKBS11]. From Table 4.1, we see that our unfolding transformation approach presented in Section 4.5 is more efficient than the approaches in [KM08], [FKBS11], [SLA12], and [ZBS13].

So far, we considered only the unfolding transformation. Now, we would like to focus on the algorithm for finding the proper unfolding factors for actors when a graph is mapped onto resource-constrained platform and scheduled by a hard real-time scheduler such that the throughput of the graph is maximized. Here, we want to compare our algorithm in Section 4.6 with the approach in [ZBS13], because only that approach, among the related approaches, exploits the parallelism in an application under hard real-time scheduling. For example, in order to schedule graph G in Figure 4.1 on a platform with 2 processors while maximizing the throughput under hard real-time scheduling, the approach in [ZBS13] finds a vector of unfolding factors $\vec{f} = [1, 2, 4, 1, 1]$.

However, there exists a smaller vector of unfolding factors, such as $\vec{f} = [1, 1, 3, 1, 1]$, such that G is schedulable on 2 processors and the throughput is maximized. This smaller vector $\vec{f}$ is found by our algorithm in Section 4.6. Table 4.2 gives the throughput $\mathcal{R}_{\text{out}}$, latency $\mathcal{L}_{\text{in}\to\text{out}}$, buffer sizes M and code

size CS when G is unfolded and mapped on $m = 2$ processors by applying the approach in [ZBS13] and by applying our algorithm presented in Section 4.6. We can see from the table that by applying our algorithm we obtain under ISPS 1.67 times shorter latency, 1.94 times smaller buffers, and 1.38 smaller code size than the approach in [ZBS13]. From these results and the results given in Table 4.1, we clearly show the necessity and usefulness of the graph unfolding transformation presented in Section 4.5, and the algorithm for finding proper values for the unfolding factors presented in Section 4.6.

4.5 New Unfolding Transformation for SDF Graphs

Our new unfolding transformation method is given in Algorithm 3. The algorithm takes an SDF graph G and a vector of unfolding factors $\vec{f}$ and produces an unfolded graph G', which is a CSDF graph. The initial SDF graph and its unfolded version given in the form of a CSDF graph are functionally equivalent, meaning that both of them generate the same sequence of output data tokens for a given sequence of input data tokens. The algorithm consists of three phases. The first phase is given in lines 1 to 4 in Algorithm 3. Given that the execution semantics of the SDF model allows any integer multiple of the basic repetition vector also as a valid repetition vector, in line 1 of Algorithm 3 the basic repetition vector $\vec{q}$ of G is replaced by $\vec{q}^f = \text{lcm}(\vec{f}) \cdot \vec{q}$, where $\text{lcm}(\vec{f})$ is the least common multiple of all elements in $\vec{f}$. Then in lines 2 to 4, for each channel e_u in G, a matrix d is constructed containing as many columns as the number of tokens produced/consumed on the channel during one iteration of G with repetition vector $\vec{q}^f$. Each column in d contains in raw 0 an index p, $d[0][t] = p$, which is the index of the firing of the producer actor, $p \geq 0$, which produces the t^{th} token, and an index c in raw 1, $d[1][t] = c$, representing the index of the firing of the consumer actor, $c \geq 0$, which consumes the t^{th} token on e_u. Constructing matrix d for channel e_2 of graph G in Figure 4.1 when $\vec{f} = [1, 2, 3, 1, 1]$ is given in Figure 4.3, lines 2 to 4.

In the second phase the topology of the equivalent CSDF graph G' is created, which is given in lines 5 to 14 in Algorithm 3. In the equivalent CSDF graph G', every actor is replicated a certain number of times, as determined by the unfolding vector, lines 6 to 8. Then each channel e_u in the initial graph is replicated a certain number of times in the equivalent graph such that each replica of the producer on e_u is connected to each replica of the consumer on e_u, as given in lines 9 to 12 of Algorithm 3. The motivation behind unfolding is to equally distribute the workload of an actor in the initial graph by running in parallel replicas corresponding to that actor. The workload of an actor within

Algorithm 3: Procedure to unfold an SDF graph.

Input: An SDF graph $G = (\mathcal{V}, \mathcal{E})$, a vector of unfolding factors $\vec{f}$.
Output: The equivalent CSDF graph $G' = (\mathcal{V}', \mathcal{E}')$.

Take $\vec{q}^f = [\text{lcm}(\vec{f}) \cdot q_1, \cdots, \text{lcm}(\vec{f}) \cdot q_N]$ as a repetition vector of G;
for *communication channel* $e_u = (v_i, v_j) \in \mathcal{E}$ **do**
    Get production rate prd and consumption rate cns on e_u;
    Construct a matrix d, $d[0][t] = p$, $d[1][t] = c$, $t \in [0, prd \cdot q_i^f - 1]$, p is the index of v_i firing which produces t^{th} token, c is the index of v_j firing which consumes t^{th} token on e_u;
$\mathcal{V}' \leftarrow \emptyset$, $\mathcal{E}' \leftarrow \emptyset$;
for *actor* $v_i \in \mathcal{V}$ **do**
    for $k = 0$ *to* $f_i - 1$ **do**
        Add replica $v_{i,k}$ to $\mathcal{V}'$;
for *communication channel* $e_u = (v_i, v_j) \in \mathcal{E}$ **do**
    for *replica* $v_{i,k}$ *of* v_i **do**
        for *replica* $v_{j,l}$ *of* v_j **do**
            Add $e'_u = (v_{i,k}, v_{j,l})$ to $\mathcal{E}'$;
    for $t = 0$ *to* $prd \cdot q_i^f - 1$ **do**
        $d[0][t] = d[0][t] \bmod f_i$, $d[1][t] = d[1][t] \bmod f_j$;
for *communication channel* $e_u = (v_i, v_j) \in \mathcal{E}$ **do**
    Get production rate prd and consumption rate cns on e_u;
    Create empty/zero matrices $P^{i,k}$ with size $f_j \times q_{i,k}$, $k \in [0, f_i - 1]$;
    Create empty/zero matrices $C^{j,l}$ with size $f_i \times q_{j,l}$, $l \in [0, f_j - 1]$;
    for $h = 0$ *to* $q_{i,0} - 1$ **do**
        for $k = 0$ *to* $f_i - 1$ **do**
            Initialize a prod. counter seq. cnt_{prod} of length f_j to 0;
            for $o = 0$ *to* $prd - 1$ **do**
                $cnt_{\text{prod}}[d[1][h \cdot k \cdot prd + o]] = cnt_{\text{prod}}[d[1][h \cdot k \cdot prd + o]] + 1$;
            for $l = 0$ *to* $f_j - 1$ **do**
                $P^{i,k}[l][h] = cnt_{\text{prod}}[l]$;
    for $h = 0$ *to* $q_{j,0} - 1$ **do**
        for $l = 0$ *to* $f_j - 1$ **do**
            Initialize a cons. counter seq. cnt_{cons} of length f_i to 0;
            for $o = 0$ *to* $cns - 1$ **do**
                $cnt_{\text{cons}}[d[0][h \cdot l \cdot cns + o]] = cnt_{\text{cons}}[d[0][h \cdot l \cdot cns + o]] + 1$;
            for $k = 0$ *to* $f_i - 1$ **do**
                $C^{j,l}[k][h] = cnt_{\text{cons}}[k]$;
    for $k = 0$ *to* $f_i - 1$ **do**
        for $l = 0$ *to* $f_j - 1$ **do**
            if *all entries in raw* $P^{i,k}[l][]$ *are 0* **then**
                Delete a channel e'_u connecting replicas $v_{i,k}$ and $v_{j,l}$;
            else
                Associate production sequence $P^{i,k}[l][]$ and consumption sequence $C^{j,l}[k][]$ with $e'_u = (v_{i,k}, v_{j,l})$;
return G';

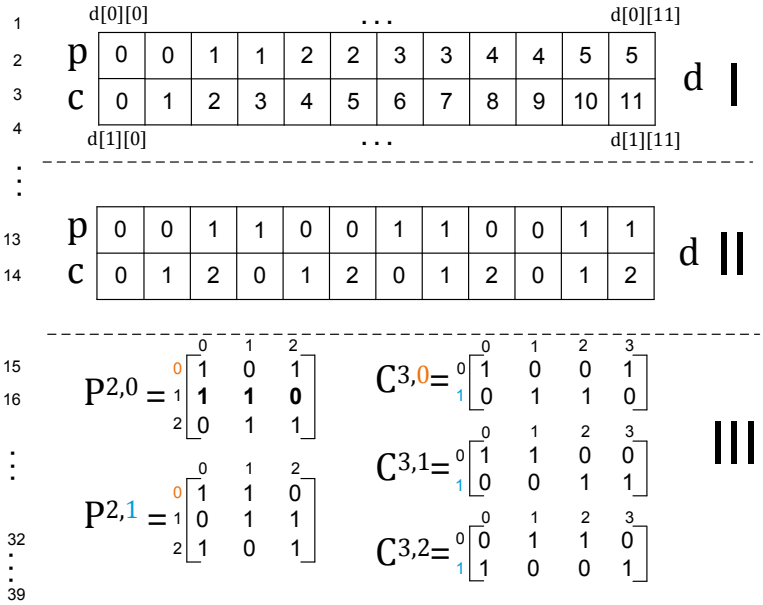


Figure 4.3: *Unfolding channel e_2 from the graph in Figure 4.1 by using Algorithm 3 when* $\vec{f} = [1,2,3,1,1]$.

one graph iteration is determined by the corresponding repetition value of the actor. Thus, each replica $v_{i,k} \in G'$ of an actor $v_i \in G$ will have the repetition $q_{i,k}$:

$$q_{i,k} = \frac{q_i^f}{f_i} = \frac{q_i \cdot \text{lcm}(\vec{f})}{f_i}. \tag{4.1}$$

For example, after the unfolding of the SDF graph in Figure 4.1 with the unfolding vector $\vec{f} = [1,2,3,1,1]$ we obtain the graph shown in Figure 4.2(d) with the repetition vector $\vec{q}' = [6,3,3,4,4,4,6,6]$, where $q_{2,0} = q_{2,1} = \frac{1 \cdot lcm(1,2,3,1,1)}{2} = 3$. Lines 13 to 14 convert the firing production/consumption indexes in $d[0][t]/d[1][t]$ for each token t produced/consumed on channel e_u into the indexes corresponding to the index k of the replica which produces/consumes t. This is illustrated for channel e_2 in Figure 4.3, lines 13 and 14.

Lines 15 to 39 represent the third phase of Algorithm 3 and they derive the production and consumption sequences for new channels and perform final placement of the new channels between the corresponding actor replicas. More specifically, for each source replica $v_{i,k}$ and destination replica $v_{j,l}$ of a channel, a production matrix $P^{i,k}$ and consumption matrix $C^{j,l}$ is created from matrix d in lines 15 to 32. The index of each raw in a production matrix

$P^{i,k}$ corresponds to the index l of a destination replica $v_{j,l}$. The index of each column in a production matrix $P^{i,k}$ corresponds to the firing index of source replica $v_{i,k}$. Elements in a production matrix of a source replica contain the number of tokens produced by a certain firing of that replica. Similar holds for elements in a consumption matrix. The created matrices $P^{2,0}$, $P^{2,1}$, $C^{3,0}$, $C^{3,1}$, $C^{3,2}$ for the source replicas $v_{2,0}$, $v_{2,1}$ and destination replicas $v_{3,0}$, $v_{3,1}$, $v_{3,2}$ on channel e_2 are given in Figure 4.3. For example, the value 0 in element $P^{2,0}[\mathbf{2}][\mathbf{0}]$ says that 0 tokens are produced by the **0** firing of source replica $v_{2,0}$ for the destination replica $v_{3,\mathbf{2}}$. Once these matrices are constructed, the production and consumption sequences on channel replicas are extracted from the corresponding raws in matrices, as given in lines 33 to 38 in Algorithm 3. For example, the production sequence on the channel between $v_{2,0}$ and $v_{3,\mathbf{1}}$ in Figure 4.2(d) is extracted from raw $P^{2,0}[\mathbf{1}][]$ in matrix $P^{2,0}$ and is equal to $[1, 1, 0]$. The extracted production/consumption sequences on replicas of channel e_2 can be seen in Figure 4.2(d). The unfolded graph G' is returned in line 39 of Algorithm 3.

4.6 The Algorithm for Finding Proper Unfolding Factors

In order to efficiently utilize the parallelism available in an application when mapping the application on a resource-constrained platform under hard real-time scheduling, proper unfolding factors for actors of the application have to be determined. Therefore, in this section, we present an algorithm which derives the proper unfolding factors which maximize the utilization of the platform, that is, maximize the application throughput.

The algorithm is given in Algorithm 4. It takes an SDF graph G, where the actors are scheduled by ISPS presented in Chapter 3, a platform with m processors, a scheduling algorithm A [LL73], an allocation heuristic H [CGJ96] and a quality factor ρ. A quality factor $\rho \in (0, 1]$ determines how much of the platform processing resources we want to utilize, with $\rho = 1$ corresponding to full utilization. The algorithm returns the best solution vector of unfolding factors $\vec{f}^{best}$.

Line 1 in Algorithm 4 initializes each unfolding factor of an actor in G to 1 and G' to G. Then, the upper bound $\hat{f}_i$ of unfolding factor f_i for each actor v_i in G is computed in line 2 in Algorithm 4 by using Equation (4.2) which is

similar to Equation (3) in [ZBS13]:

$$\hat{f}_i = \frac{\text{lcm}\{x_1, x_2, \cdots, x_n\}}{x_i}, \tag{4.2}$$

where $x_i = \frac{\text{lcm}\{W_1, W_2, \cdots, W_n\}}{W_i}$ and $W_i = \sum_{\varphi=1}^{\varphi=\phi_i} C_i(\varphi) \cdot r_i$ is the workload of actor v_i during one hyperperiod. The logic behind the computation of upper bounds on unfolding factors is the same as in [ZBS13]. That is, by unfolding every actor v_i in G by an upper bound $\hat{f}_i$ we obtain a CSDF graph G', for which the minimum period $\breve{T}_{i,k}$ of each replica $v_{i,k}$ under ISPS is equal to $\sum_{\varphi=1}^{\varphi=\phi_{i,k}} C_{i,k}(\varphi)$, meaning that each actor in the unfolded graph fully utilizes the processor which it runs on, hence, leading to the maximum throughput. Line 3 finds the utilization of graph G' when G' is scheduled on m processors by invoking Algorithm 5. The best utilization of G' is initialized in line 4 to be the first schedulable solution on m processors found by Algorithm 5 in line 3. Line 5 finds the bottleneck actor in G'. The bottleneck actor $v_{b,k}$ is the actor with the heaviest workload during one hyperperiod, $W_{b,k} = \max_{v_{i,k} \in \mathcal{V}'} W_{i,k}$. If multiple actors have the same maximum workload, then the one with the smallest code size is selected to be the bottleneck. If the current utilization $u_{G'}$ does not meet the quality requirement checked in line 6, the unfolding factor f_b of the bottleneck actor $v_{b,k}$ is increased in line 7 and the graph is unfolded by using Algorithm 3 in line 8. Note that stateful actors and input and output actors are not unfolded, that is, the upper bound on their unfolding factors is 1. The utilization $u_{G'}$ of the unfolded graph G' mapped on m processors is calculated in line 9 by Algorithm 5. If the current utilization $u_{G'}$ is higher than the best utilization in line 10, then in line 11 the best utilization becomes the one found in line 9 and the best solution vector of unfolding factors becomes the current vector of unfolding factors. Line 12 finds the bottleneck actor in the unfolded graph G'. Lines 6 to 12 are repeated and the algorithm terminates when either a pre-specified quality factor ρ is satisfied ($u_{G'} \geq \rho \cdot m$) or the unfolding factor of a bottleneck actor exceeds its upper bound $\hat{f}_b$ ($f_b \geq \hat{f}_b$).

We see that Algorithm 4 uses Algorithm 5 for finding the utilization of the unfolded graph G' when mapped on a platform with m processors. Algorithm 5 takes the unfolded CSDF graph G', a platform with m processors, a scheduling algorithm A [LL73] and an allocation heuristic H [CGJ96] as inputs. Line 1 calculates periods of actors in G' scheduled by ISPS presented in Chapter 3 by using Equation (3.5). Equation (3.5) can be written as Equation (4.3) and Equation (4.4):

$$T_i = \frac{\text{lcm}(\vec{r})}{r_i} \cdot s, \forall v_i \in \mathcal{V}, \tag{4.3}$$

Algorithm 4: Finding proper unfolding factors for an SDF graph mapped onto resource-constrained platform.

Input: An SDF graph G, the number of processors in a platform m, quality factor ρ, a scheduling algorithm A, an allocation heuristic H.
Output: Vector of unfolding factors $\vec{f}^{best}$.

$\vec{f} = [1, 1, \cdots, 1]$; $G' = G$;
Compute the upper bound $\hat{\vec{f}}$ of $\vec{f}$ by Equation (4.2);
Get $u_{G'}$ of G' by Algorithm 5 when scheduled by A and H on m;
$u_{G^{best}} = u_{G'}$; $\vec{f}^{best} = \vec{f}$;
Find the bottleneck actor $v_{b,k}$ in G';
while $u_{G'} < \rho \cdot m$ *and* $f_b < \hat{f}_b$ **do**
    $f_b = f_b + 1$;
    Get G' by unfolding G by Algorithm 3;
    Get $u_{G'}$ of G' by Algorithm 5 when scheduled by A and H on m;
    if $u_{G'} > u_{G^{best}}$ **then**
        $u_{G^{best}} = u_{G'}$; $\vec{f}^{best} = \vec{f}$;
    Find the bottleneck actor $v_{b,k}$ in G';
return $\vec{f}^{best}$.

Algorithm 5: Procedure to find the utilization of a CSDF graph mapped onto resource-constrained platform.

Input: A CSDF graph G', the number of processors in a platform m, a scheduling algorithm A, an allocation heuristic H.
Output: Graph utilization $u_{G'}$.

Calculate s by Equation (4.4); calculate T_i by Equation (4.3) by using the calculated s;
Calculate $u_{G'}$ by Equation (4.5);
while G' *is not schedulable on* m *by* A *and* H **do**
    $s = s + 1$;
    Calculate T_i by using s in Equation (4.3); calculate $u_{G'}$ by Equation (4.5);
return $u_{G'}$.

$$s = \left\lceil \frac{\hat{W}}{\mathrm{lcm}(\vec{r})} \right\rceil, \tag{4.4}$$

where $\mathrm{lcm}(\vec{r})$ is the least common multiple of all repetition entries in $\vec{r}$ and $\hat{W} = \max_{v_i \in \mathcal{V}} W_i$ is the maximum workload during one hyperperiod. Note that periods computed by Equation (4.3) are the minimum periods for actors scheduled by ISPS and that there exist other larger valid periods for actors by taking any integer $s > \left\lceil \frac{\hat{W}}{\mathrm{lcm}(\vec{r})} \right\rceil$. Once the actor periods are computed, the

utilization of actor v_i, denoted as u_i, can be computed as $u_i = \sum_{\varphi=1}^{\varphi=\phi_i} C_i(\varphi)/T_i$, where $u_i \in (0, 1]$. For a graph G, u_G is the total utilization of G given by:

$$u_G = \sum_{v_i \in \mathcal{V}} u_i = \sum_{v_i \in \mathcal{V}} \frac{\sum_{\varphi=1}^{\varphi=\phi_i} C_i(\varphi)}{T_i}. \tag{4.5}$$

The total utilization of a graph directly determines the minimum number of processors needed to schedule the graph, as explained earlier in Section 2.2.5. The utilization $u_{G'}$ of G' is calculated in line 2 in Algorithm 5 by using Equation (4.5). Actor periods computed by Equation (4.3) and Equation (4.4) represent the minimum periods when the actors are scheduled under ISPS on a platform with unlimited number of processors. It may happen that these minimum periods lead to a graph which is not schedulable on a platform with only m processors. Hence, in line 3 by using the utilization $u_{G'}$ calculated in line 2 we check if G' can be scheduled on m processors by using the corresponding schedulability test for A and H [DB11]. If G' is not schedulable on the platform, we decrease $u_{G'}$ until G' becomes schedulable by increasing the actor periods T_i. This is done in lines 4 and 5 in Algorithm 5. Once the graph G' becomes schedulable on m processors by A and H, Algorithm 5 returns the utilization of the unfolded graph G' in line 6.

4.7 Evaluation

We present two experiments to evaluate the techniques proposed in Section 4.5 and Section 4.6. In the first experiment, we evaluate the efficiency of our unfolding transformation in comparison to the unfolding transformation methods in [KM08], [FKBS11], [SLA12], and [ZBS13]. In the second experiment, we evaluate the efficiency of Algorithm 4 presented in Section 4.6 in terms of performance and time complexity by comparing our approach to the related approach in [ZBS13]. The experiments were performed on the real-life applications from the StreamIt benchmarks suit [TA10], given in Table 4.3. These applications were modeled as SDF graphs and $|\mathcal{V}|$ denotes the number of actors in an SDF graph, while $|\mathcal{E}|$ denotes the number of communication channels. The results of the evaluations are shown in Figure 4.4, Figure 4.5, and Figure 4.6. In all these figures, each vertical line shows the variations in the corresponding results among all the applications. The upper and lower ends of a vertical line represent the maximum and minimum values of the corresponding result while the marker at the middle of each vertical line represents the geometric mean of the result. Note that the Y axis in Figure 4.4 to

Table 4.3: *Benchmarks used for evaluation.*

Benchmark	$\lvert\mathcal{V}\rvert$	$\lvert\mathcal{E}\rvert$
Discrete cosine transform (DCT)	8	7
Fast Fourier transform (FFT)	17	16
Time delay equalization (TDE)	29	28
Data encryption standard (DES)	53	60
Bitonic Sorting	40	46
Channel Vocoder	55	70
Filterbank	85	99
Serpent	120	128
MPEG2	23	26
Vocoder	114	147
FMRadio	43	53

Figure 4.6 has a logarithmic scale. We run all the experiments on an Intel Core i7-2620M CPU running at 2.70 GHz with Linux Ubuntu 12.4.

4.7.1 Efficiency of the Proposed Unfolding Transformation

In this section, we evaluate the performance of our unfolding transformation method proposed in Section 4.5 by comparison to the related unfolding transformation methods in [KM08], [FKBS11], [SLA12], and [ZBS13]. In this experiment, first we use Algorithm 4 to find a vector of unfolding factors for each application in Table 4.3 mapped on a platform with 64 processors with partitioned First-Fit Decreasing Earliest Deadline First (FFD-EDF) scheduler and quality factor $\rho = 0.9$. Then, for each application, we use the found vector of unfolding factors to unfold the application graph by applying our transformation method and the related transformation methods [KM08], [FKBS11], [SLA12], [ZBS13]. Finally, we use the ISPS framework presented in Chapter 3 to calculate the latency, buffer sizes and code size when the unfolded graphs are scheduled by FFD-EDF on 64-processor platform. The ratios between the results obtained by *related* transformation methods and *our* transformation in terms of application latency ($\mathcal{L}$), buffer sizes (M) and code size (CS) are given in Figure 4.4. We can see that our method outperforms all the related methods, and delivers on average 2.82, 3.95, and 1.43 times shorter latency and 1.98, 2.5, and 1.08 times smaller buffers than the method in [KM08] and [FKBS11], [SLA12], and [ZBS13], respectively. Although the methods in [KM08] and [FKBS11] introduce additional actors for data management, the average increase in the total code size is only 1%. The other two transformation methods, [SLA12]

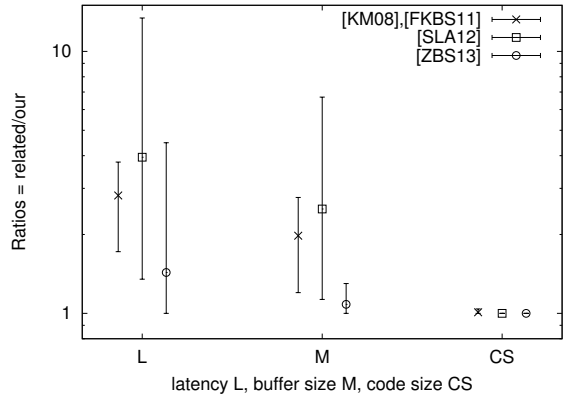
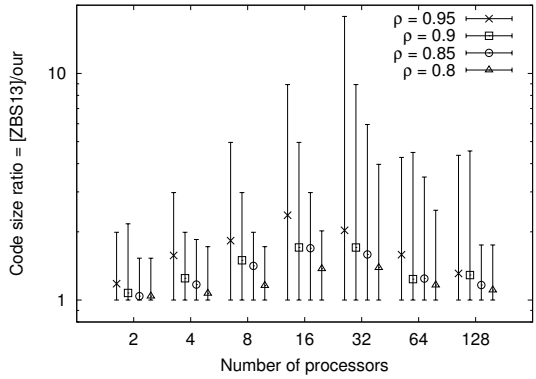


Figure 4.4: *Comparison of our unfolding transformation to the approaches in [KM08], [FKBS11], [SLA12], [ZBS13].*

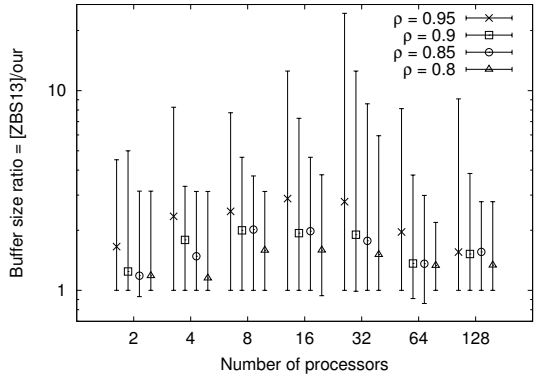
and [ZBS13], have the same code size as our method. Note that all the methods achieve the same application throughput.

4.7.2 Performance of Algorithm 4

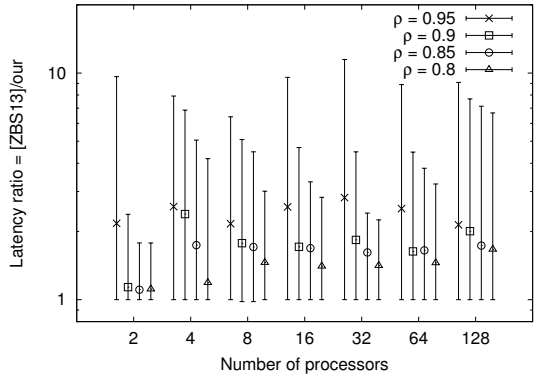
We evaluate the performance of Algorithm 4 by comparison to the related approach in [ZBS13]. For each application *app* in Table 4.3, we construct 28 system configurations (app, m, ρ) with number of processors $m \in \{2, 4, 8, 16, 32, 64, 128\}$, and utilization quality $\rho \in \{0.8, 0.85, 0.9, 0.95\}$. We run Algorithm 4 with FFD-EDF scheduler for each (app, m, ρ) configuration to obtain a vector of unfolding factors $\vec{f}^{best}$. Then, for each configuration, we unfold the corresponding application graph by the obtained vector $\vec{f}^{best}$ by using Algorithm 3. Finally, we use the ISPS framework presented in Chapter 3 to calculate the latency of an application, buffer sizes and code size when the unfolded graphs are scheduled by FFD-EDF on the corresponding platform. We perform the same experiment by running the related algorithm proposed in [ZBS13] and using the ISPS framework in Chapter 3 for each (app, m, ρ). The obtained ratios for the total code size, total buffer sizes, and latency between the approach in [ZBS13] and our approach are given in Figure 4.5(a), Figure 4.5(b), and Figure 4.5(c), respectively. We can see that by using Algorithm 4 we can achieve up to 17.85 times smaller code size (see Figure 4.5(a), ρ=0.95, m=32), up to 24.4 times smaller buffers (see Fig. 4.5(b), ρ=0.95, m=32) and up to 11.47 shorter latency (see Figure 4.5(c), ρ=0.95, m=32) than the approach in [ZBS13]. Note that both approaches meet



(a) Code size ratio (higher is better)



(b) Buffer size ratio (higher is better)



(c) Latency ratio (higher is better)

Figure 4.5: *Results of performance evaluation of our proposed approach in comparison to the approach in [ZBS13].*

the same throughput requirements. Regarding the buffer sizes, we obtain in 5 experiments out of 308 experiments larger buffer sizes by up to 1.16 times than the approach in [ZBS13] (see for example Figure 4.5(b), ρ=0.85, m=64). However, for all these experiments we do less unfolding, so we obtain smaller code size. In the case of latency, in 2 experiments out of 308 we get latency which is by 2% larger than the corresponding latency when the approach in [ZBS13] is applied (see Figure 4.5(c), m=8). However, in these two cases, we obtain smaller code size and smaller buffer sizes than the approach in [ZBS13].

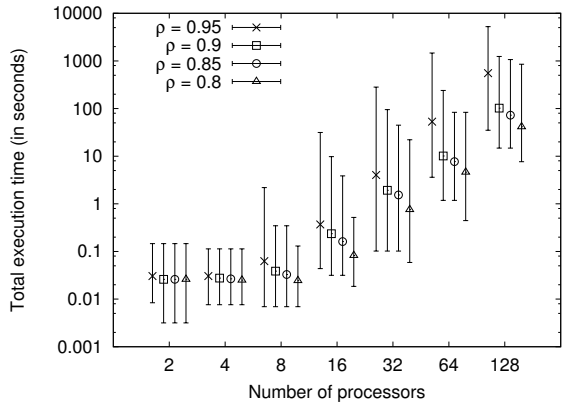
4.7.3 Time Complexity of Algorithm 4

We evaluate the efficiency of our algorithm for finding proper values of unfolding factors in terms of the execution time of our Algorithm 4 to find a solution. The execution times for different quality factors and different number of processors in a platform are given in Figure 4.6(a). We compare these execution times with the corresponding execution times of the related approach in [ZBS13]. The comparison is given in Figure 4.6(b).

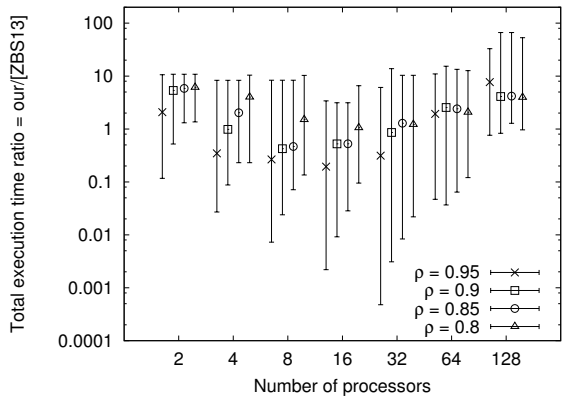
As can be seen from Figure 4.6(a), for platforms containing up to 16 processors, our Algorithm 4 takes in the worst case 32 seconds to find a solution, and less than 1 second on average for all values of quality factor ρ. For a platform with 32 processors, the execution time of our algorithm is 5 minutes in the worst case, and up to 4 seconds on average. In the case of a 64-processor platform our algorithm needs 25 minutes in the worst case to find a solution, and up to 53 seconds on average. Finally, for a platform with 128 processors Algorithm 4 takes 88 minutes in the worst case and up to 9 minutes on average to find a solution. In addition, it can be seen in Figure 4.6(b) that our approach is on average up to 8 times slower than the approach in [ZBS13] which is acceptable given that our approach delivers solutions of better quality, as shown in Section 4.7.2, within a matter of minutes.

4.8 Discussion

As a solution to a problem of exploiting the right amount of parallelism with the aim to achieve the maximum achievable throughput when mapping a streaming application modeled by an SDF graph on a resource-constrained platform under hard real-time scheduling, we presented in this chapter a new unfolding graph transformation and an algorithm which uses the transformation to adapt the parallelism in the application when mapping the application on the platform. Experiments on a set of real-life streaming applications



(a) Running time (in seconds) of Algorithm 4



(b) Execution time ratio (lower is better)

Figure 4.6: *Results of time evaluation of our proposed approach in comparison to the approach in [ZBS13]*

demonstrate that: 1) our unfolding transformation gives shorter latency and smaller buffer sizes when compared to the related approaches; and 2) our algorithm finds, in a matter of minutes, a solution with smaller code size, smaller buffer sizes and shorter latency in 98% of the experiments, while meeting the same performance and timing requirements when compared to an existing approach.

Chapter 5

Exploiting Parallelism in Hard Real-Time Systems to Minimize Energy

Jelena Spasic, Di Liu, Todor Stefanov, "Energy-Efficient Mapping of Real-Time Applications on Heterogeneous MPSoCs using Task Replication", *In Proceedings of the IEEE/ACM/IFIP International Conference on HW/SW Codesign and System Synthesis (CODES+ISSS'16)*, pp. 28:1–28:10, Pittsburgh, Pennsylvania, USA, October 2-7, 2016.

IN this chapter, we devise an approach to exploit the right amount of parallelism in streaming applications in order to minimize the energy consumption of streaming applications with throughput constraints when they are mapped on cluster heterogeneous MPSoCs. That is, this chapter describes our solution to **Problem 3** given in Section 1.3.

The problem of energy minimization by exploiting parallelism in streaming applications is further described in Section 5.1. Then, our contributions are summarized in Section 5.2. The related work is addressed in Section 5.3. Section 5.4 gives a motivational example to demonstrate the need for a new energy minimization approach. The considered system model and energy model are described in Sections 5.5 and 5.6. Then, the new energy minimization approach, that is, our proposed solution approach, is given in Section 5.7. Our proposed approach is experimentally evaluated in Section 5.8. The concluding discussion is given in Section 5.9.

5.1 Problem Statement

As introduced in Sections 1.2.2 and 1.3, cluster heterogeneous MPSoCs with per-cluster VFS capability are recognized as energy-efficient platforms for embedded systems. In order to efficiently utilize these cluster heterogeneous platforms to achieve all the desired requirements, the underlying hardware platform and the running streaming application have to be closely related. This requires the embedded designer to expose the right amount of parallelism available in the application and to decide how to allocate and schedule the tasks of the application on the available processing elements such that the platform is utilized efficiently and the timing and energy consumption constraints are met. As explained in Chapter 4 already, the given initial parallel application specification is often constructed without fully considering the computational capacity and power consumption profile of an MPSoC platform. This may lead to an application specification which consists of highly imbalanced tasks in terms of the task workload, that is, task utilization. This may further lead to unnecessary increase in the energy consumption of such a system because when several tasks are mapped onto the same cluster in cluster heterogeneous MPSoCs, the one with the heaviest utilization will determine the required voltage and frequency of the whole cluster and will significantly increase the energy consumption of the other tasks mapped on the same cluster. As shown in Chapter 4, by applying task replication to application tasks with heavy utilization, their utilization can be decreased while still providing the same application performance. Thus, to better utilize the underlying MPSoC platform while minimizing the energy consumption, the initial specification of an application, that is, the initial task graph, should be transformed to an alternative one that exposes more parallelism while preserving the same application behavior and timing performance. However, as mentioned in Chapter 4, having more tasks' replicas than necessary introduces more overheads in code and data memory, scheduling and inter-tasks communication, which in turn will result in higher energy consumption. Therefore, in this chapter, we investigate how to **exploit the right amount of parallelism, that is, find the proper values of replication (unfolding) factors, depending on the underlying MPSoC platform, to achieve the required performance and timing guarantees while minimizing the energy consumption**.

5.2 Contributions

As a solution to the research problem described in Section 5.1, we propose a novel algorithm to efficiently map real-time streaming applications onto cluster heterogeneous MPSoCs, which are subject to throughput constraints, such that the energy consumption of the cluster heterogeneous MPSoC is reduced by using task replication and per-cluster VFS. The specific novel contributions of this chapter are the following:

- We propose a novel polynomial-time algorithm, called Data Parallel Energy Minimization (DPEM), to map and schedule hard real-time streaming applications modeled as acyclic SDF graphs onto a cluster heterogeneous MPSoC such that the energy consumption is minimized while the throughput constraints are guaranteed. By using the hard real-time scheduling framework for CSDF graphs, presented in Chapter 3, we propose within our DPEM algorithm an efficient way to determine a suitable processor type for each task in an (C)SDF graph such that the energy consumption is minimized and the throughput constraint is met. Then, by using the unfolding graph transformation in Chapter 4, we propose a method in DPEM to determine a replication factor for each task in an SDF graph such that the distribution of the workload on the same type of processors is balanced, which enables processors to run at a lower frequency, hence reducing the energy consumption.
- We show, on a set of real-life streaming applications, that our proposed energy minimization approach outperforms related approaches in terms of energy consumption while meeting the same throughput constraints.

5.3 Related Work

Energy-efficient mapping and scheduling of streaming applications represented as dataflow graphs which guarantees certain throughput has been extensively studied. The related works can be divided into several categories depending on the MPSoC platform they consider: *homogeneous* [SDK13, DSB+13, ZSJ08, LW13, NMM+11, BL13, HNP+15], or *heterogeneous* [HMGM13, SJE11]. Depending on the VFS technique they apply to minimize the energy consumption, the related works can be divided into those considering *per-core VFS* [SDK13, DSB+13, ZSJ08, LW13, NMM+11, HMGM13, BL13], those considering *global VFS* [HMGM13, HNP+15] and the works which do not consider VFS but they utilize platform heterogeneity to achieve energy-efficiency [SJE11]. The approaches in [HMGM13, LW13, NMM+11, SJE11, HNP+15] convert an

initial SDF graph into equivalent Homogeneous SDF (HSDF) graph to exploit the parallelism of an application and achieve energy-efficiency. However, the HSDF graph obtained from the initial SDF graph may grow in size exponentially, making the analysis performed on the HSDF graph time-consuming. Instead, the approaches in [SDK13, ZSJ08, DSB+13, BL13] perform energy minimization directly on an SDF graph. Works [SDK13] and [ZSJ08] perform design space exploration at design time to find the energy-efficient mapping solution of an SDF scheduled in self-timed manner on a homogeneous MPSoC platform with per-core VFS capability such that certain throughput is guaranteed. In addition, the approach in [SDK13] has a run-time phase where slack created at run time is exploited to further minimize the energy consumption. In [DSB+13] the authors propose a heuristic to find per-core voltage-frequency points for a given task mapping and the execution order such that the throughput constraint is met. The authors in [BL13] propose a technique to transform an SDF graph at run time into its equivalent SDF graph to adapt to environmental and demand changes. One possible scenario where the SDF graph should be transformed to adapt to the new circumstances is when some processors become available on a homogeneous platform with per-core VFS capability. In that case the tasks in the SDF graph are replicated such that all processors are occupied, which enables processors to run at a lower frequency hence consuming less energy. However, the authors in [BL13] focus more on the transformation itself and not on the energy minimization. In contrast to all related works, discussed above, our approach: 1) considers heterogeneous MPSoC platforms with per-cluster VFS capability, which is a good trade-off in terms of energy-efficiency and the implementation cost; 2) utilizes an unfolding graph transformation to balance the workload put on the MPSoC and to reduce energy consumption by finding how many times each task in a graph should be replicated; 3) uses preemptive hard real-time scheduling to schedule the tasks which gives more opportunities to meet the lowest frequency for schedulability supported by the platform.

Energy-efficient mapping and scheduling of periodic hard real-time tasks has been widely researched in the past. [BMAB16] gives a comprehensive review of works dealing with energy-aware scheduling for real-time systems. As stated in [BMAB16], most of the existing work considers homogeneous MPSoCs and in recent years people started considering heterogeneous platforms and platforms with voltage/frequency levels shared among multiple processors as energy-efficient design solutions. Regarding the considered heterogeneous MPSoC platforms, the closest to our work are the works in [CKR14] and [LSCS15]. The approach in [CKR14] proposes and evaluates several parti-

tioned Earliest Deadline First (EDF) scheduling strategies for real-time tasks mapped on cluster heterogeneous platforms in terms of energy-efficiency. However, because of the bin-packing issue in partitioned scheduling, the approach in [CKR14] may not fully utilize the energy-efficient cores in a cluster heterogeneous MPSoC, hence the energy minimization is limited. In contrast, by replicating the tasks with heavy utilization, we can reduce their utilization and hence fully utilize the energy-efficient cores. The approach in [LSCS15] considers cluster scheduling for cluster heterogeneous MPSoCs where tasks are allowed to migrate at run-time among processors within the same cluster in order to achieve better resource utilization. However, cluster scheduling suffers from high scheduling overhead caused by task migration and increased context switching. Moreover, the frequency of some clusters in [LSCS15] is still determined by the tasks with the heaviest utilization. In contrast, in our approach, we use partitioned scheduling which has low scheduling overhead and we avoid the capacity loss and we lower the operating frequency by replicating the tasks with heavy utilizations.

The works in [XKD12], [WYK+10] and [Lee09] consider parallel execution of task replicas to achieve energy efficiency, as we do. The authors in [XKD12] consider frame-based tasks with an implicit deadline and a homogeneous platform with per-core VFS capability where the frequency of a core may be changed for each task. In contrast, in our work, we consider more general periodic task model and more realistic heterogeneous platform with per-cluster VFS capability, hence our approach is more applicable in practice than the approach in [XKD12]. The approach in [WYK+10] exploits the data parallelism in an application by replicating the tasks of the application over all processors available in an MPSoC. This means that, in distributed memory architectures, the code of the whole application has to be replicated on all the processors in an MPSoC. By contrast, in our approach, only certain tasks of the application have to be replicated, which reduces significantly the memory overhead of our approach compared to the one in [WYK+10]. Moreover, the work in [WYK+10] assumes homogeneous systems with per-core VFS and continuous frequencies, while we consider heterogeneous systems with per-cluster VFS capability, which is more practical in modern embedded systems. The approach presented in [Lee09] replicates computation-intensive tasks which yields to a more balanced load on processors, and in turn allows the system to run at a lower frequency. In addition, the authors in [Lee09] consider systems with discrete set of operating frequencies and homogeneous platforms with per-core VFS capability. As discussed earlier, per-core VFS is not practical in modern many-core systems. Hence, our work considers het-

erogeneous platforms with per-cluster VFS capability. The approach in [Lee09] is devised and, hence efficient only for platforms with performance-efficient processors. This means that the approach in [Lee09] would never replicate the tasks which are going to be mapped on energy-efficient processors. In addition, if the total number of tasks with heavy utilization is equal to the number of processors in a platform, tasks will not be replicated in [Lee09]. In contrast, our approach will replicate the tasks mapped on energy-efficient processors and it will replicate the tasks even if the number of heavy tasks is equal to the number of processors if this leads to more energy-efficient design.

5.4 Motivational Example

In the first part of this section, we motivate the need for using the unfolding graph transformation, presented in Chapter 4, to achieve energy-efficient MPSoC design under a throughput constraint. We first show the drawback of the energy minimization approaches for heterogeneous MPSoCs and hard real-time scheduling, that is, the approaches in [CKR14] and [LSCS15]. We analyze three different designs obtained by mapping the SDF graph G in Figure 5.1 to a heterogeneous platform consisting of one performance efficient (PE) cluster with 2 PE processors and one energy-efficient (EE) cluster with 2 EE processors, namely, the platform given in column 1, row 2 in Table 5.1, under a throughput constraint of 1 output token per 100 μs. An example of PE and EE clusters and processors is given in Figure 1.1, where a cluster of 'big' cores corresponds to a PE cluster, and a cluster of 'LITTLE' corresponds to an EE cluster.

The first design out of the three analyzed is obtained by using the best mapping approach evaluated in [CKR14] and we refer to that approach as CKR. The CKR approach allocates actors v_2 and v_3 to PE processors in one-to-one manner, and it allocates actors v_1 and v_4 to one EE processor, while the other EE processor is switched-off. Once the actors are allocated, the minimum frequency which ensures the schedulability of the actors mapped to a processor in a cluster is selected from a discrete set of frequencies per cluster. The energy consumption of such design is given in Table 5.1, column 2, row 2. After applying the approach in [LSCS15], we obtain the second design where actors v_2 and v_3 are allocated to the PE cluster, while actors v_1 and v_4 are allocated to the EE cluster. The corresponding energy consumption after applying the approach in [LSCS15], denoted by FDM, is given in Table 5.1, column 3, row 2. If we apply our approach presented in Section 5.7 which uses the unfolding transformation in Chapter 4 on graph G in Figure 5.1,

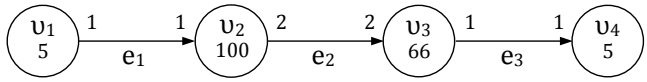


Figure 5.1: *An SDF graph G.*

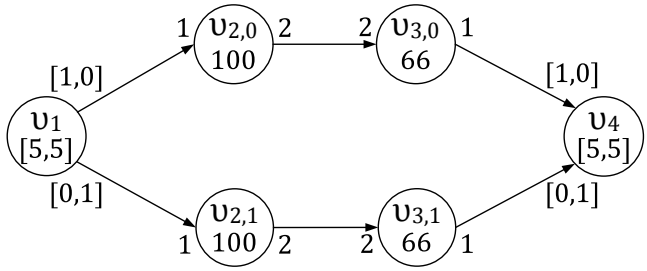


Figure 5.2: *A CSDF graph G' obtained by unfolding SDF graph G in Figure 5.1 with* $\vec{f} = [1, 2, 2, 1]$.

Table 5.1: *Different MPSoC designs for G in Figure 5.1.*

MPSoC	CKR $[\mu J]$	FDM $[\mu J]$	SDK $[\mu J]$	our $[\mu J]$	WYL $[\mu J]$
(2PE)(2EE)	343.55	343.55	346.30	97.76	343.55
(PE)(PE)(PE)(PE)	357.94	392.18	389.09	192.80	240.32

under the same throughput constraint as in the CKR and FDM approaches, we can lower the utilization of the actors with high utilization, v_2 and v_3, and achieve better load balancing on the processors of the same type and hence, the frequency of the power-hungry processors can be lowered further than in [CKR14], [LSCS15]. For example, by unfolding actors v_2 and v_3 twice, as given in Figure 5.2, our approach in Section 5.7 allocates $v_{2,0}$ and $v_{3,0}$ one-to-one to PE processors, and it allocates $v_{2,1}$ to an EE processor and $v_{3,1}$, v_1 and v_4 to another EE processor. The energy consumption value for this third design is given in Table 5.1, column 5, row 2. We can see that our approach reduces the energy consumption by 71% when compared to the CKR and FDM approaches.

Now, we would like to analyze an approach which was devised for homogeneous platforms with per-core VFS capability, that is, the approach in [SDK13], denoted by SDK in Table 5.1. To this end, we compare the energy consumption of two designs in which the SDF graph G in Figure 5.1 is mapped to a homogeneous MPSoC consisting of four PE clusters with 1 processor per cluster, under a throughput constraint of 1 output token per 100 μs. The SDK approach will allocate actors to processors in one-to-one manner, while our

approach, proposed in Section 5.7, will replicate actors v_2 and v_3 twice, as shown in Figure 5.2, to lower their utilization. We can see from columns 4 and 5, row 3 in Table 5.1 that our approach reduces the energy consumption by 50% when compared to the SDK approach. The main reason is that we are using the unfolding graph transformation to reduce the influence of heavy actors and hence minimize the energy of an MPSoC. Another reason is that the SDK approach uses self-timed scheduling which is non-preemptive, hence, less flexible for scheduling and that the SDK only minimizes dynamic energy consumption. The energy consumption values of the approaches CKR, FDM and SDK in Table 5.1 which were not discussed above are given only for completeness. However, we can see that these values are always higher than the corresponding energy consumption of our approach in Section 5.7. Thus, our approach outperforms these related approaches.

Above, we motivated the need to use the unfolding transformation, presented in Chapter 4, within our new approach described in Section 5.7 to achieve energy-efficiency for MPSoCs under a throughput constraint. Now, we would like to motivate the need for our whole approach, presented in Section 5.7, which efficiently finds task replication factors and task mappings to achieve further reductions in the energy consumption. Although the approach in [Lee09] exploits the energy-saving capability of data-parallel execution for a homogeneous MPSoC with per-core VFS capability, that approach is not efficient in terms of energy reduction, especially in the case of platforms with EE processors. Below we show its inefficiency for the homogeneous platform in column 1, row 3, and for the heterogeneous MPSoC platform in column 1, row 2, in Table 5.1. The approach in [Lee09], called the WYL approach, considers platforms which power consumption curve is increasing "fast" with the increase of processor utilization. Such power consumption curve corresponds to PE processors. Let us consider the mapping of the SDF graph in Figure 5.1 on the homogeneous platform under a throughput constraint of 1 output token per 100 μs. The WYL approach will classify actors v_2 and v_3 as "heavy" tasks, that is, tasks eligible for replication. However, because the platform contains only 4 processors, the WYL will decide that the number of processors is not sufficient to replicate both actors and it will only replicate actor v_2 twice. In contrast, our algorithm will replicate both actors v_2 and v_3 twice, which will lead to an energy reduction of 20%, see Table 5.1, row 3, columns 5 and 6.

Let us now analyze the designs obtained by applying the WYL and our new approach for mapping graph G in Figure 5.1 on the heterogeneous platform in column 1, row 2, in Table 5.1. Given that the power consumption curve

of EE processors is a "slowly" increasing curve with the increase of processor utilization, the WYL approach will never replicate actors assigned to EE processors. In contrast, our approach presented in Section 5.7 will replicate actors assigned to EE processors as well if their replication leads to more energy-efficient MPSoC. We can see in row 2, columns 5 and 6, in Table 5.1, that our approach leads to a design with 71% reduction in energy consumption when compared to the WYL approach, for the heterogeneous MPSoC with one PE and one EE cluster each containing 2 processors. This happens because after the classifications of actors into PE and EE in order to satisfy the throughput constraint of 1 output token per 100 μs, EE actors will not be considered for replication in the WYL approach, while PE actors will be considered yet never replicated because of the algorithm in [Lee09] which does not replicate the actors once the number of "heavy" actors is equal to the number of (PE) cores, which happens for this platform.

From the above examples, we can see the necessity and usefulness of our approach, presented in Section 5.7, which uses the graph unfolding transformation, given in Chapter 4, to obtain energy-efficient cluster heterogeneous MPSoC designs.

5.5 System Model

In this section, we describe the system model we use in this chapter. We consider a cluster heterogeneous MPSoC containing two types of clusters – performance-efficient (PE) clusters and energy-efficient (EE) clusters. Each cluster has a number of identical PE processors, denoted as N_p^{PE}, or a number of EE processors, denoted as N_p^{EE}. Thus, in total, a cluster heterogeneous MPSoC contains $N_c^{PE} \times N_p^{PE}$ PE processors and $N_c^{EE} \times N_p^{EE}$ EE processors, where N_c^{PE} and N_c^{EE} represent the total number of PE clusters and the total number of EE clusters, respectively. All processors on the same cluster operate at the same voltage and frequency level. The voltage and frequency level of a cluster can be changed to control the power consumption. A cluster can be switched-off, thereby consuming no power.

Since the actors of a (C)SDF graph G modeling an application may run on two different types of processors (PE and EE), the worst-case execution time value $C_i(\varphi)$ for each phase φ of an actor v_i has two values – $C_i^{PE}(\varphi)$ and $C_i^{EE}(\varphi)$. The total utilizations of the actors/tasks assigned to PE cluster j and

EE cluster k can be calculated by:

$$u_j^{PE} = \sum_{v_i \in \mathcal{V}_j^{PE}} \frac{\sum_{\varphi=1}^{\varphi=\phi_i} C_i^{PE}(\varphi)}{T_i}, \; u_k^{EE} = \sum_{v_i \in \mathcal{V}_k^{EE}} \frac{\sum_{\varphi=1}^{\varphi=\phi_i} C_i^{EE}(\varphi)}{T_i}, \tag{5.1}$$

where $\mathcal{V}_j^{PE}$ and $\mathcal{V}_k^{EE}$ represent sets of CSDF actors/tasks assigned to PE cluster j and EE cluster k, respectively.

5.6 Energy Model

This section defines the energy model used in this chapter. Given that all processors in the same cluster operate at the same voltage and frequency level, we can reduce the energy consumption of a cluster heterogeneous MPSoC by using per-cluster VFS and by switching-off some clusters. The authors in [LSCS15] give a power model for cluster heterogeneous MPSoC systems with discrete voltage and frequency levels based on real measurements performed on the ODROID XU-3 [ODR] board containing an MPSoC with two clusters – one quad core Cortex A15 big (PE) cluster and one quad core Cortex A7 LITTLE (EE) cluster. The power model of a cluster is given by:

$$P(f) = \alpha f^b + \beta N_{p,ac} + P_s(f), \tag{5.2}$$

where the first term is the dynamic power consumption, β is the static power consumption of one processor and $N_{p,ac}$ is the number of active processors on the cluster, $P_s(f)$ is the "uncore" power consumption and f is the frequency level. The "uncore" power consumption is the power consumption from some components not pertaining to a processor, such as a shared cache, an integrated memory controller, and others. Parameters α, b and β, and $P_s(f)$ depend on the platform and cluster type, and they are determined in [LSCS15].

We calculate the total energy consumption for an application graph G mapped onto a cluster heterogeneous MPSoC over one hyperperiod T_G by:

$$E = E^{PE} + E^{EE}. \tag{5.3}$$

E^{PE} in Equation (5.3) contains the total energy consumption of PE clusters and is given by:

$$E^{PE} = T_G \Bigg(\sum_{j=1}^{N_{ac}^{PE}} \Big(u_j^{PE} \alpha^{PE} (f_j)^{b^{PE}} + \beta^{PE} N_{p,ac_j}^{PE} + P_s^{PE}(f_j) \Big) \Bigg), \tag{5.4}$$

where N_{ac}^{PE} is the number of active PE clusters, N_{p,ac_j}^{PE} is the number of active processors on PE cluster j, u_j^{PE} is the total utilization for tasks successfully scheduled by a partitioned scheduling algorithm on the corresponding PE cluster j, f_j is the operating frequency for the corresponding PE cluster j, and α^{PE}, b^{PE} and β^{PE} are the power parameters for PE clusters taken from [LSCS15].

The total energy consumption of EE clusters, E^{EE} in Equation (5.3), is given by:

$$E^{EE} = T_G\Bigg(\sum_{k=1}^{N_{ac}^{EE}}\Big(u_k^{EE}\alpha^{EE}(f_k)^{b^{EE}} + \beta^{EE}N_{p,ac_k}^{EE} + P_s^{EE}(f_k)\Big)\Bigg), \tag{5.5}$$

where N_{ac}^{EE} is the number of active PE clusters, N_{p,ac_k}^{EE} is the number of active processors on EE cluster k, u_k^{EE} is the total utilization for tasks successfully scheduled by a partitioned scheduling algorithm on EE cluster k, f_k is the operating frequency for the corresponding EE cluster k, and α^{EE}, b^{EE} and β^{EE} are the power parameters for EE clusters taken from [LSCS15].

5.7 The Proposed Energy Minimization Approach

In this section, we present our novel energy minimization approach called Data-Parallel Energy Minimization (DPEM) which energy-efficiently exploits a given cluster heterogeneous MPSoC platform when mapping a hard real-time streaming application under a throughput constraint. The logic behind our energy minimization approach is the following: our approach replicates the tasks with heavy utilization to reduce their utilization and lower the operating frequency, thereby reducing the energy consumption; it tries to map as many tasks as possible to EE processors such that the energy consumption is further reduced, while the throughput constraint is met. The DPEM approach is given in Algorithm 6 and explained in Section 5.7.1 while its constituents are desribed in Section 5.7.2 and Section 5.7.3.

5.7.1 The Data-Parallel Energy Minimization Algorithm

In this section, we present our integral algorithm for Data-Parallel Energy Minimization (DPEM). The inputs to DPEM are an SDF graph G, a cluster heterogeneous MPSoC, and a throughput constraint $\mathcal{R}_{\text{out}}$. The outputs are a vector of unfolding factors $\vec{f}_{best}$ according to which each actor in the initial SDF graph should be replicated, the task mapping to processors in the clusters $\mathcal{C}_{best}$, a vector of operating frequencies for clusters $\vec{F}_{best}$ and the minimum

energy consumption of the system E_{best}. The DPEM algorithm is shown in Algorithm 6. Line 1 in Algorithm 6 initializes each unfolding factor of an actor in graph G to 1. In Lines 2 and 3, the initial graph G is converted to periodic tasks by the ISPS approach in Chapter 3, where periods for each actor in G are set, by using scaling factor s in Line 3, to be as large as possible while meeting the throughput constraint $\mathcal{R}_{\text{out}}$. The corresponding hyperperiod T_G of graph G is calculated as well in Line 3. Line 4 finds the bottleneck actor in G. The bottleneck actor is the actor with the heaviest workload among the actor workloads for PE type of processors during one hyperperiod. If multiple actors have the same maximum workload, then the one with the smallest code size is selected to be the bottleneck. Note that stateful actors and input and output actors are not unfolded. In Line 5, Algorithm 7, explained in Section 5.7.2, is applied to classify actors into two groups – EE and PE. Here, by splitting actors into two groups, the required throughput of G under ISPS is guaranteed. Line 6 uses Algorithm 8, described in Section 5.7.3, to energy-efficiently map graph G on the input MPSoC platform. It may happen that the input platform is not big enough to map the input application, that is, graph G. In that case Algorithm 8 will return an empty mapping, $\mathcal{C}_{best} = \emptyset$. If this happens, the algorithm terminates and signals failure in Line 8. Otherwise, after obtaining the initial energy-efficient solution in Line 6, we further search to reduce the energy consumption by exploiting task replication via the unfolding, Lines 9 to 20.

Line 9 checks if the upper bound on the unfolding factor for the bottleneck actor has been reached and if the bottleneck actor is one of the actors which cannot be unfolded (input, output actors and stateful actors). If one of these happens, Algorithm 6 terminates and returns in Line 21 the most energy-efficient solution found so far. Otherwise, the initial SDF graph is transformed into an equivalent CSDF graph by replicating, in Line 10, the bottleneck actor previously found in Line 4. The graph transformation is performed in Line 11 by using the unfolding transformation method given by Algorithm 3, described in Chapter 4. Given that the transformed graph contains more actors than the original one, the WCETs of the actors have to be recomputed because the worst-case communication time may change. This is done in Line 12. Once the WCETs in the CSDF graph are recalculated, actors in the CSDF graph are transformed into periodic tasks by using the ISPS approach in Chapter 3. The unfolding graph transformation is usually used to increase the throughput of a graph by exposing more parallelism through task replication. However, here we want just to meet the same throughput constraint $\mathcal{R}_{\text{out}}$ as the initial graph, and use the unfolding transformation to change the utilization of the periodic

Algorithm 6: Data-Parallel Energy Minimization (DPEM).

Input: An SDF graph $G = (\mathcal{V}, \mathcal{E})$, a cluster heterogeneous MPSoC and a throughput constraint $\mathcal{R}_{\text{out}}$.

Output: Vector of unfolding factors $\vec{f}_{best}$, task mapping to processors in the clusters $\mathcal{C}_{best}$, vector of operating frequencies for clusters $\vec{F}_{best}$ and the minimum energy consumption E_{best}.

1 $\vec{f} = [1, 1, \cdots, 1]$;
2 Calculate WCETs for each actor v_i in G by using Equation (3.2);
3 Calculate period T_i for PE type of processors for each actor v_i in G by using Equation (4.3) and $s = \left\lfloor \frac{\phi_{out} \cdot r_{out}}{\mathcal{R}_{\text{out}} \cdot \text{lcm}(\vec{r})} \right\rfloor$; $T_{best} = T_G = r_{out} \cdot T_{out}$;
4 Find the bottleneck actor $v_{b,k}$ in G;
5 $\mathcal{V}^{EE}$, $\mathcal{V}^{PE} \leftarrow$ Classify actors in G by Algorithm 7(G, $\frac{\phi_{out}}{T_{out}}$);
6 Find $\mathcal{C}_{best}$, $\vec{F}_{best}$, E_{best} by Algorithm 8($\mathcal{V}^{EE}$, $\mathcal{V}^{PE}$);
7 **if** $\mathcal{C}_{best} = \emptyset$ **then**
8 $\quad$ **return** Unschedulable;
9 **while** $f_b < (N_c^{EE} \times N_p^{EE} + N_c^{PE} \times N_p^{PE}) \wedge v_{b,k}$ *not stateful/in/out* **do**
10 $\quad$ $f_b = f_b + 1$;
11 $\quad$ Get G' by unfolding G using the method in Chapter 4 (Algorithm 3);
12 $\quad$ Calculate WCETs for each actor v'_i in G' by using Equation (3.2);
13 $\quad$ Calculate period T'_i for each actor v'_i by using Equation (4.3) and $s = \left\lfloor \frac{\phi'_{out} \cdot r'_{out}}{\mathcal{R}_{\text{out}} \cdot \text{lcm}(\vec{r}')} \right\rfloor$; $T_{G'} = r'_{out} \cdot T'_{out}$;
14 $\quad$ Find the bottleneck actor $v_{b,k}$ in G';
15 $\quad$ $\mathcal{V}'^{EE}$, $\mathcal{V}'^{PE} \leftarrow$ Classify actors in G' by Algorithm 7(G', $\frac{\phi'_{out}}{T'_{out}}$);
16 $\quad$ Find $\mathcal{C}_{best,u}$, $\vec{F}_{best,u}$, $E_{best,u}$ by Algorithm 8($\mathcal{V}'^{EE}$, $\mathcal{V}'^{PE}$);
17 $\quad$ **if** $\mathcal{C}_{best,u} = \emptyset$ **then**
18 $\quad\quad$ **go to 9**;
19 $\quad$ **if** $\frac{\text{lcm}(T_{G'}, T_{best})}{T_{G'}} \cdot E_{best,u} < \frac{\text{lcm}(T_{G'}, T_{best})}{T_{best}} \cdot E_{best}$ **then**
20 $\quad\quad$ $E_{best} = E_{best,u}$, $T_{best} = T_{G'}$, $\vec{F}_{best} = \vec{F}_{best,u}$, $\mathcal{C}_{best} = \mathcal{C}_{best,u}$, $\vec{f}_{best} = \vec{f}$;
21 **return** $\vec{f}_{best}$, $\mathcal{C}_{best}$, $\vec{F}_{best}$, E_{best};

tasks. To meet throughput constraint $\mathcal{R}_{\text{out}}$ and keep the throughput as close as possible to the initial throughput in Line 3, we scale the periods of the periodic tasks obtained after the conversion by scaling factor s, which is given in Line 13. Then, we find in Line 14 the bottleneck actor in the equivalent CSDF graph G', which is replicated in the next pass of the algorithm. The actors in G' are classified into PE and EE actors and the minimum energy of mapping the tasks corresponding to actors in G' onto the MPSoC is calculated in Lines 15 and 16. If there is no feasible mapping we continue with the task replication, Lines 17

and 18. On the other hand, if we could map G' on the MPSoC, the obtained energy is compared against the best, that is, the minimum, energy obtained so far over the same time interval in Line 19. If we detect that the energy consumption of the current solution is smaller than the energy consumption of the best solution found so far, the current solution becomes the best one in Line 20. Line 9 checks whether the termination criteria for Algorithm 6 is met. If it is not, the algorithm will repeat Lines 10 to 20. Otherwise, the best solution is returned in Line 21.

Finally, we can analyze the time complexity of our DPEM algorithm in the worst case. The complexity of Algorithm 6 is determined by the **while loop** in Lines 9 to 20. In the worst case, the while loop will be executed until all the actors in the initial graph are replicated in the equivalent graph maximum number of times, which is equal to the number of processors N in the platform. So, the while loop will be executed $|\mathcal{V}|N$ times in the worst case. The complexity of the graph unfolding algorithm in Chapter 4, Algorithm 3, which is called in Line 11, is $O(|\mathcal{E}|N^2\phi)$, where ϕ is the maximum number of execution phases per actor in the equivalent CSDF graph obtained after unfolding, $\phi = \max_{v_i \in \mathcal{V}'}\{\phi_i\}$. The complexity of the other parts of the while loop is determined by Algorithm 8, see Section 5.7.3. Thus, the worst-case complexity of Algorithm 6 is $O(N|\mathcal{V}| \cdot (N^2\phi|\mathcal{E}| + (N|\mathcal{V}|)^2 \log(N|\mathcal{V}|)))$, which is polynomial.

5.7.2 Task Classification for Energy Minimization

In Algorithm 6, we used Algorithm 7 in Lines 5 and 15 to classify tasks of a graph into two groups, depending on the processor type they should be executed. Selecting the processor type to execute a task in an application is very important because different type of processors in a heterogeneous MPSoC have significantly different power and timing profiles. Algorithm 7 gives our task classification method. It takes a CSDF graph G and a throughput requirement $\frac{\phi_{out}}{T_{out}}$ as inputs and it produces PE and EE subsets of tasks in G.

First, we sort the tasks in order of increasing workload assuming all of them are assigned to EE processors – see Line 1 in Algorithm 7. Then, with the sorted tasks, we use the hyperperiod $r_{out} \cdot T_{out}$ as the classification threshold such that throughput requirement $\frac{\phi_{out}}{T_{out}}$ is met and the energy consumption is minimized, and deploy a binary search algorithm in Line 2 to find the pivotal point by which we can split the sorted tasks into two sets, one for the EE type of processor and another for the PE type of processor. The goal is to put as many tasks as possible to EE processors to reduce the energy consumption while satisfying the throughput requirement. All the tasks, which do not

Algorithm 7: Procedure to classify tasks according to processor type.

Input: A CSDF graph $G = (\mathcal{V}, \mathcal{E})$ and a throughput constraint $\frac{\phi_{out}}{T_{out}}$.
Output: Subsets $\mathcal{V}^{PE}$ and $\mathcal{V}^{EE} \subset \mathcal{V}$.
1 $V \leftarrow$ Sort actors v_i in $\mathcal{V}$ in increasing order of W_i^{EE};
2 $b \leftarrow$ Binary search to find the position in $\mathcal{V}$ with the biggest index where actor v_i can meet $W_i^{EE} \leq r_{out} T_{out}$;
3 $\mathcal{V}^{EE} \leftarrow \mathcal{V}[0 : b]$;
4 $\mathcal{V}^{PE} \leftarrow \mathcal{V} - \mathcal{V}^{EE}$;
5 **return** $\mathcal{V}^{EE}$, $\mathcal{V}^{PE}$;

violate the throughput constraint, that is, the hyperperiod $r_{out} \cdot T_{out}$, when assigned to EE processors are classified as EE tasks, Line 3, and all the rest as PE tasks, Line 4. In this way we guarantee that the throughput requirement will be met while minimizing the energy consumption.

Since the sorting algorithm in Line 1 has the worst-case complexity of $O(|\mathcal{V}| \log |\mathcal{V}|)$ and the worst-case complexity of the binary search in Line 2 is $O(\log |\mathcal{V}|)$, the worst-case complexity of Algorithm 7 is $O(|\mathcal{V}| \log |\mathcal{V}|)$.

5.7.3 Task Mapping for Energy Minimization

In Algorithm 6, once the actors in a graph are classified by Algorithm 7 in Lines 5 and 15 into two sets of EE and PE actors, each set is mapped by Algorithm 8 in Lines 6 and 16 onto the corresponding type of clusters, EE and PE clusters, such that the energy consumption of the whole cluster heterogeneous MPSoC is minimized. Our algorithm of energy-efficient tasks mapping is given in Algorithm 8.

Algorithm 8 takes sets $\mathcal{V}^{EE}$ and $\mathcal{V}^{PE}$ of actors and a cluster heterogeneous MPSoC, and it returns the task mapping on processors in the clusters $\mathcal{C}$, a vector of operating frequencies for clusters $\vec{F}$ and the minimum energy consumption E. The authors in [AY03] showed that the most balanced workload distribution leads to the least energy consumption, and that the the most balanced distribution is obtained when the Worst-Fit Decreasing (WFD) heuristic [CGJ96] is used to allocate tasks to processors. Thus, in this work, we use the WFD heuristic for task allocation. First, Algorithm 8 checks in Lines 1 to 4 whether the input MPSoC has enough resource to map (allocate) and schedule the tasks by using the WFD allocation heuristic [CGJ96], applied among the processors of the same type, and a given per-processor schedulability test [LL73] when processors are running at the maximum available frequency for each processor type. If there is no enough EE type of processors,

Algorithm 8: Procedure to find the minimum energy when the given tasks are mapped onto a cluster heterogeneous MPSoC.

Input: Sets of actors $\mathcal{V}^{EE}$ and $\mathcal{V}^{PE}$ and a cluster heterogeneous MPSoC.
Output: Task mapping to processor in the clusters $\mathcal{C}$, vector of operating frequencies for clusters $\vec{F}$ and the minimum energy consumption E.

1 **if** $\mathcal{V}^{EE}$ *cannot be scheduled on* $N_c^{EE} \times N_p^{EE}$ *processors by WFD algorithm and max frequency* f_{max}^{EE} **then**
2 Move some actors $v_i \in \mathcal{V}^{EE}$ to PE set $\mathcal{V}^{PE}$ in order of non-increasing u_i such that $\mathcal{V}^{EE}$ is schedulable on $N_c^{EE} \times N_p^{EE}$ processors;
3 **if** $\mathcal{V}^{PE}$ *cannot be scheduled on* $N_c^{PE} \times N_p^{PE}$ *processors by WFD algorithm and max frequency* f_{max}^{PE} **then**
4 **return** $\mathcal{C} \leftarrow \emptyset$, $\vec{F} \leftarrow \emptyset$, $E = \infty$;
5 **if** $|\mathcal{V}^{EE}| = 0$ **then**
6 $\mathcal{C}^{EE} \leftarrow \emptyset$, $\vec{F}^{EE} \leftarrow \emptyset$, $E^{EE} = 0$;
7 **else**
8 $n_{lb}^{EE} = \left\lceil \frac{\lceil u^{EE} \rceil}{N_p^{EE}} \right\rceil$, $n_{ub}^{EE} = \min\{ \left\lceil \frac{|\mathcal{V}^{EE}|}{N_p^{EE}} \right\rceil, N_c^{EE} \}$;
9 Find $\mathcal{C}^{EE}$, $\vec{F}^{EE}$, E^{EE} by Algorithm 9(n_{lb}^{EE}, n_{ub}^{EE}, $\mathcal{V}^{EE}$, Equation (5.5));
10 **if** $|\mathcal{V}^{PE}| = 0$ **then**
11 $\mathcal{C}^{PE} \leftarrow \emptyset$, $\vec{F}^{PE} \leftarrow \emptyset$, $E^{PE} = 0$;
12 **else**
13 $n_{lb}^{PE} = \left\lceil \frac{\lceil u^{PE} \rceil}{N_p^{PE}} \right\rceil$, $n_{ub}^{PE} = \min\{ \left\lceil \frac{|\mathcal{V}^{PE}|}{N_p^{PE}} \right\rceil, N_c^{PE} \}$;
14 Find $\mathcal{C}^{PE}$, $\vec{F}^{PE}$, E^{PE} by Algorithm 9(n_{lb}^{PE}, n_{ub}^{PE}, $\mathcal{V}^{PE}$, Equation (5.4));
15 $\mathcal{C} = \{\mathcal{C}^{EE}, \mathcal{C}^{PE}\}$, $\vec{F} = \{\vec{F}^{EE}, \vec{F}^{PE}\}$, $E = E^{EE} + E^{PE}$;
16 **return** $\mathcal{C}$, $\vec{F}$, E;

we select some actors from set $\mathcal{V}^{EE}$ and assign them to set $\mathcal{V}^{PE}$. The actors are selected in order of decreasing utilization and the selection is terminated as soon as the tasks corresponding to actors in set $\mathcal{V}^{EE}$ are schedulable on the EE processors. However, if there is no enough PE type of processors, that means the application is not schedulable on the input MPSoC. The algorithm terminates and signals the failure by returning an empty set for tasks-to-processors mapping $\mathcal{C}$ in Line 4. Line 5 checks if there are tasks that should be mapped on processors in EE clusters. If no task should be mapped to EE clusters, then EE clusters will not be used within the input MPSoC, hence they will not contribute to the total energy consumption, Line 6. Otherwise, the bounds on the number of active EE clusters are calculated in Line 8 and the energy consumption of mapping task set $\mathcal{V}^{EE}$ to EE clusters is calculated in Line 9.

The lower bound n_{lb}^{EE} corresponds to the minimum possible number of active clusters to schedule the tasks because it is determined according to the ceiling of the utilization u^{EE} of EE tasks. The upper bound n_{ub}^{EE} is selected to be the minimum value among the case when tasks are mapped onto processors in one-to-one manner, and the case when all clusters available on the platform are active. We find the minimum energy for mapping the tasks on EE clusters by using Algorithm 9 (described later) in Line 9. Similarly, Line 10 checks whether there are tasks that should be mapped onto processors in PE clusters. If there are such tasks, lower and upper bounds of active PE clusters are calculated in Line 13 and the minimum energy for mapping the tasks on PE clusters by using Algorithm 9 is obtained in Line 14. Finally, the EE solution and the PE solution mappings are grouped together in Line 15 and the integral solution mapping of the given tasks onto the given MPSoC which results in minimum energy consumption is returned in Line 16 of Algorithm 8.

Within Algorithm 8, described above, Algorithm 9 is used to map the tasks which are in the same group, EE or PE, such that the energy consumption is minimized. Algorithm 9 takes the bounds on the number of active clusters of certain type (PE or EE), n_{lb} and n_{ub}, tasks $\mathcal{V}$ that are going to be mapped onto PE/EE clusters, the corresponding equation, Equation (5.4) or (5.5) – see Section 5.6, for the calculation of the energy consumption and returns the task partitions among the processors in the clusters $\mathcal{C}_{best}$ and a vector of operating frequencies for clusters $\vec{F}_{best}$ which lead to the minimal energy consumption E_{best}. In Lines 2 to 15 in Algorithm 9, the best task mapping and the frequency assignment is determined among different number of active clusters in the range from n_{lb} to n_{ub}. For each number of active clusters n, $n \in [n_{lb}, n_{ub}]$, the algorithm in Line 4 performs the WFD allocation heuristic [CGJ96] and uses a given per-processor schedulability test [LL73] to check the schedulability of the tasks. In this way, we want to achieve load balancing among the processors of the same type. If all tasks are allocated on processors, Line 5, we group processors into clusters according to their workload such that all processors in one cluster run at the frequency which matches their workload as much as possible. This is done in Lines 6 and 7, where processors $\pi_j \in \Pi$ are first sorted in non-increasing order of their workload, that is, their utilization u_j, and then starting from the processor with the highest utilization, every N_p processors are grouped into a cluster. For each cluster, we select the smallest frequency which guarantees the schedulability and is supported by the cluster type, that is, it is in the set $\mathcal{F}$ of available frequencies, Lines 9 to 11. The energy consumption of the mapping is calculated in Lines 12 and 13 of Algorithm 9. In Lines 14 and 15, we check whether the energy consumption obtained by

Algorithm 9: Procedure to find the minimum energy when the given tasks are mapped onto the same type of clusters.

Input: Lower n_{lb} and upper n_{ub} bound on the number of clusters, set $\mathcal{V}$ of tasks that should be mapped onto clusters, equation *Eq.* for calculating the energy consumption.

Output: Task mapping to processor in the clusters $\mathcal{C}_{best}$, vector of operating frequencies for clusters $\vec{F}_{best}$ and the minimum energy consumption E_{best}.

1 $E_{best} = \infty$, $\vec{F}_{best} \leftarrow \emptyset$, $\mathcal{C}_{best} \leftarrow \emptyset$;
2 **for** $n = n_{lb}$ *to* n_{ub} **do**
3 Create a set Π of $n \times N_p$ empty processors, $\forall \pi_j \in \Pi : u_j = 0$;
4 Perform WFD allocation heuristic and a corresponding schedulability test for all tasks in $\mathcal{V}$;
5 **if** *all* $v_i \in \mathcal{V}$ *can be scheduled on* Π **then**
6 $\Pi \leftarrow$ Sort Π in non-increasing order of u_j;
7 $\mathcal{C} \leftarrow$ group every N_p processors in Π to a cluster C_k, $k \in [1, n]$;
8 $E = 0$, $F_k = 0, k \in [1, n]$;
9 **for** *cluster* $C_k \in \mathcal{C}$ **do**
10 Find processor $\pi_j \in C_k$ with the highest utilization u_j, $u_{max} = u_j$;
11 Compute frequency of C_k as $F_k \geq u_{max} \cdot f_{max} \wedge F_k \in \mathcal{F}$;
12 Calculate energy E_k for cluster C_k by using *Eq.*;
13 $E = E + E_k$;
14 **if** $E < E_{best}$ **then**
15 $E_{best} = E$, $\vec{F}_{best} \leftarrow \vec{F}$, $\mathcal{C}_{best} \leftarrow \mathcal{C}$;
16 **return** E_{best}, $\vec{F}_{best}$, $\mathcal{C}_{best}$;

mapping the tasks on the current number of active clusters n is the smallest one obtained so far. If that is the case, the mapping on the current number of active clusters becomes the best mapping solution. Finally, in Line 16, Algorithm 9 returns the minimum energy E_{best} obtained after mapping the tasks on clusters of the same type, the frequency assignment $\vec{F}_{best}$ for clusters and the cluster partitions $\mathcal{C}_{best}$.

Let us now analyze the time complexity of Algorithm 9 and Algorithm 8 in the worst case. The complexity of Algorithm 9 is determined by the **for loop** in Lines 2 to 15. Due to the sorting algorithms used within the WFD heuristic, in Lines 4, and in Line 6, the complexity of Algorithm 9 is $O(N_c|\mathcal{V}|\log|\mathcal{V}|)$, where N_c is the number of active clusters. The worst-case complexity of Algorithm 8 is then determined by Line 2, which is executed in the worst case $|\mathcal{V}|$ times, and every time the WFD allocation heuristic is applied, thus the complexity of Algorithm 8 is $O(|\mathcal{V}|^2\log|\mathcal{V}|)$.

Table 5.2: *Benchmarks used for evaluation.*

Benchmark	$\lvert\mathcal{V}\rvert$	$\lvert\mathcal{E}\rvert$	$\mathcal{R}_{\text{out}}$[1/time unit]
Discrete cosine transform (DCT)	8	7	1/47616
Fast Fourier transform (FFT)	17	16	1/12032
Filterbank	85	99	1/11312
Time delay equalization (TDE)	29	28	1/36960
Data encryption standard (DES)	53	60	1/1024
Serpent	120	128	1/3336
Bitonic Sorting	40	46	1/95
MPEG2	23	26	1/7680
Vocoder	114	147	1/9105
FMRadio	43	53	1/1434
Channel Vocoder	55	70	1/35500

5.8 Evaluation

We have performed three experiments to evaluate the efficiency of our DPEM approach in comparison to the related energy minimization approaches in [CKR14], [LSCS15], [SDK13] and [Lee09]. We have selected the approaches in [CKR14] and [LSCS15] for comparison because they consider the same task and system models as we do. We selected to compare with the approach in [SDK13] because it is a very good representative among the approaches for energy-efficient mapping and scheduling of streaming applications modeled as SDF graphs. Finally, we compare our approach with the approach in [Lee09] which is the only approach among the related approaches which considers task replication for energy minimization for classical periodic real-time tasks. In the first two experiments, we compare the approaches when the streaming applications are executed on a cluster heterogeneous platform. We apply our task classification method, given in Algorithm 7, for the approaches in [SDK13] and [Lee09] which were originally devised for homogeneous platforms and then we apply these approaches on the two sets of tasks, PE and EE, obtained by the classification. Since two of the related approaches, [SDK13] and [Lee09], originally consider homogeneous platforms with per-core VFS capability, in the third experiment, we compare our approach with these related approaches on this type of platform.

The experiments have been performed on the real-life applications from the StreamIt benchmarks suit [TA10], given in Table 5.2. $\lvert\mathcal{V}\rvert$ denotes the number of actors in an SDF graph, while $\lvert\mathcal{E}\rvert$ denotes the number of communication channels. $\mathcal{R}_{\text{out}}$ is the maximum achievable throughput, computed by using Equation (3.5) and (3.22), when the applications are scheduled by the ISPS approach described in Chapter 3. We consider these throughput values as the

throughput constraints in our experiments.

In the experiments on heterogeneous MPSoC platforms, we consider the same MPSoC platforms considered in [LSCS15]. These platforms have the same number of PE processors and EE processors but they have different cluster granularities, that is, different number of processors per cluster, and hence, different number of clusters. We use the same MPSoC notation MPSoC_-x_pe_ee as in [LSCS15]. For example, MPSoC_2_20_28 corresponds to an MPSoC platform with 2 processors per cluster, 20 PE clusters and 28 EE clusters. The approaches in [CKR14], [LSCS15] and [Lee09] use hard real-time scheduling algorithms to schedule the tasks on an MPSoC while the approach in [SDK13] uses self-timed scheduling. The application tasks are permanently assigned to processors in [CKR14], [Lee09] and [SDK13], while in [LSCS15], the tasks are permanently assigned to clusters, but within a cluster tasks are scheduled by a global scheduling algorithm, hence, they can migrate. In the experiments, we use the EDF [LL73] scheduling algorithm within our DPEM approach which is also used in [CKR14] and [Lee09]. In all experiments, we use the power parameters in [LSCS15] obtained from real measurements performed on the ODROID XU-3 [ODR] board. The results of the evaluations are shown in Figure 5.3, Figure 5.4 and Figure 5.5. In all these figures, we show the energy reduction obtained by our DPEM approach in comparison with the related approaches. The energy reduction r is computed by:

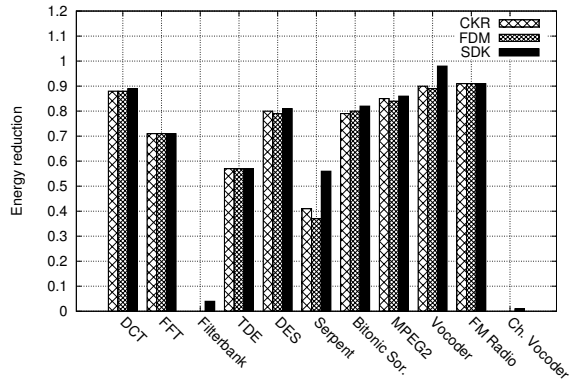
$$r = \frac{E_{rel} - E_{DPEM}}{E_{rel}}, \tag{5.6}$$

where E_{rel} is the energy consumption of an application to MPSoC mapping configuration obtained by a related approach and E_{DPEM} denotes the energy consumption achieved by our DPEM approach.

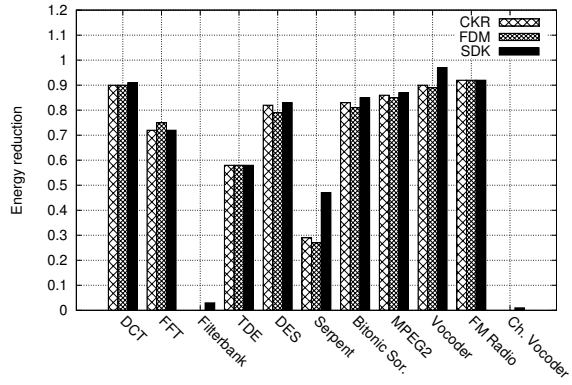
5.8.1 Comparison with [CKR14], [LSCS15], [SDK13] on Heterogeneous MPSoCs

In this section, we compare the energy consumption on cluster heterogeneous MPSoCs obtained by our proposed DPEM approach with the energy consumption delivered by the related approaches which do not consider task replication [CKR14] – CKR, [LSCS15] – FDM, [SDK13] – SDK.

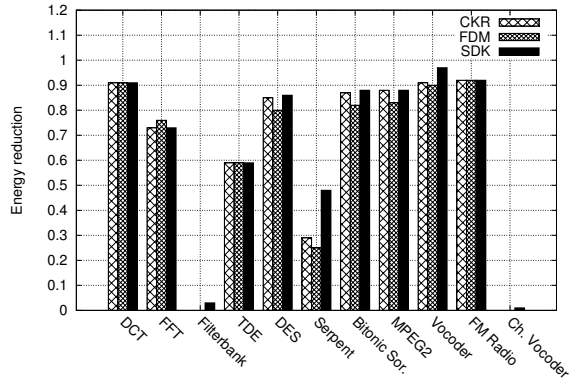
The comparison results with the CKR, FDM and SDK approaches on the three considered heterogeneous MPSoCs are given in Figure 5.3(a)-5.3(c). In each of these figures, the x-axis shows the application benchmarks and the y-axis shows the energy reduction. Both approaches CKR and FDM are devised for cluster heterogeneous MPSoCs and both of them use preemptive



(a) MPSoC_2_20_28 (higher is better)



(b) MPSoC_4_10_14 (higher is better)



(c) MPSoC_8_5_7 (higher is better)

Figure 5.3: *Comparison of our proposed DPEM approach with related approaches on heterogeneous MPSoCs.*

hard real-time scheduling algorithms, which is also the case in our DPEM algorithm. We can see in Figure 5.3 that our DPEM approach reduces the energy consumption when compared to CKR and FDM for all but two considered benchmarks. The two benchmarks for which our approach results in the same energy consumption as CKR and FDM are Filterbank and Channel Vocoder. The workload which these two benchmarks put on the considered MPSoCs is balanced among processors, hence our approach will not replicate tasks of these benchmarks which leads to the same energy consumption as obtained by the CKR and FDM approaches. The average energy reduction of our approach when compared to the CKR approach is 62%, 62% and 63.1% for the three MPSoCs with 2, 4 and 8 processors per cluster, respectively. When compared to the FDM approach the corresponding average energy reductions are 61.6%, 61.4% and 61.6%. When compared to the SDK approach, our approach achieves energy reduction for all benchmarks because we use both task replication and preemptive scheduling. Note that we only use the design-time phase in the SDK approach for the comparison because our approach is a design-time approach. Our approach obtains on average the energy reduction of 65%, 65.1% and 66% for the three MPSoCs with 2, 4 and 8 processors per cluster, respectively, when compared to the SDK approach. We can conclude from these results that our approach achieves large energy reduction by utilizing task replication.

5.8.2 Comparison with [Lee09] on Heterogeneous MPSoCs

In this section, we compare the energy consumption on cluster heterogeneous MPSoCs of our DPEM approach with the related approach in [Lee09], denoted by WYL, which considers task replication as well. The results are given in Figure 5.4. Here again, both approaches will not replicate tasks in Filterbank and Channel Vocoder and hence both approaches will lead to the same energy consumption in these two cases. Given that the task classification in the WYL approach is based on the power consumption curve of a processor, the WYL approach will never replicate tasks assigned to EE processors. In addition, the WYL approach will never replicate the tasks of an application once the total number of heavy tasks is equal to the number of processors on an MPSoC platform. All these limitations of WYL explain the energy reduction achieved when our approach is used to map the benchmarks in Table 5.2 onto the three considered MPSoCs. The average energy reduction obtained by our DPEM approach is 51.3%, 57.2% and 60.7% for the MPSoCs with 2, 4 and 8 processors per cluster, respectively.

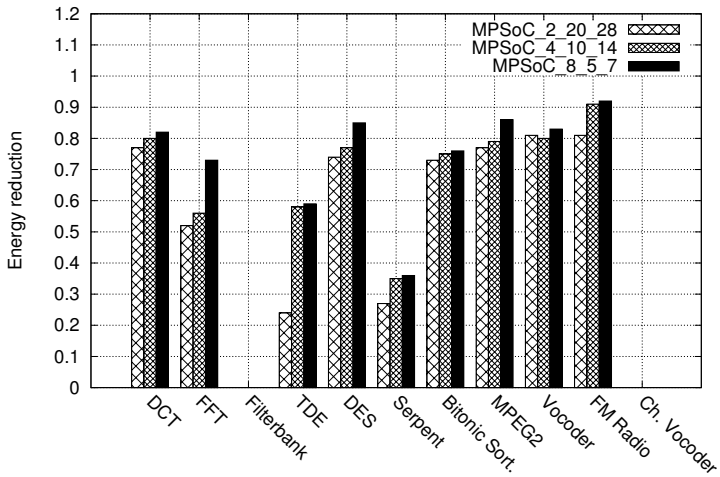


Figure 5.4: *Comparison between DPEM and WYL on heterogeneous MPSoCs.*

5.8.3 Comparison on Homogeneous MPSoC

Given that both the SDK and WYL approaches were originally proposed for homogeneous platforms with per-core VFS capability, in this section, we compare the energy consumption on such systems when our DPEM approach is used with the energy consumption values when the SDK and WYL approaches are used. The results of the energy reduction on a homogeneous MPSoC platform consisting of 96 PE processors with per-core VFS capability are given in Figure 5.5. Here, we also give the results of energy reduction when our DPEM approach is compared with the CKR and FDM approaches for completeness.

The benchmarks Filterbank and Channel Vocoder were the only two benchmarks for which our approach could not obtain any reduction in energy consumption on heterogeneous MPSoCs when compared to the approaches which use hard real-time scheduling algorithms – CKR, FDM and WYL. In the case of a homogeneous platform, we can see in Figure 5.5 that there is still no difference in energy consumption between our DPEM and the CKR, FDM and WYL for the Filterbank benchmark. This happens because when mapped onto a homogeneous MPSoC, Filterbank has balanced workload among the processors, hence both our DPEM and the WYL approaches will not replicate tasks. However, in the case of the Channel Vocoder benchmark, we see in Figure 5.5 that the situation changes, that is, now there is a reduction in the energy consumption when our approach is compared to the CKR and FDM, because our approach will replicate tasks to balance the workload of Channel

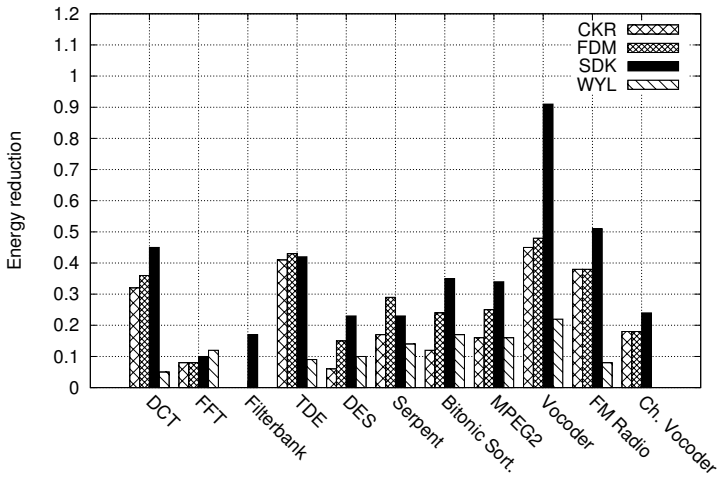


Figure 5.5: *Comparison on homogeneous MPSoC.*

Vocoder on a homogeneous platform. The WYL approach will replicate tasks as well, leading to the same energy consumption as obtained by our DPEM approach. Although the WYL approach was devised for homogeneous platforms with types of processors which match the PE type and with per-core VFS capability, still our DPEM approach outperforms the WYL approach by reducing the energy consumption on average by 10.4%, and in the best case up to 22%. The reason is that our task replication procedure is more flexible than the procedure in the WYL approach.

When compared to the another approach devised for homogeneous MPSoCs with per-core VFS capability, that is, the SDK approach, our DPEM approach leads to an energy reduction of 36% on average and up to 90% in the best case. The reason is that our approach replicates tasks to lower the utilization per-processor, and hence, lower operating frequencies can be achieved. In addition, the SDK approach minimizes only the dynamic energy consumption and uses non-preemptive scheduling which both lead to higher total energy consumption.

Finally, when compared to the CKR and FDM approaches on a homogeneous platform, our DPEM approach delivers systems with energy reduction of 21.2% and 25.6% on average, respectively. The main reason is the task replication which our approach uses to lower the utilization per processor while keeping the application throughput.

We performed an additional experiment to evaluate the influence of the

number of processors in an MPSoC on the energy reduction of our DPEM approach in comparison with the related approaches. In this experiment, beside the MPSoC platform with 96 PE processors, we considered two additional platforms with 48 and 192 PE processors with per-core VFS capability. On the 48-processor platform, our DPEM approach resulted in the energy reduction of 6%, 18.5%, 20.5% and 30.3% when compared with the WYL, CKR, FDM and SDK approaches, respectively. In comparison with the same related approaches, our DPEM approach obtains on the 192-processor platform the following energy reductions – 10.7%, 24.9%, 29.4% and 39.8%. We can conclude that the energy reduction of our approach with regard to the related approaches slowly increases with the increase of the number of processors in the platform.

5.8.4 Overhead and Time Complexity Analysis

In this section, we briefly discuss the code and data memory overhead of our approach when compared to the related approaches and the time complexity of our and the related approaches. The code and data memory overhead of our approach on heterogeneous platforms when compared to the WYL approach is 2 times higher on average, and 2.3 times higher on average than the approaches which do not consider task replication, that is, approaches CKR, FDM and SDK. The memory overhead of our DPEM approach on the homogeneous platform is 16% higher on average when compared to the WYL approach, and 85% higher on average when compared to the CKR, FDM and SDK approaches. Given that the actual memory increase in the worst case is 213 KB and given the size of memory available in modern embedded systems, we can conclude that the memory overhead introduced by our approach is acceptable.

The time complexity in the worst-case of our DPEM approach and the approaches CKR, FDM and WYL is polynomial, while the worst-case time complexity of the SDK approach is exponential. In the worst-case, our approach needs 62 minutes, the WYL approach needs 5 minutes, the CKR approach takes 11 minutes, the FDM less than 1 second and the SDK approach needs 6 days to find an energy-efficient solution. Given that our DPEM approach is a design-time approach and that it delivers solutions of better quality, we can conclude that our approach outperforms the related approaches.

5.9 Discussion

In this chapter, we proposed a novel energy minimization mapping approach to reduce the energy consumption of embedded multiprocessor streaming systems with throughput constraints. To map energy-efficiently an SDF graph onto cluster-heterogeneous MPSoC, our polynomial-time solution approach: 1) determines a processor type for each task in an SDF graph such that the throughput constraint is met and the energy consumption is minimized; 2) determines a replication factor for each task in an SDF graph such that the distribution of the workload on the same type of processors is balanced, which enables processors to run at a lower frequency, hence reducing the energy consumption. The experiments on a set of real-life streaming applications showed that our approach reduces energy consumption by 66% on average among all the performed experiments while meeting the same throughput requirement when compared to related energy minimization mapping approaches.

Chapter 6

An Accurate Energy Modeling of Streaming Systems

Jelena Spasic and Todor Stefanov, "An Accurate Energy Model for Streaming Applications Mapped on MPSoC Platforms", *In Proceedings of the IEEE International Conference on Embedded Computer Systems: Architectures, MOdeling, and Simulation (IC-SAMOS'13)*, pp. 205–212, Samos, Greece, July 15-18, 2013.

THE solution to the problem of accurate energy modeling of an application-to-MPSoC mapping, that is, **Problem 4** introduced in Section 1.3, is presented in this chapter.

The investigated research problem is further described in Section 6.1. It is followed by a summary of our contributions in Section 6.2. The related work is addressed in Section 6.3. The considered system model is described in Section 6.4. The energy model formulation and the procedure to extract the parameters of the energy model are given in Section 6.5. The model is experimentally evaluated in Section 6.6. The concluding discussion is given in Section 6.7.

6.1 Problem Statement

As discussed in Sections 1.2.2 and 1.3, finding an efficient application-to-platform mapping is the key issue for optimizing the energy consumption and performance of streaming MPSoC systems. Since there are many possible application-to-platform mapping combinations forming a design space, this design space should be efficiently explored by using high-level system

performance/energy models. Early in the design process of a system with certain performance/energy requirements, the design space is very large and decisions taken at higher level of abstraction have greater impact on the final design in terms of system performance and energy consumption. Therefore, high-level performance/energy models of a system should be accurate enough to steer the selection of optimal design points under given constraints in the right direction. Model accuracy is usually traded-off for modeling and evaluation effort. Especially accuracy of energy models is very important as the International Technology Roadmap for Semiconductors (ITRS) [Int11] reports that the power/energy consumption is the dominating constraint in the new generations of embedded systems.

In the embedded systems domain the research and results on performance modeling are very mature, while the research on system-level power/energy modeling and estimation has received attention only in recent years. So far, research on power/energy modeling has been mainly done for a single system component in isolation [QKUP00, LJSM04, SC01, VJD+07, BTM00, YVKI00, CAC, PPKD10, BZZ04, KLPS09, BVC04]. Only in a few cases, the power/energy consumption of the whole system has been modeled [LPB04, C+10, HLF+11, RAN+11, SRH+11, PP12]. However, in most cases power/energy consumption due to the contention on shared resources is not considered. Moreover, in most cases, characterization and validation of the models have been done by using lower-level simulators or data-sheet values [QKUP00, BTM00, YVKI00, KLPS09, BVC04, LPB04, HLF+11, PP12], introducing additional inaccuracy in the model. Therefore, in order to find accurately an energy optimal application-to-platform mapping: 1) the energy model should describe the system as a whole and take into account the parallel nature of MPSoCs and possible energy consumption due to contention on shared resources; 2) the energy modeling and estimation should be done with high level of accuracy and efficiency. For the above mentioned reasons, we address the problem of accurate and efficient energy modeling of an application-to-platform mapping when a streaming application is modeled using the Polyhedral Process Network (PPN) [VNS07] MoC and mapped onto a tile-based MPSoC platform with distributed memory.

6.2 Contributions

Our energy model describes the system as a whole as well as it considers and models accurately the energy consumption due to data communication among the processors in a platform and the contention on non-contention-free communication infrastructures. The model is based on the well-defined prop-

erties of the PPN application model and the values of important energy model parameters are obtained by real measurements of energy consumption for the accuracy reason. It models the total (static and dynamic) energy consumption and is applicable to different types of processors. The energy model is integrated in the existing Daedalus design flow [TNS+07], enabling a system designer to explore a large design space starting from a high-level description of the system behavior and having energy consumption as a primary design constraint.

6.3 Related Work

Research on power/energy modeling has been mainly done for individual system components in isolation – processors [QKUP00, LJSM04, SC01, VJD+07, BTM00, YVKI00], memories [CAC], interconnections [PPKD10, BZZ04, KLPS09, BVC04]. In contrast, our energy model models the system as a whole and thus enables more accurate energy estimation and exploration of different application-to-platform mappings.

Only a few works deal with power/energy modeling of the whole system. [LPB04] analyzes power distribution among components in a homogeneous shared bus based MPSoC platform. However, there is no accuracy information for any model of a component in the system. In contrast, our energy model is more general in the sense that it can model platforms with contention-free and different configurations of non-contention-free communication infrastructures. In addition, we provide accuracy information concerning the obtained energy estimates.

[C+10] presents the performance and power modeling of multi-programmed multi-core systems. In this work, it is assumed that there is no data dependency between the processes running on a platform. The model is characterized and validated by real measurements. However, real applications usually consist of data dependent processes, and thus the energy consumption due to communication between the processes should be considered. In contrast to [C+10], our model considers the data dependency between the processes, and hence the energy consumption due to interprocessor communication is modeled.

[HLF+11] presents a multi-core power modeling and estimation tool flow which consists of two tools: *PowerMixer*^{IP}, an IP power model builder, and *PowerDepot*, a power estimation tool which generates and embeds power monitors into a SystemC simulation environment. Power model characterization and validation are done by using transistor-level and gate-level simulations.

The authors report an average power estimation error of 2% compared to gate-level simulations which accuracy is not known. In addition, the contention on shared resources is not discussed in [HLF+11]. In contrast, our energy model considers the contention on different kinds of non-contention-free communication structures with the energy estimates close to real energy measurements.

The FLPA power estimation methodology for MPSoCs is presented in [RAN+11]. The power consumption estimation consists of two parts: 1) power model development – a system is divided into functional blocks, and the power consumption is evaluated for selected activity parameters; 2) activity estimation and power calculation – a transaction level SystemC simulator and an Instruction Set Simulator (ISS) are used for detection of the activities. Models are characterized and validated by real measurements. Power modeling of shared resources and the contention on shared resources are not discussed in detail. In contrast, we give a general methodology for modeling the energy consumption for both contention-free and non-contention-free communication infrastructures. By considering the energy consumption due to the contention on non-contention-free communication structures we achieve energy estimates close to real energy measurements.

[SRH+11] proposes a top-down power and performance estimation methodology for MPSoCs. The system architecture is modeled by a set of resources – processors, memories, interconnects, and dedicated hardware resources. Each resource is characterized by power and performance attributes. Power costs of the power attributes are extracted from measurements. There is no information about the accuracy of the proposed model and modeling of contention on non-contention-free communication infrastructures is not considered. In contrast, our energy model is more detailed, and consequently highly accurate with accuracy numbers obtained by comparison with real measurements. In addition, our work considers various contention-free and non-contention-free communication infrastructures in the energy modeling.

In terms of application and platform models, the closest work to ours is [PP12]. An application is modeled as a Kahn Process Network (KPN) where every process has *read*, *execute* and *write* events. The proposed power modeling technique estimates the power consumption of an application-to-FPGA MPSoC mapping based on "event signatures". The "event signatures" for *execute*, *read* and *write* events are used together with a micro-architecture description, lower-level simulators and some additional parameters obtained from literature and through synthesis to calculate the power consumption of an application-to-MPSoC mapping. The model is validated by comparison

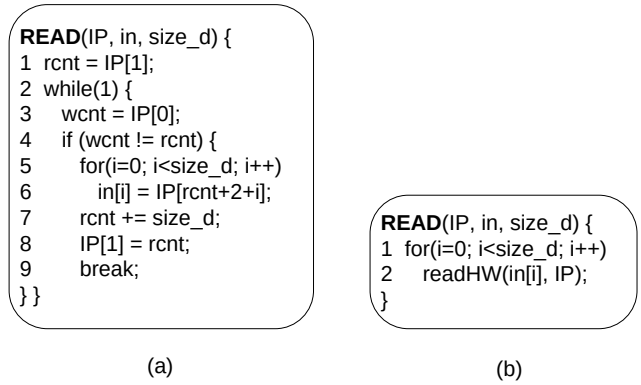


Figure 6.1: *The read primitive implemented in software (a) and hardware (b).*

to measurements. However, it is not clear how the "scaling factors" used for pre-calibration of the power models for interconnections and memories are obtained and what the relation is between these factors and application/MPSoC properties. This fact does not give high credibility to the accuracy of the model. Moreover, the authors assume that the data communication transactions performed by the KPN application model are not interleaved at the architecture level. In contrast, our energy model considers contention on shared resources and its parameters are extracted from measurements, which make the model very accurate. In addition, we do not use scaling factors and thus the accuracy of our model is highly credible.

6.4 System Model

Since our energy model is based on the well-defined properties of the PPN application model and the MPSoC platform model, in this section, we first give more details on the PPN application model presented in Chapter 2, and then describe the MPSoC platform model we consider in this chapter.

6.4.1 Application Model

An example of a PPN and the structure of its process P_3 is given in Figure 2.2. Each process has a set of channels it reads from, a set of channels it writes to, and a function that represents a computation performed on input data that generates output data. A read/write from/to a channel is realized by blocking read/write primitives implemented in *software* (SW) or *hardware*

(HW). Figure 6.1 gives the structure of the read primitive implemented in software and hardware. In case of the SW read primitive, blocking FIFO access is implemented in software: *check for data*, see Figure 6.1(a) lines 1, 3 and 4, *read data*, see Figure 6.1(a) lines 5 and 6, and *release space*, see Figure 6.1(a) lines 7 and 8. In case of the HW read primitive, blocking FIFO access is encapsulated in the *readHW* function and realized in hardware, see Figure 6.1(b). As explained in Chapter 2, the execution of a PPN process represents a process domain, described by using the polytope model [Fea96b]. In addition, accessing input and output ports of the PPN process is represented by the corresponding input and output port domains which are subsets of the process domain. By counting the integer points in the process domain polytope, we can determine the number of iterations each process function is executed. Similarly, by counting the integer points in the corresponding input/output port domain we can determine the number of read/write accesses for each channel of a process. Counting of the integer points in a polytope can be done automatically by using the *Barvinok* library in the `pn` compiler [VNS07]. The counting ability of the PPN model is used in Section 6.5.2 for the computation of the so-called N energy model parameters N^{r_k}, N^{w_k} and N^{F_k}. In the example given in Figure 2.2(b), by counting the integer points (i, j) in the process domain D_{P3} we can see that function F is executed 128 times and by counting the integer points in the port domains D_{IP1} and D_{OP2} we obtain that channel $CH3$ is read 48 times and channel $CH5$ is written 48 times.

6.4.2 Platform Model

In this work, we consider tile-based MPSoC platforms with distributed memory. The generic architecture template of our platforms is shown in Figure 6.2. A programmable processor with its local data and program memory, a timer, and a bus bridge constitute a processing tile within the platform. Different processing tiles can have different types of processors. The communication infrastructure consists of a contention-free or non-contention-free communication component and distributed communication memories (every tile has its own communication memory). A contention-free communication component is a point-to-point (P2P) medium where every channel in the PPN application model has its own communication link. Non-contention-free communication components are mediums with shared communication links – a shared bus (ShB) or a crossbar switch (CB). Communication memories are assumed to be dual-port memories. This means that the communication memory can be accessed by its own processing tile and a remote processing tile at the same time. The processing tile produces data to its communication memory, lo-

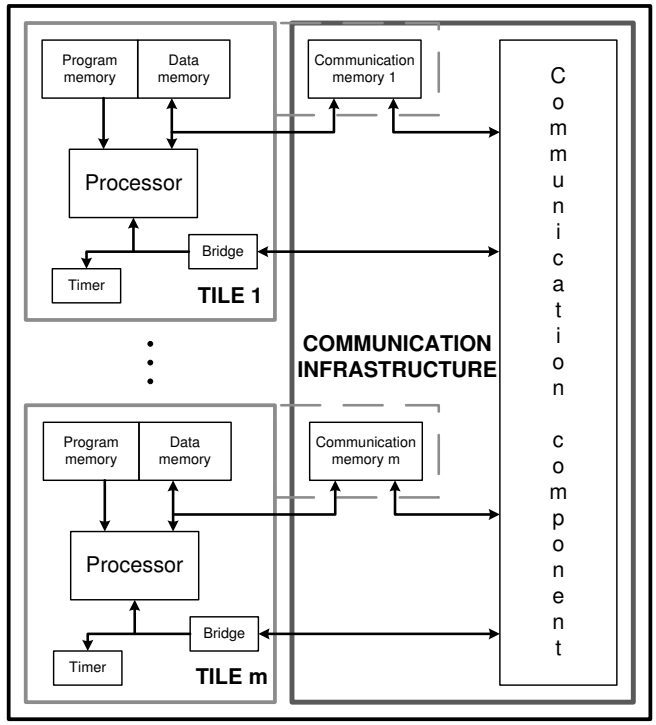


Figure 6.2: *The architecture template of MPSoC platforms.*

cally accessing it through the local data bus, and consumes data from its own and/or other communication memories remotely through the communication component. Within our platforms, HW read/write primitives are used for P2P communication components, while SW read/write primitives can be used for both P2P and shared (CB, ShB) communication components.

More formally, a platform can be represented as a directed graph $\Pi = (\pi, CL)$, where $\pi = \{\pi_1, \pi_2, ..., \pi_m\}$ is a set of m processing tiles (homogeneous or heterogeneous) and $CL \subseteq \pi \times \pi$ is a set of physical communication links between the tiles.

6.4.3 Application-to-Platform Mapping

The mapping of an application modeled as a PPN $G = (\mathcal{P}, \mathcal{C})$ onto a platform $\Pi = (\pi, CL)$ can be expressed as a tuple $M = (\mathcal{P}^m, \mathcal{C}^m)$, where $\mathcal{P}^m = \{P_1^m, P_2^m, ..., P_m^m\}$ is an m-partition of set $\mathcal{P}$ of the processes, and $\mathcal{C}^m = \{CH_1^m, CH_2^m, ..., CH_m^m\}$ is a set of communication channels constructed from the set $\mathcal{C}$ of all channels in a PPN. A subset P_i^m represents the set of processes mapped

onto tile π_i. These processes produce data to the communication memory that is assigned to tile π_i. If the number of processes of a PPN is greater than the number of processing tiles in a platform, then some of the tiles execute more than one process. In this case, static schedule of processes is derived for every tile. This is done automatically by using the `pn` compiler [VNS07]. Each channel $CH_l^m \in \mathcal{C}^m$ corresponds to one channel $CH_l = (P_i, P_j) \in \mathcal{C}$ and is given by a tuple $(proc(P_i), proc(P_j))$, where $proc(P_i) = \pi_i$ represents the processor π_i on which process P_i is mapped.

Recall that in our platforms FIFO channels reside in the communication memories and reading from channels is performed remotely through the communication component, while writing to channels is done locally by a processor through the local tile's bus on which the processor is the only master, see Section 6.4.2. This means that writing is always contention-free, while reading is not because non-contention-free communication component may be used in the platform. In case of non-contention-free communication components, for each application-to-platform mapping, we define *Read contention* matrix $\mathbf{R} \in \mathbb{N}^{m \times m}$ as:

$$R_{ij} = \begin{cases} 1, & \exists CH_l^m = (\pi_i, \pi_j) \in \mathcal{C}^m \\ 0, & \text{otherwise} \end{cases} \tag{6.1}$$

The matrix is used in Section 6.5.2 to analyze the influence of contention on the energy consumption of an application-to-platform mapping.

6.5 Energy Model

The proposed energy model is used to estimate the energy consumption of a mapping of a streaming application modeled as a PPN described in Sections 2.1.2 and 6.4.1 onto an MPSoC platform modeled as described in Section 6.4.2. The energy model relies on the properties of the PPN application model and the platform model. The following subsections describe our energy model.

6.5.1 Model Formulation

Without loss of generality and for the sake of clarity, we assume in the following that each process within an application is mapped to a different processing tile in a platform, that is, the number of processes is equal to the number of processing tiles. In general, the proposed model is applicable to any application-to-platform mapping given that multiple processes of an application can be

grouped and represented as a single process by finding a sequential schedule between the processes, as explained in Section 6.4.3.

Since the PPN representation of an application is a set of concurrent processes, we can express the energy consumption of the application-to-platform mapping $E_{app \to pla}$ as the sum of energies consumed by processes E_{P_k}:

$$E_{app \to pla} = \sum_{k=1}^{n} E_{P_k}. \tag{6.2}$$

A PPN process reads input data from (a part of) input channels, performs computation on input data and generates output data which is further written to (a part of) output channels (see Figure 2.2(b)). Read and write accesses to channels are blocked if the required data is not available or if there is no space for new data. Having this in mind, we can express the energy consumed by a process E_{P_k} as:

$$E_{P_k} = E_{RD_k} + E_{EXE_k} + E_{WR_k} + E_{BLK_k} + E_{CTRL_k}, \tag{6.3}$$

where E_{RD_k} and E_{WR_k} are the energies consumed by reading from and writing to channels without blocking, respectively; E_{EXE_k} is the energy consumed by performing the computation in the process; E_{BLK_k} is the energy consumed while the process is blocked on read and write, and E_{CTRL_k} is the energy consumed by control structures in the process code. In the example given in Figure 2.2(b), E_{CTRL_k} corresponds to the control structures in lines 1, 2, 4, 6, 9 and 11. Further, E_{RD_k} and E_{WR_k} can be expressed as:

$$E_{RD_k} = \sum_{r_k} N^{r_k} (E^{r_k}_{RD_k} + c \cdot E^{r_k}_{c}) \tag{6.4}$$

and

$$E_{WR_k} = \sum_{w_k} N^{w_k} \cdot E^{w_k}_{WR_k}, \tag{6.5}$$

where r_k/w_k is a communication channel process P_k reads from/writes to; N^{r_k}/N^{w_k} is the number of times P_k accesses each read/write channel, and $E^{r_k}_{RD_k}/E^{w_k}_{WR_k}$ is the energy profile of one read/write from/to a channel. Recall that in our platforms writing to channels is local and reading from channels is remote (Section 6.4.2). This means that reading from channels may go through a non-contention-free communication component and hence, energy consumed by reading from channels contains the contention dependent part $c \cdot E^{r_k}_{c}$. If the communication component is contention-free, c is 0, if it is non-contention-free, c is 1, while $E^{r_k}_{c}$ is the energy consumed while P_k is waiting for

data from channel r_k when the communication component is non-contention-free. Similarly to $E^{r_k}_{RD_k}$ and $E^{w_k}_{WR_k}$, E_{EXE_k} becomes:

$$E_{EXE_k} = N^{F_k} \cdot E_{F_k}, \tag{6.6}$$

where N^{F_k} is the number of times process P_k executes its computation function F_k, and E_{F_k} is the energy profile of the function. The energy E_{BLK_k} consumed while the process is blocked can be divided to energy $E^{RD}_{BLK_k}$ consumed while the process is blocked on reading due to unavailable data and energy $E^{WR}_{BLK_k}$ consumed while the process is blocked on writing due to unavailable space. E_{BLK_k} can be expressed as:

$$E_{BLK_k} = E^{RD}_{BLK_k} + E^{WR}_{BLK_k}. \tag{6.7}$$

The energies $E^{RD}_{BLK_k}$ and $E^{WR}_{BLK_k}$ can be further expressed as:

$$E^{RD}_{BLK_k} = \frac{T^{total}_{BLKRD_k}}{(T^{1}_{BLKRD_k} + c \cdot T^{1k}_{c})} \cdot (E^{rd}_{BLK_k} + c \cdot E^{1k}_{c}) \tag{6.8}$$

and

$$E^{WR}_{BLK_k} = \frac{T^{total}_{BLKWR_k}}{T^{1}_{BLKWR_k}} \cdot E^{wr}_{BLK_k}, \tag{6.9}$$

where $T^{total}_{BLKRD_k}/T^{total}_{BLKWR_k}$ is the time spent in blocking on read/write by all the channels during the whole execution of the process P_k, $T^{1}_{BLKRD_k}/T^{1}_{BLKWR_k}$ is the time spent in one blocking on read/write by a channel, and $E^{rd}_{BLK_k}/E^{wr}_{BLK_k}$ is the energy profile of one blocking on read/write by a channel. During blocking on read, the process checks the write counter of the corresponding FIFO channel by reading its value through the communication component – see Figure 6.1(a) line 3. If contention may occur (c is 1), checking the write counter on average will last longer with additional time T^{1k}_{c}, and the energy consumed by the checking will increase on average with E^{1k}_{c}.

The above mentioned energy profiles $E^{r_k}_{RD_k}$, $E^{w_k}_{WR_k}$, E_{F_k}, $E^{rd}_{BLK_k}$, $E^{wr}_{BLK_k}$ and E_{CTRL_k} associated with an application process are obtained by first converting the corresponding part of the process code to its assembly equivalent, then counting the number of times N_{inst} each assembly instruction *inst* is executed in the corresponding assembly equivalent, and finally assigning the energy cost E_{inst} to each instruction in the processor ISA. Therefore, each energy profile is the sum of the number of times N_{inst} each instruction *inst* is executed in the corresponding assembly equivalent multiplied by the energy cost E_{inst}

of the given instruction *inst* on a selected platform type. Hence, each of the above mentioned energy profiles can be represented with:

$$(E^{r_k}_{RD_k}, E^{w_k}_{WR_k}, E_{F_k}, E^{rd}_{BLK_k}, E^{wr}_{BLK_k}, E_{CTRL_k}) = \sum_{inst} N_{inst} E_{inst}. \tag{6.10}$$

The contention dependent energy $E^{r_k}_c$ consumed by one read from a channel through a non-contention-free communication component can be expressed as:

$$E^{r_k}_c = \frac{T^{r_k}_{stall}}{T_{1stall}} \cdot E_{stall}, \tag{6.11}$$

where $T^{r_k}_{stall}$ is the total estimated stall time during one read access on channel r_k through non-contention-free communication component, T_{1stall} is the latency of one stall, the ratio $T^{r_k}_{stall}/T_{1stall}$ is the estimated number of stalls on the communication component for one read access on channel r_k, and E_{stall} is the energy cost of one stall.

The contention dependent energy E^{1k}_c consumed by one checking of the write FIFO counter through a non-contention-free communication component can be expressed as:

$$E^{1k}_c = \frac{T^{1k}_c}{T_{1stall}} \cdot E_{stall} = \frac{T^{1k}_{stall}}{T_{1stall}} \cdot E_{stall}, \tag{6.12}$$

where T^{1k}_{stall} is the total estimated stall time through non-contention-free communication component for one check for data availability and the ratio $T^{1k}_{stall}/T_{1stall}$ is the estimated number of stalls for one check for data availability.

6.5.2 Derivation of Model Parameters

From the energy model formulation in Section 6.5.1 we can see that the energy model has three types of parameters – N parameters such as N^{r_k}, N^{w_k}, N^{F_k}, N_{inst}; T parameters such as $T^{total}_{BLKRD_k}$, $T^{total}_{BLKWR_k}$, $T^{1}_{BLKRD_k}$, $T^{1}_{BLKWR_k}$, $T^{r_k}_{stall}$, $T^{1k}_{stall}(T^{1k}_c)$, T_{1stall}; and E parameters such as E_{inst}, E_{stall}. This section explains how the value of each of the parameters is obtained. Parameters N^{r_k}, N^{w_k} and N^{F_k} are obtained by counting integer points in input, output and process domain polytopes of P_k, see Section 6.4.1, which can be done automatically by using the *Barvinok* library in the `pn` compiler [VNS07]. It is done only once per application and the obtained parameters can be used for any mapping of that application to any MPSoC platform. Parameter N_{inst} is obtained by counting how many times an instruction from the processor ISA is executed in the corresponding assembly equivalent of the process code. This is obtained by using

Instruction Set Simulators (ISS) or some hardware tracing circuits and our profiler tool. It is done only once per application for a selected processor type. Parameters $T^{total}_{BLKRD_k}$ and $T^{total}_{BLKWR_k}$ are obtained from a cycle-accurate SystemC timing simulation of PPNs [vHHK10]. This SystemC simulation should be performed for each application-to-platform mapping, because the blocking time, that is, waiting for data/space, depends on the specific mapping of the processes of an application to the platform. Parameters $T^{1}_{BLKRD_k}$ and $T^{1}_{BLKWR_k}$ are obtained by using ISS or some hardware tracing circuits. It is done only once for a selected processor type and for a selected implementation of the read/write primitives. Parameters $T^{r_k}_{stall}$ and T^{1k}_{stall} are obtained for each mapping, by performing the analysis explained later in Section 6.5.2. Parameter T_{1stall} is obtained from data-sheets or from measurements. The energy cost E_{inst} for each instruction *inst* and the energy cost E_{stall} for a stall are obtained from measurements, and this is done only once per platform type (processor type, communication infrastructure type, selected technology).

Extraction of the Energy Costs

In this subsection we will describe how the energy costs E_{inst} for each instruction *inst* and the energy cost E_{stall} for a stall are derived.

Since our platforms consist of processing tiles and communication infrastructure, the energy costs E_{inst} and E_{stall} can be expressed as:

$$E_{inst} = E_{inst_{tile}} + E_{comm} = (p_{inst_{tile}} + \frac{p_{comm}}{m}) l_{inst} \tag{6.13}$$

and

$$E_{stall} = E_{stall_{tile}} + E_{comm} = (p_{stall_{tile}} + \frac{p_{comm}}{m}) l_{stall}, \tag{6.14}$$

where $E_{inst_{tile}}$ and $E_{stall_{tile}}$ are tile-dependent energy costs and E_{comm} is a communication infrastructure-dependent energy cost. The energy costs are obtained by multiplying the corresponding power costs $p_{inst_{tile}}$, $p_{stall_{tile}}$, p_{comm} with the instruction latency l_{inst}, and the stall latency l_{stall}. The power consumption $p_{inst_{tile}}$ is the power consumed by an instruction *inst* during its execution on a processing tile. The power cost $p_{stall_{tile}}$ is the power consumption of a tile when a stall occurs. p_{comm} is the power consumed by the communication infrastructure when there is no communication over the infrastructure, while m is the maximum number of tiles that the interconnect allows. The power consumption of communication is captured within the $p_{inst_{tile}}$ power cost for *load* and *store* instructions. These power costs are extracted from measurements.

There may be instructions with different latencies depending on cases they are used. An example is a conditional branch instruction which can be taken

or not taken with different latencies for both cases. In this case, we consider an instruction as a set of instructions with finite number of elements equal to the number of possible cases. We consider every instruction from that set as an individual instruction and assign a power cost to each of them.

For each instruction *inst* we determine its power cost $p_{inst_{tile}}$ by measuring the power consumption with minimum activity and maximum activity of the instruction. The final power cost is an average of the measured maximum and minimum power consumption. In order to measure the maximum and minimum power consumption, we create simple test codes with the instruction under test in a loop and run them on the tile. In the "minimum activity" case an instruction performs its action each time on the same operands, so there is no switching activity on processor core buses. In the "maximum activity" case an instruction performs its action each time on different operands such that switching activity on the buses is maximized. The power cost of a stall $p_{stall_{tile}}$ is obtained by measuring the power consumption of a system when stall occurs. The power cost p_{comm} is measured on a platform with maximum number of tiles m, which the corresponding interconnect allows, while there is no communication between the tiles. These estimations of energy costs are performed only once for the selected processor type and only once for the selected communication infrastructure.

Extraction of the Energy Profiles

In order to create the energy profiles $E^{r_k}_{RD_k}$, $E^{w_k}_{WR_k}$, E_{F_k}, $E^{rd}_{BLK_k}$, $E^{wr}_{BLK_k}$ and E_{CTRL_k} associated with an application process P_k, we should first obtain the assembly instruction profiles of the corresponding parts of the process code. The instruction profile of a code consists of instruction counters which show how many times each instruction from a processor ISA is executed in the corresponding code. In case of branch instructions we also need the number of taken and the number of not-taken branches for each branch instruction. We need the execution trace of an application in order to obtain the needed instruction profiles. Since the PPN application consists of processes repeated a number of times, we do not need the instruction trace of the whole execution of an application and we only need the traces of each process, the read and write primitives for each channel and the control structures. Each process of an application is executed as many times as many different execution traces can occur for that process. The execution traces can be obtained by ISS or by some hardware tracing circuits. The execution traces usually contain program counter values, instructions and can also contain some additional information (such as branch is taken or not, and others). By analyzing program counter

values we can determine if a branch is taken or not. Our profiler tool reads the execution traces of an application and creates the instruction profiles of the application. The final instruction profile of the process with many possible execution traces is the average profile, where counters of each instructions are averaged. Profiling of an application is done only once for the selected processor type and selected implementation of read/write primitives (HW or SW).

Analysis of Communication Contention

The derivation of the energy model parameters $T_{stall}^{r_k}$ and T_{stall}^{1k} related to non-contention-free communication components is explained in this subsection. Since our procedure analyzes the contention on a remote tile-to-communication memory link, that is, $\pi_j \leftarrow \pi_i$ link, we will use in the following the notation r_{ij} for a read channel of a process mapped onto tile π_j that reads from communication memory of a tile π_i. Thus, $T_{stall}^{r_k}$ and T_{stall}^{1k} become $T_{stall}^{r_{ij}}$ and T_{stall}^{1j} from a tile point of view.

The communication contention may occur if the communication component within the MPSoC platform is an arbitrated structure. In this work, we consider two types of non-contention-free communication components – a crossbar switch (CB) and a shared bus (ShB), where the CB and ShB interconnections have a round-robin arbitration policy.

The procedure to derive $T_{stall}^{r_{ij}}$ and T_{stall}^{1j} for each read channel r_ij in G for CB communication component is given in Algorithm 10. Inputs of the algorithm are the contention matrix $\mathbf{R}$ defined in Section 6.4.3, the number m of processing tiles in the platform, the size $s_{r_{ij}}$ of a data token transmitted through a channel r_{ij} and latencies of the interconnect read and write arbiters a_R, h_R, ra_R, a_W, h_W, ra_W. During one read and write access through the CB before the transferring of data the corresponding arbiters first arbitrate the requests for access, with associated arbitration latency a_R and a_W, and then ensure the communication link, with associated handshaking latency h_R and h_W. Additional latency ra_R and ra_W may occur on a master-slave link if there is a re-arbitration, which happens when the requested slave unit is different from the last granted slave unit. The parameter $s_{r_{ij}}$ is obtained from the PPN model of an application, while arbiters' latencies a_R, h_R, ra_R, a_W, h_R, ra_W are obtained from measurements or data-sheets. The contention on a CB component may occur when at least two processes from at least two different tiles perform read operation to the same communication memory at the same time.

Algorithm 10 gives the procedure to derive $T_{stall}^{r_{ij}}$ and T_{stall}^{1j} for the CB and

Algorithm 10: Procedure to derive $T_{stall}^{r_{ij}}$ and T_{stall}^{1j} for CB.

Input: Contention matrix **R**, number of tiles m, read channels r_{ij} of processes in G, size of data tokens $s_{r_{ij}}$ for each read channel r_{ij}, latencies of the communication infrastructure a_R, h_R, ra_R, a_W, h_R, ra_W.

Output: Arrays $\mathcal{T}_{stall}$ and $\mathcal{T}_{stall}^{1}$ of time parameters $T_{stall}^{r_{ij}}$ and T_{stall}^{1j}.

for $1 \leq i \leq m$ **do**
  $c_j = 0$;
  for $1 \leq j \leq m$ **do**
    $c_j = c_j + R_{ij}$;
  if $c_j > 1$ **then**
    $c_j = 1$;
  else
    $c_j = 1$;
for $1 \leq j \leq m$ **do**
  $q_j = 0$;
  for $1 \leq i \leq m$ **do**
    $q_j = q_j + R_{ij}$;
  if $q_j > 1$ **then**
    $q_j = 1$;
  else
    $q_j = 0$;
for $1 \leq j \leq m$ **do**
  $l = 0$;
  $T_{stall}^{1j} = 0$;
  for $1 \leq i \leq m$ **do**
    if $R_{ij} = 1$ **then**
      $L_{r_{ij}}^{1bc} = h_R$;
      for r_{ij} *such that* $\pi_i \rightarrow \pi_j$ **do**
        $L_{r_{ij}}^{bc} = (2 + s_{r_{ij}}) \cdot h_R + q_j \cdot 0.5 \cdot ra_R + h_W + q_j \cdot 0.5 \cdot r_W$;
        $L_{r_{ij}}^{wc} = L_{r_{ij}}^{bc} + c_i \sum_{o=1, o \neq j}^{n} R_{io}((2 + s_{r_{ij}})(h_R + a_R) + q_o \cdot 0.5 \cdot ra_R + h_W + a_W + q_o \cdot 0.5 \cdot r_W)$;
        $T_{stall}^{r_{ij}} = (L_{r_{ij}}^{bc} + L_{r_{ij}}^{wc})/2$;
        $L_{r_{ij}}^{1wc} = L_{r_{ij}}^{1bc} + c_i \cdot \sum_{k=1, k \neq j}^{n} R_{ik}(h_R + a_R)$;
        $T_{stall}^{1j} = T_{stall}^{1j} + (L_{r_{ij}}^{1bc} + L_{r_{ij}}^{1wc})/2$;
        $l = l + 1$;
  $T_{stall}^{1j} = T_{stall}^{1j}/l$;
return $\mathcal{T}_{stall}$, $\mathcal{T}_{stall}^{1}$;

it consists of three parts: 1) for each communication memory it is determined whether contention may occur – lines 1 to 8; 2) for each processing tile it is determined whether re-arbitration may occur – lines 9 to 16 in the algorithm; and 3) the estimation of $T_{stall}^{r_{ij}}$ and T_{stall}^{1j} is performed at lines 17 to 31. Since the circular round-robin arbitration pointer is statistically located in the middle of the search space, we estimate $T_{stall}^{r_{ij}}$ at line 26 in Algorithm 10 as the average value of the best case stall time $L_{r_{ij}}^{bc}$, line 24, and the worst case stall time $L_{r_{ij}}^{wc}$, line 25. The best case stall time is when only one tile wants to read from a communication memory (so there is no arbitration latency a_R, a_W). The worst case stall times are calculated by analyzing if contention may happen, and if it may happen, then latencies h_R, h_W, a_R, a_W, ra_R, ra_W for all the tiles that compete for the same communication memory are summed up and added to the best case stall time. Recall that our platforms with shared communication infrastructures use SW read/write primitives – see Section 6.4.2. During one read SW primitive on a channel r_{ij}, $s_{r_{ij}} + 2$ reads and one write are performed – see lines 1, 3, 6 and 8 in Figure 6.1(a), where $s_{r_{ij}}$ corresponds to *size_d* in Figure 6.1(a). Here, re-arbitration may happen only on the first read (out of $s_{r_{ij}} + 2$ reads) and on a write. The frequency of the re-arbitration depends on both the application structure and mapping. Here, if the re-arbitration may happen we assume that the re-arbitration on a read access to the channel happens every second time on the first read and every second time on a write, that is, we multiply r_R and r_W by 0.5 in lines 24 and 25 in Algorithm 10. In the case of data checking, there is no possibility for re-arbitration because waiting for data represents the reading of the write counter (second read within the SW read primitive). Since from the SystemC timing simulation we obtain information about the blocking time on a tile basis, for estimation of T_{stall}^{1j}, at line 30, we sum up the average values $L_{r_{ij}}^{1}$ of each channel that a tile accesses for reading and divide the result by the number of the accessed channels for reading by that tile.

Let us now analyze the ShB case. The contention may happen when at least two processes from at least two different tiles perform read operation at the same time to any of the communication memories in the platform. The procedure to derive $T_{stall}^{r_{ij}}$ and T_{stall}^{1j} for ShB is given in Algorithm 11. The input parameters are similar to the CB case with the difference that here we have only one arbiter. First, we determine the number of tiles *n_r* that do not read from any communication memory, lines 1 to 7 in Algorithm 11. Then in the following lines, we estimate $T_{stall}^{r_{ij}}$ and T_{stall}^{1j}, line 14, 17, as the average of the best case stall time $L_{r_{ij}}^{bc}$, L^{1bc}, line 12, 15, and the worst case stall time $L_{r_{ij}}^{wc}$, L^{1wc},

Algorithm 11: Procedure to derive $T_{stall}^{r_{ij}}$ and T_{stall}^{1j} for ShB.

Input: Contention matrix **R**, number of tiles m, read channels r_{ij} of processes in G, size of data tokens $s_{r_{ij}}$ for each read channel r_{ij}, latencies of the communication infrastructure a, h.

Output: Arrays $\mathcal{T}_{stall}$ and $\mathcal{T}_{stall}^{1}$ of time parameters $T_{stall}^{r_{ij}}$ and T_{stall}^{1j}.

```
1  n_r = 0;
2  for 1 ≤ j ≤ m do
3      y_r_j = 0;
4      for 1 ≤ i ≤ m do
5          y_r_j = y_r_j + R_ij;
6      if y_r_j = 0 then
7          n_r = n_r + 1;
8  for 1 ≤ j ≤ m do
9      for 1 ≤ i ≤ m do
10         if R_ij = 1 then
11             for r_ij such that π_i → π_j do
12                 L^bc_{r_ij} = (3 + s_{r_ij})h;
13                 L^wc_{r_ij} = L^bc_{r_ij} + (n − n_r − 1)(3 + s_{r_ij})(h + a);
14                 T^{r_ij}_stall = (L^bc_{r_ij} + L^wc_{r_ij})/2;
15     L^1bc = h;
16     L^1wc = L^1bc + (n − n_r − 1)(h + a);
17     T^1j_stall = (L^1bc + L^1wc)/2;
18 return T_stall, T^1_stall;
```

line 13, 16. In the best case, only one tile wants to read from a communication memory. By computing how many tiles $(n - n\_r - 1)$ read from any of the communication memories, we determine how long the tile may wait in the worst case.

6.6 Evaluation of the Energy Model

We evaluate our energy model, proposed in Section 6.5, by showing its accuracy considering various application-to-platform mappings. The obtained energy estimates by using our energy model are compared to real energy measurements obtained from real implementations of the considered systems, that is, applications, platforms and mappings. These real measurements are 100% accurate, thereby can be used as credible reference points. We show that the proposed energy model is highly accurate for contention-free and

different kinds of non-contention-free communication components, different applications and mappings, and different number of processing tiles in a platform.

The proposed energy model is evaluated on MPSoC systems prototyped on the Virtex-6 FPGA board ML605. Since the MicroBlaze [Mic] processor is the only available processor type on Virtex-6, we use MPSoC platforms with different number of MicroBlaze based tiles and with the AXI-4 [AXI] interconnect as a non-contention-free communication component, and a P2P interconnect as a contention-free communication component. We use the AXI interconnect configured in CB and ShB modes with a round-robin arbitration policy. The energy model is evaluated for two applications with SW read/write primitives. The first application is a Sobel edge-detection filter and the second application is a MJPEG video encoder. The PPN model for the Sobel consists of 5 lightweight processes in terms of computation and 15 channels, thus the Sobel application is data communication-dominant which introduces a lot of contention on the CB and ShB. The MJPEG PPN model consists of 6 processes and 5 channels with much higher computation/communication ratio, and hence the MJPEG is a computation-dominant application. Since the maximum number of processes among these two applications is 6, we performed energy estimates for platforms with 2 to 6 processing tiles in a platform. The corresponding power consumptions of application-to-platform mappings are measured by using the ML605 on-board power monitoring device and an additional MicroBlaze processor which reads the corresponding power measurements from the monitoring device. Instruction traces for the applications are obtained by monitoring the Trace interface of a MicroBlaze processor. All the platforms run at a frequency of 100 MHz.

Applying our model, described in Section 6.5, we estimate the energy consumption for each application-to-platform mapping, specified in the first column of Table 6.1, for three types of communication infrastructures – the CB, ShB and P2P. In the first column, each mapping is denoted as $app\_n_{tiles}\_m_{map}$, where app is the application, n_{tiles} is the number of tiles in the platform, and m_{map} is the index of a mapping (as an application can be mapped onto a platform in many possible ways). The E_m columns contain the reference values of energy consumption of application-to-platform mappings, obtained by real measurements. The E_e columns contain the energy estimates of the same application-to-platform mappings obtained by using our energy model. The e_{rr} column for each type of interconnect gives the energy estimation error calculated as $e_{rr} = (E_e - E_m)/E_m \cdot 100\%$. It can be seen from Table 6.1 that our energy model is highly accurate for all three types of interconnects, with an

Table 6.1: *Accuracy of the energy model for CB, ShB and P2P MPSoC platforms*

$app \rightarrow pla$	CB			ShB			$P2P$		
	E_m $[mWs]$	E_e $[mWs]$	e_{rr} [%]	E_m $[mWs]$	E_e $[mWs]$	e_{rr} [%]	E_m $[mWs]$	E_e $[mWs]$	e_{rr} [%]
Sobel_2_m1	59.9	61.66	+2.94	54.71	58.18	+6.34	53.95	52.34	-2.98
MJPEG_2_m1	48.82	51.96	+6.43	49.56	51.99	+4.9	58.82	56.98	-3.13
Sobel_3_m1	82.12	73.32	-10.72	68.75	73.67	+7.16	74.43	73.51	-1.24
Sobel_3_m2	69.74	66.63	-4.46	60.98	62.13	+1.89	49.62	50.42	+1.61
MJPEG_3_m1	44.1	42.73	-3.1	40.5	42.19	+4.17	43.64	44.39	+1.72
MJPEG_3_m2	76.74	69.95	-8.85	68.84	67.58	-1.83	86.56	79.67	-7.96
Sobel_4_m1	58.32	58.7	+0.66	52.18	56.86	+8.97	68.07	68.06	-0.01
MJPEG_4_m1	96.72	95.05	-1.73	93.8	93.69	-0.12	107.97	103.68	-3.97
Sobel_5_m1	71.5	71.03	-0.65	68.78	77.46	+12.62	79.52	85.87	+7.99
MJPEG_5_m1	125.63	121.65	-3.17	127.75	119.31	-6.61	137.94	126.84	-8.05
MJPEG_6_m1	77.4	77.15	-0.32	74.7	75.27	+0.76	84.42	79.32	-6.04

average energy estimation error of 4.34% and a standard deviation of 3.35% among all the interconnection types.

In order to analyze the influence of the communication contention on the energy consumption of an application-to-platform mapping, we perform the energy estimation for each application-to-platform mapping with CB and ShB interconnects without considering the contention in the energy model. The results are given in Table 6.2. By comparing Table 6.1 and Table 6.2, we can see, first, that if the contention is not considered, the energy of a mapping is always underestimated, and second that the energy estimates are less accurate than the estimates when considering the contention in the energy model. Therefore, in our proposed energy model special attention is paid to modeling the contention on communication infrastructures.

From the results shown in Table 6.1 it is clear that our energy model is very accurate. Now, we would like to discuss the efficiency of our model in terms of the time required to estimate the energy consumption of a single application-to-platform mapping. For every mapping, listed in the first column of Table 6.1, we measured the time needed for the energy estimation. The average model evaluation time for a mapping is 2.5 minutes, where a few milliseconds are needed for evaluation of the formulas in Section 6.5.1 and derivation of $T_{stall}^{r_{ij}}$ and T_{stall}^{1j} parameters, and the rest of the time is spent on getting the $T_{BLKRD_j}^{total}$ and $T_{BLKWR_j}^{total}$ parameters. The derivation of the other model parameters is not considered in this model evaluation time because they are derived only once at the beginning when the model is calibrated and they are independent of the mapping. The time efficiency of the proposed energy model is very good given its high accuracy. Note that the majority of the evaluation time (99%) is spent

Table 6.2: *Accuracy of the energy estimation when contention is not considered in the model*

$app \rightarrow pla$	CB			ShB		
	E_m $[mWs]$	E_e $[mWs]$	e_{rr} [%]	E_m $[mWs]$	E_e $[mWs]$	e_{rr} [%]
Sobel_2_m1	59.9	47.2	-21.2	54.71	46.76	-14.53
MJPEG_2_m1	48.82	44.54	-8.77	49.56	45.5	-8.19
Sobel_3_m1	82.12	53.64	-34.68	68.75	55.05	-19.93
Sobel_3_m2	69.74	54.78	-21.45	60.98	53.04	-13.02
MJPEG_3_m1	44.1	38.65	-12.36	40.5	38.23	-5.6
MJPEG_3_m2	76.74	57.92	-24.53	68.84	54.82	-20.37
Sobel_4_m1	58.32	46.89	-19.6	52.18	45.56	-12.69
MJPEG_4_m1	96.72	82.27	-14.94	93.8	81.17	-13.46
Sobel_5_m1	71.5	54.86	-23.27	68.78	58.15	-15.46
MJPEG_5_m1	125.63	109.39	-12.93	127.75	106.79	-16.41
MJPEG_6_m1	77.4	71.51	-7.62	74.7	68.24	-8.65

in the SystemC cycle accurate simulation. We run cycle accurate SystemC simulation for each mapping in order to obtain very accurate $T^{total}_{BLKRD_j}$ and $T^{total}_{BLKWR_j}$. We need as accurate estimation of these blocking times as possible in order to have accurate energy estimates because these blocking times are significant part of the total execution time of an application, and hence the energy consumed in blocking could be also significant part of the total energy consumed by a mapping.

6.7 Discussion

We have proposed, in this chapter, an accurate energy model for streaming applications modeled using the PPN model and mapped onto MPSoC platforms. Special attention in our model is paid to the contention on non-contention-free communication infrastructures which is important to estimate accurately the energy consumption of a mapping. Experimental results on two applications with very different computation and communication characteristics mapped onto MPSoC platforms with different communication infrastructures show that the proposed modeling and estimation methodology is highly accurate for different kinds of applications, different kinds of communication infrastructures within MPSoC platforms, and various application-to-platform mappings. On average, the energy estimation error is 4.34% with a standard deviation of 3.35% in comparison to real energy measurements for all considered communication infrastructures. The average model evaluation time of 2.5 minutes

per single design point is very good given the high accuracy of the proposed energy model.

Chapter 7

Summary and Conclusions

The continuous increase in user demands and very fast technology improvement have led to more and more complex embedded streaming systems. Nowadays, many embedded systems are based on Multi-Processor System-on-Chip (MPSoC) platforms. In modern MPSoCs, it is desirable to execute multiple applications, a mixture of streaming applications and control (hard) real-time applications, simultaneously in order to efficiently utilize the resources in an MPSoC. To deliver high-quality output of multiple running applications, together with the ability to dynamically start/stop applications without affecting other already running applications, streaming applications have tight timing requirements that often make it necessary to treat them as hard real-time applications. Moreover, given that the embedded systems are very often battery-powered, a very important requirement in the design of embedded streaming MPSoCs is the energy-efficiency. Designing such an embedded system imposes several challenges: a streaming application should be represented in a way that reveals the parallelism of the application, and it should be mapped and scheduled on a platform such that the timing requirements are satisfied and the energy consumption minimized. To exploit the parallel nature of MPSoC platforms, application behavior is usually specified using a certain parallel Model of Computation (MoC), in which the application is represented as parallel executing and communicating tasks. Finding an efficient tasks-to-platform mapping, that is, *spatial scheduling*, and execution order of tasks in time, that is, *time scheduling*, are the key issues for optimizing the energy consumption and performance of these systems.

In Chapter 3, we have proposed a *conversion approach* that converts tasks in a MoC to real-time tasks to solve the problem of providing timing guarantees for embedded streaming systems. In particular, we have devised an

improved hard real-time scheduling approach to schedule streaming applications modeled as acyclic CSDF graphs on an MPSoC platform. Our proposed approach converts each actor in a CSDF graph to a set of real-time periodic tasks. The conversion enables application of many hard real-time scheduling algorithms that offer fast calculation of the required number of processors for scheduling the tasks. In addition, in Chapter 3, we have proposed a method to reduce the graph latency when the converted tasks are scheduled as real-time periodic tasks. Our proposed scheduling approach gives tighter guarantee on the throughput and better processor utilization with acceptable increase in terms of communication memory requirements when compared with related scheduling approaches. Although it has been shown in [TA10] that the majority of streaming applications, around 90%, can be represented as acyclic SDF graphs, extending the scheduling framework presented in Chapter 3 to support streaming applications modeled as cyclic (C)SDF graphs deserves further investigation.

To exploit efficiently the available parallelism in an MPSoC platform to guarantee performance, energy, and timing constraints, the right amount of parallelism available in a streaming application should be exposed. Given that an initial application specification is often not the most suitable one for the given MPSoC platform, in Chapter 4, we have proposed an *unfolding transformation approach* to transform an initial application specification into an alternative application specification which closely matches the given MPSoC platform. Our proposed approach transforms an initial SDF graph to an alternative CSDF graph which exploits the proper amount of parallelism in the given homogeneous MPSoC to maximize the system performance and provide timing guarantees. The tasks in the transformed graph are scheduled according to our scheduling approach presented in Chapter 3. The experiments on a set of real-life applications showed that our proposed approach delivers, in a matter of minutes, solutions with smaller code size, smaller buffer sizes, and shorter application latency while meeting the same performance and timing requirements as the related approaches.

In Chapter 5, we have investigated and proposed an approach which utilizes our unfolding transformation presented in Chapter 4 and our scheduling framework given in Chapter 3 to better map the given streaming application to the given MPSoC platform in terms of energy consumption under throughput constraints. Our *energy-minimization mapping approach* applies the unfolding transformation to an initial SDF graph to the extent that when combined with Voltage-Frequency Scaling (VFS) techniques on cluster-heterogeneous MPSoCs leads to an energy-efficient design which also meets throughput con-

strains. In particular, our proposed approach is a polynomial-time approach which first determines a processor type for each task in an SDF graph such that the throughput constraint is met, and second, determines a replication factor for each task in an SDF graph to achieve load-balancing on processors of the same type, which enables processors to run at a lower frequency, thereby consuming less energy. The experimental evaluation performed on a set of real-life streaming applications showed that our approach reduces energy consumption by 66% on average among all the experiments while meeting the same throughput requirement when compared to related energy minimization mapping approaches.

Finally, in Chapter 6, we have proposed a novel very *accurate energy model* for streaming applications modeled as PPN graphs and mapped onto tile-based MPSoC platforms with distributed memory. The energy model is based on the well-defined properties of the PPN application model. To guarantee the accuracy of the energy model, values of important model parameters were obtained by real measurements. The proposed energy model is applicable to different types of processors and communication infrastructures within an MPSoC platform. The energy model was evaluated on FPGA-based MPSoC platforms against real measurements of the energy consumption from the FPGA. The obtained energy consumption estimates are highly accurate with an average error of 4% and a standard deviation of 3%. The average model evaluation time per design point takes 2.5 minutes for the considered cases, which is very good given the high accuracy of the model. The majority of the evaluation time is spent in SystemC cycle accurate simulations, which are used for obtaining the *time parameters* of the energy model. Although they lead to highly accurate estimates of the energy consumption, these simulations might lead to significantly high overall design space exploration time. Thus, to maintain the high estimation accuracy while providing time-efficient evaluation of the energy consumption, a future work could combine our energy model based on SystemC simulations with a model based on less accurate but faster techniques to estimate the time parameters. The two energy models should be properly interleaved in the design space exploration process to achieve an adequate trade-off between the model accuracy and evaluation time. In that case, our highly accurate energy model would be used to estimate the energy consumption of a set of design points preselected by using the less energy accurate model in the process of design space exploration.

Bibliography

[ABD08] J. H. Anderson, V. Bud, and U. C. Devi. An edf-based restricted-migration scheduling algorithm for multiprocessor soft real-time systems. *Real-Time Systems*, 38(2):85–131, 2 2008. doi:10.1007/s11241-007-9035-0.

[ABR+93] N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. J. Wellings. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Eng. J.*, 8:284–292(8), 1993.

[ABRW91] N. C. Audsley, A. Burns, M. M. Richardson, and A. J. Wellings. Hard real-time scheduling: The deadline-monotonic approach. In *Proceedings of 8th IEEE Workshop on Real-Time Operating Systems and Software*, pages 133–137, 1991.

[ARM] ARM. AMBA Specifications. URL: https://www.arm.com/products/system-ip/amba-specifications.php [cited June 8, 2016].

[AS04] K. Albers and F. Slomka. An event stream driven approximation for the analysis of real-time systems. In *Proceedings of the 16th Euromicro Conference on Real-Time Systems*, ECRTS '04, pages 187–195, Washington, DC, USA, 2004. IEEE Computer Society. URL: http://dx.doi.org/10.1109/ECRTS.2004.4, doi:10.1109/ECRTS.2004.4.

[AT06] B. Andersson and E. Tovar. Multiprocessor scheduling with few preemptions. In *Proceedings of the 12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, RTCSA '06, pages 322–334. IEEE, 2006. doi:10.1109/RTCSA.2006.45.

[AXI] ARM Ltd., AMBA AXI Protocol - Version: 2.0 : Specification, 2010. http://www.arm.com.

[AY03] H. Aydin and Q. Yang. Energy-aware partitioning for multiprocessor real-time systems. In *Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, IPDPS '03, pages 113.2–, Washington, DC, USA, 2003. IEEE Computer Society. URL: http://dl.acm.org/citation.cfm?id=838237.838347.

[Bam12] M. Bamakhrama, 2012. http://daedalus.liacs.nl/darts.

[BB04] D. Bertozzi and L. Benini. Xpipes: a network-on-chip architecture for gigascale systems-on-chip. *Circuits and Systems Magazine, IEEE*, 4(2):18–31, September 2004. URL: http://dx.doi.org/10.1109/mcas.2004.1330747, doi:10.1109/mcas.2004.1330747.

[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.

[BDM02] L. Benini and G. De Micheli. Networks on chips: A new soc paradigm. *Computer*, 35(1):70–78, January 2002. URL: http://dx.doi.org/10.1109/2.976921, doi:10.1109/2.976921.

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

[BF05] S. Baruah and N. Fisher. The partitioned multiprocessor scheduling of sporadic task systems. In *Proceedings of the 26th IEEE International Real-Time Systems Symposium*, RTSS '05, pages 321–329, Washington, DC, USA, 2005. IEEE Computer Society. URL: http://dx.doi.org/10.1109/RTSS.2005.40, doi:10.1109/RTSS.2005.40.

[BL13] D. Bui and E. A. Lee. Streamorph: A case for synthesizing energy-efficient adaptive programs using high-level abstractions. In *Proceedings of the Eleventh ACM International Conference on Embedded Software*, EMSOFT '13, pages 20:1–20:10, Piscataway, NJ, USA, 2013. IEEE Press. URL: http://dl.acm.org/citation.cfm?id=2555754.2555774.

[BMAB16] M. Bambagini, M. Marinoni, H. Aydin, and G. Buttazzo. Energy-aware scheduling for real-time systems: A survey. *ACM Trans. Embed. Comput. Syst.*, 15(1):7:1–7:34, January 2016. URL: http://doi.acm.org/10.1145/2808231, doi:10.1145/2808231.

[BMKdD13] B. Bodin, A. Munier-Kordon, and B. 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.

[BMMKM10] M. Benazouz, O. Marchetti, A. Munier Kordon, and T. Michel. A new method for minimizing buffer sizes for cyclo-static dataflow graphs. In *Proceedings of the 8th IEEE Symposium on Embedded Systems for Real-Time Multimedia*, ESTIMedia 2010, pages 11–20, 2010.

[BRH90] S. K. Baruah, L. E. Rosier, and R. 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] M. Bamakhrama and T. Stefanov. Hard-real-time scheduling of data-dependent tasks in embedded streaming applications. In *Proceedings of the ninth ACM International Conference on Embedded Software*, EMSOFT '11, pages 195–204, New York, NY, USA, 2011. ACM. doi:10.1145/2038642.2038672.

[BS13] M. A. Bamakhrama and T. P. Stefanov. On the hard-real-time scheduling of embedded streaming applications. *Design Automation for Embedded Systems*, 17(2):221–249, 2013. URL: http://dx.doi.org/10.1007/s10617-012-9086-x, doi:10.1007/s10617-012-9086-x.

[BTM00] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. In *Proc. of ISCA*, pages 83–94, 2000.

[BTV12] A. Bouakaz, J.-P. Talpin, and J. 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.

[BVC04] N. Banerjee, P. Vellanki, and K.S. Chatha. A power and performance model for network-on-chip architectures. In *Proc. of DATE - Volume 2*, 2004.

[BZNS12] M. A. Bamakhrama, J. Teddy Zhai, H. Nikolov, and T. 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.

[BZZ04] A. Bona, V. Zaccaria, and R. Zafalon. System level power modeling and simulation of high-end industrial network-on-chip. In *Proc. of DATE - Volume 3*, 2004.

[C+10] X. Chen et al. Performance and power modeling in a multi-programmed multi-core environment. In *Proc. of DAC*, pages 813–818, 2010.

[CAC] HP Labs, CACTI. http://www.hpl.hp.com/research/cacti.

[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.

[CKR14] A. Colin, A. Kandhalu, and R. Rajkumar. Energy-efficient allocation of real-time applications onto heterogeneous processors. In *RTSCA*, 2014.

[CRJ06] H. Cho, B. Ravindran, and E. D. Jensen. An optimal real-time scheduling algorithm for multiprocessors. In *Proceedings of the 27th IEEE International Real-Time Systems Symposium*, RTSS '06, pages 101–110, Washington, DC, USA, 2006. IEEE Computer Society. URL: http://dx.doi.org/10.1109/RTSS.2006.10, doi:10.1109/RTSS.2006.10.

[DB11] R. I. Davis and A. 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.

[DK89] M. L. Dertouzos and A. Ka-Lau Mok. Multiprocessor online scheduling of hard-real-time tasks. *IEEE Transactions on Software Engineering*, 15(12):1497–1506, 1989. doi:10.1109/32.58762.

[DSB+13] M. Damavandpeyma, S. Stuijk, T. Basten, M. Geilen, and H. Corporaal. Throughput-constrained dvfs for scenario-aware dataflow graphs. In *RTAS*, 2013.

[EBSA+11] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger. Dark silicon and the end of multicore scaling. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, pages 365–376, New York, NY, USA, 2011. ACM. URL: http://doi.acm.org/10.1145/2000064.2000108, doi:10.1145/2000064.2000108.

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

[Fea96a] P. 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.

[Fea96b] P. Feautrier. Automatic parallelization in the polytope model. In *The Data Parallel Programming Model*. Springer-Verlag, 1996.

[Fis07] N. Fisher. *The multiprocessor real-time scheduling of general task systems*. PhD thesis, Department of Computer Science, The University of North Carolina at Chapel Hill, Chapel Hill, NC., 2007.

[FKBS11] S. M. Farhad, Y. Ko, B. Burgstaller, and B. Scholz. Orchestration by approximation: Mapping stream programs onto multicore architectures. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 357–368, New York, NY, USA, 2011. ACM. URL: http://doi.acm.org/10.1145/1950365.1950406, doi:10.1145/1950365.1950406.

[fSI] International Technology Roadmap for Semiconductors (ITRS). ITRS Reports. URL: http://www.itrs2.net/itrs-reports.html [cited June 12, 2016].

[GB14] M. Grant and S. Boyd. CVX: Matlab software for disciplined convex programming, ver. 2.1, 2014.

[GDR05] K. Goossens, J. Dielissen, and A. 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.

[GGS+06] A. H. Ghamarian, M. C. W. Geilen, S. Stuijk, T. Basten, B. D. Theelen, M. R. Mousavi, A. J. M. Moonen, and M. J. G. Bekooij. Throughput analysis of synchronous data flow graphs. In *Proceedings of the Sixth International Conference on Application of Concurrency to System Design*, ACSD '06, pages 25–36, Washington, DC, USA, 2006. IEEE Computer Society. URL: http://dx.doi.org/10.1109/ACSD.2006.33, doi:10.1109/ACSD.2006.33.

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

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

[Gre11] P. Greenhalgh. Big.LITTLE processing with ARM Cortex-A15 & Cortex-A7, 2011.

[HA06] P. Holman and J. H. Anderson. Group-based Pfair scheduling. *Real-Time Systems*, 32(1–2):125–168, 2006. doi:doi:10.1007/s11241-006-4687-8.

[HGWB13] J. P. H. M. Hausmans, S. J. Geuns, M. H. Wiggers, and M. 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.

[HLF+11] C.-W. Hsu, J.-L. Liao, S.-C. Fang, C.-C. Weng, S.-Y. Huang, W.-T. Hsieh, and J.-C. Yeh. Powerdepot: Integrating ip-based power modeling with esl power analysis for multi-core soc designs. In *Proceedings of the 48th Design Automation Conference*, DAC '11, pages 47–52, New York, NY, USA, 2011. ACM. URL: http://doi.acm.org/10.1145/2024724.2024736, doi:10.1145/2024724.2024736.

[HM07] S. Herbert and D. Marculescu. Analysis of dynamic voltage/frequency scaling in chip-multiprocessors. In *Proceedings of the 2007 International Symposium on Low Power Electronics and Design*, ISLPED '07, pages 38–43, New York, NY, USA, 2007. ACM. URL: http://doi.acm.org/10.1145/1283780.1283790, doi:10.1145/1283780.1283790.

[HMGM13] P. Huang, O. Moreira, K. Goossens, and A. Molnos. Throughput-constrained voltage and frequency scaling for real-time heterogeneous multiprocessors. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, SAC '13, pages 1517–1524, New York, NY, USA, 2013. ACM. URL: http://doi.acm.org/10.1145/2480362.2480645, doi:10.1145/2480362.2480645.

[HNP+15] S. Holmbacka, E. Nogues, M. Pelcat, S. Lafond, D. Menard, and J. Lilius. Energy-awareness and performance management with parallel dataflow applications. *J. of Signal Processing Systems*, pages 1–16, 2015.

[HP06] J. L. Hennessy and D. A. Patterson. *Computer Architecture, Fourth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.

[IBM12] IBM ILOG CPLEX Optimization Studio V12.4, 2012.

[Int11] The International Technology Roadmap for Semiconductors (ITRS), System Drivers, 2011. Available on: http://www.itrs.net.

[Joh74] D. 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.

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

[Kah13] A. B. Kahng. The itrs design technology and system drivers roadmap: Process and status. In *Proceedings of the 50th Annual Design Automation Conference*, DAC '13, pages 34:1–34:6, New York, NY, USA, 2013. ACM. URL: http://doi.acm.org/10.1145/2463209.2488776, doi:10.1145/2463209.2488776.

[KLPS09] A. B. Kahng, B. Li, L.-S. Peh, and K. Samadi. Orion 2.0: A fast and accurate noc power and area model for early-stage design space exploration. In *Proc. of DATE*, pages 423–428, 2009.

[KM08] M. Kudlur and S. Mahlke. Orchestrating the execution of stream programs on multicore platforms. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 114–124, New York, NY, USA, 2008. ACM. URL: http://doi.acm.org/10.1145/1375581.1375596, doi:10.1145/1375581.1375596.

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

[KS97] A. Khemka and R.K. Shyamasundar. An optimal multiprocessor real-time scheduling algorithm. *J. Parallel Distrib. Comput.*, 43(1):37–45, May 1997. URL: http://dx.doi.org/10.1006/jpdc.1997.1327, doi:10.1006/jpdc.1997.1327.

[KY07] S. Kato and N. Yamasaki. Real-time scheduling with task splitting on multiprocessors. In *Proceedings of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, RTCSA '07, pages 441–450. IEEE, 2007. doi:10.1109/RTCSA.2007.61.

[LB03] G. Lipari and E. Bini. Resource partitioning among real-time applications. In *Proceedings of 15th the Euromicro Conference on Real-Time Systems*, ECRTS '03, pages 151–158, 2003.

[Lee09] W. Y. Lee. Energy-saving dvfs scheduling of multiple periodic real-time tasks on multi-core processors. In *Proceedings of the 2009 13th IEEE/ACM International Symposium on Distributed Simulation and Real Time Applications*, DS-RT '09, pages 216–223, Washington, DC, USA, 2009. IEEE Computer Society. URL: http://dx.doi.org/10.1109/DS-RT.2009.12, doi:10.1109/DS-RT.2009.12.

[Leu89] J. Y. T. Leung. A new algorithm for scheduling periodic, real-time tasks. *Algorithmica*, 4(1):209–219, 1989. URL: http://dx.doi.org/10.1007/BF01553887, doi:10.1007/BF01553887.

[LJSM04] J. Laurent, N. Julien, E. Senn, and E. Martin. Functional level power analysis: An efficient approach for modeling the power consumption of complex processors. In *Proceedings of the Conference on Design, Automation and Test in Europe - Volume 1*, DATE '04, pages 10666–, Washington, DC, USA, 2004. IEEE Computer Society. URL: http://dl.acm.org/citation.cfm?id=968878.968987.

[LL73] C. L. Liu and J. 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] E. A. Lee and D. G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, 1987. doi:10.1109/PROC.1987.13876.

[LPB04] M. Loghi, M. Poncino, and L. Benini. Cycle-accurate power analysis for multiprocessor systems-on-a-chip. In *Proc. of ACM Great Lakes symposium on VLSI*, pages 406–410, 2004.

[LSCS15] D. Liu, J. Spasic, G. Chen, and T. Stefanov. Energy-efficient mapping of real-time streaming applications on cluster heterogeneous mpsocs. In *ESTIMedia*, 2015.

[LSZ+14] D. Liu, J. Spasic, J. T. Zhai, T. Stefanov, and G. Chen. Resource optimization for csdf-modeled streaming applications

with latency constraints. In *Proceedings of the Conference on Design, Automation & Test in Europe*, DATE '14, pages 188:1–188:6, 3001 Leuven, Belgium, Belgium, 2014. European Design and Automation Association. URL: http://dl.acm.org/citation.cfm?id=2616606.2616837.

[LW82] J. Y.-T. Leung and J. 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.

[LW13] D. Li and J. Wu. Energy-aware scheduling for acyclic synchronous data flows on multiprocessors. *Journal of Interconnection Networks*, 14(4), 2013.

[Mar06] P. Marwedel. *Embedded System Design*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[MB07] O. M. Moreira and M. 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.

[Mic] Xilinx Inc., MicroBlaze Soft Processor Core. http://www.xilinx.com.

[Mit15] T. Mitra. Heterogeneous multi-core architectures. *IPSJ Transactions on System LSI Design Methodology*, 8:51–62, 2015. doi:10.2197/ipsjtsldm.8.51.

[NMM+11] A. Nelson, O. Moreira, A. Molnos, S. Stuijk, B. T. Nguyen, and K. Goossens. Power minimisation for real-time dataflow applications. In *DSD*, 2011.

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

[NVI15] NVIDIA. NVIDIA Tegra X1: NVIDIA'S New Mobile Superchip, 2015.

[ODR] ODROID. http://www.hardkernel.com.

[OH04] H. Oh and S. 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.

[PDG06] J. Parkhurst, J. Darringer, and B. Grundmann. From single core to multi-core: Preparing for a new exponential. In *Proceedings of the 2006 IEEE/ACM International Conference on Computer-aided Design*, ICCAD '06, pages 67–72, New York, NY, USA, 2006. ACM. URL: http://doi.acm.org/10.1145/1233501.1233516, doi:10.1145/1233501.1233516.

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

[PP12] R. Piscitelli and A. D. Pimentel. A signature-based power model for mpsoc on fpga. *VLSI Design J.*, 2012(6), January 2012.

[PPKD10] S. Pasricha, Y.-H. Park, F. J. Kurdahi, and N. Dutt. Capps: A framework for power-performance tradeoffs in bus-matrix-based on-chip communication architecture synthesis. *IEEE Trans. on VLSI Systems*, 18:209–221, February 2010.

[PZMA04] O. U. Pereira Zapata and P. Mejía Alvarez. EDF and RM multiprocessor scheduling algorithms: Survey and performance evaluation. Technical Report CINVESTAV-CS-RTG-02, 2004.

[QKUP00] G. Qu, N. Kawabe, K. Usami, and M. Potkonjak. Function-level power estimation methodology for microprocessors. In *Proceedings of the 37th Annual Design Automation Conference*, DAC '00, pages 810–813, New York, NY, USA, 2000. ACM. URL: http://doi.acm.org/10.1145/337292.337786, doi:10.1145/337292.337786.

[RAN+11] S.K. Rethinagiri, R.B. Atitallah, S. Niar, E. Senn, and J.-L. Dekeyser. Hybrid system level power consumption estimation for fpga-based mpsoc. In *Proc. of IEEE ICCD*, pages 239–246, 2011.

[Sama] Samsung. Exynos 8 Octa (8890). URL: http://www.samsung.com/semiconductor/minisite/Exynos/w/solution/mod_ap/8890/ [cited June 8, 2016].

[Samb] Samsung. http://www.samsung.com.

[SC01] A. Sinha and A. P. Chandrakasan. Jouletrack - a web based tool for software energy profiling. In *Proc. of DAC*, pages 220–225, 2001.

[SDK13] A. K. Singh, A. Das, and A. Kumar. Energy optimization by exploiting execution slacks in streaming applications on multiprocessor systems. In *Proceedings of the 50th Annual Design Automation Conference*, DAC '13, pages 115:1–115:7, New York, NY, USA, 2013. ACM. URL: http://doi.acm.org/10.1145/2463209.2488875, doi:10.1145/2463209.2488875.

[SEL08] I. Shin, A Easwaran, and I. Lee. Hierarchical scheduling framework for virtual clustering of multiprocessors. In *Proceedings of 20th the Euromicro Conference on Real-Time Systems*, ECRTS '08, pages 181–190, 2008. doi:10.1109/ECRTS.2008.28.

[SGB06] S. Stuijk, M. Geilen, and T. Basten. SDF^3: SDF for free. In *Proc. of ACSD*, pages 276–278, 2006.

[SGB08] S. Stuijk, M. Geilen, and T. Basten. Throughput-buffering trade-off exploration for cyclo-static and synchronous dataflow graphs. *IEEE Trans. on Computers*, 57(10):1331–1345, 2008.

[SGTB11] S. Stuijk, M. Geilen, B. Theelen, and T. Basten. Scenario-aware dataflow: Modeling, analysis and implementation of dynamic applications. In *International Conference on Embedded Computer Systems (SAMOS), 2011*, pages 404–411, July 2011. doi:10.1109/SAMOS.2011.6045491.

[SJE11] M. Sackmann, D. Janssens, and P. Ebraert. A fast heuristic for scheduling parallel software with respect to energy and timing constraints. *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum*, 00:1397–1406, 2011. doi:doi.ieeecomputersociety.org/10.1109/IPDPS.2011.284.

[SLA12] A. Stulova, R. Leupers, and G. Ascheid. Throughput driven transformations of synchronous data flows for mapping to heterogeneous MPSoCs. In *ICSAMOS*, pages 144–151. IEEE, 2012.

[SLCS15] J. Spasic, D. Liu, E. Cannella, and T. Stefanov. Improved hard real-time scheduling of csdf-modeled streaming applications. In *Proceedings of the 10th International Conference on Hardware/Software Codesign and System Synthesis*, CODES '15, pages 65–74, Piscataway, NJ, USA, 2015. IEEE Press. URL: `http://dl.acm.org/citation.cfm?id=2830840.2830848`.

[SLCS16] J. Spasic, D. Liu, E. Cannella, and T. Stefanov. On the improved hard real-time scheduling of cyclo-static dataflow. *ACM Transactions on Embedded Computing Systems*, 15(4):68:1–68:26, August 2016. URL: `http://doi.acm.org/10.1145/2932188`, `doi:10.1145/2932188`.

[SLS16a] J. Spasic, D. Liu, and T. Stefanov. Energy-efficient mapping of real-time applications on heterogeneous mpsocs using task replication. In *Proceedings of the 11th International Conference on Hardware/Software Codesign and System Synthesis*, CODES '16, pages 65–74, Piscataway, NJ, USA, 2016. IEEE Press. URL: `http://dl.acm.org/citation.cfm?id=2830840.2830848`.

[SLS16b] J. Spasic, D. Liu, and T. Stefanov. Exploiting resource-constrained parallelism in hard real-time streaming applications. In *2016 Design, Automation & Test in Europe Conference & Exhibition, DATE 2016, Dresden, Germany, March 14-18, 2016*, pages 954–959, 2016. URL: `http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=7459445`.

[SRH+11] M. Streubuhr, R. Rosales, R. Hasholzner, C. Haubelt, and J. Teich. Esl power and performance estimation for heterogeneous mpsocs using systemc. In *Proc. of the Forum on Specification, Verification and Design Languages*, pages 1–8, 2011.

[SS13] J. Spasic and T. Stefanov. An accurate energy model for streaming applications mapped on mpsoc platforms. In *2013 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation, SAMOS 2013, Agios Konstantinos, Samos Island, Greece, July 15-18, 2013*, pages 205–212,

2013. URL: http://dx.doi.org/10.1109/SAMOS.2013.6621124, doi:10.1109/SAMOS.2013.6621124.

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

[TNS+07] M. Thompson, H. Nikolov, T. Stefanov, A. D. Pimentel, C. Erbas, S. Polstra, and E. 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. URL: http://doi.acm.org/10.1145/1289816.1289823, doi:10.1145/1289816.1289823.

[vHHK10] S. van Haastregt, E. Halm, and B. Kienhuis. Cost modeling and cycle-accurate co-simulation of heterogeneous multiprocessor systems. In *Proc.of DATE*, pages 1297–1300, 2010.

[VJD+07] A. Varma, B. Jacob, E. Debes, I. Kozintsev, and P. Klein. Accurate and fast system-level power modeling: An xscale-based case study. *ACM Trans. Embed. Comput. Syst.*, 6(4), September 2007. URL: http://doi.acm.org/10.1145/1274858.1274864, doi:10.1145/1274858.1274864.

[VNS07] S. Verdoolaege, H. Nikolov, and T. 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.

[WBJS07] M. Wiggers, M. Bekooij, P. G. Jansen, and G. J. M. Smit. Efficient computation of buffer capacities for cyclo-static real-time systems with back-pressure. In *IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 281–292. IEEE Computer Society, 2007.

[WYK+10] Y.-H. Wei, C.-Y. Yang, T.-W. Kuo, S.-H. Hung, and Y.-H. Chu. Energy-efficient real-time scheduling of multimedia tasks on

multi-core processors. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, pages 258–262, New York, NY, USA, 2010. ACM. URL: http://doi.acm.org/10.1145/1774088.1774142, doi:10.1145/1774088.1774142.

[XKD12] H. Xu, F. Kong, and Q. Deng. Energy minimizing for parallel real-time tasks based on level-packing. In *Proceedings of the 2012 IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, RTCSA '12, pages 98–103, Washington, DC, USA, 2012. IEEE Computer Society. URL: http://dx.doi.org/10.1109/RTCSA.2012.10, doi:10.1109/RTCSA.2012.10.

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

[YVKI00] W. Ye, N. Vijaykrishnan, M. Kandemir, and M. J. Irwin. The design and use of simplepower: A cycle-accurate energy estimation tool. In *Proceedings of the 37th Annual Design Automation Conference*, DAC '00, pages 340–345, New York, NY, USA, 2000. ACM. URL: http://doi.acm.org/10.1145/337292.337436, doi:10.1145/337292.337436.

[ZB09] F. Zhang and A. 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] J. T. Zhai, M. A. Bamakhrama, and T. Stefanov. Exploiting just-enough parallelism when mapping streaming applications in hard real-time systems. In *Proceedings of the 50th Annual Design Automation Conference*, DAC '13, pages 170:1–170:8, New York, NY, USA, 2013. ACM. doi:10.1145/2463209.2488944.

[Zha15] J. T. Zhai. *Adaptive Streaming Applications: Analysis and Implementation Models*. PhD thesis, Leiden University, Netherlands, 2015.

[ZK00] C. L. Zitnick and T. Kanade. A cooperative algorithm for stereo matching and occlusion detection. *IEEE Trans. Pattern Anal. Mach. Intell.*, 22(7):675–684, July 2000. doi:10.1109/34.865184.

[ZSJ08] J. Zhu, I. Sander, and A. Jantsch. Energy efficient streaming applications with guaranteed throughput on mpsocs. In *Proceedings of the 8th ACM International Conference on Embedded Software*, EMSOFT '08, pages 119–128, New York, NY, USA, 2008. ACM. URL: http://doi.acm.org/10.1145/1450058.1450075, doi:10.1145/1450058.1450075.

Samenvatting

Dit proefschrift richt zich op het probleem van het ontwerpen van prestatie- en energie-efficiënte embedded streaming-systemen. Embedded streaming-systemen verwerken een stroom van inputgegevens vanuit de omgeving en genereren een stroom van outputgegevens naar de omgeving. Het correct functioneren van embedded streaming-systemen hangt af van zowel de juistheid van de outputgegevens als van de tijd waarop de gegevens zijn geproduceerd. Daarom zijn embedded streaming-systemen real-time-systemen. Enkele voorbeelden van real-time embedded streaming-systemen kunnen gevonden worden in verscheidene autonome mobiele systemen, zoals vliegtuigen, zelfrijdende auto's en drones.

Om de gewenste prestatie en energieconsumptie te bereiken van zulke real- time embedded streaming-systemen, zijn moderne ingebedde systemen uitgerust met hardwareplatformen die meerdere processors bevatten op een enkele chip waarmee een goede prestatie van het systeem kan worden bereikt door parallelle uitvoering, terwijl energie-efficiëntie verkregen kan worden door het operationele voltage en de frequentie te verlagen. Deze hardwareplatformen worden Multi-Processor Systems-on-Chip (MPSoCs) genoemd. Om de gewenste prestatie en energieconsumptie te leveren, moet de streaming-applicatie die op een MPSoC-platform zal worden uitgevoerd, worden gespecificeerd als een serie taken die gegevensafhankelijk zijn, maar die parallel kunnen worden uitgevoerd. Deze taken worden ruimtelijk gepland, dat wil zeggen, in kaart gebracht op processors, waar ze worden gepland in de tijd en uitgevoerd. Dit proefschrift stelt methodes en technieken voor het omzetten van een streaming-applicatie naar een serie van parallelle taken die sterk overeenkomt met een MPSoC-platform en het plannen van deze parallelle taken zodanig dat de gewenste prestatie en energieconsumptie worden bereikt.

Het eerste deel van dit proefschrift voert een aanpak van planning aan om een streaming-applicatie uit te voeren als een serie van real-time periodieke taken. Een normaal gedrag van streaming-applicaties is dat verschil-

lende uitvoeringen van dezelfde applicatietaak verschillen in uitvoeringstijd. Daarom zet de voorgestelde aanpak iedere taak van een applicatie om in een serie van real-time periodieke taken door taakparameters periodes, starttijden en deadlines) af te leiden, rekening houdend met verschillende uitvoeringstijden voor verschillende uitvoeringen van iedere applicatietaak. De omzetting maakt de toepassing mogelijk van vele moeilijke real-time planningsalgoritmes die de snelle berekening bieden van het benodigde aantal processors voor het plannen van de taken met een gegarandeerde prestatie, dat wil zeggen, doorvoercapaciteit en latentietijd. Dit proefschrift laat zien dat het rekening houden met verschillende uitvoeringstijden voor verschillende uitvoeringen van appalicatietaken en moeilijke real-time planningstheorie leidt tot een hogere applicatiedoorvoercapaciteit en een kortere applicatielatentietijd terwijl het aantal processors, nodig voor het plannen van een gegeven applicatie, vermindert. Deze prestatievoordelen gaan echter gepaard met een verhoogde behoefte aan geheugen om de datacommunicatie tussen de taken te implementeren.

Het tweede deel van dit proefschrift draagt technieken aan voor de transformatie van een initiële representatie van een streaming-applicatie, dat wil zeggen, een initiële applicatiegrafiek, naar een gelijkwaardige input-output representatie, op zo'n manier dat de nieuwe representatie meer overeenkomt met het MPSoC-platform, wat leidt tot een betere prestatie en energieconsumptie. In het bijzonder, repliceert de voorgestelde transformatietechniek taken in een initiële applicatiegrafiek en verdeelt data zorgvuldig over taakreplica's, wat meer parallelle uitvoering van taken mogelijk maakt en leidt tot een kortere applicatielatentietijd en een kleiner communicatiegeheugen vergeleken met verwante benaderingen. Deze transformatietechniek wordt vervolgens gebruikt met onze planningsaanpak binnen een nieuw voorgesteld algoritme om de prestatie van embedded streaming-systemen te maximaliseren. Dit algoritme past het parallellisme in de applicatiegrafiek aan in overeenstemming met de bronnen in een MPSoC om de maximale prestatie te bereiken. Daarnaast stelt dit proefschrift een nieuwe benadering voor om real-time streamingapplicaties efficiënt te mappen op MPSoCs met doorvoerlimieten, zodat de energieconsumptie verminderd wordt, door onze transformatietechniek en onze planningsaanpak te gebruiken met gepaste selectie van het operationele voltage en de frequentie. De voorgestelde benadering voor minimalisering van energie presteert beter dan verwante benaderingen in termen van energieconsumptie terwijl aan dezelfde doorvoerlimieten wordt voldaan.

De nauwkeurigheid van het modelleren van energie is belangrijk voor een efficiënt energiebeheer. Daarom stelt het laatste deel van dit proefschrift

een nauwkeurig energiemodel voor, voor embedded streaming-applicaties in kaart gebracht op MPSoC platformen. Om exactere energieconsumptieschattingen te verkrijgen, moet een energiemodel nauwer verbonden zijn met het daadwerkelijke draaiende systeem. Tegelijkertijd moet het model efficiënt zijn wat betreft moeite en tijd besteed aan modelleren en evalueren. Daarom is het voorgestelde exacte energiemodel gebaseerd op een applicatiemodel dat een beter inzicht geeft in de uiteindelijke implementatie van applicatietaken op een platform, en de waardes van belangrijke modelparameters worden verkregen van echte metingen.

List of Publications

Journal Articles

- **Jelena Spasic**, Di Liu, Emanuele Cannella, Todor Stefanov, "On the Improved Hard Real-Time Scheduling of Cyclo-Static Dataflow", *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 15, Issue 4, Article 68, August 2016.

Peer-Reviewed Conference Proceedings

- **Jelena Spasic**, Di Liu, Todor Stefanov, "Energy-Efficient Mapping of Real-Time Applications on Heterogeneous MPSoCs using Task Replication", *In Proceedings of the IEEE/ACM/IFIP International Conference on HW/SW Codesign and System Synthesis (CODES+ISSS'16)*, pp. 28:1–28:10, Pittsburgh, Pennsylvania, USA, October 2-7, 2016.
- **Jelena Spasic**, Di Liu, Todor Stefanov, "Exploiting Resource-constrained Parallelism in Hard Real-Time Streaming Applications", *In Proceedings of the International Conference on Design, Automation and Test in Europe (DATE'16)*, pp. 954–959, Dresden, Germany, March 14-18, 2016.
- **Jelena Spasic**, Di Liu, Emanuele Cannella, Todor Stefanov, "Improved Hard Real-Time Scheduling of CSDF-modeled Streaming Applications", *In Proceedings of the IEEE/ACM/IFIP International Conference on HW/SW Codesign and System Synthesis (CODES+ISSS'15)*, pp. 65–74, Amsterdam, The Netherlands, October 4-9, 2015. **Nominated for the 2015 CODES+ISSS Best Paper Award.**
- **Jelena Spasic** and Todor Stefanov, "An Accurate Energy Model for Streaming Applications Mapped on MPSoC Platforms", *In Proceedings of the IEEE International Conference on Embedded Computer Systems: Architectures, MOdeling, and Simulation (IC-SAMOS'13)*, pp. 205–212, Samos, Greece, July 15-18, 2013.

- Di Liu, **Jelena Spasic**, Gang Chen, Nan Guan, Songran Liu, Todor Stefanov, Wang Yi, "EDF-VD Scheduling of Mixed-Criticality Systems with Degraded Quality Guarantees", *In Proceedings of the IEEE International Real-Time Systems Symposium (RTSS'16)*, pp. 35–46, Porto, Portugal, November 29-December 02, 2016.
- Di Liu, **Jelena Spasic**, Peng Wang, Todor Stefanov, "Energy-Efficient Scheduling of Real-Time Tasks on Heterogeneous Multicores Using Task Splitting", *In Proceedings of the 22nd IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'16)*, pp. 149–158, Daegu, South Korea, August 17-19, 2016.
- Di Liu, **Jelena Spasic**, Gang Chen, Todor Stefanov, "Energy-Efficient Mapping of Real-Time Streaming Applications on Cluster Heterogeneous MPSoCs", *In Proceedings of the 13th Int. IEEE Symposium on Embedded Systems for Real-Time Multimedia (ESTIMedia'15)*, pp. 1–10, Amsterdam, The Netherlands, October 8-9, 2015.
- Di Liu, **Jelena Spasic**, Jiali Teddy Zhai, Todor Stefanov, Gang Chen, "Resource Optimization for CSDF-modeled Streaming Applications with Latency Constraints", *In Proceedings of the International Conference on Design, Automation and Test in Europe (DATE'14)*, pp. 188:1–188:6, Dresden, Germany, March 24-28, 2014.

Curriculum Vitae

Jelena Spasić was born on January 27, 1984 in Trgovište, Serbia. She obtained her Dipl.Ing. (M.Sc.) degree in Electronics Engineering from University of Belgrade, Serbia in 2008. Her M.Sc. thesis was in the field of computer systems in control applications. After obtaining her M.Sc. degree, she worked for three years as an R&D engineer in embedded systems at the Institute Mihailo Pupin in Belgrade. There she worked on embedded systems design, design for electromagnetic compatibility and design of automated test systems. In August 2011, she joined the Leiden Embedded Research Center, part of the Leiden Institute of Advanced Computer Science (LIACS) at Leiden University, as a Ph.D. candidate. Her research work, which resulted in this thesis, was funded by the NWO project CREED. Besides her work as a researcher, she was involved as a teaching assistant in the Digital Technique, Computer Architecture, and Embedded Systems and Software courses. Since September 2016, she has been working at the European Organization for Nuclear Research (CERN) as an electronics engineer developing electronics for protection of magnet circuits in the Large Hadron Collider.

Acknowledgments

The work presented in this thesis would not have been possible without support and help of many people for whom I would like to express my gratitude.

First, I would like to thank the colleagues I had pleasure to work with at the Leiden Embedded Research Center (LERC): Hristo Nikolov for helping me to get familiar with Daedalus and particularly ESPAM tool, Mohammad Al Hissi for transferring his knowledge on power modeling of embedded systems, Sven van Haastregt for helping with PNgen and SystemC simulation tool, Mohamed Bamakhrama for sharing his work on real-time scheduling, Sobhan Niknam, Peng Wang, Hongchan Shan, and Christian Fuchs, for having interesting discussions on their research ideas. Thanks to LERC, I met Tsvetan Shoshkov, whose optimism and kindness I appreciate very much.

I would like to say a big "thank you" to Teddy Zhai for his kind help with both research and Linux. And maybe more importantly, for organizing our enjoyable gatherings. I would also like to thank Shanshan Yang, for making the "computer science/electronics club" even more interesting. A big "thank you" goes to Emanuele Cannella for sharing and discussing research ideas, and giving a very appreciated feedback on my work. I enjoyed very much lunches and dinners we had together. Additionally, I would like to thank Alina Wang for being a pleasant company. Guys and girls, I hope that we are going to continue having fun together in future.

I was very lucky to have a fellow Ph.D. student Di Liu from almost the beginning of my Ph.D. studies to their end. We shared not only an office during our Ph.D. journey, but all the Ph.D.-related problems, our fears and finally successes. It was an excellent opportunity that we could embark on a new research field for us at the same time, discuss our understandings of the field, findings and our ideas. I am very happy that, despite our numerous fears that it might not be even possible, we finally made it Di! Moreover, I am happy that I learned a lot about Chinese culture and cuisine from Di. Our outside office gatherings with Di's wife Yan Liu and little daughter Ruolai Liu were very joyful moments for me. I hope that we are going to maintain our

friendship in the future regardless of our physical distance.

The research work described in this thesis has been funded by the NWO project CREED, through which I had a chance to work with Roberta Piscitelli and Simon Polstra from University of Amsterdam. I would like to thank to Roberta and Simon for their help and support with Sesame tool.

The years of my Ph.D. studies passed much easier due to the support of my friends from Serbia. I would like to thank Amela Zeković for listening about my Ph.D research, giving advices, and always having time to meet me when I come to Belgrade, and moreover, arrange some activities for us. I would like to thank Zoran Jakšić for much enjoyed discussions we had about research and life in general. Especially, I appreciate his motivating and encouraging attitude. A big "thank you" goes to Milana and Miloš Kaljević, for always having time to meet me in Belgrade, even on a short notice, and visiting me in The Netherlands. I am very grateful to Milana for being an easy person to talk to and for always giving sincere feedback. Many thanks to my "cimer" Lejla Tani-Papić for being always optimistic when it comes to my Ph.D. studies. I would like to thank Sanja Petrović and Ivan Josifović for a very nice time we spent together. All our conversations and gatherings are great memories for me. I would like to thank Ljubica and Dalibor Čvorić for spending a pleasant time together in The Netherlands.

A big "thank you" is for my sister Borislava Vujanović for constantly supporting me and being the best big-sister who always takes the best care of her little-sister. Many thanks to my brother-in-law Milenko Vujanović for being a great host in Belgrade. A special "thank you" is for my little nephew Andrej for the best joyful moments I had in the past year. I would like to express my greatest gratitude to my parents Snežana and Stojadin Spasić for their unconditional love, care and support. Thank you mum for always being optimistic and a patient listener, and thank you dad for strongly encouraging me throughout my education. I am thankful to Miloš's family – Stojanka, Vojislav, Marko and Biljana, for much enjoyed moments I spent with them in Kula. Thank you for your kindness and support!

Finally, I would like to give a very special "thank you" to a very special person, Miloš Ačanski. Miloš, I am afraid that I cannot find proper words to express how happy and lucky I am for having you beside me. You were always extremely patient to listen to my research ideas and solutions, being always honest and strict, and providing valuable advices. I learned from you to always strive for simplicity, clarity, and high-quality in doing research and engineering. Without your love, encouragement, and support, this thesis would not have been possible.