ISBN 978-90-90-33402-8

Generalized Strictly Periodic Scheduling Analysis, Resource Optimization, and Implementation of Adaptive Streaming Applications - Sobhan Niknam

Generalized Strictly Periodic Scheduling Analysis,Resource Optimization, and Implementation of Adaptive Streaming Applications

Generalized Strictly Periodic Scheduling Analysis, Resource Optimization, and Implementation of Adaptive Streaming Applications

Sobhan Niknam

Generalized Strictly Periodic Scheduling Analysis, Resource Optimization, and Implementation of Adaptive 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 25 augustus 2020
klokke 15:00 uur

door

Sobhan Niknam
geboren te Tehran, Iran
in 1990

Promotor:	Dr. Todor Stefanov	Universiteit Leiden
Second-Promotor:	Prof. dr. Harry Wijshoff	Universiteit Leiden

Promotion Committee:	Prof. dr. Akash Kumar	TU Dresden
	Prof. dr. Jeroen Voeten	TU Eindhoven
	Prof. dr. Paul Havinga	Universiteit Twente
	Prof. dr. Frank de Boer	Universiteit Leiden
	Prof. dr. Aske Plaat	Universiteit Leiden
	Prof. dr. Marcello Bonsangue	Universiteit Leiden

The research was supported by NWO under project number 12695 (CPS-3).

Generalized Strictly Periodic Scheduling Analysis, Resource Optimization, and Implementation of Adaptive Streaming Applications
Sobhan Niknam. -
Dissertation Universiteit Leiden. - With ref. - With summary in Dutch.

ISBN: 978-90-9033402-8
Printed by Ipskamp Printing, Enschede.

To my family

Contents

List of Figures

List of Tables

List of Abbreviations

BFD	Best-Fit Decreasing
CDP	Constrained-Deadline Periodic
CSDF	Cyclo-Static Data Flow
DSE	Design Space Exploration
DVFS	Dynamic VFS
EDF	Earliest Deadline First
EE	Energy Efficient
FFD	First-Fit Decreasing
FFID-EDF	First-Fit Increasing Deadlines EDF
FIFO	First-In First-Out
GSPS	Generalized Strictly Periodic Scheduling
HRT	Hard Real-Time
IDP	Implicit-Deadline Periodic
MADF	Mode-Aware Data Flow
MCR	Mode Change Request
MoC	Model of Computation
MOO	Maximum-Overlap Offset
MPSoC	Multi-Processor System-on-Chip

PE	Performance Efficient
RM	Rate Monotonic
RTOS	Real-time Operating System
SDF	Synchronous Data Flow
SPS	Strictly Periodic Scheduling
SRT	Soft Real-Time
TDP	Thermal Design Power
VFS	Voltage-Frequency Scaling
WCET	Worst-Case Execution Time
WFD	Worst-Fit Decreasing

Chapter 1

Introduction

IN the last few decades, tremendous developments in the field of electronics have made a significant impact on human lives. Nowadays, electronic systems have become an inevitable part of our modern-day life. They are prevalent and exist almost everywhere around us, even sometimes without noticing their presence, from our smartwatch, cell-phones, tablets to our cars and home appliances, improving the quality of our life from almost every aspect. For instance, thanks to the electronics technology, the patients' health status, e.g., vital signals such as ECG, EEG, and skin temperature, can be remotely monitored on a daily basis and accessed by hospital physicians using *wearable health-care monitoring devices* to diagnose medical symptoms like epilepsy or sleep disorders, e.g., e-Glass [77] for detection of epileptic seizures, while the patients can do their normal activities with no need of staying at a hospital or using a conventional clinical setting. As another example, we can refer to *advanced driver-assistance systems*, supporting vehicle drivers on the road and improving their safety and comfort. Examples of such systems include *the active cruise control*, which autonomously adjusts the distance to the front car, *the collision avoidance*, which warns and prompts the driver to prevent a collision with incoming unexpected obstacles, e.g., a pedestrian, and if needed autonomously brakes shortly before the collision when the driver is not responsive to the given warning, *the rearview system*, which increases the field of view for the driver, and many others.

In all of the above cases, each electronic system is enclosed into a larger entity like a device, product, or another system for which it provides a dedicated functionality. These electronic systems are known as **embedded systems**. Embedded systems are widespread in the world and use 98% of all processors according to recent studies [36, 48]. The global market for embedded systems

was valued over $165 billion in 2015 and it is anticipated to be nearly $260 billion by 2023 [1]. In this market, automotive and health-care embedded systems have gained the first- and second-largest share due to the increasing demand for smart vehicles and portable medical devices, respectively [1].

Different from general-purpose systems such as Personal Computers (PC), embedded systems are application-domain specific because they perform specific functions tightly coupled with the environment where they operate. They collect environmental information using sensors, process it, and perform an action accordingly using actuators. An important class of embedded systems is **embedded streaming systems**. Typically, these systems run software programs, called **streaming applications**, that process a continuous infinite, stream of data items coming from the environment. In these applications, data items in the stream are processed in-order using the same set of operations. Processing each data item takes a limited time and there is a little control flow between the operations. As a result, a continuous infinite, stream of data items are produced and fed into the environment. Examples of streaming applications include a wide range of applications from different application domains such as image processing, video/audio processing, network protocol processing, computer vision, navigation, digital signal processing, and many others. For instance, a popular streaming application, widely used in our daily life, on mobile phones, is watching a movie from YouTube. In such application, a video stream is continuously being received over the internet using a software defined radio protocol like WLAN, 3G, or 4G. Simultaneously, video and audio decoding like MPEG-4 and MP3 are performed on the received data stream and the decoded video and audio streams are continuously being played on the screen and speaker, respectively.

1.1 Design Requirements for Embedded Streaming Systems

In general, embedded systems are subjected to a wide range of strict design requirements compared to general-purpose systems. Some of these design requirements are common among all classes of embedded systems, including embedded streaming systems, while others are dependent on the environment where the embedded systems are deployed. In this section, we introduce explicitly the non-functional design requirements, i.e., timing, cost, and energy efficiency, that are considered in this thesis. Functional requirements, such as deadlock-free execution, etc., are implicitly considered as well.

For many embedded systems, the *timing* is a critical design requirement. In

such systems, the correct *behavior* depends not only on producing the correct output but also on whether the output is produced before a *deadline*. This timing requirement for the correct behavior of embedded systems is called a **real-time requirement** and a system with real-time requirements is called a **real-time system**. Regarding the criticality of a failure to satisfy the real-time requirements, the real-time systems can be classified into the following categories:

- **Soft Real-Time (SRT) Systems**: not always satisfying the real-time requirements does not lead to a system failure but only degrades the system performance provided that the deadline misses are within a certain threshold which the system can tolerate.
- **Hard Real-Time (HRT) Systems**: not always satisfying the real-time requirements leads to a system failure, which can have catastrophic consequences in safety- or life-critical systems.

For instance, in a video system which is an example of a SRT system, to watch a video smoothly through YouTube, a huge amount of data should be received regularly over the internet and processed in a short period of time. Otherwise, the video is played slow-motion, blurry, and jerky which greatly degrades the user experience. In contrast, in a HRT system such as the collision avoidance system found in a smart car, the collected data from camera and laser sensors mounted on the car must be processed always within a pre-defined and fixed time interval, such that the car can detect an incoming obstacle and react in time to avoid a collision. Otherwise, catastrophic consequences can happen, e.g., loss of human life. In the case of embedded streaming systems, timing requirements that are typically considered and guaranteed are **throughput** and/or **latency**. The throughput represents the rate at which the output is produced by a streaming application, whereas the latency represents the elapsed time between the arrival of a data item to the application and the output of the processed data item by the application.

For high-volume embedded systems, especially in consumer electronics, keeping the *cost* of a system competitive in mass markets is extremely important for survival [57]. Therefore, embedded system designers should make efficient use of hardware resources (i.e., processors, memories, etc.), either by reducing the amount of resources needed to implement a required functionality or by utilizing the available resources on a single hardware platform efficiently by running as many required applications as possible. In the latter case, different applications may share resources. Such resource sharing, however, should not affect the timing requirements and guarantees for the different applications. This property is known as *temporal isolation*, that is, the

ability to start or stop applications at run-time without violating the timing requirements of other concurrently running applications on a shared hardware platform.

Usually, embedded systems operate using stand-alone power supply such as batteries. As frequently replacing/recharging the batteries is not desirable/-possible for many embedded systems, the *energy efficiency* is another important design requirement in order to prolong the operational time of such systems on a single battery charge.

1.2 Trends in Embedded Streaming Systems Design

At the beginning of this chapter, we have introduced the embedded systems and explained their importance in our daily life. We have also pointed out, in Section 1.1, the set of non-functional design requirements for embedded streaming systems, considered in this thesis. In this section, therefore, we discuss the current trends in designing embedded streaming systems to satisfy the aforementioned design requirements.

1.2.1 Multi-Processor System-on-Chip (MPSoC)

Traditionally, embedded (streaming) systems were implemented on top of uniprocessors for a long period of time. Following the same trend as in general-purpose systems, the embedded (streaming) systems designers relied on enhancing the computational power of uniprocessors by scaling up their operational clock frequency as well as employing advanced micro-architectural innovations, such as pipelining, branch prediction, out-of-order execution, cache memory hierarchy and others, to satisfy the tight timing requirements, i.e., high throughput and/or low latency, in streaming applications [41]. This enhancement of the computational power had been driven by the fast development of the technology node which had enabled chip manufacturers to produce thinner and faster transistors, the fundamental elements in digital electronic circuits, and made it possible to integrate more and more transistors on a chip, as the result of the *Moore's Law*[1] coupled with the *Dennard scaling*[2] [68]. However, by reaching a technology node below 100 nanometers,

[1] Moore's Law refers to Moore's prediction in 1965 that the number of transistors on a chip doubles every 18 months.

[2] In 1974, Dennard *et al.* [30] postulated that the power density in a chip remains roughly constant by scaling the transistor size from one technology node to another, widely known as "*Dennard Scaling*", i.e., the power consumption of transistors scales down as long as their size is reduced.

the *Dennard's Scaling* fails due to the extremely increased leakage power consumption of transistors, i.e., the consumed power caused by currents that leak through transistors when transistors are idle. In addition, when the size of transistors decreases, their density increases on a chip resulting in increased on-chip power density which leads to overheating issues and makes on-chip thermal hotspots [73]. To avoid the overheating issues, the power consumption of chips is constrained severely with a safe power level, called *thermal design power* (TDP), provided by chip manufacturers [59]. To keep the power consumption within the TDP budget, uniprocessors have to operate at a lower operational clock frequency instead of the maximum possible frequency [59]. Moreover, the usage of many micro-architectural innovations in uniprocessors quickly reached the point of diminishing return in performance and increased design complexity. As a consequence, chip manufacturers were forced to look for an alternative to the uniprocessor paradigm.

As a solution to enhance the system performance even further while coping with the aforementioned high power consumption, chip manufacturers have shifted their design scheme towards **multi-processor platforms** in order to effectively utilize the growing number of transistors on a chip. In such platforms, the issue of increased power consumption has been partially resolved by replacing a complex processor running at a high operational voltage and clock frequency with multiple relatively simpler processors running at a lower operational voltage and clock frequency. In this way, the system performance can be enhanced through parallel processing while keeping the power and complexity under control. Nowadays, due to the advances in the chip fabrication technology, embedded system designers can integrate all components, including multiple processors, memories, interconnections, and other hardware peripherals, necessary for an application into a single chip, the so-called **Multi-Processor System-On-Chip (MPSoC)** [44]. Indeed, MPSoCs are a suitable way of implementing embedded streaming systems as they can provide high-performance, timing guaranteed, low-cost, compact, light, and low power/energy products. To further reduce the power/energy consumption, MPSoC platforms are usually armed with a Voltage and Frequency Scaling (VFS) mechanism [71]. In general, a VFS mechanism trades performance for power/energy consumption by adjusting the voltage and operating frequency of processors.

An example of an MPSoC is the Samsung Exynos 5 Octa (5422) [70], shown in Figure 1.1, which can be found in the Samsung Galaxy S5 mobile phones. This MPSoC is based on the big.LITTLE architecture [40] and has one cluster of four performance-efficient ARM Cortex-A15 cores and one cluster of four

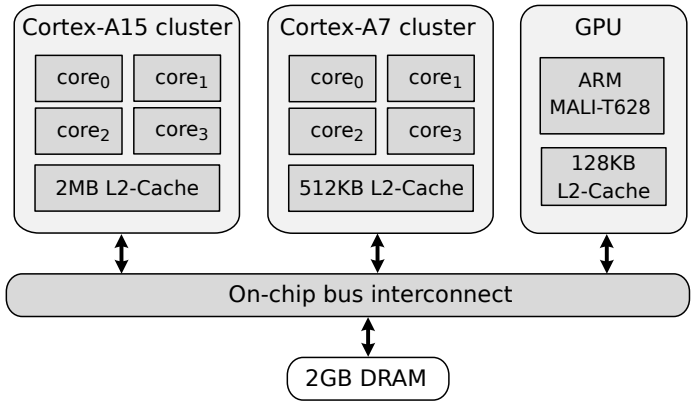


Figure 1.1: *Samsung Exynos 5422 MPSoC [70].*

energy-efficient Cortex-A7 cores. Additionally, it has the ARM Mali-T628 GPU containing 6 cores for graphical processing and 2GB DRAM on-chip memory. All the processors are connected through an on-chip bus interconnect. For the Cortex-A15 cluster, the frequency can be varied between 200 MHz to 2000 MHz whereas for the Cortex-A7 cluster, it can be varied between 200 MHz to 1400 MHz, with a step of 100 MHz in both clusters. Note that the voltage is adjusted by the firmware automatically according to pre-set pairs of voltage-frequency values.

1.2.2 Model-based Design

To satisfy the tight timing requirements of streaming applications (introduced in Section 1.1), the computational capacity of MPSoC platforms (introduced in Section 1.2.1) must be efficiently exploited. To facilitate this, streaming applications must be expressed primarily in a parallel fashion. The common practice for expressing the parallelism in an application is to use parallel **Models of Computation (MoCs)** in which the application is specified, at a high level of abstraction, as a set of parallel or concurrent tasks with specific communication and synchronization semantics. In particular, a parallel MoC defines, in a formal way, the rules by which the tasks of an application compute, communicate, and synchronize among each other. As a consequence, adopting MoCs during a design process enables system designers to reason about both functional and non-functional properties of an application. A design process which exploits MoCs is called **Model-based Design**.

In the past three decades, a variety of parallel MoCs have been proposed [43, 53]. This variety enables designers to choose the most suitable

parallel MoCs for the considered application domain. For streaming applications, that are the main focus of this thesis, *dataflow* MoCs have been identified as the most suitable parallel MoCs [88]. Within a dataflow MoC, a streaming application is modeled as a directed graph, where the graph nodes represent the application tasks and the graph edges represent data dependencies among the tasks. Thus, the parallelism is explicitly specified in the model. In general, dataflow MoCs differ among each other by their *expressiveness*, *analyzability*, and *implementation efficiency* [86]. The expressiveness of a model indicates what type of applications the model is capable of modeling and how compact the model is. The analyzability of a model is determined by the availability of design-time analysis techniques for checking (non-)functional requirements of the modeled application, e.g., liveness[3], boundedness[4], and throughput/latency, as well as by the computational complexity of the analysis techniques. Finally, the implementation efficiency of a model is influenced by the complexity of the scheduling problem and the code size of the resulting schedules. Basically, the expressiveness and analyzability are inversely related, meaning that, MoCs with high expressiveness exhibit low analyzability, and vice versa. Similarly, MoCs with high expressiveness generally have lower implementation efficiency. Therefore, there is no a single MoC which performs superior among all existing MoCs in all of the three aforementioned criteria. Consequently, designers have to choose a suitable MoC depending on their needs. A detailed and complete comparison of different dataflow MoCs is provided in [86, 93].

In this thesis, we use two well-known dataflow MoCs to specify streaming applications, namely, Synchronous Data Flow (SDF) [52] and its generalization Cyclo-Static Data Flow (CSDF) [16], due to their high analyzability. For these MoCs, various powerful analysis methods have been developed over the past two decades to evaluate liveness/boundedness [34], to compute throughput/latency [9,10,19,35,56,78,82], buffer sizes [9,10,78,85,91], and so on. These MoCs are mainly suitable and used to specify streaming applications with static behavior. But, modern streaming applications may exhibit adaptive/dynamic behavior at run-time. For example, a computer vision system processes different parts of an image continuously to obtain information from several regions of interest depending on the actions taken by the external environment [94]. To model such adaptive behavior while having a certain degree of

[3] An application is *live* if each task of the application can execute infinitely, i.e., no deadlock occurs.

[4] An application is *bounded* if the application can execute infinitely with a bounded amount of memory needed for communication/synchronization among its tasks, i.e., no buffer overflow occurs.

analyzability, in this thesis, we use a more expressive dataflow MoC, namely, Mode-Aware Data Flow (MADF) [94], which is proposed and deployed as an extension of the CSDF MoC, as well. MADF can capture the behavior of an adaptive streaming application as a collection of different static behaviors, called *modes*, which are individually analyzable at design-time. The formal definitions of the aforementioned dataflow MoCs are given in Chapter 2.

1.3 Two Important Design Challenges

Although dataflow MoCs resolve the problem of explicitly exposing the available parallelism in an application, two challenges remain, namely, how to execute the tasks of a dataflow-modeled application spatially, i.e., *task mapping*[5], and temporally, i.e., *task scheduling*, on an MPSoC platform such that all timing requirements are satisfied while making efficient utilization of available resources (e.g, processors, memory, energy, etc.) on the platform. More precisely, the task mapping determines how tasks are distributed among the processors whereas the task scheduling determines the time periods in which each task is executed on a processor. These two challenges have been identified as two of the most urgent design challenges needed to be solved for implementing embedded systems [58,75]. To address these challenges, several scheduling policies have been proposed for streaming applications, specified using dataflow MoCs and executed on MPSoC platforms. For a long period of time, *self-timed scheduling* was considered as the most appropriate scheduling policy for streaming applications [51]. Under self-timed scheduling, a task executes *as soon as possible* when its input data is ready. This scheduling policy, however, has two significant drawbacks: 1) it does not provide temporal isolation (introduced in Section 1.1) among applications concurrently running on a shared MPSoC platform; 2) it needs a complex design space exploration (DSE) to determine the minimum number of required processors and the mapping of tasks to these processors in an MPSoC platform such that all timing requirements are satisfied.

In contrast, many scheduling algorithms from the classical hard real-time scheduling theory for multiprocessors [21,29] have the following attractive properties: 1) the minimum number of processors needed to schedule a certain set of tasks and their mapping on processors can be calculated in a fast, yet accurate analytical way; 2) temporal isolation among different applications is guaranteed; 3) fast admission and scheduling decisions for new incoming applications can be performed at run-time. In these scheduling algorithms,

[5] Also referred as *tasks allocation* in the literature. Both are used interchangeably in this thesis.

the tasks of an application are specified using a *real-time task model*. The most influential example of such a task model is the *periodic real-time* task model [54] in which a task is invoked in a strictly periodic way, with a constant interval between invocations. Each task invocation has a constant execution time which must be completed before a certain deadline. These scheduling algorithms, however, typically assume sets of *independent* periodic or sporadic tasks. Thus, such a simple task model is not directly applicable to streaming applications that have data-dependent tasks.

In recent years, several approaches [8–10, 78, 79] have been proposed to bridge the gap between the dataflow MoCs that support data-dependent tasks and the classical hard real-time scheduling theory which mainly considers independent periodic/sporadic tasks. Using these approaches, the dependent tasks of an application, specified by an *acyclic* CSDF graph, can be converted to a set of real-time periodic tasks. Therefore, this conversion enables the utilization of many scheduling algorithms from the classical hard real-time scheduling theory that offer properties such as temporal isolation and fast calculation of the number of processors needed to guarantee the required performance. Motivated by the above discussion, we use the approach proposed in [8] as a basis and research driver in this thesis.

1.4 Research Questions

After introducing some important requirements, trends, and challenges in the design of embedded streaming systems in Section 1.1, Section 1.2, and Section 1.3, respectively, in this section, we formulate the specific research questions addressed in this thesis concerning the design of embedded streaming systems. Recall that we consider the scheduling framework proposed in [8], namely the so-called strictly periodic scheduling (SPS) framework, as the basis and research driver in this thesis. To easily introduce the research questions, addressed in this thesis, and the logical connection between them, a design flow which incorporates the SPS framework, as the main component, is illustrated in Figure 1.2. The design flow involves three phases, namely, analysis, resource optimization, and implementation, each of them highlighted with a different color. The rectangular boxes represent the input(s)/output(s) to/from each phase of the design flow, whereas the ellipsoid boxes represent the operations performed in the phases. The dashed lines and boxes denote the research questions and contributions of this thesis, respectively. In the following subsections, we shortly explain each phase of the design flow and introduce the research question belonging to each phase.

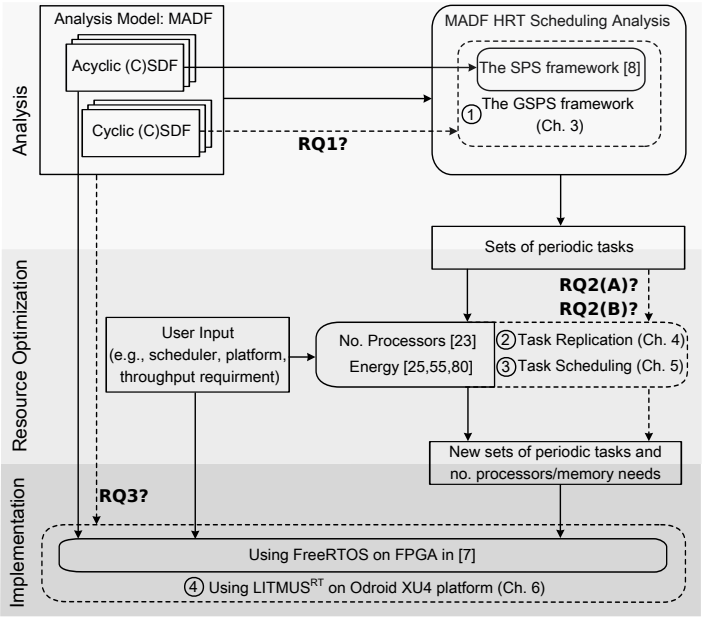


Figure 1.2: *Overview of the research questions and contributions in this thesis using a design flow.*

1.4.1 Phase 1: Analysis

The input to the first phase of the design flow is an adaptive streaming application specified using the MADF MoC [94]. Note that if the application has static behavior, its MADF specification has only one mode which is specified by a (C)SDF graph. Then, a HRT scheduling analysis is performed on the (C)SDF specification of each mode of the application using the SPS framework [8]. The result of this analysis is a derived set of periodic tasks for each mode of the application. To verify whether the timing requirements of the application are satisfied, a HRT analysis for the application execution during mode transitions, when the application's behavior is switching from one mode to another one, is provided in [94].

The SPS framework, however, as mentioned in Section 1.3, only accepts, as input, streaming applications specified as *acyclic* CSDF graphs, thereby enabling the utilization of many scheduling algorithms from classical hard real-time scheduling theory only for *acyclic* CSDF graphs. Consequently, these well-developed hard real-time scheduling algorithms cannot be applied to many streaming applications that are specified as **cyclic** CSDF graphs, i.e., graphs where the tasks have cyclic data dependencies. Thus, we formulate

the first research question addressed in this thesis as follows.

RQ1: How to apply the hard real-time scheduling theory to streaming applications, specified as CSDF graphs, with cyclic dependencies?

1.4.2 Phase 2: Resource Optimization

The inputs to the second phase of the design flow are sets of periodic tasks, derived in the first phase, and some user inputs such as the platform on which the tasks will execute, the (hard) real-time scheduling algorithm used to schedule the tasks on the platform, and timing requirements (e.g., throughput). Then, in this phase, the number of required processors on the platform and the task mapping for each mode of the application are analytically computed using the scheduling algorithm, selected by the user, such that all timing requirements are satisfied. The outputs of this phase are a new derived sets of periodic tasks along with their task mapping, number of processors required to satisfy the timing requirements, and the memory needed for data communication/synchronization among the tasks.

Regarding the design requirements, mentioned in Section 1.1, in this phase, further improvements can be performed on the tasks mapping and scheduling to more efficiently utilize the limited resources, i.e., the number of processors and energy budget, available on the platform. To this end, several task mapping and scheduling approaches using the SPS framework have been proposed in [23, 25, 55, 80]. As the computational capacity of the processors is underutilized under partitioned scheduling algorithms[6] due to the capacity fragmentation issue, i.e., no single processor has sufficient remaining capacity to schedule any other task in spite of the existence of a total large amount of unused capacity on the platform, a mapping and scheduling approach is proposed in [23] to more efficiently exploit the computational capacity of the processors by allowing only certain tasks to migrate between multiple processors while the rest of the tasks are statically allocated on the processors. Although this approach can result in better processor utilization, it increases the memory needs and latency of the application significantly. Thus, we formulate the second research question addressed in this thesis as follows.

RQ2(A): How to alleviate the capacity fragmentation issue introduced by partitioned scheduling algorithms and reduce the number of processors required for an application with a given throughput requirement while imposing less overhead on the memory needs and latency of the application?

[6] Where periodic tasks of an application are statically mapped on the processors, as introduced in Section 2.2.3 on page 24.

To achieve energy efficiency, [25, 55, 80] propose energy-efficient task mapping and scheduling approaches using the VFS mechanism mentioned in Section 1.2.1. The general idea behind these approaches is to efficiently exploit available idle (i.e., slack) times in the schedule of an application in order to slow down the execution of running tasks of the application by using the VFS mechanism to reduce the energy consumption while satisfying the throughput requirement of the application. By using the SPS framework, however, only a set of application throughputs can be guaranteed for the application. Therefore, given a required application throughput that is not in the set of guaranteed throughputs by the SPS framework, the mapping and schedule that provide the closest higher throughput to the required one must be selected from the set. This, however, reduces the amount of slack time in the schedule of the application that can be potentially exploited using the VFS mechanism to reduce the energy consumption. Thus, we formulate the third research question addressed in this thesis as follows.

RQ2(B): How to exploit more slack times in the schedule of an application with a given throughput requirement using the VFS mechanism to achieve more energy efficiency?

1.4.3 Phase 3: Implementation

Finally, the third phase of the design flow, shown in Figure 1.2, is to implement and execute the analyzed application on an MPSoC platform. The inputs to this phase are the MADF-modeled application, the selected MPSoC platform, scheduling algorithm, and timing requirements by the user, and the sets of periodic tasks derived in the second phase along with their task mapping, number of required processors, and memory needs for data communication/synchronization among the tasks. Note that since the SPS framework converts an application into a set of real-time periodic tasks, the implementation and execution of the application must be performed on top of a real-time operating system (RTOS) which provides real-time multiprocessor scheduling algorithms (e.g., Earliest Deadline First (EDF) or Rate Monotonic (RM)) needed to schedule the periodic tasks on the MPSoC platform. In this regard, [7] adopts the FreeRTOS [72], which is an open-source RTOS, and proposes an implementation and execution approach for static streaming applications, specified as acyclic (C)SDF graphs, running on a Xilinx FPGA board. Concerning adaptive streaming applications, modeled and analyzed with the MADF MoC, however, no attention has been paid so far at this implementation phase. Thus, we formulate the fourth research question addressed in this thesis as follows.

RQ3: How to implement and execute an adaptive streaming application, modeled and analyzed with the MADF MoC, on an MPSoC platform, such that the properties of the analyzed model are preserved?

1.5 Research Contributions

To address the research questions, outlined in Section 1.4, this thesis provides four research contributions represented as the dashed boxes in Figure 1.2. We summarize these research contributions in the following sub-sections.

1.5.1 Generalized Strictly Periodic Scheduling Framework

To address research question **RQ1**, we propose a novel scheduling framework, called Generalized Strictly Periodic Scheduling (GSPS), published in [64] and presented in Chapter 3, that can handle cyclic (C)SDF graphs. To this end, we first propose a sufficient test to check for the existence of a strictly periodic schedule for a streaming application modeled as a cyclic (C)SDF graph. If a strictly periodic schedule exists for the application, the tasks of the application are converted to a set of periodic tasks by computing their periods, deadlines, and earliest start times. As a consequence, this conversion enables the utilization of many well-developed HRT scheduling algorithms [21, 29] on streaming applications modeled as cyclic (C)SDF graphs to benefit from the properties of these algorithms such as HRT guarantees, fast admission control, temporal isolation, and fast calculation of the number of required processors. The experimental results, on a set of real-life benchmarks, demonstrate that our approach can schedule the tasks in an application, modeled as a cyclic CSDF graph, with guaranteed throughput equal or comparable to the throughput obtained by existing scheduling approaches while providing HRT guarantees for every task in the application thereby enabling temporal isolation among concurrently running tasks/applications on a multi-processor platform.

1.5.2 Algorithm to Find an Alternative Application Task Graph for Efficient Utilization of Processors

To address research question **RQ2(A)**, we propose a novel algorithm, published in [63] and presented in Chapter 4, to find an alternative application task graph that exposes more parallelism, particularly in the form of data-level parallelism, while preserving the same application behavior and throughput. This is needed due to the fact that a given initial application task graph is not

the most suitable one for a given MPSoC platform because the application developers, providing the initial graph, typically focus on realizing certain application behavior while neglecting the efficient utilization of the available resources on MPSoC platforms. Therefore, the main innovation in our proposed algorithm is that by using the unfolding graph transformation, introduced in Section 4.4.1, we propose a method to determine a replication factor for each task of an application, specified as an acyclic SDF graph, such that the distribution of the workloads among more parallel tasks, in the obtained graph after the transformation, results in a better resource utilization, which can alleviate the capacity fragmentation introduced by partitioned scheduling algorithms, hence reducing the number of required processors. The experimental results, on a set of real-life streaming applications, demonstrate that our approach can reduce the minimum number of processors required to schedule an application while imposing considerably less overhead, i.e., an average of up to 31.43% and 44.09% less overhead in terms of memory needs and application latency, respectively, compared to related approaches while satisfying the same throughput requirement.

1.5.3 Energy-Efficient Periodic Scheduling Approach

To address research question **RQ2(B)**, we propose a novel energy-efficient periodic scheduling approach, published in [62] and presented in Chapter 5. In this approach, the execution of an application, specified as a CSDF graph, is periodically switched at run-time between a few off-line determined energy-efficient schedules in order to satisfy the application throughput requirement in a long run. As a result, this approach can reduce the energy consumption significantly by exploiting slack times in the schedules of the application more efficiently using a Dynamic VFS (DVFS) mechanism, where multiple voltage and operating frequencies are selected at design-time for the processors to be periodically switched at run-time. The experimental results, on a set of real-life streaming applications, show that our novel scheduling approach can achieve up to 68% energy reduction depending on the application and the throughput requirement compared to related approaches.

1.5.4 MADF Implementation and Execution Approach

To address research question **RQ3**, we propose a generic parallel implementation and execution approach, published in [65] and presented in Chapter 6, for adaptive streaming applications, specified and analyzed using the MADF MoC. Our implementation and execution approach conforms to the analysis

model and its operational semantics. We demonstrate our approach using LITMUS^{RT} [22] which is one of the existing real-time extensions of the Linux kernel. To show the practical applicability of our parallel implementation and execution approach and its conformity to the analysis model, we present a case study where we implement and execute a real-life adaptive streaming application on the Odroid XU4 platform [66] with LITMUS^{RT}. Odroid XU4 features the MPSoC shown in Figure 1.1.

1.6 Thesis Outline

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

Chapter 2 presents an overview of the dataflow MoCs considered in this thesis, some relevant analysis techniques from the hard real-time (HRT) scheduling theory, and the HRT scheduling analysis of (C)SDF and MADF graphs. All of these concepts and techniques are necessary to understand the contributions of this thesis.

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

Chapter 3 presents our novel HRT scheduling framework, called GSPS, for streaming applications modeled as cyclic (C)SDF graphs. This chapter is based on our publication [64].

Chapter 4 presents our novel algorithm to optimize the number of processors needed for executing streaming applications modeled as acyclic SDF graphs under partitioned scheduling algorithms. This chapter is based on our publication [63].

Chapter 5 presents our energy-efficient periodic scheduling approach for streaming applications modeled as (C)SDF graphs. This chapter is based on our publication [62].

Chapter 6 presents the final contribution of this thesis, which is our parallel implementation and execution approach for adaptive streaming applications modeled as MADF graphs. This chapter is based on our publication [65].

Finally, **Chapter 7** ends this thesis by providing a summary of the research works done in this thesis along with some conclusions.

Chapter 2

Background

THIS chapter is dedicated to an overview of the background material needed to understand the novel research contributions of this thesis presented in the following chapters. We first provide a summary of some mathematical notations used throughout this thesis in Table 2.1.

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 of a set x
$\lceil x \rceil$	The smallest integer that is greater than or equal to x
$\lfloor x \rfloor$	The greatest integer that is smaller than or equal to x
$\hat{x}$	The maximum value of x
$\check{x}$	The minimum value of x
$\vec{x}$	The vector x
lcm	The least common multiple operator
mod	The integer modulo operator
${}^{x}V$	An x-partition of a set V (see Definition 2.2.1)

Table 2.1: *Summary of mathematical notations.*

Then, in Section 2.1, we present the dataflow MoCs that are used in this thesis. In Section 2.2, we present some results and definitions from the hard real-time (HRT) scheduling theory relevant to the context of this thesis. Finally, in Section 2.3 and 2.4, we describe the HRT analysis for the adopted dataflow MoCs.

2.1 Dataflow Models of Computation

As mentioned in Section 1.2.2, dataflow MoCs have been identified as the most suitable parallel MoCs to express the available 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 MADF MoC is given in Section 2.1.2.

2.1.1 Cyclo-Static/Synchronous Data Flow (CSDF/SDF)

An application modeled as a CSDF [16] is defined as a directed graph $G = (\mathcal{A}, \mathcal{E})$. G consists of a set of actors $\mathcal{A}$, which corresponds to the graph nodes, that communicate with each other through a set of communication channels $\mathcal{E} \subseteq \mathcal{A} \times \mathcal{A}$, which corresponds to the graph edges. Actors represent computations while communication channels represent data dependencies among actors. A communication channel $E_u \in \mathcal{E}$ is a first-in first-out (FIFO) buffer and it is defined by a tuple $E_u = (A_i, A_j)$, which implies a directed connection from actor A_i (called *source*) to actor A_j (called *destination*) to transfer data, which is divided in atomic data objects called *tokens*. An actor receiving an input data stream of the application from the environment is called *input actor* and an actor producing an output data stream of the application to the environment is called *output actor*.

An actor fires (executes) when there are enough tokens on all of its input channels. Every actor $A_i \in \mathcal{A}$ has an *execution sequence* $[f_i(1), f_i(2), \cdots, f_i(\phi_i)]$ of length ϕ_i, i.e., it has ϕ_i phases. This means that the execution of each phase $1 \leq \phi \leq \phi_i \in \mathbb{N}$ of actor A_i is associated with a certain function $f_i(\phi)$. As a consequence, the execution time of actor A_i is also a sequence $[C_i(1), C_i(2), \cdots, C_i(\phi_i)]$ consisting of the worst-case execution time (WCET) values for each phase. Every output channel E_u of actor A_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 from every input channel E_u of actor A_i is a predefined sequence $[y_i^u(1), y_i^u(2), \cdots, y_i^u(\phi_i)]$, called *consumption sequence*. Therefore, the $k-$th time that actor A_i is fired, it executes function $f_i(((k-1) \mod \phi_i)+1)$, produces $x_i^u(((k-1) \mod \phi_i)+1)$ tokens on each output channel E_u, and consumes $y_i^u(((k-1) \mod \phi_i)+1)$ tokens from each input channel E_u. The total number of produced tokens by actor A_i on channel E_u during its first n invocations and the total number of consumed tokens from the same channel by A_j during its first n invocations are $X_i^u(n) = \sum_{l=1}^{n} x_i^u(((l-1) \mod \phi_i)+1)$ and $Y_j^u(n) = \sum_{l=1}^{n} y_j^u(((l-1) \mod \phi_j)+1)$, respectively.

An important property of the CSDF model is the ability to derive a schedule

for the actors at design-time. 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 [16]). *In a CSDF graph G, a repetition vector $\vec{q} = [q_1, q_2, \cdots, q_{|\mathcal{A}|}]^T$ is given by*

$$\vec{q} = \Theta \cdot \vec{r} \quad with \quad \Theta_{ik} = \begin{cases} \phi_i & if\ i = k \\ 0 & otherwise \end{cases} \tag{2.1}$$

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

$$\Gamma \cdot \vec{r} = \vec{0} \tag{2.2}$$

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

$$\Gamma_{ui} = \begin{cases} X_i^u(\phi_i) & if\ actor\ A_i\ produces\ on\ channel\ E_u \\ -Y_i^u(\phi_i) & if\ actor\ A_i\ consumes\ from\ channel\ E_u \\ 0 & otherwise. \end{cases} \tag{2.3}$$

Theorem 2.1.1 shows that a repetition vector and hence a valid static schedule can only exist if the balance equation, given as Equation (2.2), has a non-trivial solution [16]. A graph G that meets this requirement is said to be consistent. An entry $q_i \in \vec{q} = [q_1, q_2, \cdots, q_{|\mathcal{A}|}]^T \in \mathbb{N}^{|\mathcal{A}|}$ denotes how many times an actor $A_i \in \mathcal{A}$ executes in every graph iteration of G. If a deadlock-free schedule can be found, G is said to be live. When every actor $A_i \in \mathcal{A}$ in G has a single phase, i.e., $\phi_i = 1$, the graph G is a Synchronous Data Flow (SDF) [52] graph, meaning that the SDF MoC is a subset of the CSDF MoC.

For example, Figure 2.2(b) shows a CSDF graph. The graph has a set $\mathcal{A} = \{A_1, A_2, A_3, A_4, A_5\}$ of five actors and a set $\mathcal{E} = \{E_1, E_2, E_3, E_4, E_5\}$ of five FIFO channels that represent the data dependencies between the actors. In this graph, there is one input actor (i.e., A_1) and one output actor (i.e., A_5). Each actor has different number of phases, an execution time sequence, and production/consumption sequences on different channels. For instance, actor A_1 has two phases, i.e., $\phi_1 = 2$, its execution time sequence (in time units) is $[C_1(1), C_1(2)] = [1, 1]$ and its token production sequence on channel E_4 is $[0, 1]$. Then, according to Equations (2.1), (2.2), and (2.3) in Theorem 2.1.1, we can derive the repetition vectors $\vec{q}$ as follows:

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

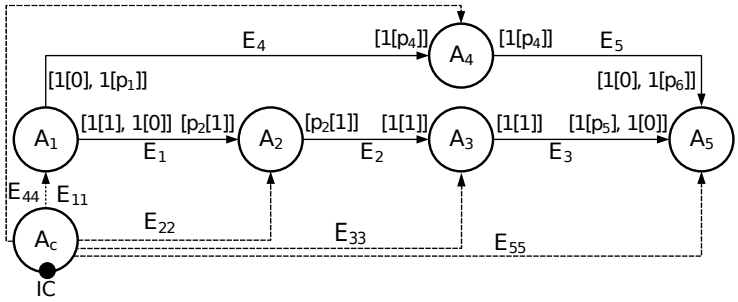
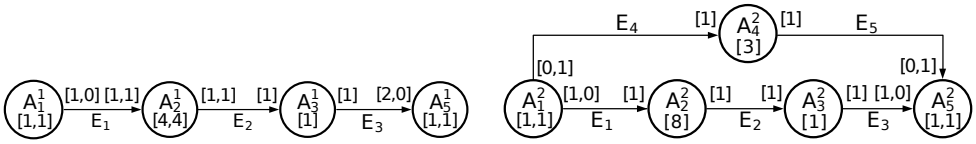


Figure 2.1: *Example of an MADF graph (G_1).*



(a) CSDF graph G_1^1 of mode SI^1. (b) CSDF graph G_1^2 of mode SI^2.

Figure 2.2: *Two modes of the MADF graph in Figure 2.1.*

2.1.2 Mode-Aware Data Flow (MADF)

MADF [94] is an adaptive MoC which can capture multiple application modes associated with an adaptive streaming application, where each individual mode is represented as a CSDF graph [16]. Formally, an MADF is a multigraph defined by a tuple $(\mathcal{A}, A_c, \mathcal{E}, P)$, where $\mathcal{A}$ is a set of dataflow actors, A_c is the control actor to determine modes and their transitions, $\mathcal{E}$ is the set of edges for data/parameter transfer, and $P = \{\vec{p}_1, \vec{p}_2, \cdots, \vec{p}_{|\mathcal{A}|}\}$ is the set of parameter vectors, where each $\vec{p}_i \in P$ is associated with a dataflow actor $A_i \in \mathcal{A}$. The detailed formal definitions of all components of the MADF MoC can be found in [94].

Here, we explain the MADF intuitively by an example. The MADF graph G_1 of an adaptive streaming application with two different modes is shown in Figure 2.1. This graph consists of a set of five actors A_1 to A_5 that communicate data over FIFO channels, i.e., the edges E_1 to E_5. Also, there is an extra actor A_c which controls the switching between modes through control FIFO channels, i.e., the edges E_{11}, E_{22}, E_{33} E_{44}, and E_{55}, at run-time. Each data FIFO channel contains a production and a consumption pattern, and some of these production and consumption patterns are parameterized. Having different values of parameters and WCET of the actors determine different

modes. For example, to specify the consumption pattern with variable length on a data FIFO channel in graph G_1, the parameterized notation $[a[b]]$ is used to represent a sequence of a elements with integer value b, e.g., $[2[1]] = [1, 1]$ and $[1[2]] = [2]$. For the MADF example in Figure 2.1, $P = \{\vec{p}_1 = [p_1], \vec{p}_2 = [p_2], \vec{p}_3 = [], \vec{p}_4 = [p_4], \vec{p}_5 = [p_5, p_6]\}$. Now let assume that the parameter vector $[p_1, p_2, p_4, p_5, p_6]$ can take only two values $[0, 2, 0, 2, 0]$ and $[1, 1, 1, 1, 1]$. Then, A_c can switch the application between two corresponding modes SI^1 and SI^2 by setting the parameter vector to the first value and the second value, respectively, at run-time. Figure 2.2(a) and Figure 2.2(b) show the corresponding CSDF graphs of modes SI^1 and SI^2.

While the operational semantics of an MADF graph [94] in steady-state, i.e., when the graph is executed in each individual mode, are the same as that of a CSDF graph [16], the transition of MADF graph from one mode to another is the crucial part that makes MADF fundamentally different from CSDF. The protocol for mode transitions has a strong impact on the design-time analyzability and implementation efficiency, discussed in Section 1.2.2. In the existing adaptive MoCs like FSM-SADF [32], a protocol, referred as self-timed transition protocol, has been adopted which specifies that tasks are scheduled as soon as possible during mode transitions. This protocol, however, introduces timing interference of one mode execution with another one that can significantly affect and fluctuate the latency of an adaptive streaming application across a long sequence of mode transitions. To avoid such undesirable behavior caused by the self-timed transition protocol, MADF employs a simple, yet effective transition protocol, namely the maximum-overlap offset (MOO) transition protocol [94] when switching an application's mode by receiving a mode change request (MCR) from the external environment via the IC port of actor A_c (see the black dot in Figure 2.1). The MOO protocol can resolve the timing interference between modes upon mode transitions by properly offsetting the starting time of the new mode by $x^{o \to n}$ computed as follows:

$$x^{o \to n} = \begin{cases} \max_{A_i \in \mathcal{A}^o \cap \mathcal{A}^n} (S_i^o - S_i^n) & \text{if } \max_{A_i \in \mathcal{A}^o \cap \mathcal{A}^n} (S_i^o - S_i^n) > 0 \\ 0 & \text{otherwise,} \end{cases} \tag{2.4}$$

where S_i^o and S_i^n are the start times of actor A_i in mode SI^o and SI^n, i.e., the current and the new mode, respectively.

For instance, consider the valid schedules of modes SI^1 and SI^2 shown in Figure 2.3(a) and (b), respectively. In these figures, H is the *iteration period*, also called *hyper period*, that represents the duration needed by the graph to complete one iteration and L is the *iteration latency* that represents the time

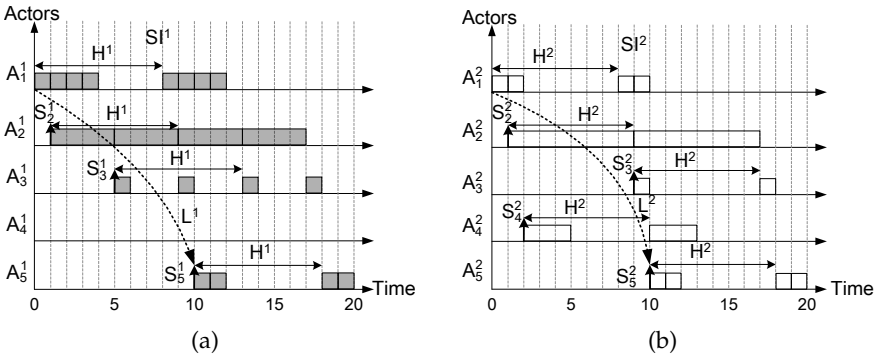


Figure 2.3: *Execution of two iterations of both modes SI^1 and SI^2. (a) Mode SI^1 in Figure 2.2(a). (b) Mode SI^2 in Figure 2.2(b).*

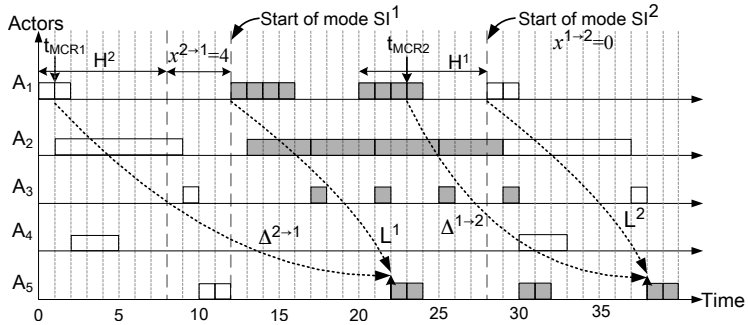


Figure 2.4: *Execution of graph G_1 with two mode transitions under the MOO protocol.*

distance between the starting times of the input actor and the output actor. Then, the offset $x^{1\to2}$ for the mode transition from SI^1 to SI^2 is computed by the following equations: $S_1^1 - S_1^2 = 0 - 0 = 0, S_2^1 - S_2^2 = 1 - 1 = 0, S_3^1 - S_3^2 = 5 - 9 = -4, S_5^1 - S_5^2 = 10 - 10 = 0$, and is $\max(0, 0, -4, 0) = 0$. Similarly, the offset $x^{2\to1}$ for the mode transition from SI^2 to SI^1, using the equations $S_1^2 - S_1^1 = 0, S_2^2 - S_2^1 = 0, S_3^2 - S_3^1 = 4, S_5^2 - S_5^1 = 0$, is $\max(0, 0, 4, 0) = 4$. An execution of G_1 with the two mode transitions and the computed offsets is illustrated in Figure 2.4, in which, the iteration latency L of the schedule of the modes, in Figure 2.3(a) and (b), are preserved during mode transitions.

To quantify the responsiveness of a transition protocol, a metric, called *transition delay* and denoted by $\Delta^{o\to n}$, is also introduced in [94] and calculated as

$$\Delta^{o\to n} = \sigma_{out}^{o\to n} - t_{\text{MCR}} \tag{2.5}$$

where $\sigma_{out}^{o\to n}$ is the earliest start time of the output actor in the new mode

SI^n and t_{MCR} is the time when the mode change request MCR occurred. In Figure 2.4, we can compute the transition delay for MCR1 occurred at time $t_{MCR1} = 1$ as $\Delta^{2\to1} = 22 - 1 = 21$ time units.

2.2 Real-Time Scheduling Theory

In this section, we introduce the real-time periodic task model [29] and some important real-time scheduling concepts and algorithms [29] which are instrumental to the contributions we present in this thesis.

2.2.1 System Model

To present the important results from the real-time scheduling theory relevant to this thesis, we consider a *homogeneous* multiprocessor system composed of a set $\Pi = \{\pi_1, \pi_2, \cdots, \pi_m\}$ of m identical processors. However, the results of our research contributions, presented in this thesis, are applicable to heterogeneous multiprocessor systems as well. This is because the processor heterogeneity can be captured within the WCET of real-time periodic tasks, which will be explained in Chapter 4.

2.2.2 Real-Time Periodic Task Model

Under the real-time periodic task model, applications running on a system are modeled as a set $\Gamma = \{\tau_1, \tau_2, \cdots, \tau_n\}$ of n periodic tasks, that can be preempted at any time. Every periodic task $\tau_i \in \Gamma$ is represented by a tuple $\tau_i = (C_i, T_i, S_i, D_i)$, where C_i is the WCET of the task, T_i is the period of the task in relative time units, S_i is the start time of the task in absolute time units, and D_i is the deadline of the task in relative time units. The task τ_i is said to be a *constrained-deadline periodic* (CDP) task if $D_i \leq T_i$. When $D_i = T_i$, the task τ_i is said to be an *implicit-deadline periodic* (IDP) task. Each task τ_i executes periodically in a sequence of task invocations. Each task invocation releases a *job*. The $k-$th job of task τ_i, denoted as $\tau_{i,k}$, is released at time instant $s_{i,k} = S_i + kT_i, \forall k \in \mathbb{N}_0$ and executed for at most C_i time units before reaching its deadline at time instant $d_{i,k} = S_i + kT_i + D_i$.

The **utilization** of task τ_i, denoted as u_i, is defined as $u_i = C_i/T_i$, where $u_i \in (0, 1]$. For a task set Γ, u_Γ is the total utilization of Γ given by $u_\Gamma = \sum_{\tau_i \in \Gamma} u_i$. Similarly, the **density** of task τ_i is $\delta_i = C_i/D_i$ and the total density of Γ is $\delta_\Gamma = \sum_{\tau_i \in \Gamma} \delta_i$.

2.2.3 Real-Time Scheduling Algorithms

When a multiprocessor system Π and a set of real-time period tasks Γ are given, a real-time scheduling algorithm is needed to execute the tasks on the system such that all task deadlines are always met. According to [29], real-time scheduling algorithms for multiprocessor systems try to solve the following two problems:

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

Depending on how the scheduling algorithms solve the allocation problem, they can be classified as follows [29]:

- *No migration*: each task is statically allocated on a processor and no migration is allowed.
- *Task-level migration*: jobs of a task can execute on different processors. However, each job can only execute on one processor.
- *Job-level migration*: jobs of a task can migrate and execute on different processors. However, each job cannot execute on more than one processor at the same time.

A scheduling algorithm that allows migration, either at task-level or job-level, among all processors is called a **global** scheduling algorithm, while an algorithm that does not allow migration at all is called a **partitioned** scheduling algorithm. Finally, an algorithm that allows migration, either at task-level or job-level, only for a subset of tasks among a subset of processors is called a **hybrid** scheduling algorithm.

Depending on how the scheduling algorithms solve the priority assignment problem, they can be classified as follows [29]:

- *Fixed task priority*: each task has a single fixed priority that is used for all its jobs.
- *Fixed job priority*: jobs of a task may have different priorities. However, each job has only a single fixed priority.
- *Dynamic priority*: a single job of a task may have different priorities at different times during its execution.

The scheduling algorithms can be further classified into [29]:

- *Preemptive*: tasks can be preempted by a higher priority task at any time.
- *Non-preemptive*: once a task starts executing, it will not be preempted and it will execute until completion.

A task set Γ is said to be **feasible** with respect to a given system Π if there exists a scheduling algorithm that can construct a schedule in which all task deadlines are always met. A scheduling algorithm is said to be **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. A task set is said to be **schedulable** on a system under a given scheduling algorithm, if all tasks can execute under the scheduling algorithm on the system without violating any deadline. To check whether a task set is schedulable on a system under a given scheduling algorithm, the real-time scheduling theory provides various analytical **schedulability tests**. Generally, schedulability tests can be classified as follows [29]:

- *Sufficient*: if all task sets that are deemed schedulable by a schedulability test are in fact schedulable.
- *Necessary*: if all task sets that are deemed unschedulable by a schedulability test are in fact unschedulable.
- *Exact*: if a schedulability test is both sufficient and necessary.

Uniprocessor Schedulability Analysis

In this thesis, we use the *preemptive* earliest deadline first (EDF) scheduling algorithm [54], which is the most studied and popular *dynamic-priority* scheduling algorithm on uniprocessor systems, as the basis scheduling algorithm. The EDF algorithm schedules jobs of tasks according to their absolute deadlines. More specifically, jobs of tasks with earlier deadlines will be executed at higher priorities [21]. The EDF algorithm has been proven to be the optimal scheduling algorithm for periodic tasks on uniprocessor systems [21, 54]. An *exact* schedulability test for an implicit-deadline periodic task set on a uniprocessor system under EDF is given in the following theorem.

Theorem 2.2.1 (From [54])**.** *Under EDF, an implicit-deadline periodic task set* Γ *is schedulable on a uniprocessor system if and only if:*

$$u_\Gamma = \sum_{\tau_i \in \Gamma} u_{\tau_i} \leq 1. \tag{2.6}$$

For a *constrained-deadline* periodic task set, however, Equation (2.6) serves as a *necessary* test. An *exact* schedulability test for a constrained-deadline periodic task set on a uniprocessor under EDF is given in the following lemma.

Lemma 2.2.1 (From [13])**.** *Under EDF, a periodic task set* Γ *is schedulable on a uniprocessor system if and only if* $u_\Gamma \leq 1$ *and* $dbf(\Gamma, t_1, t_2) \leq (t_2 - t_1)$ *for all*

$0 \leq t_1 < t_2 < \hat{S} + 2H$, *where* $dbf(\Gamma, t_1, t_2)$, *termed as* ***processor demand bound function***, *denotes the total execution time that all tasks of* Γ *demand within time interval* $[t_1, t_2]$ *and is given by*

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

$\hat{S} = \max\{S_1, S_2, \cdots, S_{|\Gamma|}\}$, *and* $H = \text{lcm}\{T_1, T_2, \cdots, T_{|\Gamma|}\}$.

However, this schedulability test is computationally expensive because it needs to check all absolute deadlines, which can be a large number, within the time interval. To improve the efficiency of the EDF exact test, a new exact test for the EDF scheduling is proposed in [95] which checks a smaller number of time points within the time interval.

Multiprocessor Schedulability Analysis

On multiprocessor systems, there are several *optimal* global scheduling algorithms for implicit-deadline periodic tasks, such as Pfair [12] and LLREF [27], which exploit job-level migrations and dynamic priority. Under these scheduling algorithms, an exact schedulability test for an implicit-deadline periodic task set Γ on m processors is:

$$u_\Gamma = \sum_{\tau_i \in \Gamma} u_{\tau_i} \leq m. \tag{2.7}$$

Based on the above equation, the absolute minimum number of processors, denoted as $\check{m}_{\text{OPT}}$, needed by an optimal scheduling algorithm to schedule an implicit-deadline periodic task set Γ is:

$$\check{m}_{\text{OPT}} = \lceil u_\Gamma \rceil. \tag{2.8}$$

In the case of constrained-deadline periodic tasks, however, no optimal algorithm for global scheduling exists [29]. Under global dynamic priority schedulings, a *sufficient* schedulability test for a constrained-deadline periodic task set Γ on m processors is [6, 31]:

$$\delta_\Gamma = \sum_{\tau_i \in \Gamma} \delta_{\tau_i} \leq m. \tag{2.9}$$

According to this test, the minimum number of processors needed by a global dynamic priority scheduling to schedule a constrained-deadline periodic task set Γ is:

$$\check{m} = \lceil \delta_\Gamma \rceil. \tag{2.10}$$

The other class of multiprocessor scheduling algorithms for periodic task sets are partitioned scheduling algorithms [29] that do not allow task migration. Under partitioned scheduling algorithms, a task set is first partitioned into subsets (according to Definition 2.2.1) that will be executed statically on individual processors. Then, the tasks on each processor are scheduled using a given uniprocessor scheduling algorithm.

Definition 2.2.1. *(Partition of a set).* Let V be a set. An x-partition of V is a set, denoted by ${}^{x}V$, where

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

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

$$ \bigcap_{i=1}^{x} {}^{x}V_i = \emptyset \quad \text{and} \quad \bigcup_{i=1}^{x} {}^{x}V_i = V. $$

In this regard, the minimum number of processors needed to schedule a task set Γ by a partitioned scheduling algorithm is:

$$ \check{m}_{\text{PAR}} = \min\{x \in \mathbb{N} \mid \exists x\text{-partition of } \Gamma \wedge \forall i \in [1, x] : {}^{x}\Gamma_i \text{ is schedulable on } \pi_i\}. \tag{2.11} $$

The derived x-partition of a task set, using Equation (2.11), is optimal because of requiring the least amount of processors to allocate all tasks while guaranteeing schedulability on all processors. Deriving such optimal partitioning is inherently equivalent to the well-known *bin packing* problem [45]. In the bin packing problem, items of different sizes must be packed into bins with fixed capacity such that the number of needed bins is minimized. However, finding an optimal solution for the bin packing problem is known to be NP-hard [46]. Therefore, several heuristic algorithms have been developed to solve the bin packing problem and obtain approximate solutions in a reasonable time interval. Below, we introduce the most commonly used heuristics [28, 46].

- **First-Fit (FF)** algorithm: places an item to the first (i.e., lowest index) bin that can accommodate the item. If no such bin exists, a new bin is opened and the item is placed on it.
- **Best-Fit (BF)** algorithm: places an item to a bin that can accommodate the item and has the minimal remaining capacity after placing the item. If no such bin exists, a new bin is opened and the item is placed on it.
- **Worst-Fit (WF)** algorithm: places an item to a bin that can accommodate the item and has the maximal remaining capacity after placing the item. If no such bin exists, a new bin is opened and the item is placed on it.

The performance of these heuristic algorithms can be improved by sorting the items according to a certain criteria, such as their size. Then, we obtain the **First-Fit Decreasing (FFD)**, **Best-Fit Decreasing (BFD)**, and **Worst-Fit Decreasing (WFD)** heuristics.

2.3 HRT Scheduling of Acyclic CSDF Graphs

As mentioned in Section 1.3, recently, a scheduling framework, namely, the Strictly Periodic Scheduling (SPS) framework, has been proposed in [8] which enables the utilization of many scheduling algorithms from the classical hard real-time scheduling theory (briefly introduced in Section 2.2) to applications modeled as acyclic CSDF graphs. The main advantages of these scheduling algorithms are that they provide: 1) temporal isolation and 2) fast, yet accurate calculation of the minimum number of processors that guarantee the required performance of an application and mapping of the application's tasks on processors. The basic idea behind the SPS framework is to convert a set $\mathcal{A} = \{A_1, A_2, \cdots, A_n\}$ of n actors of a given CSDF graph to a set $\Gamma = \{\tau_1, \tau_2, \cdots, \tau_n\}$ of n real-time implicit-deadline periodic tasks[1]. In particular, for each actor $A_j \in \mathcal{A}$ of the CSDF graph, the SPS framework derives the parameters, i.e., the period (T_j) and start time (S_j), of the corresponding real-time periodic task $\tau_j = (C_j, T_j, S_j, D_j = T_j) \in \Gamma$. The period T_i of task τ_j corresponding to actor A_j under the SPS framework can be computed as:

$$T_j = \frac{\text{lcm}(\vec{q})}{q_j} \cdot s, \tag{2.12}$$

$$s \geq \check{s} = \left\lceil \frac{\hat{W}}{\text{lcm}(\vec{q})} \right\rceil \in \mathbb{N}, \tag{2.13}$$

where $\text{lcm}(\vec{q})$ is the least common multiple of all repetition entries in $\vec{q}$ (explained in Section 2.1.1), $\hat{W} = \max_{A_j \in \mathcal{A}}\{C_j \cdot q_j\}$ is the maximum actor workload of the CSDF graph, and $C_j = \max_{1 \leq \phi \leq \phi_j}\{C_j(\phi)\}$, where $C_j(\phi)$ includes both the worst-case computation time and worst-case data communication time required by a phase ϕ of actor A_j. *Note that $C_j(\phi)$ includes the worst-case data communication time in order to ensure the feasibility of the derived schedule regardless of the variance of different task allocations*. In general, the derived period vector $\vec{T}$ satisfies the condition:

$$q_1 T_1 = q_2 T_2 = \cdots = q_n T_n = H \tag{2.14}$$

[1] Throughout this thesis, we may use the terms *task* and *actor* interchangeably.

where H is the iteration period. Once the period of each task has been computed, the throughput $\mathcal{R}$ of the graph can be computed as:

$$\mathcal{R} = \frac{1}{T_{out}} \tag{2.15}$$

where T_{out} is the period of the task corresponding to output actor A_{out}. *Note that when the scaling factor $s = \check{s} = \lceil \hat{W} / \operatorname{lcm}(\vec{q}) \rceil$, the minimum period ($\check{T}_j$) is derived using Equation (2.12) which determines the maximum throughput achievable by the SPS framework.*

Then, to sustain the strictly periodic execution of the tasks corresponding to actors of the CSDF graph with the periods derived by Equation (2.12), the earliest start time S_j of each task τ_j corresponding to actor A_j, such that τ_j is never blocked on reading data tokens from any input FIFO channel connected to it during its periodic execution, is calculated using the following expression:

$$S_j = \begin{cases} 0 & if\ \operatorname{prec}(A_j) = \emptyset \\ \max_{A_i \in \operatorname{prec}(A_j)} (S_{i \to j}) & otherwise, \end{cases} \tag{2.16}$$

where $\operatorname{prec}(A_j)$ represents the set of predecessor actors of A_j and $S_{i \to j}$ is given by:

$$\begin{aligned} S_{i \to j} = \min_{t \in [0, S_i + H]} \Big\{ t : & \operatorname*{Prd}_{[S_i, \max\{S_i, t\} + k)} (A_i, E_u) \\ & \geq \operatorname*{Cns}_{[t, \max\{S_i, t\} + k]} (A_j, E_u),\ \forall k \in [0, H], k \in \mathbb{N} \Big\} \end{aligned} \tag{2.17}$$

where $\operatorname{Prd}_{[t_s, t_e)}(A_i, E_u)$ is the total number of tokens produced by a predecessor actor A_i to channel E_u during the time interval $[t_s, t_e)$ with the assumption that token production happens as **late** as possible at the deadline of each invocation of actor A_i, $\operatorname{Cns}_{[t_s, t_e]}(A_j, E_u)$ is the total number of tokens consumed by actor A_j from channel E_u during the time interval $[t_s, t_e]$ with the assumption that token consumption happens as **early** as possible at the release time of each invocation of actor A_j, and S_i is the earliest start time of actor A_i.

The authors in [8] also provide a method to calculate the minimum buffer size needed for each FIFO communication channel and the latency of the CSDF graph scheduled in a strictly periodic fashion. In this framework, once the start time of each task has been calculated, the minimum buffer size of each FIFO communication channel $E_u = (A_i, A_j) \in \mathcal{E}$, denoted with b_u, is calculated as follows:

$$b_u = \max_{k \in [0, H]} \left\{ \operatorname*{Prd}_{[S_i, \max(S_i, S_j) + k)} (A_i, E_u) - \operatorname*{Cns}_{[S_j, \max(S_i, S_j) + k)} (A_j, E_u) \right\} \tag{2.18}$$

with the assumption that token production happens as **early** as possible at the release time of each invocation of actor A_i and token consumption happens as **late** as possible at the deadline of each invocation of actor A_j. Indeed, b_u is the maximum number of unconsumed data tokens in channel E_u during the execution of A_i and A_j in one graph iteration period. Finally, the latency $\mathcal{L}$ of the graph can be calculated as follows:

$$\mathcal{L} = \max_{w \in W}(S_{out} + g^C_{out} T_{out} + D_{out} - (S_{in} + g^P_{in} T_{in})) \tag{2.19}$$

where w is one path of set W which includes all paths in the CSDF graph from the input actor to the output actor, S_{in} and S_{out} are the earliest start times of the tasks corresponding to the input and output actors, respectively, T_{in} and T_{out} are the periods of the tasks corresponding to the input and output actors, respectively, D_{out} is the deadline of the task corresponding to the output actor, and g^C_{out} and g^P_{in} are two constants which denote the number of invocations the actor waits for the non-zero production/consumption on/from a path $w \in W$.

2.4 HRT Scheduling of MADF Graphs

Based on the proposed MOO protocol for mode transitions, briefly described in Section 2.1.2, a hard real-time analysis and scheduling framework for the MADF MoC is proposed in [94] which is an extension of the SPS framework, briefly described in Section 2.3, developed for CSDF graphs. As explained in Section 2.3, the key concept of the SPS framework is to derive a periodic task set representation for a CSDF graph. Since an MADF graph in steady-state can be considered as a CSDF graph, it is thus straightforward to represent the steady-state of an MADF graph as a periodic task set (see Section 2.3) and schedule the resulting task set using any well-known hard real-time scheduling algorithm.

Using the SPS framework, we can derive the two main parameters for each task τ^o_i corresponding to an MADF actor A_i in mode SI^o, namely the period (T^o_i using Equation (2.12)) and the earliest start time (S^o_i using Equation (2.16)). Then, the offset $x^{o \to n}$ for mode transition of the MADF graph from mode SI^o to mode SI^n can be simply computed using Equation (2.4). For instance, by applying the SPS framework for graphs G^1_1 and G^2_1, shown in Figure 2.2(a) and 2.2(b), corresponding to modes SI^1 and SI^2 of graph G_1 shown in Figure 2.1, the task set $\Gamma^1_1 = \{\tau^1_1 = (C^1_1 = 1, T^1_1 = 2, S^1_1 = 0, D^1_1 = T^1_1 = 2), \tau^1_2 = (4, 4, 2, 4), \tau^1_3 = (1, 4, 6, 4), \tau^1_5 = (1, 4, 14, 4)\}$ of four IDP tasks and the task set $\Gamma^2_1 = \{\tau^2_1 = (C^2_1 = 1, T^2_1 = 4, S^2_1 = 0, D^2_1 = T^2_1 = 4), \tau^2_2 = (8, 8, 4, 8), \tau^2_3 =$

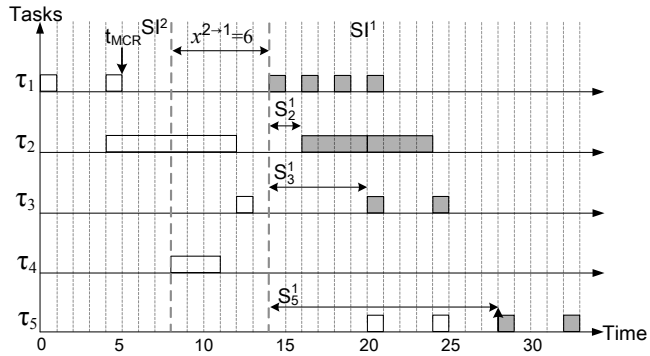


Figure 2.5: *Execution of graph G_1 with a mode transition from mode SI^2 to mode SI^1 under the MOO protocol and the SPS framework.*

$(1, 8, 12, 8)$, $\tau_4^2 = (3, 8, 8, 8)$, $\tau_5^2 = (1, 4, 20, 4)\}$ of five IDP tasks can be derived, respectively. An execution of graph G_1 with a mode transition from mode SI^2 to mode SI^1, using the derived task sets Γ_1^1 and Γ_1^2, is shown in Figure 2.5, where the offset $x^{2\to1}$ is computed by the following equations (see Equation (2.4)): $S_1^2 - S_1^1 = 0 - 0 = 0$, $S_2^2 - S_2^1 = 4 - 2 = 2$, $S_3^2 - S_3^1 = 12 - 6 = 6$, $S_5^2 - S_5^1 = 20 - 14 = 6$, and is $\max(0, 2, 6, 6) = 6$. However, this offset is only the lower bound because the task allocation on processors is not yet taken into account. This means, the execution of tasks using the schedule, shown in Figure 2.5, is valid when each task is allocated on a separate processor.

In a system where multiple tasks are allocated on the same processor, the processor may be potentially overloaded during mode transitions due to the presence of executing tasks in both modes. To avoid overloading of processors, a larger offset may be needed to delay the start time of tasks in the new mode. In [94], this offset, referred as $\delta^{o\to n}$, is calculated as follows:

$$\delta^{o\to n} = \min_{t\in[x^{o\to n}, S^o_{\text{out}}]} \{t : u_{\pi_j}(k) \leq UB,\ \forall k \in [t, S^o_{\text{out}}] \wedge \forall \pi_j \in \Pi\}. \tag{2.20}$$

This equation simply tests all time instants when tasks in both modes SI^o and SI^n are present in the system and checks whether the processors are consequently overloaded or not. If yes, the starting time of the new mode SI^n, which already was delayed by $x^{o\to n}$, is further delayed to $\delta^{o\to n}$. Thus, $\delta^{o\to n}$ of interest for the mode transition from mode SI^o to mode SI^n is the minimum time t in the bounded interval $[x^{o\to n}, S^o_{\text{out}}]$ such that the total utilization does not exceed the utilization bound (UB), e.g., 1 for EDF, for all remaining time instants in the interval. To compute the total utilization of all tasks allocated

on processor π_j in any time instant k, the following equation is used in [94].

$$u_{\pi_j}(k) = \underbrace{\sum_{\tau_i^o \in {}^x\Gamma_j} \left(u_i^o - h(k - S_i^o) \cdot u_i^o\right)}_{u_{\pi_j}^o(k)} + \underbrace{\sum_{\tau_i^n \in {}^x\Gamma_j} \left(h(k - S_i^n - t) \cdot u_i^n\right)}_{u_{\pi_j}^n(k)} \tag{2.21}$$

In this equation, the terms denoted by $u_{\pi_j}^o(k)$ and $u_{\pi_j}^n(k)$ refers to the total utilization of tasks that are allocated on processor π_j and are executing in the current mode SI^o and the new mode SI^n, respectively, at time instant k. h(t) is the Heaviside step function.

For instance, consider the execution of the tasks in the schedule, shown in Figure 2.5, on platform $\Pi = \{\pi_1, \pi_2\}$ with two processors and the tasks allocation ${}^2\Gamma = \{{}^2\Gamma_1 = \{\tau_1, \tau_3, \tau_4, \tau_5\}, {}^2\Gamma_2 = \{\tau_2\}\}$. In this schedule, the earliest start time of the new mode SI^1 is at time instant 14 corresponding to $\delta^{2\to1} = x^{2\to1} = 6$. Then, the total utilization of processor π_1 demanded by the tasks in the old mode SI^2 at time instant 14, i.e., $u_{\pi_1}^2(6)$, can be computed as follows using Equation (2.21):

$$\begin{aligned} u_{\pi_1}^2(6) &= \sum_{\tau_i^2 \in {}^2\Gamma_1} u_i^2 - h(6 - S_i^2) \cdot u_i^2,\ i \in \{1,3,4,5\} \\ &= u_1^2 - h(6) \cdot u_1^2 + u_3^2 - h(-6) \cdot u_3^2 + u_4^2 - h(-2) \cdot u_4^2 + u_5^2 - h(-14) \cdot u_5^2 \\ &= 0 + u_3^2 + u_4^2 + u_5^2 = \frac{1}{8} + \frac{3}{8} + \frac{1}{4} = \frac{3}{4}. \end{aligned}$$

Now, releasing task τ_1^1 in the new mode SI^1 at time 14 would yield

$$u_{\pi_1}(6) = u_{\pi_1}^2(6) + u_1^1 = \frac{3}{4} + \frac{1}{2} > UB = 1,$$

thereby leading to being unschedulable on processor π_1. In this case, the earliest start times of the new mode SI^1 must be delayed by $\delta^{2\to1} = 8$ time units to time instant 16 as shown in Figure 2.6. At time instant 16, the total utilization of processor π_1 demanded by the tasks in the old mode SI^2 is

$$\begin{aligned} u_{\pi_1}^2(8) &= \sum_{\tau_i^2 \in {}^2\Gamma_1} u_i^2 - h(8 - S_i^2) \cdot u_i^2,\ i \in \{1,3,4,5\} \\ &= u_1^2 - h(8) \cdot u_1^2 + u_3^2 - h(-4) \cdot u_3^2 + u_4^2 - h(0) \cdot u_4^2 + u_5^2 - h(-12) \cdot u_5^2 \\ &= 0 + u_3^2 + 0 + u_5^2 = \frac{1}{8} + \frac{1}{4} = \frac{3}{8}. \end{aligned}$$

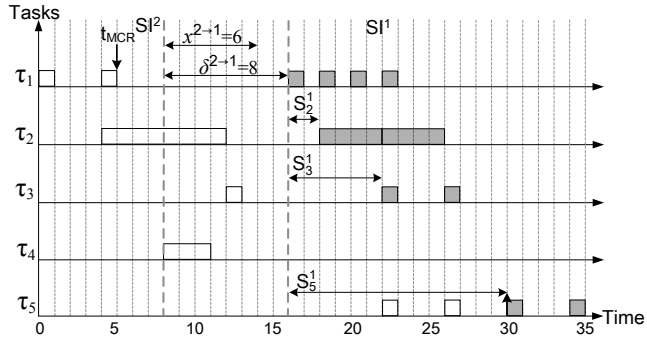


Figure 2.6: *Execution of graph G_1 with a mode transition from mode SI^2 to mode SI^1 under the MOO protocol and the SPS framework with task allocation on two processors.*

Now, releasing task τ_1^1 in the new mode SI^1 at time instant 16 results in the total utilization of processor π_1 as

$$u_{\pi_1}(8) = u_{\pi_1}^2(8) + u_1^1 = \frac{3}{8} + \frac{1}{2} < 1.$$

Next, assuming that the new mode SI^1 starts at time instant 16, the above procedure should be repeated for the remaining tasks in the new mode SI^1, namely τ_3^1 and τ_5^1, to ensure that they can start execution with S_3^1 and S_5^1, respectively, without overloading processor π_1. Then, if processor π_1 is overloaded again, a larger offset $\delta^{2\to1}$ is needed that can be calculated using Equation (2.20).

Chapter 3

Hard Real-Time Scheduling of Cyclic CSDF Graphs

Sobhan Niknam, Peng Wang, Todor Stefanov. "Hard Real-Time Scheduling of Streaming Applications Modeled as Cyclic CSDF Graphs". *In Proceedings of the International Conference on Design, Automation and Test in Europe (DATE'19)*, pp. 1528-1533, Florence, Italy, March 25 - 29, 2019.

IN this chapter, we present our Generalized Strictly Periodic Scheduling (GSPS) framework, which corresponds to the first research contribution, briefly introduced in Section 1.5.1, to address research question **RQ1**, described in Section 1.4.1. The remainder of this chapter is organized as follows. Section 3.1 introduces, in more details, the problem statement and the addressed research question. 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. A motivational example is given in Section 3.4. Then, Section 3.5 presents our proposed GSPS framework. Section 3.6 presents the experimental evaluation of our proposed GSPS framework. Finally, Section 3.7 ends the chapter with conclusions.

3.1 Problem Statement

Recall, from Section 2.3, that the Strictly Periodic Scheduling (SPS) framework [8] has been recently proposed to convert a streaming application, modeled as an acyclic CSDF graph, to a set of implicit-deadline periodic tasks. As a result, a variety of hard real-time scheduling algorithms for periodic

tasks, from the classical hard real-time scheduling theory [21,29] (briefly introduced in Section 2.2), can be applied to schedule such streaming applications with a certain guaranteed performance, i.e., throughput/latency, on MPSoC platforms. These algorithms can perform fast admission control and scheduling decisions for new incoming applications in an MPSoC platform using fast schedulability analysis while providing hard real-time guarantees and temporal isolation. In addition, these algorithms provide a fast analytical calculation of the minimum number of processors needed to schedule the tasks in an application instead of performing a complex and time-consuming design space exploration needed by conventional static scheduling of streaming applications, i.e., self-timed scheduling [85]. The SPS framework, however, is limited to acyclic CSDF graphs and cannot schedule a streaming application modeled as a cyclic CSDF graph, i.e., a graph where the actors have cyclic data dependencies. Consequently, hard real-time scheduling algorithms cannot be applied to many streaming applications modeled as cyclic CSDF graphs. Thus, in this chapter, we investigate the possibility to apply scheduling algorithms from the classical hard real-time scheduling theory to streaming applications modeled as cyclic CSDF graphs.

3.2 Contributions

In order to address the problem described in Section 3.1, in this chapter, we propose a novel scheduling framework, called Generalized Strictly Periodic Scheduling (GSPS), that can handle cyclic CSDF graphs. As a consequence, our framework enables the application of a variety of proven hard real-time scheduling algorithms [21,29] for multiprocessor systems on a wider range of applications compared to the SPS framework. More specifically, the main novel contributions of this chapter are summarized as follows:

- We propose a sufficient test to check for the existence of a strictly periodic schedule for a streaming application modeled as a cyclic (C)SDF graph;
- If a strictly periodic schedule exists for an application, the tasks of the application are converted to a set of constrained-deadline periodic tasks by computing their periods, deadlines, and earliest start times. As a consequence, this conversion enables the utilization of many well-developed hard real-time scheduling algorithms [29] on streaming applications modeled as cyclic (C)SDF graphs to benefit from the properties of these algorithms such as hard real-time guarantees, fast admission control, temporal isolation, and fast calculation of the number of required processors;

- We show, on a set of real-life streaming applications, that our approach can schedule the tasks in an application, modeled as a cyclic (C)SDF graph, as strictly periodic tasks with hard real-time guaranteed throughput which is equal or comparable to the throughput obtained by existing scheduling approaches.

3.3 Related Work

In this section, we compare our hard real-time scheduling framework with the existing hard real-time and periodic scheduling approaches [3, 8, 18, 79, 85] for streaming applications. In [8] and [78], the authors convert each actor in an *acyclic* CSDF graph to an implicit-deadline periodic task, by deriving the actor's earliest start time and period. In addition, the minimum buffer sizes of FIFO channels, that guarantee the strictly periodic execution of the tasks, are computed in [8] and [78]. These approaches, however, are limited to applications modeled as *acyclic* (C)SDF graphs. In contrast, our approach is more general than the approaches in [8] and [78] and can schedule an application, modeled as a **cyclic** (C)SDF graph, in strictly periodic fashion, if a strictly periodic schedule exists. As a result, many well-developed hard real-time scheduling algorithms [29] for periodic tasks can be applied to schedule the actors in a **cyclic** CSDF graph to provide temporal isolation between concurrently running applications, fast admission control of new incoming applications, and to compute the minimum number of required processors, using fast schedulability tests.

Ali et al. [3] propose an algorithm to convert the tasks in an application to a set of constrained-deadline periodic tasks by extracting the tasks' offset, arbitrary deadline, and period. Similar to our approach, this algorithm can deal with cyclic data dependencies in the application. However, this approach considers streaming applications modeled as Homogeneous SDF (HSDF) graphs derived by applying a certain transformation on initial (C)SDF graphs. Transforming a graph from (C)SDF to HSDF is a crucial step in which the number of tasks in the streaming application can exponentially grow, e.g., the HSDF graph of the application Echo [18], derived from a cyclic CSDF graph with 38 actors, has over 42000 actors. Such exponential growth of the application in terms of number of tasks can lead to a time-consuming analysis. Moreover, such exponential growth results in a significant memory overhead for storing the tasks' code and significant scheduling overhead due to excessive task preemptions at runtime. In addition, the derived schedule, of a transformed (C)SDF graph to a HSDF graph, is valid if all multi-rate actors

in the (C)SDF graph are transformed to functionally equivalent single-rate actors in the HSDF graph which requires modification of the actors' code. In contrast, our approach can be directly applied to streaming applications modeled with a more expressive MoC, i.e., (C)SDF graph, which avoids the significant memory and scheduling overheads introduced by large HSDF graphs as well as modification of the actors' code is not required. In addition, our approach is faster because it avoids the exponentially complex conversion of (C)SDF to HSDF.

In [18], the authors propose a framework to derive the maximum throughput of a CSDF graph under a periodic schedule and to calculate the minimum buffer sizes under a given throughput constraint. These are formulated as linear programming (LP) problems and solved approximately. In [85], a scheduling framework for exploration of the trade-off between throughput and minimum buffer sizes of (C)SDF graphs under self-timed scheduling is proposed. In [18], however, the calculation of the minimum number of processors required for the derived schedule is not taken into consideration. Moreover, the approaches in [18] and [85] do not provide hard real-time guarantees for every task in an application. Therefore, they do not ensure temporal isolation among tasks/applications. As a consequence, the schedule of already running applications has to be recalculated when a new application comes in the system. In contrast, our approach converts the tasks in applications to constrained-deadline periodic tasks. This conversion enables the utilization of many hard real-time scheduling algorithms [29] to provide temporal isolation and fast calculation of the minimum number of processors needed to schedule the tasks under certain throughput constraint. Moreover, we propose a simple analytical approach to test for the existence of a strictly periodic schedule and derive the maximum throughput of a CSDF graph under the strictly periodic schedule instead of approximately solving LP problems as done in [18].

3.4 Motivational Example

The goal of this section is to show how the actors in the cyclic CSDF graph G, shown in Figure 3.1, can be scheduled in strictly periodic fashion using our GSPS framework proposed in Section 3.5. First, assume that G has no backward edge E_5. Then, G has no cycles and the SPS framework [8] (described in Section 2.3) can convert the actors in G to IDP tasks represented by the following tuples: $\tau_1 = (C_1 = 2, \check{T}_1 = 2, S_1 = 0, D_1 = \check{T}_1 = 2)$, $\tau_2 = (2, 3, 3, 3)$, $\tau_3 = (3, 6, 4, 6)$, and $\tau_4 = (3, 3, 9, 3)$. The schedule for this periodic task set is shown in Figure 3.2. Considering E_5, however, this schedule is not valid

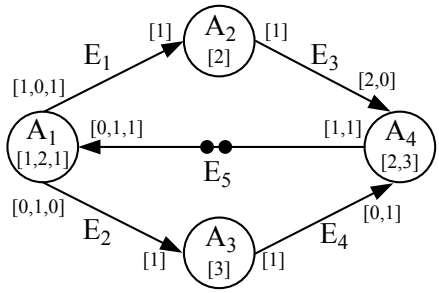


Figure 3.1: *A **cyclic** CSDF graph G. The backward edge E_5 in G has 2 initial tokens that are represented with black dots.*

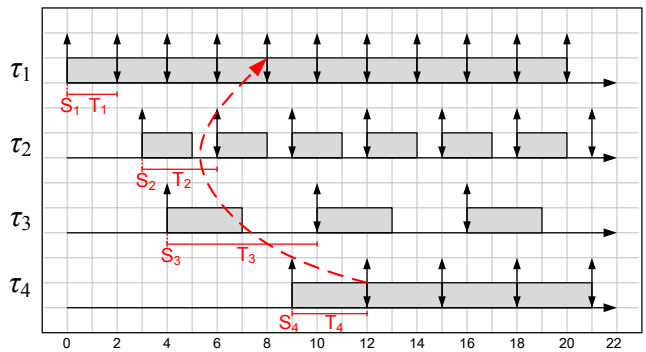


Figure 3.2: *The SPS of the CSDF graph G in Figure 3.1 without considering the backward edge E_5. Up arrows are job releases and down arrows job deadlines.*

because there is no data token available on E_5 for task τ_1 (corresponding to actor A_1) to consume at time 8 and therefore the strict periodicity of tasks' execution is no longer guaranteed. To solve this problem, we must ensure that task τ_4 (corresponding to actor A_4) can produce a data token before the fifth firing of task τ_1, as shown by the dashed line in Figure 3.2. Therefore, E_5 introduces a latency constraint between tasks τ_1 and τ_4. Please note that the derived periods of the tasks, for the schedule shown in Figure 3.2, are the minimum periods ($\check{T}_i$) by using the scaling factor $s = \check{s} = \lceil \hat{W}/\text{lcm}(\vec{q}) \rceil = 1$ in Equation (2.12). But, there exist other longer valid periods for a task by using any integer $s > \check{s} = \lceil \hat{W}/\text{lcm}(\vec{q}) \rceil = 1$ in Equation (2.12). By taking $s = 3$, a new schedule can be derived that can respect the latency constraint introduced by backward edge E_5 to guarantee strict periodicity of the tasks' execution, as shown in Figure 3.3. In this schedule, the tasks are CDP tasks that are represented by the following tuples in task set $\Gamma = \{\tau_1 = (C_1 = 2, T_1 =$

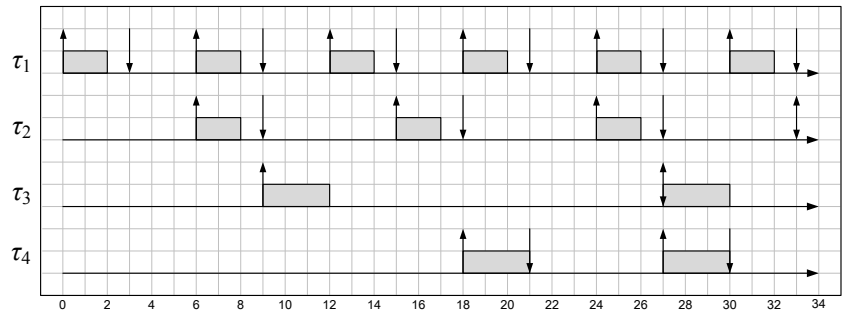


Figure 3.3: *The GSPS of the CSDF graph G in Figure 3.1.*

$6, S_1 = 0, D_1 = 3)$, $\tau_2 = (2, 9, 6, 3)$, $\tau_3 = (3, 18, 9, 18)$, $\tau_4 = (3, 9, 18, 3)\}$. Please note that the deadline of each task is derived with the goal of minimizing the number of required processors to schedule the tasks. The above example shows that the actors in the cyclic CSDF graph G can be converted to a set of CDP tasks, thus, a variety of hard real-time scheduling algorithms [29] can be applied to the cyclic CSDF graph G in order to provide temporal isolation, fast admission control, and easy calculation of the minimum required processors. For instance, for the set Γ of CDP tasks in Figure 3.3, $\delta_\Gamma = 2.5$ and the minimum number of processors for global and partitioned First-Fit Increasing Deadlines EDF (FFID-EDF) [29] schedulers are $\check{m} = 3$ and $\check{m}_{\text{PAR}} = 3$ according to Equation (2.10) and Equation (2.11), respectively. Therefore, the goal of our GSPS framework proposed in Section 3.5 is to test for the existence and to derive such strictly periodic schedule for an application modeled as a **cyclic** CSDF graph which implies that the actors in the graph can be converted to a set of CDP tasks.

3.5 Our Proposed Framework

In this section, we present our analytical GSPS framework for scheduling and converting the actors in a **cyclic** CSDF graph to a set of CDP tasks. First, we test for the existence of a strictly periodic schedule for a **cyclic** (C)SDF graph in Section 3.5.1. Then, if a strictly periodic schedule exists, each actor A_i of the graph is converted to a CDP task τ_i by deriving the period (T_i), deadline (D_i), and earliest start time (S_i) of the task, in Section 3.5.2, such that all data dependencies between the tasks are satisfied with the goal of minimizing the number of required processors to schedule the CDP tasks.

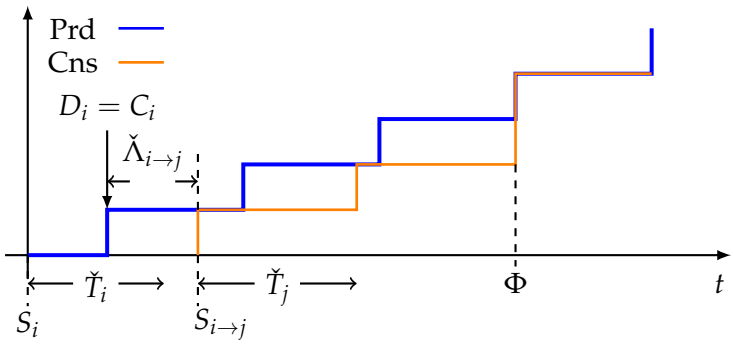


Figure 3.4: *Production and consumption curves on edge $E_u = (A_i, A_j)$.*

3.5.1 Existence of a Strictly Periodic Schedule

As explained in Section 3.4, to find a strictly periodic schedule for a **cyclic** (C)SDF graph, an appropriate scaling factor $s \geq \check{s}$ has to be determined such that all latency constraints introduced by backward edges are satisfied. Therefore, to test for the existence of a strictly periodic schedule, the existence of such scaling factor s must be tested. To do so, we need to analyze the start times of the tasks corresponding to the actors belonging to each cycle in the (C)SDF graph. Using Equation (2.17) and the minimum periods of the tasks ($\check{T}_i$), we can define interval $\check{\Lambda}_{i\to j}$ for each edge $E_u = (A_i, A_j) \in \mathcal{E}$ as follows:

$$\check{\Lambda}_{i\to j} = S_{i\to j} - S_i - D_i \tag{3.1}$$

that is the minimum distance between the deadline (D_i) of task τ_i corresponding to actor A_i and the earliest start time ($S_{i\to j}$) of task τ_j corresponding to actor A_j due to edge E_u. This means that task τ_j cannot start execution earlier than $\check{\Lambda}_{i\to j}$ time units after the deadline of task τ_i, i.e.,

$$S_i + D_i + \check{\Lambda}_{i\to j} \leq S_j. \tag{3.2}$$

Otherwise, task τ_j cannot find enough data tokens on edge E_u to read in order to execute in strictly periodic fashion. The data token production and consumption curves on edge E_u along with the $\check{\Lambda}_{i\to j}$ interval are illustrated in Figure 3.4, when $D_i = C_i$. To execute task τ_j in strictly periodic fashion, the cumulative data token production of task τ_i on channel E_u must always be greater than or equal to the cumulative data token consumption of task τ_j from E_u. This is ensured by shifting the consumption curve by $\check{\Lambda}_{i\to j}$ time units to the right after the deadline of task τ_i, as shown in Figure 3.4. In Figure 3.4,

point Φ is a critical point determining that the consumption curve cannot be shifted to the left because the consumption curve will be above the production curve. Thus task τ_j cannot start execution earlier than $S_{i\to j}$.

To compute $S_{i\to j}$ using Equation (2.17) for edge E_u, S_i must be known. Therefore, to use Equation (2.17) for each edge independently, we assume

$$S_i = \left(\left\lfloor \frac{\gamma}{Y_j^u(q_j)} \right\rfloor + 1 \right) H, \tag{3.3}$$

where γ is the number of initial tokens on channel E_u, $Y_j^u(q_j) = \sum_{l=1}^{q_j} y_j^u(((l-1) \bmod \phi_j)+1)$ is the amount of tokens that task τ_j corresponding to actor A_j consumes from E_u during one graph iteration, $\lfloor \gamma / Y_j^u(q_j) \rfloor$ is the maximum number of graph iterations where task τ_j can execute before starting task τ_i, H is the iteration period. This S_i is sufficiently large to ensure that actual $\check{\Lambda}_{i\to j}$ can be computed. For example, using Equation (3.1), Equation (2.17), and Equation (3.3) for G in Figure 3.1, we have $\check{\Lambda}_{1\to 2} = 1$, $\check{\Lambda}_{1\to 3} = 2$, $\check{\Lambda}_{2\to 4} = 3$, $\check{\Lambda}_{3\to 4} = -3$, and $\check{\Lambda}_{4\to 1} = -7$.

The $\check{\Lambda}_{i\to j}$ interval is the key component in our analysis to find a strictly periodic schedule for the actors in a **cyclic** (C)SDF graph. Since the $\check{\Lambda}_{i\to j}$ interval is calculated using the minimum period computed by Equation (2.12) with scaling factor $s = \check{s}$, we need to find how interval $\check{\Lambda}_{i\to j}$ changes by taking scaling factor $s > \check{s}$. This is provided by the following lemma.

Lemma 3.5.1. *The $\Lambda_{i\to j}$ interval changes proportionally to the scaling factor s as follows:*

$$\Lambda_{i\to j} = \frac{\check{\Lambda}_{i\to j}}{\check{s}} \cdot s \tag{3.4}$$

where $\check{s}$ is the minimum scaling factor computed by Equation (2.13) and $\check{\Lambda}_{i\to j}$ is the minimum interval computed by Equation (3.1).

Proof. Consider an arbitrary edge $E_u = (A_i, A_j) \in \mathcal{E}$ where the data token production and consumption curves can be visualized similarly to Figure 3.4. For the minimum periods ($\check{T}_i$ and $\check{T}_j$) of tasks τ_i and τ_j corresponding to actors A_i and A_j computed using Equation (2.12) with $s = \check{s}$, we assume that the critical point Φ happens after x and y executions of tasks τ_i and τ_j, respectively, e.g., 3 executions of task τ_i and 2 executions of task τ_j in Figure 3.4, that implies

$$S_i + D_i + x \cdot \check{T}_i = S_{i\to j} + y \cdot \check{T}_j \overset{(3.1)}{\iff} x \cdot \check{T}_i = y \cdot \check{T}_j + \check{\Lambda}_{i\to j} \tag{3.5}$$

$$\overset{(2.12)}{\iff} \left(x \cdot \frac{\operatorname{lcm}(\vec{q})}{q_i} - y \cdot \frac{\operatorname{lcm}(\vec{q})}{q_j}\right) = \frac{\check{\Lambda}_{i\to j}}{\check{s}}. \tag{3.6}$$

Now, we assume that after taking scaling factor $s > \check{s}$, a new critical point Φ' exists after x' and y' executions of tasks τ_i and τ_j, respectively. Therefore, we have

$$x' \cdot T_i = y' \cdot T_j + \Lambda_{i\to j} \overset{(2.12)}{\iff} (x' \cdot \frac{\text{lcm}(\vec{q})}{q_i} - y' \cdot \frac{\text{lcm}(\vec{q})}{q_j}) = \frac{\Lambda_{i\to j}}{s}. \tag{3.7}$$

Moreover, for the previous critical point Φ, we know that y executions of task τ_j cannot finish before finishing x executions of task τ_i because the consumption curve cannot be above the production curve. Therefore, after taking scaling factor $s > \check{s}$, we still have

$$x \cdot T_i \leq y \cdot T_j + \Lambda_{i\to j} \overset{(2.12)}{\iff} (x \cdot \frac{\text{lcm}(\vec{q})}{q_i} - y \cdot \frac{\text{lcm}(\vec{q})}{q_j}) \leq \frac{\Lambda_{i\to j}}{s}. \tag{3.8}$$

Then, by substituting Equation (3.6) and Equation (3.7) in Equation (3.8), we have

$$\frac{\check{\Lambda}_{i\to j}}{\check{s}} \leq (x' \cdot \frac{\text{lcm}(\vec{q})}{q_i} - y' \cdot \frac{\text{lcm}(\vec{q})}{q_j}) \overset{(2.12)}{\iff} y' \cdot \check{T}_j + \check{\Lambda}_{i\to j} \leq x' \cdot \check{T}_i. \tag{3.9}$$

However, $y' \cdot \check{T}_j + \check{\Lambda}_{i\to j} < x' \cdot \check{T}_i$ is not possible due to the fact that y' executions of task τ_j cannot finish before finishing x' executions of task τ_i for the critical point Φ' because the consumption curve cannot be above the production curve. Therefore, from Equation (3.9), we can only have

$$y' \cdot \check{T}_j + \check{\Lambda}_{i\to j} = x' \cdot \check{T}_i \overset{(3.5)}{\iff} x' \cdot \check{T}_i - y' \cdot \check{T}_j = x \cdot \check{T}_i - y \cdot \check{T}_j$$

$$\overset{(2.12)}{\iff} (x' \cdot \frac{\text{lcm}(\vec{q})}{q_i} - y' \cdot \frac{\text{lcm}(\vec{q})}{q_j}) = (x \cdot \frac{\text{lcm}(\vec{q})}{q_i} - y \cdot \frac{\text{lcm}(\vec{q})}{q_j}). \tag{3.10}$$

From Equation (3.6), Equation (3.7), and Equation (3.10) we can conclude that

$$\frac{\Lambda_{i\to j}}{s} = \frac{\check{\Lambda}_{i\to j}}{\check{s}} \Leftrightarrow \Lambda_{i\to j} = \frac{\check{\Lambda}_{i\to j}}{\check{s}} \cdot s.$$

■

Now, we propose a sufficient test for the existence of a strictly periodic schedule for a **cyclic** (C)SDF graph by formulating a theorem and prove it by using Lemma 3.5.1.

Theorem 3.5.1. *For the tasks corresponding to actors in a **cyclic** (C)SDF graph G, a strictly periodic schedule exists if for every cyclic path* $\vartheta = \{A_{\vartheta 1} \leftrightarrow A_{\vartheta 2} \leftrightarrow \cdots \leftrightarrow A_{\vartheta x} \leftrightarrow A_{\vartheta 1}\} \in \mathcal{V}$ *in G:*

$$\sum_{i=1}^{x} \check{\Lambda}_{\vartheta i \rightarrow \vartheta((i \bmod x)+1)} < 0. \tag{3.11}$$

where $\mathcal{V}$ *is a set of all cyclic paths in G and* $\check{\Lambda}_{\vartheta i \rightarrow \vartheta((i \bmod x)+1)}$ *is computed using Equation (3.1).*

Proof. In a cyclic path $\vartheta = \{A_{\vartheta 1} \leftrightarrow A_{\vartheta 2} \leftrightarrow \cdots \leftrightarrow A_{\vartheta x} \leftrightarrow A_{\vartheta 1}\} \in \mathcal{V}$ and assuming an arbitrary scaling factor $s_\vartheta \geq \check{s}$, the *earliest start time* $S_{\vartheta x}$ of task $\tau_{\vartheta x}$ corresponding to actor $A_{\vartheta x}$, when $D_i = C_i$, $\forall \tau_i \in \Gamma$, can be computed by considering task $\tau_{\vartheta(x-1)}$ corresponding to actor $A_{\vartheta(x-1)}$, that is a predecessor actor of actor $A_{\vartheta x}$, using Equation (3.2) as follows:

$$S_{\vartheta x} = S_{\vartheta(x-1)} + C_{\vartheta(x-1)} + \Lambda_{\vartheta(x-1) \rightarrow \vartheta x}.$$

Now, by recursively computing $S_{\vartheta(x-1)}$ and substituting it in the above equation, the *earliest start time* $S_{\vartheta x}$ of actor $A_{\vartheta x}$ is:

$$S_{\vartheta x} = S_{\vartheta 1} + \sum_{i=1}^{x-1} C_{\vartheta i} + \sum_{i=1}^{x-1} \Lambda_{\vartheta i \rightarrow \vartheta(i+1)}. \tag{3.12}$$

Due to the edge from actor $A_{\vartheta x}$ to actor $A_{\vartheta 1}$, the *start time* $S_{\vartheta 1}$ of task $\tau_{\vartheta 1}$ corresponding to actor $A_{\vartheta 1}$ is constrained by Equation (3.2) as follows:

$$S_{\vartheta x} + C_{\vartheta x} + \Lambda_{\vartheta x \rightarrow \vartheta 1} \leq S_{\vartheta 1}. \tag{3.13}$$

By using Equation (3.4) (Lemma 3.5.1) and Equation (3.12) in Equation (3.13), we have

$$S_{\vartheta 1} + \sum_{i=1}^{x} C_{\vartheta i} + \frac{s_\vartheta}{\check{s}} \cdot \sum_{i=1}^{x} \check{\Lambda}_{\vartheta i \rightarrow \vartheta((i \bmod x)+1)} \leq S_{\vartheta 1}$$

$$\Leftrightarrow \sum_{i=1}^{x} C_{\vartheta i} + \frac{s_\vartheta}{\check{s}} \cdot \sum_{i=1}^{x} \check{\Lambda}_{\vartheta i \rightarrow \vartheta((i \bmod x)+1)} \leq 0. \tag{3.14}$$

Equation (3.14) holds only if $\sum_{i=1}^{x} \check{\Lambda}_{\vartheta i \rightarrow \vartheta((i \bmod x)+1)} < 0$, because $\sum_{i=1}^{x} C_{\vartheta i}$, $\check{s}$, and s_ϑ are positive numbers by definition and we can always select sufficiently large scaling factor $s_\vartheta \geq \check{s}$. ■

3.5.2 Deriving Period, Earliest Start Time, and Deadline of Tasks

Recall that under our GSPS framework, every actor A_i in a **cyclic** CSDF is converted to a CDP task $\tau_i = (C_i, T_i, S_i, D_i)$. Therefore, in this section, we derive the period, deadline, and earliest start time of each task τ_i corresponding to an actor A_i in a **cyclic** (C)SDF graph scheduled in strictly periodic fashion, if such schedule exists according to Theorem 3.5.1.

(a) Period: Considering Equation (3.14), the minimum scaling factor s_ϑ that satisfies Equation (3.14) is:

$$s_\vartheta = \check{s} \cdot \frac{\sum_{i=1}^{x} C_{\vartheta i}}{-\sum_{i=1}^{x} \check{\Lambda}_{\vartheta i \to \vartheta((i \bmod x)+1)}}.$$

Since there may exist several cyclic paths in the graph, the minimum scaling factor s for the graph that guarantees strictly periodic execution of all tasks corresponding to actors is:

$$s = \left\lceil \check{s} \cdot \max\Big(\max_{\forall\ \vartheta \in \mathcal{V}} \Big(\frac{\sum_{i=1}^{x} C_{\vartheta i}}{-\sum_{i=1}^{x} \check{\Lambda}_{\vartheta i \to \vartheta((i \bmod x)+1)}}\Big), 1\Big) \right\rceil.$$

Then, using Equation (2.12) and the above computed scaling factor s, the periods of the tasks corresponding to actors can be derived.

(b) Deadline: Since the number of processors needed to schedule CDP tasks depends on the total density δ_Γ of the task set Γ [29], our objective to derive the deadline of the tasks corresponding to actors is to minimize δ_Γ in order to minimize the number of processors. Therefore, we formulate our optimization problem as follows:

$$\text{Minimize} \quad \delta_\Gamma = \sum_{\tau_i \in \Gamma} \frac{C_i}{D_i} \tag{3.15a}$$

$$\text{subject to:} \quad S_i + D_i - S_j \leq -\Lambda_{i \to j} \quad \forall E_u = (A_i, A_j) \in \mathcal{E} \tag{3.15b}$$

$$-D_i \leq -C_i,\ D_i \leq T_i \quad \forall \tau_i \in \Gamma \tag{3.15c}$$

where Equation (3.15a) is the objective function and D_i is an optimization variable. In addition, Equations (3.15b) are the constraints given by Equation (3.2), and Equations (3.15c) bound all optimization variables in the objective function by the WCET C_i and period T_i derived in Section 3.5.2(a). S_i and S_j are implicit variables which are not in the objective function Equation (3.15a), but still need to be considered in the optimization procedure.

(c) Earliest Start Time: To derive the earliest start times of the tasks corresponding to actors, we use the derived deadline of the tasks corresponding to

actors in Section 3.5.2(b) in the following optimization problem:

$$\text{Minimize} \quad \sum_{\tau_i \in \Gamma} S_i \tag{3.16a}$$

$$\text{subject to:} \; S_i - S_j \leq -\Lambda_{i \to j} - D_i \quad \forall E_u = (A_i, A_j) \in \mathcal{E} \tag{3.16b}$$

$$-S_i \leq 0 \quad \forall \tau_i \in \Gamma \tag{3.16c}$$

where Equation (3.16a) is the objective function and S_i is an optimization variable. In addition, Equations (3.16b) are the constraints given by Equation (3.2), and Equations (3.16c) bound all optimization variables in the objective function to be greater or equal to zero. Given that all variables in both problems Equations (3.15) and (3.16) are integers and both the objective functions and the constraints are convex, the problems are integer convex programming problems [56]. To solve the problems in Equations (3.15) and (3.16), we used CVX [38,39], a package for specifying and solving convex programs.

3.6 Experimental Evaluation

In this section, we present experiments to evaluate our GSPS framework proposed in Section 3.5. As explained earlier, our GSPS framework enables the application of many hard real-time scheduling algorithms [29], which offer properties such as *hard real-time guarantees, temporal isolation, fast admission control and scheduling decisions for new incoming applications, and easy and fast calculation of the number of processors needed for scheduling the tasks*, on streaming applications modeled as cyclic (C)SDF graphs. However, having these properties is not for free. Thus, the goal of these experiments is to show what the cost is for having these properties using our GSPS framework in terms of the maximum achievable application throughput, the application latency, and the buffer sizes of the communication channels compared to scheduling frameworks, such as periodic scheduling (PS) [18] and self-timed scheduling (STS) [85], which also can be applied directly on cyclic (C)SDF graphs but do not provide such properties. The experiments have been performed on a set of ten real-life streaming applications, modeled as cyclic (C)SDF graphs, taken from different sources. These applications are listed in Table 3.1. In this table, $|\mathcal{A}|$ and $|\mathcal{E}|$ denote the number of actors and communication channels in a (C)SDF graph, respectively.

The results of the evaluation for throughput $\mathcal{R}$ (one token/time units), latency $\mathcal{L}$ (time units), and buffer sizes of the communication channels $\mathcal{M}$ (number of data tokens) of the applications under our GSPS, PS, and STS are

Table 3.1: *Benchmarks used for evaluation.*

Application	$\lvert\mathcal{A}\rvert$	$\lvert\mathcal{E}\rvert$	Source
Modem	16	35	[2]
MP3 playback	4	4	
MP3 Decoder	15	21	[87]
MPEG-4 Advanced Video Coding (AVC) Decoder	4	6	
MPEG-4 Simple Profile (SP) Decoder	5	10	
Channel Equalizer	10	22	
WLAN 802.11p transceiver	8	9	[49]
TDS-CDMA receiver	16	25	[60]
Long Term Evolution (LTE)	10	15	[76]
Echo	38	82	[18]

given in Table 3.2. The throughput, latency, and buffer sizes of the applications under our GSPS, denoted by $\mathcal{R}_{\text{GSPS}}$, $\mathcal{L}_{\text{GSPS}}$, and $\mathcal{M}_{\text{GSPS}}$, are computed using Equations (2.15), (2.19), and (2.18) and given in columns 2, 3, and 4 in Table 3.2, respectively. Columns 7 and 10 show the ratio between the throughput of our GSPS and PS and STS, respectively. Looking at column 7, we can see that our GSPS can achieve the same throughput obtained by PS for 8 out of 10 applications. Looking at column 10, we can also see that the throughput under our GSPS is equal or very close to the throughput under STS, that is the optimal scheduling in terms of throughput, for the majority of the applications. In both comparisons, the largest difference is in the case of Echo. This is mainly because, our GSPS schedules all the phases of an actor in a CSDF graph as jobs of a periodic task, where different job release of the task corresponds to one of the phases of the actor. Therefore, in contrast to PS and STS, the starting time of the execution phases of the task is delayed under our GSPS. As a consequence, if a multi-phase actor exists in a cycle, a larger scaling factor may be required by our GSPS to find a strictly periodic schedule that results in a lower throughput compared to PS and STS. From these comparisons, we can conclude that although our GSPS results in a lower throughput for a few applications compared to PS and STS, achieving the properties of the hard real-time scheduling algorithms is for free in terms of the maximum achievable throughput for the majority of the applications under our GSPS.

For processor requirements under our GSPS, we compute the minimum number of processors under global and partitioned First-Fit Increasing Deadlines EDF (FFID-EDF) [29] schedulers by using Equation (2.10) and Equation (2.11), denoted with $\check{m}$ and $\check{m}_{\text{PAR}}$ in Table 3.2, respectively. However, for PS, the calculation of the number of processors was not considered in [18], and for STS, finding the minimum number of processors requires complex

Table 3.2: *Comparison of different scheduling frameworks.*

Application	GSPS					PS [18]			STS [85]		
	$\mathcal{R}_{\text{GSPS}}[\frac{1}{\text{t.u.}}]$	$\mathcal{L}_{\text{GSPS}}$[t.u.]	$\mathcal{M}_{\text{GSPS}}$[Tkn]	$\hat{m}$	$\hat{m}_{\text{PAR}}$	$\frac{\mathcal{R}_{\text{GSPS}}}{\mathcal{R}_{\text{PS}}}$	$\frac{\mathcal{L}_{\text{GSPS}}}{\mathcal{L}_{\text{PS}}}$	$\frac{\mathcal{M}_{\text{GSPS}}}{\mathcal{M}_{\text{PS}}}$	$\frac{\mathcal{R}_{\text{GSPS}}}{\mathcal{R}_{\text{STS}}}$	$\frac{\mathcal{L}_{\text{GSPS}}}{\mathcal{L}_{\text{STS}}}$	$\frac{\mathcal{M}_{\text{GSPS}}}{\mathcal{M}_{\text{STS}}}$
Modem	1/16	64	50	10	10	1	2.78	1.25	1	2.78	1.25
MPEG-4 AVC	1/7632	15264	6	4	4	1	1.04	1	1	1.04	1
MPEG-4 SP	1/3960	11088	881	2	2	1	2.35	2.02	1	2.35	2.02
MP3 Decoder	1/3732288	33590592	42674	4	4	1	5.46	3.06	1	6.70	-
MP3 playback	1/25	46355	3958	3	4	1	1.12	1.22	0.91	1.30	-
WLAN	1/6	18	14	7	8	1	1.5	1.07	0.92	1.5	0.93
TDS-CDMA	1/675000	792829	44	7	8	1	1.62	1.19	1	1.62	1.19
LTE	1/280	1284	27	5	6	1	2.99	1.28	1	2.99	1.28
Channel Equalizer	1/9264	18989	24	7	7	0.91	1.57	1	0.66	-	1
Echo	1/26882376000	80754156016	30287	13	19	0.19	15.75	1.08	0.19	-	1.08

design space exploration to find the best allocation which delivers the maximum achievable throughput [83]. This fact shows one advantage of using our GSPS compared to using PS and STS when our GSPS gives the same throughput as PS and STS.

Let us now analyze the latency and the buffer sizes of the applications. Columns 8 and 11 give the ratio of the maximum latency of the applications under our GSPS to the latency of the applications under PS and STS, respectively. As we can see, the average latency of the applications under our GSPS is 3.8 and 2.5 times larger than the latency under PS and STS, respectively. Similarly, the ratio of the buffer sizes of the applications under our GSPS to the buffer sizes under PS and STS is given in columns 9 and 12, respectively. From these columns, we can see that the buffer sizes in our GSPS are on average 1.4 and 1.21 times larger than the buffer sizes under PS and STS. Obviously, the larger latency and buffer sizes of the channels for the applications are the main costs in our GSPS framework to enable the utilization of hard real-time scheduling algorithms on streaming applications modeled as cyclic (C)SDF graphs. Please note that, our GSPS causes larger latency and buffer sizes because of the minimization of the number of processors we perform using Equations (3.15), while PS and STS cause lower latency and buffer sizes because they do not perform such minimization. Therefore, if we also do not perform the processor minimization and only perform minimization of the start times of the tasks using Equations (3.16) with $D_i = C_i, \forall \tau_i \in \Gamma$, our GSPS can achieve latency and buffer sizes closer or equal to the latency and buffer sizes of the applications under PS and STS.

3.7 Conclusions

In this chapter, we have presented our GSPS framework to test for the existence of strictly periodic schedule for streaming applications modeled as cyclic CSDF graphs. Then, if such schedule exists, our GSPS converts each task in the graph to a constrained-deadline periodic task. This conversion enables the utilization of many hard real-time scheduling algorithms which offer properties such as temporal isolation and fast calculation of the required number of processors. Finally, we show, on a set of real-life streaming applications, that strictly periodic scheduling is capable of delivering equal or comparable throughput to existing approaches for the majority of the applications we experimented with.

Chapter 4

Exploiting Parallelism in Streaming Applications to Efficiently Utilize Processors

Sobhan Niknam, Peng Wang, Todor Stefanov. "Resource Optimization for Real-Time Streaming Applications using Task Replication". *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 37, No. 11, pp. 2755-2767, Nov 2018.

IN this chapter, we present our novel algorithm to derive an alternative application specification for efficient utilization of processors, which corresponds to the second research contribution, briefly introduced in Section 1.5.2, to address research question **RQ2(A)**, described in Section 1.4.2. The remainder of this chapter is organized as follows. Section 4.1 introduces, in more details, the problem statement and the addressed research question. It is followed by Section 4.2, which gives a summary of the contributions presented in this chapter. Section 4.3 gives an overview of the related work. Section 4.4 introduces the extra background material needed for understanding the contributions of this chapter. Section 4.5 gives a motivational example. Section 4.6 presents our proposed algorithm. Section 4.7 presents the experimental evaluation of our proposed algorithm. Finally, Section 4.8 ends the chapter with conclusions.

4.1 Problem Statement

Recall, from Section 2.2, that in real-time systems, tasks can be scheduled on multiprocessor systems using three main classes of algorithms, i.e., global, partitioned, and hybrid scheduling algorithms, based on whether a task can migrate between processors [29]. Under global scheduling algorithms, all the tasks can migrate between all processors. Such scheduling guarantees optimal utilization of the available processors but at the expense of high scheduling overheads due to extreme task preemptions and migrations. More importantly, implementing global scheduling algorithms in distributed-memory MPSoCs imposes a large memory overhead due to replicating the code of each task on every processor [24]. Under partitioned scheduling algorithms, however, no task migration is allowed and the tasks are allocated statically to the processors, hence they have low run-time overheads. The tasks on each processor are scheduled separately by a uniprocessor (hard) real-time scheduling algorithm, e.g., earliest deadline first (EDF) [54]. The third class of scheduling algorithms is hybrid scheduling that is a mix of global and partitioned approaches to take advantages of both classes. However, since hybrid scheduling algorithms allow task migration, they still introduce additional run-time task migration/preemption overheads and memory overhead on distributed-memory MPSoCs. By performing an extensive empirical comparison of global, clustered (hybrid) and partitioned algorithms for EDF scheduling, Bastoni et al. [14] concluded that the partitioned algorithm outperforms the other algorithms when hard real-time systems are considered.

Although partitioned scheduling algorithms do not impose any migration and memory overheads, they are known to be non-optimal for scheduling real-time periodic tasks [29]. This is because the partitioned scheduling algorithms fragment the processors' computational capacity such that no single processor has sufficient remaining capacity to schedule any other task in spite of the existence of a total large amount of unused capacity on the platform. Therefore, more processors are needed to schedule a set of real-time periodic tasks using partitioned scheduling algorithms compared to optimal (global) scheduling algorithms.

However, for better resource usage and energy efficiency in a real-time embedded system while taking advantages of partitioned scheduling algorithms, the number of processors needed to satisfy a performance requirement, i.e., throughput, in an application should be minimized. This can be difficult because often the given initial application specification, i.e., the initial graph, is not the most suitable one for the given MPSoC platform because the application developers typically focus on realizing certain application behavior

while neglecting the efficient utilization of the available resources on MPSoC platforms. Therefore, to better utilize the resources on an underlying MPSoC platform while using partitioned scheduling algorithms, the initial application specification should be transformed to an alternative one that exposes more parallelism while preserving the same application behavior and performance. This is mainly because by replicating a task of the application, its workload is distributed among more parallel task's replicas in the obtained transformed graph. Therefore, the task's required capacity is split up in multiple smaller chunks that can more likely fit into the left capacity on the processors and alleviate the capacity fragmentation due to partitioned scheduling algorithms. However, having more parallelism, i.e., tasks' replicas, than necessary introduces significant overheads in code and data memory, scheduling and inter-tasks communication. Thus, in this chapter, we investigate the possibility to determine the right amount of parallelism in a streaming application, modeled as an acyclic SDF graph, to minimize the number of required processors under partitioned scheduling algorithms while satisfying a given performance requirement.

4.2 Contributions

In order to address the problem described in Section 4.1, in this chapter, we propose a novel algorithm to find a proper replication factor for each task in an initial application specification, such that the obtained alternative one requires fewer processors under partitioned scheduling algorithms and a given throughput requirement is satisfied. More specifically, the main novel contributions of this chapter are summarized as follows:

- We propose a novel heuristic algorithm to allocate the tasks in a hard real-time streaming application modeled as an acyclic SDF graph, which is subject to a throughput constraint, onto a heterogeneous MPSoC such that the number of required processors is reduced under partitioned scheduling algorithms. The main innovation in this algorithm is that by using the unfolding graph transformation technique in [81], we propose an approach to determine a replication factor for each task of the application such that the distribution of the workloads among more parallel tasks, in the obtained graph after the transformation, results in a better resource utilization, which can alleviate the capacity fragmentation issue introduced by partitioned scheduling algorithms, hence reducing the number of required processors.
- We show, on a set of real-life streaming applications, that our algorithm

significantly reduces the number of required processors compared to the First-Fit Decreasing (FFD) allocation algorithm with slightly increasing the memory requirements and application latency while maintaining the same application throughput. We also show that our algorithm can still reduce the number of required processors compared to the related approaches in [4, 23, 81, 92] with significantly improving the memory requirements and application latency while maintaining the same application throughput.

Scope of work. In this chapter, we consider that streaming applications are modeled as acyclic SDF graphs. This restriction comes from the related approaches that are adopted for comparison with our proposed algorithm. These approaches can only be applied on sets of implicit deadline periodic tasks which can be derived from acyclic SDF graphs using the SPS framework, described in Section 2.3.

4.3 Related Work

In order to overcome the scheduling problems in global and partitioned scheduling algorithms, briefly explained in Section 4.1, a restricted-migration semi-partitioned scheduling algorithm, called EDF-fm, in the class of hybrid scheduling algorithms, is proposed in [4] for homogeneous platforms. In this scheduling algorithm, the tasks can be either fixed or migrating between only two processors at job boundaries. The purpose of this migration is to utilize the remaining capacity on the processors where a migrating task cannot be entirely allocated. However, this scheduler provides hard real-time guarantees only for migrating tasks and soft real-time guarantees for fixed tasks, i.e., fixed tasks can miss their deadlines by a bounded value called tardiness. In [92], another semi-partitioned scheduling algorithm, called EDF-sh, is proposed that, in contrast to EDF-fm, supports heterogeneous platforms and allows the tasks to migrate between more than two processors. In EDF-sh, however, both migrating and fixed tasks may miss their deadlines.

Similarly, [20] proposes the C=D approach to split real-time periodic tasks on homogeneous multiprocessor systems while on each processor a normal EDF scheduler is used. In the C=D approach, a task which cannot be entirely allocated on any processor is split up in two parts that can be entirely allocated on different processors. However, since the task splitting is performed in every job execution, this approach requires transferring the internal state of the splitted tasks between processors at run-time, thereby imposing high task migration overhead. Moreover, these approaches in [4, 20, 92] only consider

sets of independent tasks. In contrast, we consider a more realistic application model which consists of tasks with data dependencies. In addition, we use partitioned scheduling to allocate the tasks statically on the processors. Therefore, since task migration is not allowed in partitioned scheduling, no extra run-time overhead is imposed to the system by our algorithm in comparison to [20] and no task is subjected to a deadline miss in comparison to [4, 92]. Compared to the approaches in [4, 20] that only support homogeneous platforms, our proposed algorithm also supports heterogeneous platforms.

To allocate data-dependent application tasks to a multiprocessor platform, many techniques have already been devised [75]. Existing approaches which are close to our work are [8, 23, 81]. The authors in [8] propose the SPS framework, briefly described in Section 2.3, to only convert each actor in an acyclic (C)SDF graph to an implicit-deadline periodic task by deriving parameters such as period and start time to enable the usage of all well-developed real-time theories. In [8], however, no optimization technique for different system design metrics, such as, throughput, latency, memory, number of processors, etc., is proposed. In contrast, in this chapter, we propose a heuristic algorithm on top of the SPS framework to optimize the number of required processors when scheduling a hard real-time streaming application with a given throughput requirement onto a heterogeneous MPSoC under partitioned scheduling algorithms.

Using the SPS framework, the authors in [23] propose a heuristic under the semi-partitioned scheduling algorithm in [4] to allocate tasks to processors while taking the data dependencies into account. Although the fixed tasks can miss their deadlines in the EDF-fm scheduling approach, a hard real-time property can be guaranteed on the input/output interfaces of the application with the external environment, using the proposed extension of the SPS framework in [23]. In [4], the authors also propose three task-allocation heuristics under EDF-fm to allocate independent tasks to processors in which the one called fm-LUF requires the least number of processors. In a similar way, this heuristic can be used while taking data dependencies into account using the approach presented in [23]. However, in these approaches [4, 23], the deadline misses of the fixed tasks due to task migration have significant overheads on the memory requirements and the application latency. In contrast, we provide hard real-time guarantees for all tasks in an application modeled as an SDF graph. Moreover, we use partitioned scheduling and to utilize processors efficiently, we adopt the unfolding graph transformation technique. By using our proposed algorithm, as shown in Section 4.7, processors can be more efficiently utilized while imposing considerably lower overheads on the memory

requirements and the application latency compared to the approaches in [4,23]. In addition, our proposed algorithm supports heterogeneous platforms while the approaches in [4,23] can only support homogeneous platforms.

In [81], the authors propose an approach to increase the application throughput in a homogeneous platform with a fixed number of processors. This approach considers partitioned scheduling and exploits an unfolding transformation technique to fully utilize the platform by replicating the bottleneck tasks which are the ones with the maximum workload, i.e., highest utilization, when mapping a streaming application modeled as an SDF. However, to satisfy a given throughput requirement under limited resources, the approach in [81] does not always replicate the right tasks, as shown in Section 4.5. Consequently, this leads to more parallelism than needed which increases the memory requirements and application latency unnecessarily. In contrast, we propose an algorithm that supports heterogeneous platforms. In addition, our proposed algorithm first detects which tasks cause the capacity fragmentation in partitioned scheduling on the processors. Note that these tasks are not the bottleneck tasks identified and used in [81]. This is because, the bottleneck tasks efficiently utilize the processors' capacity and there is no need to replicate them. Then, using the unfolding transformation technique, we replicate the detected tasks causing the capacity fragmentation to distribute their workloads among more parallel tasks and utilize the platform more efficiently with less unused capacity on the processors. As a result, shown in Section 4.7, our proposed algorithm can reduce the number of required processors to guarantee the same throughput while keeping a low memory and latency overheads under partitioned scheduling in comparison to [81].

In [80], the authors use the same approach as in [81] for energy efficiency purpose under partitioned scheduling algorithms, when there are a lot of processors available on a cluster heterogeneous MPSoC. To reduce energy consumption, they iteratively take the bottleneck tasks which are limiting the processors to work at a lower frequency and replicate them. By replicating the application tasks with heavy utilization, their utilization is distributed among more task's replicas while still providing the same application performance. Consequently, the workload distribution of these bottleneck tasks enables the processors to work at a lower frequency, thereby reducing the energy consumption. In this chapter, however, we focus on and solve a totally different problem, that is, how the unfolding transformation technique can be exploited to reduce the number of required processors when a partitioned scheduling algorithm is used. In our algorithm, we do not search for and take the bottleneck task, which is taken in [80], for replication in every iteration.

In contrast, we detect which task is responsible for fragmentation of the processors' capacity when using a partitioned scheduling algorithm and try to resolve this fragmentation by replicating this task such that the number of processors is reduced. We do not replicate the bottleneck task because it can efficiently utilize the processor and it does not contribute to the fragmentation of the processors' capacity.

4.4 Background

In this section, we first introduce the unfolding transformation technique, presented in [81], that we use to replicate the tasks in an application initially modeled as an SDF graph. Then, we present the system model considered in this chapter.

4.4.1 Unfolding Transformation of SDF Graphs

The authors in [81] have shown that an SDF graph can be transformed into an equivalent CSDF graph by using a graph unfolding transformation technique to better utilize the underlying MPSoC platform by exposing more parallelism in the SDF graph. In fact, the intuition behind the unfolding, i.e., replication, of an actor in the initial SDF graph is to evenly distribute the workload of the actor among multiple of its replicas that are running concurrently. Given a vector $\vec{f} \in \mathbb{N}^{|\mathcal{A}|}$ of replication factors, where f_i denotes the replication factor for actor $A_i \in \mathcal{A}$, the unfolding transformation replaces actor A_i with f_i replicas of actor A_i, denoted by $A_{i,k}, k \in [1, f_i]$. To ensure the functional equivalence, the production and consumption sequences on FIFO channels in the obtained CSDF graph are calculated accordingly to the production and consumption rates in the initial SDF graph. After the replication, each replica $A_{i,k}$ of actor A_i will have the repetition

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

where $\text{lcm}(\vec{f})$ is the least common multiple of all replication factors in $\vec{f}$. For example, consider the SDF graph G shown in Figure 4.1 with the repetition vector $\vec{q} = [2, 1, 1, 1, 1, 2]^T$, derived using Theorem 2.1.1. After unfolding of G with replication vector $\vec{f} = [1, 1, 1, 1, 2, 1]$, the CSDF graph G' shown in

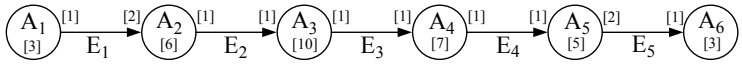
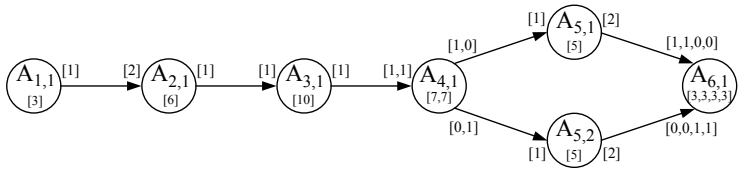
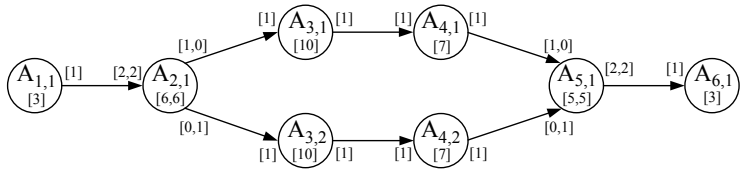


Figure 4.1: *An SDF graph G.*



(a) A CSDF graph G'



(b) A CSDF graph G''

Figure 4.2: *Equivalent CSDF graphs of the SDF graph G in Figure 4.1 obtained by (a) replicating actor A_5 by factor 2 and (b) replicating actors A_3 and A_4 by factor 2.*

Figure 4.2(a) is obtained which has the repetition vector $\vec{q}' = [4,2,2,2,1,1,4]^T$, e.g.,

$$q_{5,1} = q_{5,2} = \frac{1 \cdot \text{lcm}(1,1,1,1,2,1)}{2} = 1.$$

4.4.2 System Model

The considered MPSoC platforms in this chapter are heterogeneous containing two types of processors [1], i.e., performance-efficient (PE) and energy-efficient (EE) processors, with distributed memories. We use Π_{PE} and Π_{EE} to denote the sets containing the PE processors and the EE processors, respectively. We denote the heterogeneous MPSoCs containing all PE and EE processors by $\Pi = \{\Pi_{PE}, \Pi_{EE}\}$. Since application tasks may run on two different types of processors (PE and EE), the worst-case execution time value C_i for each periodic task $\tau_i \in \Gamma$ has two values, i.e., C_i^{PE} and C_i^{EE}, when EE and PE processors run at their maximum operating clock frequencies supported by

[1] We refer to the ARM big.LITTLE architecture [40] including Cortex A15 'big' (PE) and Cortex A7 'LITTLE' (EE) that is shown in Figure 1.1.

the hardware platform. The utilization of task τ_i on a PE processor and an EE processor, denoted as u_i^{PE} and u_i^{EE}, is defined as $u_i^{PE} = C_i^{PE}/T_i$ and $u_i^{EE} = C_i^{EE}/T_i$, respectively. Now, let us consider an x-partition ${}^x\Gamma$ of task set Γ. Then, the total utilizations of the tasks allocated on a PE processor j and an EE processor k can be calculated by:

$$u_{\pi_j^{PE}} = \sum_{\tau_i \in {}^x\Gamma_j} \frac{C_i^{PE}}{T_i}, \quad u_{\pi_k^{EE}} = \sum_{\tau_i \in {}^x\Gamma_k} \frac{C_i^{EE}}{T_i} \tag{4.2}$$

where ${}^x\Gamma_j$ and ${}^x\Gamma_k \in {}^x\Gamma$ represent sets of tasks allocated on PE processor j and EE processor k, respectively.

4.5 Motivational Example

In this section, we take the SDF graph G shown in Figure 4.1 as our motivational example to demonstrate the necessity and efficiency of our proposed algorithm, presented in Section 4.6, compared to the related approaches [81], [23], [4], and [92] in terms of memory requirements, application latency, and number of required processors on a homogeneous platform[2], i.e., including only PE processors, to schedule the actors in the SDF graph under a given throughput requirement. By applying the SPS framework [8], briefly described in Section 2.3, for graph G, the task set $\Gamma = \{\tau_1 = (C_1 = 3, T_1 = 5, S_1 = 0, D_1 = T_1 = 5), \tau_2 = (6, 10, 10, 10), \tau_3 = (10, 10, 20, 10), \tau_4 = (7, 10, 30, 10), \tau_5 = (5, 10, 40, 10), \tau_6 = (3, 5, 50, 5)\}$ of six IDP tasks can be derived. Based on these tuples, a strictly periodic schedule, as shown in Figure 4.3(a), can be obtained for this graph. Using Equation (2.15), the throughput of this schedule can be computed as $\mathcal{R} = \frac{1}{T_6} = \frac{1}{5}$. In this example, we consider this throughput as the given throughput requirement. Moreover, using Equation (2.19), the application latency $\mathcal{L}$ for this schedule is 55 which is the elapsed time between the arrival of the first sample to the application, at $t = 0$, and the departure of the processed sample from task τ_6, at $t = 55$. The minimum number of processors needed for this schedule using an optimal scheduling algorithm, according to Equation (2.8), is $\check{m}_{\text{OPT}} = \left\lceil \sum_{\tau_i \in \Gamma} u_i \right\rceil = \left\lceil \frac{3}{5} + \frac{6}{10} + \frac{10}{10} + \frac{7}{10} + \frac{5}{10} + \frac{3}{5} \right\rceil = 4$. However, using the partitioned EDF and the First-Fit Decreasing (Utilization) [28] allocation algorithm, that is proven to be the resource efficient heuristic allocation algorithm [5], 6 processors are required for this schedule with task

[2] In this section, we adopt a homogeneous platform because the related approaches [4, 23, 81] can support only such platform. Later, in Section 4.7.2, we compare our proposed approach and the approach proposed in [92] in terms of memory requirements and application latency on different heterogeneous platforms for a set of real-life applications.

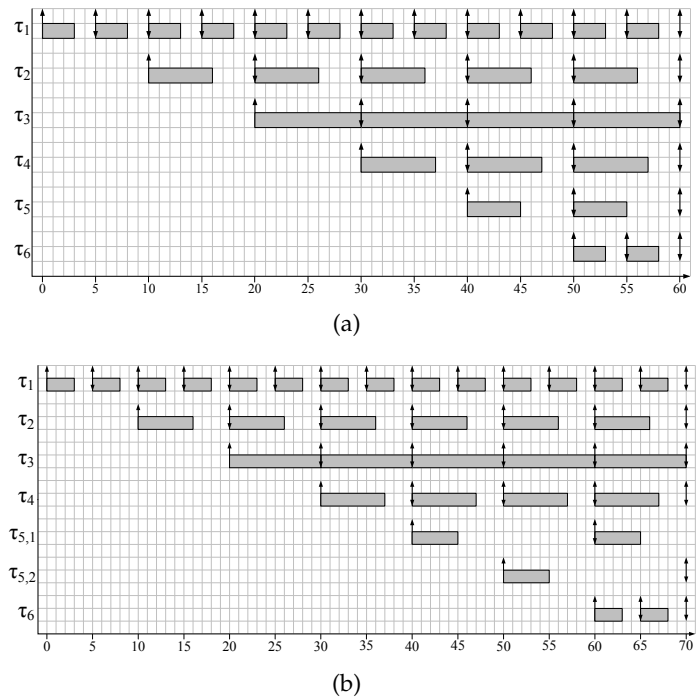


Figure 4.3: *A strictly periodic execution of tasks corresponding to the actors in: (a) the SDF graph G in Figure 4.1 and (b) the CSDF graph G′ in Figure 4.2(a). The x-axis represents the time.*

allocation ${}^6\Gamma = \{{}^6\Gamma_1 = \{\tau_3\}, {}^6\Gamma_2 = \{\tau_4\}, {}^6\Gamma_3 = \{\tau_1\}, {}^6\Gamma_4 = \{\tau_2\}, {}^6\Gamma_5 = \{\tau_6\}, {}^6\Gamma_6 = \{\tau_5\}\}$. We refer to this scheduler as partitioned First-Fit Decreasing EDF (FFD-EDF) scheduler.

To reduce the number of required processors under the FFD-EDF scheduler while satisfying the given throughput requirement $\mathcal{R} = \frac{1}{5}$, we adopt the unfolding graph transformation technique in [81], briefly explained in Section 4.4.1. Let us assume that the platform has only 5 processors. Then, to schedule the application on 5 processors under FFD-EDF scheduler, our proposed algorithm, explained in Section 4.6, replicates actor A_5 in graph G by a factor of 2. Figure 4.2(a) shows the CSDF graph G' obtained after applying the unfolding transformation on the initial graph G shown in Figure 4.1. By applying the SPS framework for graph G', the task set $\Gamma' = \{\tau_{1,1} = (3, 5, 0, 5), \tau_{2,1} = (6, 10, 10, 10), \tau_{3,1} = (10, 10, 20, 10), \tau_{4,1} = (7, 10, 30, 10), \tau_{5,1} = (5, 20, 40, 20), \tau_{5,2} = (5, 20, 50, 20), \tau_{6,1} = (3, 5, 60, 5)\}$ of seven IDP tasks can be derived which is schedulable on 5 processors under FFD-EDF scheduler, with task

allocation ${}^5\Gamma' = \{{}^5\Gamma'_1 = \{\tau_{3,1}\}, {}^5\Gamma'_2 = \{\tau_{4,1}, \tau_{5,1}\}, {}^5\Gamma'_3 = \{\tau_{1,1}, \tau_{5,2}\}, {}^5\Gamma'_4 = \{\tau_{2,1}\}, {}^5\Gamma'_5 = \{\tau_{6,1}\}\}$, while satisfying the given throughput requirement of $\frac{1}{5}$. This is because, the workload of task τ_5, corresponding to actor A_5 of graph G, with $u_5 = \frac{5}{10}$ is now evenly distributed between two tasks $\tau_{5,1}$ and $\tau_{5,2}$, corresponding to replicas $A_{5,1}$ and $A_{5,2}$ of actor A_5, i.e., $u_{5,1} = u_{5,2} = \frac{5}{20}$. Apparently, this workload distribution using the unfolding transformation can enable the FFD-EDF scheduler to more efficiently utilize the processors and schedule the tasks on fewer processors while satisfying the throughput requirement. The strictly periodic schedule of the task set Γ' is shown in Figure 4.3(b).

The approach in [81] is very close to our approach as it adopts the unfolding transformation technique to increase the throughput of an SDF graph scheduled on an MPSoC with fixed number of processors under partitioned scheduling. However, to schedule Γ on a platform with 5 processors under the throughput requirement of $\frac{1}{5}$, the approach in [81] performs differently. It first scales the period of the tasks in Γ using Equation (2.13) to make Γ schedulable on 5 processors under FFD-EDF scheduler. Due to scaling the periods, i.e., $s = 6 > \lceil \frac{10}{2} \rceil = 5$, however, the throughput is dropped to $\frac{1}{6}$. Then, to increase the throughput, the approach in [81] replicates the actor corresponding to the bottleneck task, i.e., the actor with the heaviest workload during one graph iteration, and scales again the minimum computed periods of the tasks such that the new task set can be scheduled on 5 processors under FFD-EDF scheduler. This procedure is repeated until no throughput improvement can be gained anymore by task replication under the resource constraint. For our example in Figure 4.1, the approach in [81] replicates actors A_3 and A_4 corresponding to tasks τ_3 and τ_4 by a factor of 2 that results in the throughput of $\frac{1}{3}$. Figure 4.2(b) shows the CSDF graph G'' obtained after applying the unfolding transformation on graph G. Then, to schedule the tasks on 5 processors under FFD-EDF scheduler, the periods of tasks are scaled by using Equation (2.13), i.e., $s = 5 > \lceil \frac{12}{4} \rceil = 3$, where the throughput of $\frac{1}{5}$ finally could be achieved with the derived task set $\Gamma'' = \{\tau_{1,1} = (3, 5, 0, 5), \tau_{2,1} = (6, 10, 10, 10), \tau_{3,1} = (10, 20, 20, 20), \tau_{3,2} = (10, 20, 30, 20), \tau_{4,1} = (7, 20, 40, 20), \tau_{4,2} = (7, 20, 50, 20), \tau_{5,1} = (5, 10, 60, 10), \tau_{6,1} = (3, 5, 70, 5)\}$ of eight IDP tasks and the task allocation ${}^5\Gamma'' = \{{}^5\Gamma''_1 = \{\tau_{4,1}, \tau_{1,1}\}, {}^5\Gamma''_2 = \{\tau_{4,2}, \tau_{2,1}\}, {}^5\Gamma''_3 = \{\tau_{6,1}\}, {}^5\Gamma''_4 = \{\tau_{3,1}, \tau_{3,2}\}, {}^5\Gamma''_5 = \{\tau_{5,1}\}\}$.

The approaches in [4,23], adopt differently the semi-partitioned scheduling EDF-fm to allow certain tasks to migrate between processors for efficiently utilizing the remaining capacity on the processors. Under EDF-fm scheduling, the LUF heuristic in [4] allocates the tasks in Γ to 5 processors with task

allocation ${}^5\Gamma_= \{{}^5\Gamma_1 = \{\tau_3\}, {}^5\Gamma_2 = \{\tau_4, \tau_5\}, {}^5\Gamma_3 = \{\tau_5, \tau_1\}, {}^5\Gamma_4 = \{\tau_6, \tau_2\}, {}^5\Gamma_5 = \{\tau_2\}\}$, where task τ_5 is allowed to migrate between π_2 and π_3 and task τ_2 is allowed to migrate between π_4 and π_5. In this task allocation, however, the fixed tasks τ_1, τ_4, and τ_6 that are allocated to the same processors as the migrating tasks τ_2 and τ_5, can miss their deadline by a bounded tardiness. To reduce the number of affected tasks by tardiness, the FFD-SP heuristic is proposed in [23] to restrict the task migrations. Under EDF-fm scheduling, this approach allocates the tasks in Γ to 5 processors with task allocation ${}^5\Gamma = \{{}^5\Gamma_1 = \{\tau_3\}, {}^5\Gamma_2 = \{\tau_4, \tau_5\}, {}^5\Gamma_3 = \{\tau_5, \tau_1\}, {}^5\Gamma_4 = \{\tau_6\}, {}^5\Gamma_5 = \{\tau_2\}\}$, where only task τ_5 is allowed to migrate between π_2 and π_3. Similar to the approach in [23], EDF-sh [92] allocates the tasks in Γ to 5 processors with task allocation ${}^5\Gamma = \{{}^5\Gamma_1 = \{\tau_3\}, {}^5\Gamma_2 = \{\tau_4, \tau_5\}, {}^5\Gamma_3 = \{\tau_5, \tau_1\}, {}^5\Gamma_4 = \{\tau_6\}, {}^5\Gamma_5 = \{\tau_2\}\}$, where only task τ_5 is allowed to migrate between π_2 and π_3.

The reduction on the number of required processors using our proposed algorithm and the related approaches, however, comes at the expense of more memory requirements and longer application latency either because of task replication[3], i.e., more tasks and data communication channels, or task migration, i.e., task tardiness. The throughput $\mathcal{R}$, latency $\mathcal{L}$, memory requirements $\mathcal{M}$, i.e., the sum of the buffer sizes of the communication channels in the graph and the code size of the tasks, and the number of required processors m for different scheduling/allocation approaches are given in Table 4.1. Table 4.1 clearly shows that our proposed algorithm can reduce the number of required processors while keeping a low memory and latency increase compared to the related approaches for the same throughput requirement.

Let us now assume that the platform has only 4 processors. Then, all the related approaches, except EDF-sh, fail to satisfy the throughput requirement of $\frac{1}{5}$ under this resource constraint. However, our approach finds a vector of replication factors $\vec{f} = [1, 2, 1, 1, 5, 1]$ such that the CSDF graph obtained after applying the unfolding transformation on the initial SDF graph G, is schedulable on 4 processors under FFD-EDF scheduler using the SPS framework while satisfying the throughput requirement of $\frac{1}{5}$. EDF-sh can also allocate the tasks in Γ to 4 processors with task allocation ${}^4\Gamma = \{{}^4\Gamma_1 = \{\tau_3\}, {}^4\Gamma_2 = \{\tau_4, \tau_2\}, {}^4\Gamma_3 = \{\tau_2, \tau_5, \tau_1\}, {}^4\Gamma_4 = \{\tau_5, \tau_6\}\}$, where task τ_2 is allowed to migrate between π_2 and π_3 and task τ_5 is allowed to migrate between π_3 and π_4. The memory requirement and application latency to schedule G on 4 processors

[3]When replicating an actor, the period of the task corresponding to the actor is enlarged. As a consequence, the production of data tokens that are required by its data-dependent tasks to execute are postponed which results in a further offsetting of their start time, when calculating the earliest start time of tasks in the SPS framework using Equation (2.16), hence increasing the application latency.

Table 4.1: *Throughput $\mathcal{R}$ (1/time units), latency $\mathcal{L}$ (time units), memory requirements $\mathcal{M}$ (bytes), and number of processors m for G under different scheduling/allocation approaches.*

Scheduling	Allocation	$\mathcal{R}$ [$\frac{1}{\text{t.u}}$]	$\mathcal{L}$ [t.u]	$\mathcal{M}$ [B]	$\check{\text{m}}$	$\check{\text{m}}_{\text{OPT}}$
EDF	FFD	1/5	55	155	6	4
	our	1/5	65 **(105)**	189 **(327)**	5 **(4)**	4
	FFD-EP [81]	1/5	75	228	5	4
EDF-*fm*	FFD-SP [23]	1/5	90	197	5	4
	LUF [4]	1/5	94	217	5	4
EDF-sh [92]		1/5	113 **(192)**	217 **(311)**	5 **(4)**	4

using our proposed algorithm and EDF-sh are given in the third and seventh rows of Table 4.1 in parenthesis. As a result, our proposed algorithm can decrease the application latency by 45.3% while increasing the memory requirement by only 4.9% compared to EDF-sh.

From the above example, we can see the deficiencies of the related approaches because they have significant impact on the memory requirements and application latency when reducing the number of processors. Oppositely, our proposed algorithm which adopts the graph unfolding transformation, can reduce the number of processors while introducing lower memory and latency increase compared to the related approaches for the same throughput requirement.

4.6 Proposed Algorithm

As explained and shown in Section 4.5, the partitioned scheduling algorithms, potentially, have the disadvantage that processors cannot be fully utilized, i.e., capacity fragmentation, because the static allocation of tasks on processors leaves an amount of unused capacity which is not sufficient to accommodate another task. Therefore, in this section, we present our novel algorithm that aims to exploit these unused capacity on the processors to reduce the number of processors needed to schedule the tasks in a hard real-time streaming application, modeled as an acyclic SDF graph and subjected to a throughput constraint, onto a heterogeneous MPSoC under partitioned scheduling algorithms, e.g., FFD-EDF scheduler. Our propose algorithm can achieve this goal by replicating tasks such that the required capacity of each resulting task replica is sufficiently small to make use of the available capacity on the processors.

The rationale behind our algorithm is the following: our algorithm first

detects every task which cannot be entirely allocated to any individual under-utilized processor due to insufficient free capacity while, in total, there exists sufficient remaining capacity on under-utilized processors to schedule the tasks. Then, our algorithm replicates some of these tasks to distribute their workloads equally among more parallel replicas and fit them entirely on the remaining capacity of the processors without increasing the number of processors. As a result, our algorithm can alleviate the capacity fragmentation due to the FFD-EDF scheduler and utilize the processors more efficiently. In this section, therefore, we present a novel heuristic algorithm to derive the proper replication factor for each actor in an SDF graph and the task allocation to reduce the number of required processors while satisfying a given throughput requirement.

The algorithm is given in Algorithm 1. It takes as input an SDF graph G, and a heterogeneous platform $\Pi = \{\Pi_{PE}, \Pi_{EE}\}$ with fixed number of PE and EE processors onto which the actors in the graph have to be allocated. The algorithm returns as output a CSDF graph G', that is functionally equivalent to the initial SDF graph, and a task allocation set ${}^{x}\Gamma$ if a successful allocation, i.e., $x \leq |\Pi|$, is found. Otherwise, it returns false as output.

In Line 1, the algorithm initializes the replication factor of all actors in graph G to 1, G' to G, and Π' to Π. In Line 2, the actors in the graph G' are converted to periodic tasks using the SPS framework, explained in Section 2.3, where the minimum period T'_i of each task $\tau'_{i,k}$ corresponding to actor $A_{i,k}$ in G' is calculated for PE type of processors, i.e., using C_i^{PE}, by Equation (2.12) and Equation (2.13). *In this chapter, we take the maximum throughput of graph G, achievable by the SPS framework with the minimum calculated periods, as the throughput requirement. Note that we can set another throughput requirement by scaling the minimum calculated periods.* Then, the algorithm builds a set of periodic tasks Γ in Line 3 and sorts the tasks in the order of decreasing utilization. Next, the algorithm enters to a **while loop**, Lines 4 to 37, where the task allocation is started on platform Π'. The body of the **while loop**, then, is repetitively executed to better utilize the processors' capacity using the graph unfolding transformation, explained in Section 4.4.1, and allocate the tasks on platform Π'.

In Line 5, a task allocation set ${}^{|\Pi'|}\Gamma$ is created, to keep the tasks allocated to each processor individually. *Please note that in sets Π' and ${}^{|\Pi'|}\Gamma$, the processors are ordered according to their type, where EE processors are followed by PE processors, to first utilize the energy-efficient processors.* In Line 5, an empty task set Γ_1 is also defined to keep the candidate tasks for replication. In Lines 6 to 23, the algorithm allocates every task $\tau'_{i,k} \in \Gamma$ to one of the processors according

Algorithm 1: Proposed task allocation and finding proper replication factors for an SDF graph.

Input: An SDF graph $G = (\mathcal{A}, \mathcal{E})$ and a heterogeneous MPSoC $\Pi = \{\Pi_{PE}, \Pi_{EE}\}$.
Output: *True*, an equivalent CSDF graph $G' = (\mathcal{A}', \mathcal{E}')$, and a task allocation set ${}^{x}\Gamma$ if a successful task allocation onto platform Π is found, *False* otherwise.

$\vec{f} = [1, 1, \cdots, 1]$; $G' \leftarrow G$; $\Pi' \leftarrow \Pi$;
Calculate period T'_i for PE type of processors for each task $\tau'_{i,k}$ corresponding to actor $A_{i,k}$ in G' by using Equation (2.12) and Equation (2.13);
$\Gamma \leftarrow$ Sort tasks corresponding to actors in G' in order of decreasing utilization;
while *True* **do**
  ${}^{|\Pi'|}\Gamma \leftarrow \{{}^{|\Pi'|}\Gamma_1, {}^{|\Pi'|}\Gamma_2, \cdots, {}^{|\Pi'|}\Gamma_{|\Pi'|}\}$; $\Gamma_1 \leftarrow \emptyset$;
  for $\tau'_{i,k} \in \Gamma$ **do**
    for $1 \leq j \leq |\Pi'|$ **do**
      if π_j *is an* EE *processor* **then**
        $u_{left} = \sum_{\ell=1}^{j-1} (1 - u_{\pi_\ell^{EE}})$; $u_i = u_i^{EE}$;
      if π_j *is a* PE *processor* **then**
        $u_{left} = \frac{C_i^{\text{PE}}}{C_i^{\text{EE}}} \sum_{\ell=1}^{|\Pi_{\text{EE}}|} (1 - u_{\pi_\ell^{EE}}) + \sum_{\ell=|\Pi_{\text{EE}}|+1}^{j-1} (1 - u_{\pi_\ell^{PE}})$; $u_i = u_i^{PE}$;
      Check EDF schedulability test on π_j;
      if *task* $\tau'_{i,k}$ *is not schedulable on* π_j **then continue**;
      else
        if $u_{\pi_j} = 0 \wedge u_{left} \geq u_i$ **then**
          if *actor* $A_{i,k}$ *corresponding to task* $\tau'_{i,k}$ *is not stateful/in/out* **then**
            $\Gamma_1 \leftarrow \Gamma_1 + \{\tau'_{i,k}, \pi_j\}$;
        ${}^{|\Pi'|}\Gamma_j \leftarrow \tau'_{i,k}$;
        break;
    if *task* $\tau'_{i,k}$ *is not allocated* **then**
      if $u_i > u_{left}$ **then return** False;
      $\Pi' \leftarrow \Pi' + \pi^{PE}$;
      go to 5
  for $|\Pi_{EE}| < j \leq |\Pi'|$ **do**
    if ${}^{|\Pi'|}\Gamma_j = \emptyset$ **then**
      $\Pi' \leftarrow \Pi' - \pi_j^{PE}$;
  if $|\Pi'_{PE}| \leq |\Pi_{PE}|$ **then break**;
  if $\Gamma_1 \neq \emptyset$ **then**
    $u_{left} = 0$;
    for $\{\tau'_{i,k}, \pi_j\} \in \Gamma_1$ **do**
      if $1 - u_{\pi_j} > u_{left}$ **then**
        $u_{left} = 1 - u_{\pi_j}$; $sel = i$;
  else return False;
  $f_{sel} = f_{sel} + 1$; $f_{sel} \in \vec{f}$;
  Get CSDF graph $G' = (\mathcal{A}', \mathcal{E}')$ by unfolding G with replication factors $\vec{f}$ using the method in Section 4.4.1;
  Calculate period T'_i for PE type of processors for each task $\tau'_{i,k}$ corresponding to actor $A_{i,k}$ in G' by using Equation (2.12) and Equation (2.13);
  $\Gamma \leftarrow$ Sort tasks corresponding to actors in G' in order of decreasing utilization;
return True, G', ${}^{|\Pi'|}\Gamma$;

to the FFD-EDF scheduler. In Lines 8 to 11, the total unused capacity u_{left} from the first processor π_1 to the current processor π_j is calculated. The current processor π_j can be either an EE processor or a PE processor. If it is an EE processor, all the previous processors are also EE processors due to the ordering of processors based on their type in platform Π'. In this case, the total unused capacity is calculated in Line 9 and stored in variable u_{left}. Otherwise, if π_j is a PE processor, the total unused capacity from π_1 to the current processor π_j, that includes all the EE processors followed by a subset of PE processors, is calculated in Line 11 and stored in variable u_{left}. Since the tasks have different utilization on the PE and EE processors, the total unused capacity on the EE processors are scaled accordingly by the proportion of the worst-case execution time of task $\tau'_{i,k}$ on the PE processor and EE processor, in Line 11.

In Line 12, the EDF schedulability test [54] is performed to check the schedulability of task $\tau'_{i,k}$ on processor π_j, i.e., $\tau'_{i,k}$ is schedulable if the total utilization of all tasks currently allocated to processor π_j (including $\tau'_{i,k}$) is not greater than the utilization bound of 1. If task $\tau'_{i,k}$ is not schedulable on processor π_j, the procedure of visiting the next processors is continued in Line 13. Otherwise, the candidate tasks for replication are identified first in Lines 15 to 17. If task $\tau'_{i,k}$ is allocated to an unused processor π_j while there is, in total, a sufficient unused capacity on the other under-utilized processors, the task is selected as a candidate to be replicated. This condition is checked in Line 15. *Note that stateful tasks, whose next execution depends on the current execution, and input and output tasks, which are connected to the external environment, are not replicated*. So, if task $\tau'_{i,k}$ satisfies the condition in Line 16, it is added in Line 17 to task set Γ_1 together with the processor π_j which it will be allocated to. Task $\tau'_{i,k}$ is actually allocated on processor π_j in Line 18 and the procedure of vising the next processors is terminated in Line 19.

If task $\tau'_{i,k}$ is not allocated after visiting all processors in platform Π' and if the utilization of the task is larger than the total unused capacity left on the platform, then the algorithm cannot allocate the application tasks onto the given platform and returns False in Line 21. Otherwise, a PE processor is added to platform Π' in Line 22. This is because to reasonably find all candidate tasks for replication, the algorithm first checks how the processors are finally utilized by continuing the task mapping through adding an extra processor and finding a valid tasks' allocation using the FFD-EDF scheduler. For instance, the capacity of a processor that is fragmented by a big task can be efficiently exploited later by smaller tasks. Therefore there is no need to replicate such a big task. Later, by iteratively replicating the selected tasks,

the algorithm gradually exploits the processors' capacity more efficiently and removes the extra added PE processors to finally find a valid tasks' allocation on the given platform Π. Next, the procedure is moved to Line 5 to find new tasks' allocation on the new platform Π'.

In Lines 24 to 26, the reduction of the number of required processors is performed by removing PE processors. If a PE processor with no allocated tasks is found, it means the task set Γ requires one PE processor fewer to be scheduled under FFD-EDF scheduler. Therefore, the PE processor with no allocated tasks is removed from platform Π' in Line 26. Then, Line 27 checks whether the number of PE processors in platform Π' is fewer than or equal to the number of PE processors in the given platform Π (*Note that both platforms* Π' *and* Π *have an equal number of EE processors as the algorithm only adds/removes PE processor to/from platform* Π'). If yes, then the CSDF graph G' and the task allocation set Γ_Π are returned in Line 38 and the algorithm terminates successfully.

If not, to better utilize the processors, a task is selected among the candidate tasks in Γ_1 for replication, in Lines 28 to 32. If task set Γ_1 is empty then no task could be selected for replication, therefore the algorithm cannot allocate the application tasks onto platform Π and returns False as output in Line 33. Among all the candidates in task set Γ_1, the task allocated to a processor with the largest amount of unused capacity is identified as a fragmentation-responsible task, in Lines 31 and 32. Then, the replication factor of the actor corresponding to this task in the initial SDF graph is increased by one in Line 34 and the initial SDF graph is transformed into an equivalent CSDF graph using the unfolding transformation technique with unfolding vector $\vec{f}$, in Line 35. The periods of the tasks corresponding to actors in the obtained CSDF graph are calculated again for PE type of processors using Equation (2.12) and Equation (2.13) in Line 36 and the new periodic tasks are sorted in Γ in the order of decreasing utilization, in Line 37. The body of the **while loop**, then, is repeated to either find successfully a task allocation of the transformed graph onto platform Π or fail due to lack of candidate tasks for replication, i.e., empty task set Γ_1.

4.7 Experimental Evaluation

In this section, we present the experiments to evaluate our proposed algorithm in Section 4.6. The experiments have been performed on a set of seven real-life streaming applications modeled as acyclic SDF graphs taken from [23]. These applications, from different application domains, are listed in Table 4.2. In this

Table 4.2: *Benchmarks used for evaluation taken from [23].*

Domain	Application	$\|\mathcal{A}\|$	$\|\mathcal{E}\|$
Signal Processing	Fast Fourier transform (FFT) kernel	32	32
	Multi-channel beamformer	57	70
	Time delay equalization (TDE)	35	35
Cryptography	Data Encryption Standard (DES)	55	64
	Serpent	120	128
Video processing	MPEG2 video	23	26
Sorting	Bitonic Parallel Sorting	41	48

table, $|\mathcal{A}|$ and $|\mathcal{E}|$ denote the number of actors and FIFO communication channels in the corresponding SDF graph of an application.

To demonstrate the effectiveness and efficiency of our proposed algorithm, we perform two experiments. In the first experiment, in Section 4.7.1, we consider a homogeneous platform as considered in the related works [4,23,81]. In this experiment, we compare the application latency, the memory requirements, and the minimum number of processors needed to schedule the tasks of each application under a given throughput requirement for a homogeneous platform, i.e, platform with only PE processors, obtained with six different scheduling/allocation approaches: (i) partitioned EDF with FFD heuristic; (ii) partitioned EDF with our proposed heuristic algorithm; (iii) partitioned EDF with the heuristic proposed in [81]; (iv) semi-partitioned EDF-*fm*, with the FFD-SP heuristic proposed in [23]; (v) semi-partitioned EDF-*fm*, with the LUF heuristic proposed in [4]; (vi) semi-partitioned EDF-sh [92]. These approaches are denoted in Table 4.3 with FFD, our, FFD-EP, FFD-SP, *fm*-LUF, and EDF-*sh*, respectively. In the second experiment, in Section 4.7.2, we consider heterogeneous platforms, including PE and EE processors, as considered in the related work [92]. In this experiment, we compare the application latency and the memory requirements needed to schedule the tasks of each application under a given throughput requirement obtained with partitioned EDF with our proposed heuristic algorithm and semi-partitioned EDF-sh [92] for different heterogeneous platforms. *Please note that we use the approach presented in [23] to handle data dependencies when using the scheduling/allocation approaches in [4,92] for comparison with our algorithm.* The throughput requirement $\mathcal{R}$ for each application, that is, the maximum achievable throughput under the SPS framework, is given in the second column in Table 4.3.

Table 4.3: *Comparison of different scheduling/allocation approaches.*

Benchmark	$\mathcal{R}\ [\frac{1}{\text{t.u.}}]$	OPT	Partitioned										Semi-partitioned								
			FFD			our			FFD-EP				FFD-SP			*fm*-LUF			EDF-sh		
		$\check{m}_{\text{OPT}}$	$\check{m}_{\text{FFD}}$	$\mathcal{M}_{\text{FFD}}$[B]	$\mathcal{L}_{\text{FFD}}$[t.u.]	$\check{m}_{\text{our}}$	$\frac{\mathcal{M}_{\text{our}}}{\mathcal{M}_{\text{FFD}}}$	$\frac{\mathcal{L}_{\text{our}}}{\mathcal{L}_{\text{FFD}}}$	$\check{m}_{\text{EP}}$	$\frac{\mathcal{M}_{\text{EP}}}{\mathcal{M}_{\text{FFD}}}$	$\frac{\mathcal{L}_{\text{EP}}}{\mathcal{L}_{\text{FFD}}}$	$\check{m}_{\text{SP}}$	$\frac{\mathcal{M}_{\text{SP}}}{\mathcal{M}_{\text{FFD}}}$	$\frac{\mathcal{L}_{\text{SP}}}{\mathcal{L}_{\text{FFD}}}$	$\check{m}_{\text{LUF}}$	$\frac{\mathcal{M}_{\text{LUF}}}{\mathcal{M}_{\text{FFD}}}$	$\frac{\mathcal{L}_{\text{LUF}}}{\mathcal{L}_{\text{FFD}}}$	$\check{m}_{\text{sh}}$	$\frac{\mathcal{M}_{\text{sh}}}{\mathcal{M}_{\text{FFD}}}$	$\frac{\mathcal{L}_{\text{sh}}}{\mathcal{L}_{\text{FFD}}}$
FFT	1/6016	24	30	144680	192512	24 **(26)**	1.545 **(1.115)**	1.313 **(1.063)**	24	2.420	2.344	26	1.413	1.483	26	1.485	1.676	24	3.114	3.772
Beamformer	1/5076	26	28	14492	60912	26	1.144	1.166	26	2.781	1.750	26	1.145	1.474	26	1.229	1.606	26	1.326	2.091
TDE	1/32205	20	25	516282	1127175	20 **(21)**	1.597 **(1.180)**	1.286 **(1.086)**	21	1.301	1.195	20	1.560	1.396	21	1.722	1.860	20	3.139	3.086
DES	1/704	26	33	3381	33088	26 **(27)** **(28)**	1.182 **(1.103)** **(1.073)**	1.213 **(1.106)** **(1.085)**	27	1.357	1.340	27	1.138	1.218	28	1.684	1.862	26	1.592	2.301
Serpent	1/3336	39	42	59815	370296	39 **(40)**	1.016 **(1.005)**	1.090 **(1.027)**	40	3.78	1.81	40	1.012	1.074	39	1.068	1.479	39	1.069	1.648
MPEG2	1/7680	8	9	61909	138240	8	1.104	1.055	8	1.478	1.141	8	1.290	1.217	9	3.014	3.432	8	1.665	1.544
Bitonic	1/91	11	13	2374	2275	11	1.104	1.080	11	1.102	1.120	11	1.139	1.185	11	1.413	1.395	11	1.291	1.502

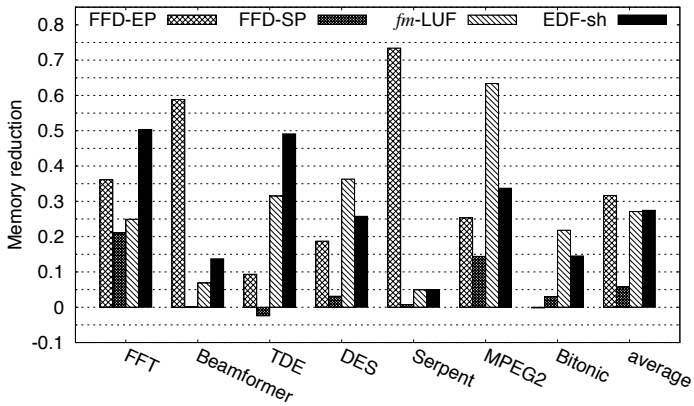
4.7.1 Homogeneous platform

Let us first compare our algorithm with the related approaches in terms of the number of required processors. The minimum number of required processors to satisfy the throughput requirement for each application using an optimal scheduler, denoted as $\check{m}_{OPT}$ and calculated using Equation (2.8), is given in the third column in Table 4.3. To find the minimum number of required processors using our proposed algorithm and the related approaches proposed in [4, 23, 81, 92], we set the number of PE processors on the homogeneous platform initially to $\check{m}_{OPT}$. Then, if the task set cannot be scheduled on the platform, we add one more PE processor and repeat the task allocation procedure again until a successful task allocation is found.

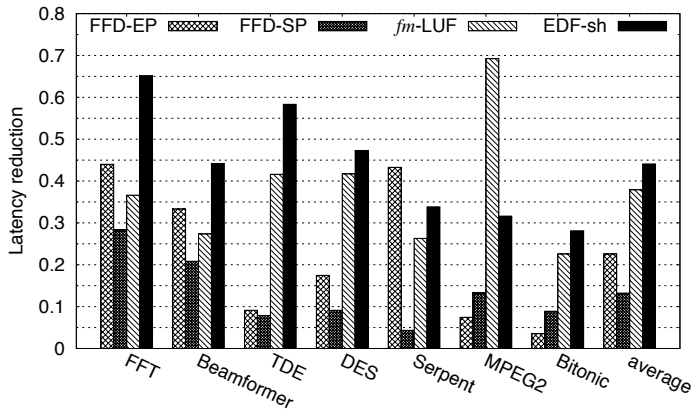
As can be seen in Table 4.3, the FFD approach requires considerably more processors, on average 17.6% more, than the number of required processors by an optimal scheduler, see column $\check{m}_{FFD}$. In contrast, our algorithm and EDF-sh require the same number of processors as the optimal scheduler while maintaining the same throughput for this set of applications, see columns $\check{m}_{our}$ and $\check{m}_{sh}$, respectively. For the other approaches, although they require fewer processors than FFD, they still require more processors than our algorithm for some applications. For instance, the approach FFD-EP requires one more processor for TDE, DES, and Serpent, see column $\check{m}_{EP}$; The approach FFD-SP requires two more processors for FFT and one more processor for DES and Serpent, see column $\check{m}_{SP}$; Finally the approach *fm*-LUF requires two more processors for FFT and DES and one more processor for TDE and MPEG2, see column $\check{m}_{LUF}$. Although this difference in terms of number of required processors is not too large, it clearly reveals that our algorithm is more capable of scheduling the applications with fewer processors compared to the FFD-EP, FFD-SP, and *fm*-LUF approaches while satisfying the same throughput requirement.

However, this reduction on the number of required processors comes at the expense of increased memory requirements and application latency. For each application, columns $\mathcal{M}_{FFD}$ and $\mathcal{L}_{FFD}$ report the memory requirements, expressed in bytes, and the application latency, expressed in time units, under FFD, respectively. The memory requirements is computed as the sum of the buffer sizes of the FIFO communication channels in the (C)SDF graph and the code size of the tasks. For each application, the increase on memory requirements and application latency by our algorithm over FFD are given in columns $\frac{\mathcal{M}_{our}}{\mathcal{M}_{FFD}}$ and $\frac{\mathcal{L}_{our}}{\mathcal{L}_{FFD}}$, respectively, that are on average 24.2% and 17.2%, respectively. Similarly, the increases on memory requirements and application latency are on average respectively 100% and 52.85% for FFD-EP, 24.3% and 29.2% for

FFD-SP, 65.9% and 90.2% for *fm*-LUF, and finally 88.5% and 127.8% for EDF-sh compared to FFD. From these numbers, we can conclude that not only our algorithm achieves fewer processors compared to the related approaches, but also it imposes, on average, lower memory and latency overheads.



(a) Memory reduction



(b) Latency reduction

Figure 4.4: *Memory and latency reduction of our algorithm compared to the related approach with the same number of processors.*

To further compare our algorithm with the related approaches, we compute the memory requirements and application latency of our algorithm when equal number of processors as the related approaches are used, see the bolded numbers in parenthesis in columns $\check{\mathrm{m}}_{\mathrm{our}}$, $\frac{\mathcal{M}_{\mathrm{our}}}{\mathcal{M}_{\mathrm{FFD}}}$, and $\frac{\mathcal{L}_{\mathrm{our}}}{\mathcal{L}_{\mathrm{FFD}}}$. To ease the interpretation of Table 4.3 for this comparison, Figure 4.4(a) and Figure 4.4(b) illustrate the memory and latency reductions obtained by our algorithm compared to

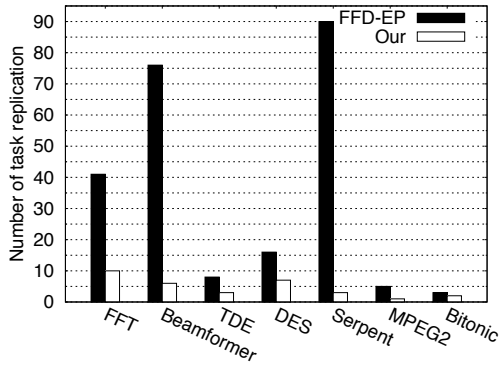


Figure 4.5: *Total number of task replications needed by FFD-EP and our proposed algorithm.*

the related approaches, respectively. For instance, the reduction on memory requirements is computed using the following equation:

$$r = \frac{\mathcal{M}_{\text{rel}} - \mathcal{M}_{\text{our}}}{\mathcal{M}_{\text{rel}}} \tag{4.3}$$

where $\mathcal{M}_{\text{rel}}$ is the memory requirements of scheduling an application using a related approach and $\mathcal{M}_{\text{our}}$ denotes the memory requirements achieved by our algorithm for the same number of processors. In Figure 4.4(a), we can see that our algorithm can reduce the memory requirements by an average of 31.43%, 5.72%, 27.11%, and 27.46% compared to FFD-EP, FFD-SP, *fm*-LUF, and EDF-sh, respectively. In Figure 4.4(a), however, there are two exceptions where our algorithm achieves 2.43% and 0.19% more memory for TDE and Bitonic compared to FFD-SP and FFD-EP, respectively. In Figure 4.4(b), we can also see that our algorithm can reduce the application latency considerably for all applications by an average of 22.60%, 13.24%, 37.92%, and 44.09% compared to FFD-EP, FFD-SP, *fm*-LUF, and EDF-sh, respectively. This comparison clearly demonstrates that for most of the applications our algorithm is more efficient than the related approaches in exploiting the available resources. Compared to FFD-EP, that is the closest approach to our algorithm as both adopt the graph unfolding transformation, our efficiency comes from significantly reducing the number of required task replications due to our novel Algorithm 1, as shown in Figure 4.5. This figure clearly shows that, by replicating the right tasks, our proposed algorithm can reduce the total number of task replications significantly, by up to 30 times, compared to FFD-EP. From Figure 4.4, it can be also observed that our proposed algorithm works better for some applications than for others compared to the related approaches. Given the (C)SDF graph of each application has different properties, e.g, the number of actors, the actors'

Table 4.4: *Runtime (in seconds) comparison of different scheduling/allocation approaches.*

Benchmark	t_{FFD}	t_{our}	$t_{FFD\text{-}EP}$	$t_{FFD\text{-}SP}$	t_{fm-LUF}	$t_{EDF\text{-}sh}$
FFT	0.001	5.95	451.48	0.22	0.17	0.024
Beamformer	0.011	5.16	126.30	0.100	0.037	0.022
TDE	0.005	3.96	138.32	0.011	0.013	0.011
DES	0.002	9.41	14.20	0.28	1.013	0.021
Serpent	0.025	56.43	960.30	1.44	0.45	0.09
MPEG2	0.001	0.015	3.25	0.002	0.002	0.004
Bitonic	0.001	0.127	0.093	0.003	0.011	0.034

workload, the graph's topology, repetition vector, etc., the applications are represented with a different set of periodic tasks by using the SPS framework in terms of the number of tasks and the utilization of tasks. Therefore, this variation on the number of tasks and the utilization of tasks in the set of periodic tasks according to each application can have different impact on the performance of different scheduling/allocation approaches.

Finally, we evaluate the efficiency of our algorithm in terms of the execution time. We compare the execution time of our algorithm with the corresponding execution times of FFD, FFD-EP, FFD-SP, *fm*-LUF, and EDF-sh. The comparison is given in Table 4.4. As can be seen from Table 4.4, the execution time of FFD and EDF-sh are always within less than 34 millisecond, while the execution times of FFD-SP and *fm*-LUF are within less than 1.5 seconds. However, the execution time of our algorithm is longer than FFD, FFD-SP, *fm*-LUF, and EDF-sh due to its iterative execution nature, but it is within less than 10 seconds for most of the cases and within less than 1 minute for one case which is reasonable given that our proposed algorithm is used at design-time and that it achieves better resource utilization. Among all the approaches, FFD-EP has the highest execution time, which is within less than 17 minutes, due to excessive number of algorithm iterations. This excessive number of iterations is due to the excessive number of required task replications in FFD-EP as shown in Figure 4.5.

4.7.2 Heterogeneous platform

To compare our proposed algorithm and EDF-sh [92] on heterogeneous platforms, in this section, we conduct experiments on a set of heterogeneous platforms including different number of PE and EE processors. To do so, we initially generate a heterogeneous platform having $\breve{m}_{FFD}-1$ PE processors (see Table 4.3 for $\breve{m}_{FFD}$) and 1 EE processor for each application and iteratively replace one PE processor with one EE processor (or more EE processors

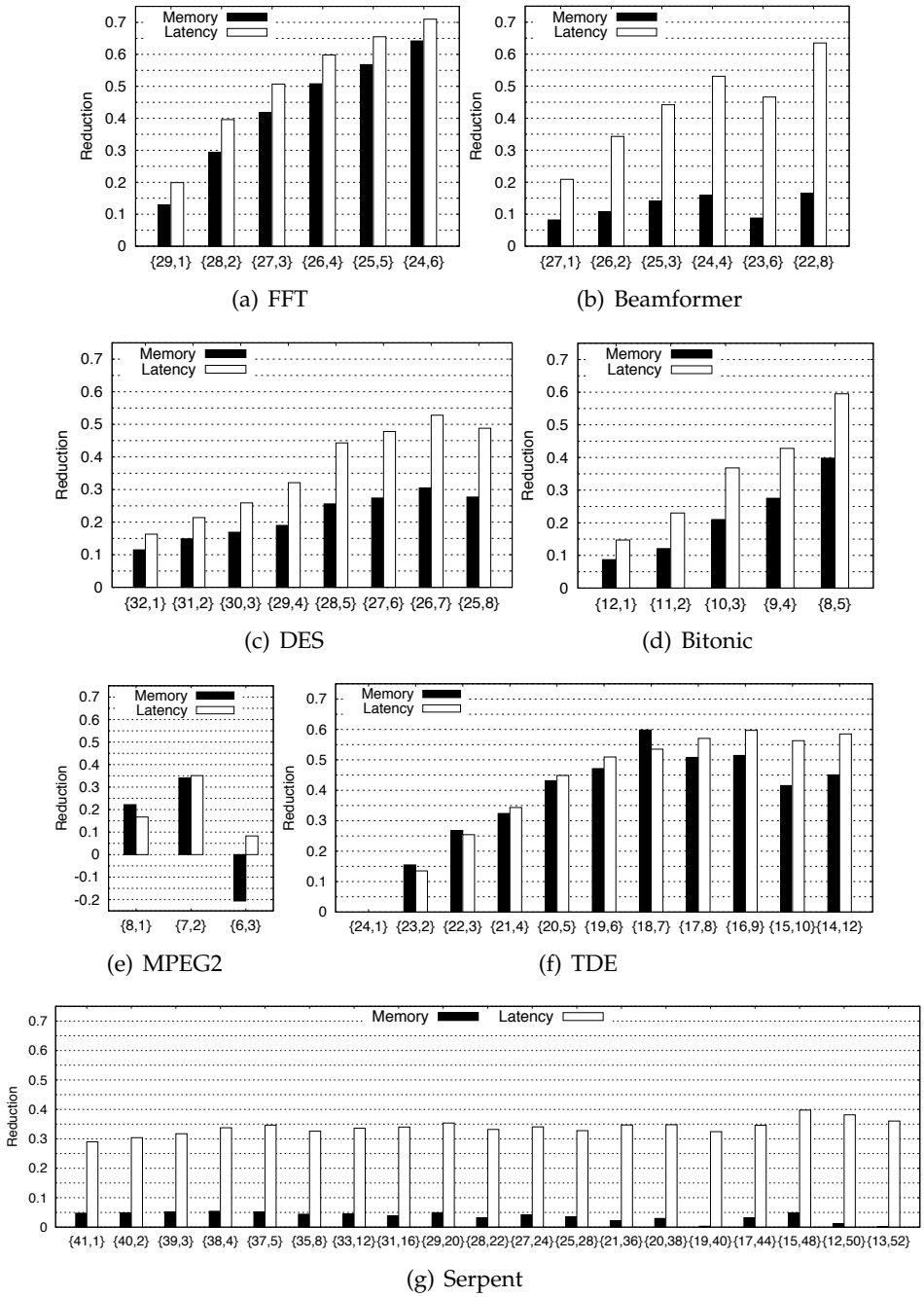


Figure 4.6: *Memory and latency reduction of our algorithm compared to EDF-sh [92] for real-life applications on different heterogeneous platforms.*

if the task set is not schedulable on the platform). However, due to the restrictive allocation rules in EDF-sh to ensure bounded tardiness for deadline misses, EDF-sh cannot find a task allocation for some heterogeneous platforms that have fewer than a certain number of PE processors. Therefore, we only compare our algorithm with EDF-sh on the heterogeneous platforms for which EDF-sh can successfully allocate the tasks for each application. Figure 4.6 shows the memory and latency reductions obtained by our algorithm compared to EDF-sh for each application individually. The reductions are computed using Equation (4.3). In Figure 4.6, the x-axis shows different heterogeneous platforms, comprised of different number of PE and EE processors denoted by {number of PEs, number of EEs}. The y-axis shows the reduction on the memory requirements and application latency.

From Figure 4.6, it can be observed that our proposed algorithm outperforms EDF-sh in terms of memory requirements and application latency for most of the cases. Compared to EDF-sh, our algorithm can reduce the memory requirements and application latency by an average of 42.6% and 51.1%, 12.4% and 43.8%, 21.7% and 36.2%, 21.8% and 35.4%, 11.9 % and 20.1%, 37.6 % and 42.2%, and 3.6 % and 33.8% for the FFT, Beamformer, DES, Bitonic, MPEG, TDE, and Serpent applications, respectively. For the MPEG application, however, our proposed algorithm increases the memory requirements compared to EDF-sh by 20.6% on a platform including 6 PE and 3 EE processors. This is because our algorithm excessively replicates a task to utilize the unused capacity left on the under-utilized processors. Therefore, the memory requirements increase significantly due to the code and data memory overheads. However, since the replicated task has low impact on the application latency, our algorithm can still reduce the application latency by 8.3% compared to EDF-sh. For the TDE application, both approaches find a task allocation without requiring either task replication (our) or task migration (EDF-sh) on a platform including 24 PE and 1 EE processors, therefore no reduction is achieved for both memory requirements and latency in this case.

In addition, it can be observed in Figure 4.6 that for most of the cases by replacing more PE processors with EE processors on the platform, our algorithm can further reduce the memory requirements and application latency compared to EDF-sh. This is mainly because, by replacing more number of PE processors with EE processors on the platform, the number of migrating tasks under EDF-sh scheduler is considerably increased while the number of task replications is only gently increased by our algorithm. As a result, more fixed tasks are affected by migrating tasks and can miss their deadlines, by a bounded tardiness, under EDF-sh scheduler that comes at the expense of

more memory requirements and longer application latency. According to the approach presented in [23], the memory requirements increase due to both the size of buffers, that have to be enlarged to handle task tardiness, and the code size overhead of task replicas, which are necessary in case of migrating tasks. In addition, the application latency increases due to the postponement of task start times needed to handle task tardiness.

4.8 Conclusions

In this chapter, we have presented a novel heuristic algorithm which determines a replication factor for each actor in an acyclic SDF graph, with a given throughput requirement, such that the number of processors needed to schedule the periodic tasks corresponding to actors in the obtained transformed graph is reduced under partitioned scheduling algorithms. By performing tasks replication, the tasks' workload is distributed among more parallel tasks' replicas with larger period and lower utilization in the obtained transformed graph. Therefore, the required capacity of the tasks which are replicated, is split up in multiple smaller chunks that can more likely fit into the left capacity on the processors and alleviate the capacity fragmentation due to partitioned scheduling algorithms, hence reducing the number of needed processors. The experiments on a set of real-life streaming applications show that our proposed algorithm can reduce the number of needed processors by up to 7 processors with increasing the memory requirements and application latency by 24.2% and 17.2% on average compared to FFD while satisfying the same throughput requirement. We also show that our algorithm can still reduce the number of needed processors by up to 2 processors and considerably improve the memory requirements and application latency by up to 31.43% and 44.09% on average compared to the other related approaches while satisfying the same throughput requirement.

Chapter 5

Energy-Efficient Scheduling of Streaming Applications

Sobhan Niknam, Todor Stefanov. "Energy-Efficient Scheduling of Throughput-Constrained Streaming Applications by Periodic Mode Switching". *In Proceedings of the 17th IEEE International Conference on Embedded Computer Systems: Architectures, MOdeling, and Simulation (SAMOS)*, Samos, Greece, July 17 - 20, 2017.

IN this chapter, we present our energy-efficient periodic scheduling approach, which corresponds to the third research contribution, briefly introduced in Section 1.5.3, to address the research question **RQ2(B)**, described in Section 1.4.2. The remainder of this chapter is organized as follows. Section 5.1 introduces, in more details, the problem statement and the addressed research question. It is followed by Section 5.2, which gives a summary of the contributions presented in this chapter. Section 5.3 gives an overview of the related work. Section 5.4 introduces the extra background material needed for understanding the contributions of this chapter. Section 5.5 gives a motivational example. Section 5.6 presents the proposed scheduling approach. Section 5.7 presents the experimental evaluation of the proposed scheduling approach. Finally, Section 5.8 ends the chapter with conclusions.

5.1 Problem Statement

As mentioned in Section 1.1, energy efficiency has become a critical challenge for the design of modern embedded systems, especially for those which are battery-powered. To address the energy efficiency challenge, many approaches

have been proposed in the past decades by several research communities [11]. These approaches mostly exploit the Voltage and Frequency Scaling (VFS) mechanism that is widely adopted in modern processors. The general idea behind these approaches is to exploit available idle, i.e., slack, time in the schedule of an application in order to slow down the execution of tasks of the application, by running processors at a lower voltage and operating clock frequency, using the VFS mechanism and to reduce the energy consumption while satisfying a given throughput requirement for the application.

Concerning the SPS framework, briefly described in Section 2.3, some heuristic approaches have been proposed in [25, 55, 80] to find an energy-efficient task mapping and scheduling using the VFS mechanism. Recall from Equation (2.12) that under the SPS framework, briefly described in Section 2.3, the period of real-time periodic tasks corresponding to the actors of a CSDF graph can be enlarged by taking any $s \geq \check{s} \in \mathbb{N}$ as long as a given application throughput requirement is satisfied. This period enlargement under the SPS framework, however, results in a set of application schedules that can only satisfy a discreet set of application throughputs, as the timing requirement. Therefore, given a required application throughput that is not in this set of guaranteed throughputs by the SPS framework, the schedule that provides the closest higher throughput to the required one must be selected from the set. As a consequence, this reduces the amount of available slack time in the application schedule, that can be potentially exploited using the VFS mechanism to reduce the energy consumption, and limits the energy-efficiency of the approaches in [25, 55, 80]. Thus, in this chapter, we investigate the possibility to exploit more slack time in the schedule of an application, modeled as a CSDF graph, under the SPS framework with a given throughput requirement using the VFS mechanism to achieve more energy efficiency.

5.2 Contributions

In order to address the problem described in Section 5.1, in this chapter, we propose a novel energy-efficient scheduling approach that combines the VFS mechanism [71] and the SPS framework [8] in a sophisticated way. In this novel approach, the execution of an application is periodically switched at run-time between a few off-line determined energy-efficient schedules, called *operating modes*, to satisfy a given throughput requirement at a long run. As a result, this approach can reduce the energy consumption significantly by exploiting the slack time in the application schedule more efficiently using the Dynamic Voltage and Frequency Scaling (DVFS) mechanism [50], where

multiple operating frequencies are computed at design-time for the processors to be used at run-time. More specifically, the main contributions of this chapter are as follows:

- A simple scheme has been devised for determining a set of discrete operating modes of a system at different operating frequencies where each operating mode provides a unique pair of throughput and minimum power consumption to achieve this throughput.
- With such a set of discrete operating modes and a given throughput requirement, we have devised an energy-efficient periodic scheduling approach which allows streaming applications to switch their execution periodically between operating modes at run-time to satisfy the throughput requirement at a long run. Using this specific switching scheme, we can benefit from adopting the DVFS mechanism to exploit the available static slack time in an application schedule efficiently.
- The experimental results, on a set of real-life streaming applications, show that our scheduling approach can achieve energy reduction by up to 68% depending on the application and the throughput requirement compared to the straightforward way of applying VFS as done in related works.

5.3 Related Work

Several approaches aiming at reducing the energy consumption of streaming applications have been presented in the past decades. Among these approaches, [26, 42, 61, 74, 96] are the closest to our work. These approaches have a common goal to reduce the energy consumption of a system by exploiting the static slack time in the schedule of throughput-constrained streaming applications using per-task [26, 61], per-core [42, 74, 96] or global [42] VFS.

The approaches in [26, 42, 61], formulate the energy optimization problem as a mixed integrated linear programming (MILP) problem to integrate the VFS capability of processors with application scheduling. Compared to these approaches, our approach mainly differs in two aspects. First, these approaches consider streaming applications modeled either as a Directed Acyclic Graph (DAG) [26, 42] or a Homogeneous SDF (HSDF) graph [61] derived by applying a certain transformation on an initial SDF graph. Therefore, these approaches cannot be directly applied to streaming applications modeled with more expressive MoCs, e.g., (C)SDF as considered in our work. In addition, transforming a graph from SDF to HSDF is a crucial step in [61] where the number of tasks in the streaming application can exponentially grow. This

growth of the application in terms of the number of tasks can lead to time-consuming analysis and significant memory overhead for storing the tasks' code. In contrast, our approach directly handles a more expressive MoC, such as (C)SDF. Second, the approach in [42] uses per-core VFS where the off-line computed operating frequencies of processors are fixed at run-time and cannot be changed. In contrast, our approach uses DVFS where a sequence of frequency changes which is computed off-line is used on the processors during execution at run-time while satisfying the throughput requirement. As a result, the DVFS mechanism enables our approach to exploit the available static slack time in the application schedule more efficiently for better energy reduction. The approaches in [26, 61] use a fine-grained DVFS, i.e., per-task VFS, where the operating frequency of processors can be changed before executing each task. Fine-grained DVFS, like in [26, 61], can be beneficial only when the overhead of DVFS is negligible. In contrast to these approaches, we adopt a coarse-grained DVFS where the operating frequencies of processors are changed at the granularity of graph iterations to avoid the large overhead associated with the operating frequency changes.

The approaches in [74, 96] perform energy reduction directly on an SDF graph. To this end, the approaches in [74, 96] perform design space exploration (DSE) at design time to find an energy-efficient schedule (in a self-timed manner) of an SDF graph mapped on an MPSoC platform with per-core VFS capability such that a given throughput requirement is satisfied. However, as shown in the motivation example in Section 5.5, applying VFS in a similar way as in [74, 96] for streaming applications scheduled using the SPS framework [8] is not energy-efficient. Compared to the approaches in [74, 96], our approach is different in two aspects. First, these approaches use self-timed scheduling for which analysis techniques suffer from a complex DSE. In contrast, we use the SPS framework that enables the utilization of many scheduling algorithms with fast analysis techniques from the classical hard real-time scheduling theory [29]. Second, these approaches use per-core VFS to exploit static slack time in the application schedule. In contrast, our approach uses a coarse-grained DVFS. As a result, the processors are able to run periodically at lower operating frequencies by exploiting available static slack time more efficiently which can result in lower energy consumption.

5.4 Background

In this section, we define the system model and present the power model considered throughout this chapter.

5.4.1 System Model

In this section, we define the system model used in this chapter. The considered MPSoC platforms in this chapter are homogeneous, i.e., a platform contains a set $\Pi = \{\pi_1, \pi_2, \cdots, \pi_m\}$ of m identical processors with distributed memories. We assume that processors are endowed with the VFS capability. In this regard, we assume that each processor supports only a discrete set $\theta = \{f_{min} = f_1, f_2, \cdots, f_n = f_{max}\}$ of n operating frequencies and different processors can operate at different frequencies at the same time. Without loss of generality, we assume that the operating frequencies in the set θ are in ascending order, in which f_1 is the lowest operating frequency and f_n is the highest operating frequency.

5.4.2 Power Model

This section defines the power model used in this chapter. According to [55], the power consumption of a (fully utilized) processor can be computed by the following equation:

$$P(f) = \alpha f^b + \beta$$

where the first term is the dynamic power consumption and includes all frequency-dependent components, the second term is the static power consumption and includes all frequency-independent components, and f is the operating frequency. Parameters α, b, and β are dependent on the platform and they are determined in [55] by performing real measurements on a real MPSoC platform. When all tasks are allocated on processors of platform Π, the power consumption of processor π_j can be computed by the following equation:

$$P_j = \alpha \cdot f_{\pi_j}^b \cdot \frac{f_{max}}{f_{\pi_j}} \sum_{\forall \tau_i \in {}^m\Gamma_j} \frac{C_i}{T_i} + \beta \tag{5.1}$$

where $f_{\pi_j} \in \theta$ is the operating frequency of π_j and ${}^m\Gamma_j \in {}^m\Gamma$ represent the set of tasks allocated on processor π_j. Therefore, the energy consumption of π_j within one graph iteration period (hyper period) is $E_j = H \cdot P_j$ and the energy consumption of the platform within one iteration period is $E = \sum_{\forall \pi_j \in \Pi} H \cdot P_j$.

5.5 Motivational Example

In this section, we motivate the necessity of devising a new energy-efficient scheduling approach using the VFS mechanism in the context of the SPS

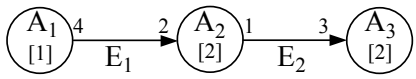


Figure 5.1: *An SDF graph G.*

framework [8]. To do so, this motivational example consists of two parts. In the first part, we show that a straightforward way of applying the VFS mechanism in the context of the SPS framework is not energy efficient. Then, in the second part, we show how we can schedule an application more energy efficient using our novel periodic scheduling approach.

5.5.1 Applying VFS Similar to Related Works

Let us consider a simple streaming application modeled as the SDF graph G shown in Figure 5.1. This graph has three actors $\mathcal{A} = \{A_1, A_2, A_3\}$ with worst-case execution times $C_1 = 1$, $C_2 = 2$, and $C_3 = 2$ at the maximum processor operating clock frequency. The repetition vector of this graph, according to Theorem 2.1.1, is $\vec{q} = [3, 6, 2]^T$. By applying the SPS framework for graph G, the task set $\Gamma = \{\tau_1 = (C_1 = 1, T_1 = 4, S_1 = 0, D_1 = 4), \tau_2 = (2, 2, 4, 2), \tau_3 = (2, 6, 10, 6)\}$ of three IDP tasks can be derived. Note that the derived periods of the tasks are the minimum periods by using the scaling factor $s = \check{s} = \lceil \frac{12}{6} \rceil = 2$ in Equation (2.12). Based on these tuples, a strictly periodic schedule, as shown in Figure 5.2(a), can be obtained for this graph. Using Equation (2.15), the throughput of this schedule can be computed as $\mathcal{R} = \frac{1}{T_3} = \frac{1}{6}$. The minimum number of processors needed for this schedule under partitioned First-Fit Decreasing (Utilization) EDF (FFD-EDF) is two. Therefore, we consider a homogeneous MPSoC platform $\Pi = \{\pi_1, \pi_2\}$ containing two processors, where we allocate task τ_2 on processor π_1 and tasks τ_1 and τ_3 on processor π_2, i.e., ${}^2\Gamma = \{{}^2\Gamma_1 = \{\tau_2\}, {}^2\Gamma_2 = \{\tau_1, \tau_3\}\}$.

So far, we have assumed that the tasks run at the maximum operating frequency of the processors. Let us assume that each processor can only support a discrete set $\theta = \{1/4, 1/2, 3/4, 1\}$(GHz) of four operating frequencies. In order to make this schedule more energy efficient, we use the VFS mechanism to exploit the available static slack time in the schedule for the purpose of slowing down the execution of tasks by decreasing the operating frequency of the processors. For this example, we can only decrease the operating frequency of processor π_2 to 3/4 GHz while still satisfying all timing requirements, i.e., job deadlines shown as down arrows in Figure 5.2(a). This slowing down of the execution of tasks is visualized by extending the gray boxes with the

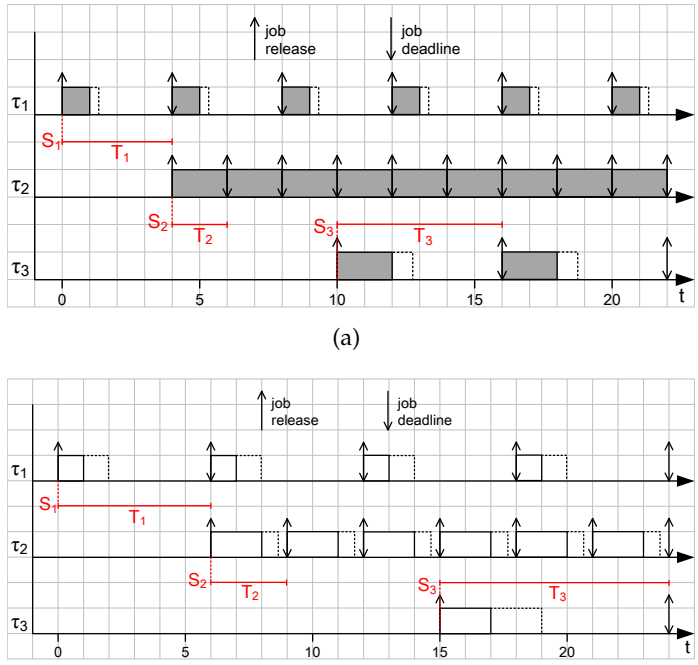


Figure 5.2: *The (a) SPS and (b) scaled SPS of the (C)SDF graph G in Figure 5.1. Up arrows represent job releases, down arrows represent job deadlines. Dotted rectangles show the increase of the tasks execution time when using the VFS mechanism.*

dotted boxes in Figure 5.2(a). Using Equation (5.1), the power consumption of this schedule is 0.61 mW. The energy consumption of this schedule for a period of 36 time units, which is equivalent to 3 graph iterations, is 21.96 mJ.

To further reduce the power consumption by decreasing the operating frequency of processors, more static slack time is needed to be created in the application schedule. To do so, we can derive larger periods for tasks by using any integer scaling factor $s > \check{s} = 2$ in Equation (2.12). We refer to this approach as *period scaling* in this chapter. In this way, if we take $s = 3$, a new schedule can be derived using the SPS framework, as shown in Figure 5.2(b), with throughput $\mathcal{R} = \frac{1}{T_3} = \frac{1}{9}$. As a result, there is more static slack time available in the application schedule which enables the processors π_1 and π_2 to run at lower operating frequencies of $3/4$ GHz and $1/2$ GHz, respectively. This is visualized by extending the white boxes with the dotted boxes in Figure 5.2(b). Using Equation (5.1), the power consumption of this schedule is 0.43 mW. The energy consumption of this schedule for a period

of 36 time units, which is equivalent to 2 graph iterations, is 15.48 mJ. As a result, the energy consumption is reduced by 29.5% using the schedule in Figure 5.2(b) corresponding to $s = 3$ compared to the schedule in Figure 5.2(a) corresponding to $s = 2$ for the same time period at the expense of decreasing the application throughput from $1/6$ to $1/9$. By increasing the value of scaling factor s and enlarging the periods of tasks as much as possible such that the corresponding schedule still satisfies a given throughput requirement, we can apply the VFS mechanism in the straightforward way, described above, similar to the related works [74, 96]. Therefore, the maximum created static slack time in the application schedule can be exploited using the VFS mechanism to reduce the energy consumption as much as possible.

Now, assume that a throughput requirement of $1/8$ has to be satisfied. Following the period scaling approach, described above, the schedule corresponding to $s = 2$ with the throughput of $1/6$, shown in Figure 5.2(a), must be selected to satisfy the throughput requirement of $1/8$. However, this schedule is not the most energy-efficient one. This is because, although the throughput requirement of $1/8$ is satisfied, more energy is consumed as a result of delivering higher throughput than needed.

5.5.2 Our Proposed Scheduling Approach

In this section, we introduce our novel energy-efficient scheduling approach for graph G in Figure 5.1 that satisfies the same throughput requirement of $1/8$ while consuming less energy compared to the scheduling approach explained in Section 5.5.1. In our approach, among all possible application schedules corresponding to different values of scaling factor s to enlarge periods, we select only Pareto optimal schedules and form a set γ of schedules called *operating modes*. For instance, the set $\gamma = \{\mathrm{SI}^1, \mathrm{SI}^2, \mathrm{SI}^3, \mathrm{SI}^4, \mathrm{SI}^5\}$ of five operating modes for graph G is given in Table 5.1. In this table, every row shows an operating mode with the iteration period H, the operating frequencies of the two processors (f_{π_1}, f_{π_2}), the pair of throughput and power consumption $(\mathcal{R}, P)$, and the energy consumption corresponding to the operating mode. In the last column, the energy consumption of the operating modes is given for a period of 720 time units which is the *least common multiply* of the iteration periods H of all operating modes. As can be seen in this column, the energy consumption of the operating modes is being reduced by slowing down the application execution during this common period of time. The value of scaling factor s corresponding to each operating mode is also given in the first column. For instance, operating mode SI^4 is the application schedule corresponding to $s = 5$ that delivers throughput of $1/15$. In this schedule, processors π_1 and

Table 5.1: *Operating modes for graph G*

Mode	H	f_{π_1}	f_{π_2}	$(\mathcal{R}\ [\frac{Token}{Time\ units}],\ P\ [mW])$	$E\ [mJ]$
SI^1 $(s=2)$	12	1	3/4	$(1/6, 0.61)$	439.2
SI^2 $(s=3)$	18	3/4	1/2	$(1/9, 0.43)$	309.6
SI^3 $(s=4)$	24	1/2	1/2	$(1/12, 0.36)$	259.2
SI^4 $(s=5)$	30	1/2	1/4	$(1/15, 0.34)$	244.8
SI^5 $(s=8)$	48	1/4	1/4	$(1/24, 0.31)$	223.2

π_2 must operate at frequencies of 1/2 GHz and 1/4 GHz in order to meet all task's job deadlines. The power consumption of this schedule is 0.34 mW and the energy consumption of this schedule for 720 time units is 244.8 mJ.

Looking at set γ of operating modes in Table 5.1, the throughput requirement of 1/8, we consider in this example, is between the throughput of operating modes SI^1 and SI^2. Therefore, we propose the idea of periodically switching the application execution between operating modes SI^1 and SI^2 to satisfy the throughput requirement. Such a periodic switching schedule is depicted for one period in Figure 5.3, where the application executes for three graph iterations according to the schedule of operating mode SI^1 and two graph iterations according to the schedule of operating mode SI^2. Different graph iterations are separated by dotted and dashed lines for consecutive executions of the application in operating mode SI^1 and SI^2, respectively, in Figure 5.3. Note that this schedule repeats periodically every 77 time units, as shown in Figure 5.3 ($Q_1 + Q_2 + o_{12} = 77$). In one period, task τ_3 executes 10 times in total during 77 time units, meaning that throughput of 10/77=1/7.7 is delivered at a long run that is more closer to the throughput requirement of 1/8 compared to the throughput of 1/6 delivered as a result of the schedule in Figure 5.2(a). More importantly, the energy consumption of our proposed novel schedule in Figure 5.3 for a period of 924 time units, which is the *least common multiply* of the period of our schedule (77 time units) and the iteration period of the schedule in Figure 5.2(a) (12 time units), is 496.68 mJ. The energy consumption of the schedule in Figure 5.2(a) in the same period of 924 time units is 563.64 mJ. Therefore, our novel scheduling approach can reduce the energy consumption by 11.87% when the throughput requirement of 1/8 has to be satisfied. The energy reduction of our proposed schedule, referred as Switching, compared to the scheduling approach explained in Section 5.5.1, referred as Scale, for a wide range of throughput requirements is given in Figure 5.4. In this figure, the x-axis shows different throughput requirements for graph G in Figure 5.1 while the y-axis shows the normalized energy consumption. From Figure 5.4, we can see that our proposed scheduling approach

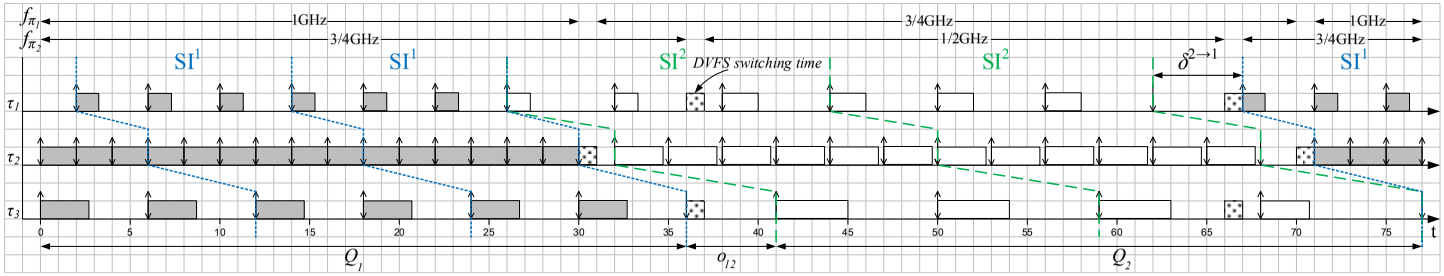


Figure 5.3: *Our proposed periodic schedule of graph G in Figure 5.1. In this schedule, graph G periodically executes according to schedules of operating mode SI^1 and operating mode SI^2 in Figure 5.2(a) and Figure 5.2(b), respectively. Note that this schedule repeats periodically. $o_{12} = 5$ and $o_{21} = 0$.*

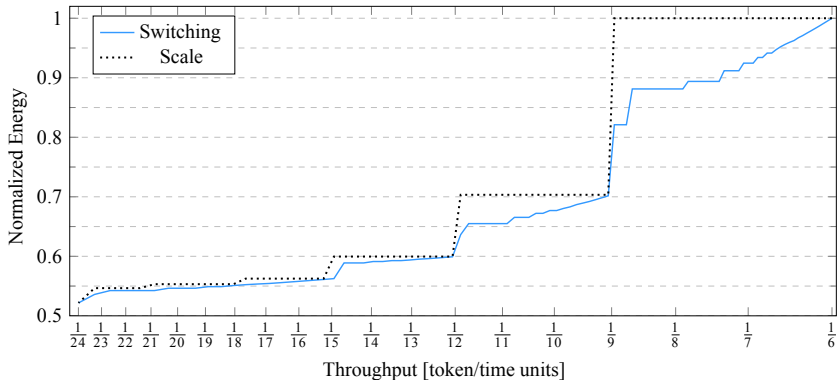


Figure 5.4: *Normalized energy consumption of the scaled scheduling and our proposed scheduling of the graph G in Figure 5.1 for a wide range of throughput requirements.*

Switching can reduce the energy consumption significantly compared to Scale for a large set of throughput requirements.

Note that our proposed scheduling approach uses the DVFS mechanism. This is because, processors run at different operating frequencies in each operating mode. Therefore, when the application switches to execute in a different operating mode, the operating frequencies of the processors are changed accordingly. The way of changing the operating frequencies of the processors, for our example, is shown by the horizontal arrows on top of Figure 5.3. Note that we also consider the switching time cost of the DVFS mechanism in our analysis that is shown by the boxes with dotted pattern in Figure 5.3.

From the above example, we can see the necessity and usefulness of our novel scheduling approach, presented in detail in Section 5.6, to obtain more energy-efficient application schedule when the VFS mechanism is used in the context of the SPS framework.

5.6 Proposed Scheduling Approach

In this section, we describe our proposed energy-efficient periodic scheduling approach for throughput-constrained streaming applications. The basis of our approach is to determine a set of *operating modes* where each operating mode provides a unique pair of throughput and minimum power consumption to achieve this throughput. Then, for a given throughput requirement, there may exist an operating mode whose throughput matches the throughput

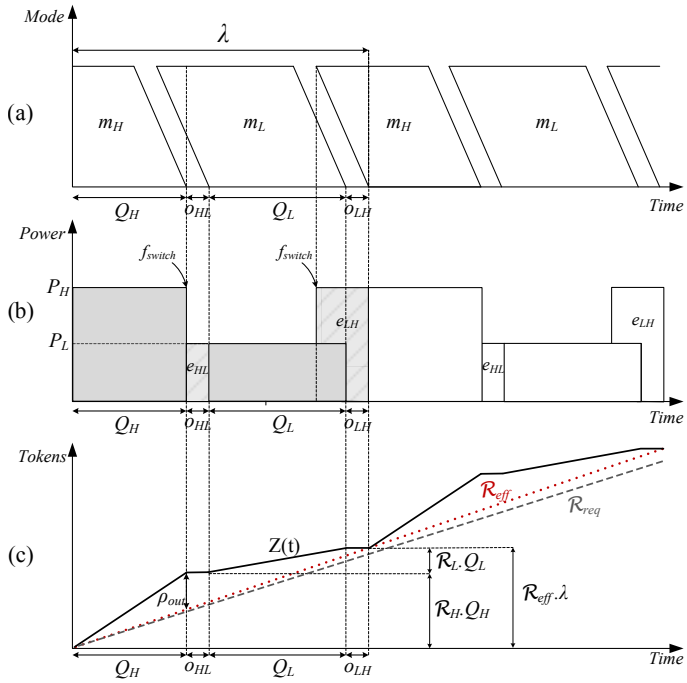


Figure 5.5: *(a) Switching scheme, (b) Associated energy consumption of the switching scheme and (c) Token production function Z(t).*

requirement. In this unlikely case, we simply select this operating mode. Otherwise, we choose the two operating modes with the closest higher and lower throughput to the throughput requirement, referred as *higher operating mode* (SI^H) and *lower operating mode* (SI^L), respectively. Then, we satisfy the throughput requirement at a long run by periodically switching the execution of the application between these two operating modes.

A general overview of our proposed switching scheme for the execution of an application between the higher and lower operating modes is illustrated in Figure 5.5. The periodic execution of the application between the higher and lower operating modes in our approach is shown in Figure 5.5(a) and the period of switching is denoted by λ. The associated energy consumption and token production of the application caused by our switching scheme corresponding to Figure 5.5(a) are also shown in Figure 5.5(b) and Figure 5.5(c), respectively. According to Figure 5.5(a), the execution of the application in each period λ consists of four parts. In the first part, the application executes in the higher operating mode for Q_H time units where the application has throughput $\mathcal{R}_H$ and power consumption P_H. Then, in the second part,

the execution of the application switches to the lower operating mode SI^L. However, this switching cannot happen immediately and it takes some time, denoted as o_{HL}, before the application can produce tokens again in the lower operating mode. Therefore, during the switching, the application does not have any token production for o_{HL} time units while consuming the energy of e_{HL}, as shown in Figure 5.5(b) and Figure 5.5(c), respectively. After completing the switching, in the third part, the application executes in the lower operating mode for Q_L time units where the application has the throughput and power consumption of $\mathcal{R}_L$ and P_L, respectively. Finally, in the fourth part, the application switches again to the higher operating mode SI^H for the next period of λ. However, this switching cannot happen immediately and it takes some time that is denoted by o_{LH}. During the switching time o_{LH}, no tokens are produced by the application while the energy of e_{LH} is consumed. As a result of the switching scheme in Figure 5.5(a), the application generates a number of tokens in total, see the curve $Z(t)$ in Figure 5.5(c), by executing in the higher and lower operating modes during every period of λ and in every λ the application effectively delivers the throughput of $\mathcal{R}_{eff}$ in a long run. The curves corresponding to the token production $Z(t)$ in our switching scheme and the effective throughput of $\mathcal{R}_{eff}$ are shown in Figure 5.5(c) with a solid line and a dotted line, respectively. The throughput requirement $\mathcal{R}_{req}$ is also shown with a dashed line in this figure. Therefore, to satisfy the throughput requirement, we have to always keep the effective throughput $\mathcal{R}_{eff}$ above the throughput requirement $\mathcal{R}_{req}$. This ensures that the number of produced tokens at any time instant is greater than or equal to what is needed.

Considering Figure 5.5(c), the effective throughput obtained by executing the application in operating mode SI^H for Q_H time units and operating mode SI^L for Q_L time units is computed by the following expression:

$$\mathcal{R}_{eff} = \frac{\mathcal{R}_H Q_H + \mathcal{R}_L Q_L}{Q_H + Q_L + o_{HL} + o_{LH}} = \frac{\mathcal{R}_H Q_H + \mathcal{R}_L Q_L}{\lambda} \tag{5.2}$$

where $\mathcal{R}_H$ and $\mathcal{R}_L$ are the throughputs of the application in the higher and lower operating modes, respectively, and $\mathcal{R}_H Q_H$ and $\mathcal{R}_L Q_L$ are the number of produced tokens in the higher and lower operating modes, respectively. Similarly, the effective power consumption for the same operating mode switching is computed as follows:

$$P_{eff} = \frac{P_H Q_H + P_L Q_L + e_{HL} + e_{LH}}{\lambda} = \frac{P_H Q_H + P_L Q_L}{\lambda} + \frac{e_{HL} + e_{LH}}{\lambda} \tag{5.3}$$

where P_H and P_L are the power consumption of the higher and lower operating modes, respectively, and $P_H Q_H$ and $P_L Q_L$ are the energy consumption in the higher and lower operating modes, respectively.

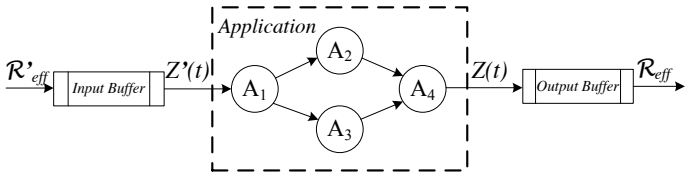


Figure 5.6: *Input and Output buffers.*

Using the periodic switching scheme, described above, we can benefit from adopting the DVFS mechanism to exploit the available static slack time in the application schedule more efficiently that can reduce the energy consumption considerably. The shaded area in Figure 5.5(b) shows the energy consumption corresponding to one period λ in our scheduling approach. Although the throughput requirement of the application is satisfied by our proposed approach, the mentioned energy reduction comes at the expense of increasing the memory requirement. This is because the application samples the input data stream and produces output data tokens in the higher operating mode more frequently than in the lower operating mode. As a consequence, this results in irregularity of sampling the input data stream and producing the output data tokens over time. Therefore, to solve this irregular sampling/production problem, we need extra memory buffers for the input and output of the application, as shown in Figure 5.6. The reason to use an output buffer is to gather the produced tokens and release them regularly over time in order to deliver the throughput requirement in a long run. In the same manner, to regularly sample the input data stream coming to the application, regardless of which operating mode the application is running in, we need an extra buffer at the input of the application. This buffer is needed to distribute the sampled data regularly over the input data stream to guarantee certain sampling accuracy instead of sampling the input data stream differently in each operating mode leading to different accuracy in every operating mode.

According to the discussion above and looking at Figure 5.5, there are some parameters in our scheduling approach that have to be determined, namely, the time duration to stay in the higher and lower operating modes (Q_H, Q_L), as well as switching costs $(o_{HL}, o_{LH}, e_{HL}, e_{LH})$. Therefore, in the rest of this section, we explain how to compute these parameters. We first explain how the operating modes are determined in Section 5.6.1. Then, we compute the switching costs, o_{HL}, e_{HL}, o_{LH} and, e_{LH} and the time duration of staying in the higher and lower operating modes, Q_H and Q_L, that are key elements in our approach, in Section 5.6.2 and Section 5.6.3, respectively. Finally, we compute the memory overhead (the input and output buffers in Figure 5.6)

Algorithm 2: Operating modes determination.

Input: A CSDF graph $G = (\mathcal{A}, \mathcal{E})$.
Input: A set $\Pi = \{\pi_1, \pi_2, \cdots, \pi_m\}$ of m identical processors.
Input: A set $\theta = \{f_{min} = f_1, f_2, \cdots, f_n = f_{max}\}$ of n discrete operating frequencies for the processors.
Input: A set ${}^m\Gamma = \{{}^m\Gamma_1, {}^m\Gamma_2, \cdots, {}^m\Gamma_m\}$ of task allocation on the processors.
Output: A set γ of operating modes.

1 $\gamma \leftarrow \emptyset$;
2 Compute $s = \check{s}$ using Equation (2.13);
3 **while** *true* **do**
4 **for** $\forall\ \tau_i \in \Gamma$ **do**
5 $T_i = \frac{\text{lcm}(\vec{q})}{q_i} \cdot s$;
6 **for** $\forall\ \pi_j \in \Pi$ **do**
7 Compute a minimum operating frequency f_{π_j} such that $u_{\pi_j} = \frac{f_{max}}{f_{\pi_j}} \sum_{\forall \tau_i \in {}^x\Gamma_j} \frac{C_i}{T_i} \leq 1$;
8 $\mathcal{R}$ = Compute the throughput of new schedule using Equation (2.15);
9 P = Compute the power consumption of new schedule corresponding to the operating frequency set $\vec{f}$ using Equation (5.1);
10 SI $\leftarrow (\mathcal{R}, P, \Gamma, \vec{f})$;
11 **if** $\{\neg\exists\ SI^i \in \gamma : \vec{f}_i = \vec{f}\}$ **then**
12 $\gamma \leftarrow \gamma +$ SI;
13 **if** *the operating frequency of all processors reaches to* f_{min} **then**
14 **return** γ;
15 $s = s + 1$;

associated with our scheduling approach in Section 5.6.4.

5.6.1 Determining Operating Modes

The procedure for determining the operating modes is given in Algorithm 2. The inputs of this algorithm are a CSDF graph G, a homogeneous platform Π containing m processors, a set θ of n discrete operating frequencies for the processors, and a set ${}^m\Gamma$ of task allocations on the processors. The output of this algorithm is a set γ of determined operating modes. First, Line 2 in this

algorithm initializes the scaling factor $s = \check{s}$ using Equation (2.13). Then, we use this initial value of s in Lines 4 and 5 to compute the minimum period of tasks corresponding to the actors in the CSDF graph G using Equation (2.12). Then, the minimum operating frequencies of the processors are computed in Lines 6 and 7 in such a way that the schedulability of the allocated tasks on each processor is still preserved. To do so, a simple utilization check is performed where the total utilization of the allocated tasks on each processor has to be less than or equal to 1, for partitioned EDF, for the selected operating frequency. These operating frequencies are then stored in the frequency set $\vec{f}$. In Lines 8 and 9, the throughput $\mathcal{R}$ and power consumption P of the periodic scheduling of task set Γ are computed using Equation (2.15) and Equation (5.1), respectively. Then, in Line 10 a new operating mode SI that is characterized with the strictly periodic task set Γ corresponding to s, throughput $\mathcal{R}$, power consumption P, and the set of operating frequencies $\vec{f}$ for the processors is created. Line 11 checks a condition whether to include the newly created mode to the set γ of operating modes. According to this condition, an operating mode is included to the set γ, in Line 12, if there does not exist any operating mode in set γ with the same operating frequency set $\vec{f}$. This is because, if there exists such an operating mode in set γ, it corresponds to smaller s than the new operating mode. Therefore, the tasks in the existing operating mode have shorter periods where less unused slack time remains in the application schedule with the same operating frequency of the processors. This selection strategy ensures that the static slack time in the application schedule is exploited more efficiently using the DVFS mechanism. Then, the explained procedure from Lines 4 to 12 repeats by incrementing s in Line 15 until the operating frequency of all processors reaches to the minimum available operating frequency. Finally, the set γ of all determined operating modes is returned by this algorithm in Line 14. As an example, following Algorithm 2, the operating modes for the graph G shown in Figure 5.1 are determined and listed in the Table 5.1.

5.6.2 Switching Costs $o_{HL}, o_{LH}, e_{HL}, e_{LH}$

In this section, we introduce the switching costs associated with our proposed switching scheme and explain the way we compute them.

(1) *Time Costs*: As shown in Figure 5.5(a), we switch the operating mode in our approach between SI^H and SI^L. In Section 2.4, mode switching has been investigated for an MADF graph to determine the earliest time that tasks in the new operating mode can start their execution during mode switching instants.

In Section 2.4, it has been shown that the tasks in the new operating mode cannot be executed immediately. Therefore, their execution has to be offset by δ time units according to Equation (2.20). As a consequence, the system may not have any token production during the operating mode switching. In our case, the time cost of switching from the higher operating mode SI^H to the lower operating mode SI^L and vice versa using the offset δ, according to Equation (2.20), can be computed as follows:

$$o_{HL} = S^L_{out} + \delta^{H\to L} - S^H_{out},\ o_{LH} = S^H_{out} + \delta^{L\to H} - S^L_{out} \tag{5.4}$$

where S^L_{out} and S^H_{out} are the starting time of the output task in the lower and higher operating modes, respectively. This time cost is exactly the elapsed time between the finishing of the output task in one operating mode and the starting time of the output task in the other operating mode. However, since the operating frequencies of the processors are changed during the switching, the computed δ offset in Equation (2.20) may not be sufficient. This is because, the time that is needed for physically changing the operating frequencies in the processors, denoted by ζ, is not considered in Equation (2.20). Apparently, the operating frequency must not be changed when the tasks in the higher operating mode are still executing in the system. Therefore, when the operating mode is switched from the higher operating mode to the lower operating mode, the operating frequency of the processors must be changed after the end of the execution of the allocated tasks on the processors in the higher operating mode. Similarly, when the operating mode is switched from the lower operating mode to the higher operating mode, the operating frequency of the processors must be changed before the start of the execution of the allocated tasks on the processors in the higher operating mode. This ensures that the tasks' job deadlines in both operating modes are met. For instance, for the proposed switching scheduling approach in Figure 5.3, the time instants of changing the operating frequencies of π_1 and π_2 are shown by the boxes with a dotted pattern where the size of these boxes denotes the frequency switching delay ζ. The δ offset in Equation (2.20) is a function of the tasks utilization. Therefore, to involve such switching delay ζ associated with the DVFS mechanism into the δ offset, we have changed the utilization of each task τ_i in the lower operating mode SI^L, i.e., τ^L_i, from C^L_i / T^L_i to $(C^L_i + \zeta) / T^L_i$ that is executing when the operating frequency change happens. As a result, using Equation (2.20), we can compute a sufficient δ with the new utilization of tasks to make sure that the job deadlines of all tasks in both operating modes are still met during operating mode switching. Clearly, the last starting time instant of the new operating mode, using Equation (2.20), can be when

the execution of the previous operating mode is completely finished and the operating frequencies of the processors are also changed. This is the safest starting time for the new operating mode while no extra schedulability test is needed as there is no overlapping execution between two operating modes. Using the method, explained above, for the proposed schedule in Figure 5.3, the starting offset of $\delta^{1\rightarrow 2} = 0$ can be computed for operating mode SI^2 when the operating mode is switched from SI^1 to SI^2. Similarly, the starting offset of $\delta^{2\rightarrow 1} = 5$ can be computed for operating mode SI^1 when the operating mode is switched from SI^2 to SI^1. Finally, the time cost of $o_{12} = 5$ and $o_{21} = 0$ can be computed using Equation (5.4) for the operating mode switching from SI^1 to SI^2 and vice versa, respectively, as can be seen in Figure 5.3.

(2) *Energy Costs*: By applying sufficient δ offset, as computed in Section 5.6.2(1) above, tasks belonging to both the lower and higher operating modes may be concurrently executing on the processors during mode switching instants. For instance, in Figure 5.3 tasks in both operating modes SI^1 and SI^2 execute from time instant 26 to 36 and from time instant 67 to 77 when the operating mode is switched from SI^1 and SI^2 and vice versa, respectively. To meet the tasks' job deadlines in both operating modes, the processors must run at the operating frequency corresponding to the higher operating mode during operating mode switching instants. Therefore, the total energy consumption of our proposed scheduling approach is more than the summation of the energy consumption of operating modes SI^H and SI^L for the execution intervals of Q_H and Q_L time unit, respectively. As a result, we define e_{HL} and e_{LH} as extra energy consumption when the operating mode is switched from the high operating mode to the low operating mode and vice versa, respectively, and we compute them using the following expressions:

$$e_{HL} = o_{HL} P_L \tag{5.5}$$

$$e_{LH} = (S^H_{out} - o_{LH})(P_H - P_L) + o_{LH} P_H = S^H_{out}(P_H - P_L) + o_{LH} P_L \tag{5.6}$$

where the S^H_{out} is the start time of the task corresponding to output actor A_{out} in the graph in the higher operating mode. These energy costs are visualized by the hatched boxes in Figure 5.5(b). These energy costs are overestimated using the above expressions because a single time instant is assumed for changing the operating frequency of all processors in each operating mode switching. This time instant is referred by f_{switch} in Figure 5.5(b). Note that we also include the energy overhead of DVFS into this energy costs.

5.6.3 Computing Q_H and Q_L

In our approach, we only allow the switching of operating modes at the graph iteration boundary. This means that the operating mode can be switched as soon as an application graph iteration is completed. Under this assumption, the time that an application is executed, in any operating mode, must be a multiple of the duration of one graph iteration. Therefore, the time that the application spends in the higher and lower operating modes can be defined as follows:

$$Q_H = N_H \cdot H_H,\ N_H \in \mathbb{N} \tag{5.7}$$

$$Q_L = N_L \cdot H_L,\ N_L \in \mathbb{N} \tag{5.8}$$

where N_H and N_L are the number of graph iterations in the higher and lower operating modes, respectively, and H_H and H_L are the graph iteration period in the higher and lower operating modes, respectively, as defined in Equation (2.14). Finally, by substituting Equation (5.7) and Equation (5.8) in Equation (5.2) and setting $\mathcal{R}_{eff} = \mathcal{R}_{req}$, the number of graph iterations to stay in the higher operating mode, N_H, can be derived as follows:

$$N_H = \left\lceil \frac{H_L N_L(\mathcal{R}_{req} - \mathcal{R}_L) + \mathcal{R}_{req}(o_{HL} + o_{LH})}{H_H(\mathcal{R}_H - \mathcal{R}_{req})} \right\rceil. \tag{5.9}$$

Note that, in the above equation, the ceiling function is used to derive an integer value for N_H such that the effective throughput $\mathcal{R}_{eff}$ can still satisfy the throughput requirement $\mathcal{R}_{req}$. This fact is shown in Figure 5.5(c) where our proposed effective throughput $\mathcal{R}_{eff}$ is higher than the throughput requirement $\mathcal{R}_{req}$. Using Equation (5.9), we have to derive the pair of N_H and N_L that satisfies the throughput requirement $\mathcal{R}_{req}$. Clearly, Equation (5.9) has more than one solution for the pair of N_H and N_L. Since all of these solutions have the same timing requirement, i.e., throughput requirement, the energy reduction is equivalent with the power reduction. Therefore, to find the less power consuming solution that consequently results in the less energy consumption, we can see from Equation (5.3) that less power is consumed when we have an arbitrarily large period λ. This is because, the contribution of the switching power consumption $\frac{e_{HL}+e_{LH}}{\lambda}$ becomes negligible in the total power consumption P_{eff}. Moreover, as the period λ is enlarged, the delivered effective throughput $\mathcal{R}_{eff}$ using our switching scheme becomes closer to the throughput requirement $\mathcal{R}_{req}$. This is because, as N_L increases in Equation (5.9), the ceiling function becomes less contributing and the pair of N_L and N_H can produce the effective throughput $\mathcal{R}_{eff}$ more closely to the throughput requirement $\mathcal{R}_{req}$. As a result, this leads to exploiting static slack

Algorithm 3: Finding the least power consuming pair of N_H and N_L.

Input: $\mathcal{R}_{req}$, SI^H, SI^L.
Output: N_L, N_H.

```
1 Prev_Power = +∞;
2 N_L = 1;
3 while True do
4     Calculate N_H using Equation (5.9) and R_req;
5     Power = Calculate power consumption by using Equation (5.3);
6     if (Prev_Power − Power)/Prev_Power × 100 < 1 then
7         return N_L, N_H;
8     Prev_Power = Power;
9     N_L = N_L + 1;
```

time in the application scheduling more efficiently leading to further power reduction. Therefore, to find a valid solution for N_H and N_L which satisfies Equation (5.9) and reduces the power consumption significantly, we search for the largest N_L where if it is further enlarged, the power reduction diminishes to less than one percent.

Algorithm 3 presents the pseudo-code of finding the least power consuming pair of N_H and N_L. The inputs of this algorithm are the throughput requirement and the higher and lower operating modes. The output of this algorithm is the pair of N_H and N_L. First, we initialize $N_L = 1$ in Line 2 and compute the corresponding N_H using Equation (5.9) in Line 4. Then, we compute the power consumption corresponding to the derived pair of N_H and N_L using Equation (5.3) in Line 5. We repeat this procedure by incrementing N_L in Line 9 until further power reduction compared to the previous iteration becomes less than one percent. This condition to terminate the procedure is given in Line 6. Then, the pair N_H and N_L is returned by the algorithm.

5.6.4 Memory Overhead

In this section, we compute the memory overhead that our approach incurs to the system, that is, the input and output buffers shown in Figure 5.6. In order to compute the output buffer, we should consider Figure 5.5(c) which shows the variable rate of token production $Z(t)$ delivered by our scheduling approach (the solid curve) and the needed constant rate of token production $\mathcal{R}_{eff}$ (the dotted line). When the application executes in the higher operating

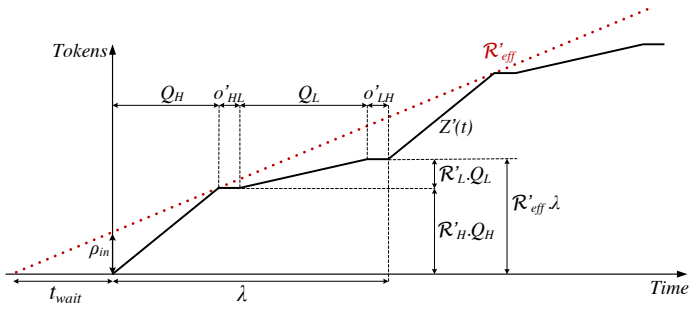


Figure 5.7: *Token consumption function $Z'(t)$. Note that, $o_{HL} + o_{LH} = o'_{HL} + o'_{LH} = \delta^{H\to L} + \delta^{L\to H}$.*

mode, it produces more tokens than needed while in the lower operating mode it produces less tokens than needed. Therefore, the purpose of using the output buffer is to accumulate the maximum difference between the number of produced and needed tokens over time. This maximum difference is given by ρ_{out} in Figure 5.5(c). Therefore, the size of the output buffer must be at least

$$B_{out} = \left\lceil \rho_{out} \right\rceil = \left\lceil Q_H(\mathcal{R}_H - \mathcal{R}_{eff}) \right\rceil \tag{5.10}$$

To compute the input buffer, the same method as for the output buffer can be used. To do so, we should consider Figure 5.7 which shows the rate of sampling data tokens $Z'(t)$ in our scheduling approach given by the solid curve. As can be seen, the application samples the data tokens in the higher operating mode more often than in the lower operating mode. To solve such irregular sampling of the input data tokens over the time, we introduce a constant rate of sampling data tokens $\mathcal{R}'_{eff}$ give by the dotted line in Figure 5.7 for the application and we compute it as follows:

$$\mathcal{R}'_{eff} = \frac{\mathcal{R}'_H Q_H + \mathcal{R}'_L Q_L}{Q_H + Q_L + o'_{HL} + o'_{LH}} \tag{5.11}$$

where $\mathcal{R}'_H$ and $\mathcal{R}'_L$ are the throughput of the input task in the higher and lower operating modes, $\mathcal{R}'_H Q_H$ and $\mathcal{R}'_L Q_L$ are the number of sampled data tokens from the input data stream in the higher and the lower operating modes, and o'_{HL} and o'_{LH} are the time overhead for the input task where no input data stream is sampled during switching from the higher to lower operating mode and vice versa, respectively. These time overheads are equal to the offset δ computed using Equation (2.20). Apparently, the constant sampling rate of $\mathcal{R}'_{eff}$ has to always provide sufficient sampled data tokens in both

operating modes. Thus, to be able to guarantee this feature, the sampling of the input data stream with the rate of $\mathcal{R}'_{eff}$ must be started t_{wait} time units before the application starts executing, as shown in Figure 5.7. This time can be computed as follows:

$$t_{wait} = \frac{(\mathcal{R}'_H - \mathcal{R}'_{eff})Q_H}{\mathcal{R}'_{eff}} \tag{5.12}$$

Finally, the size of the input buffer must be at least

$$B_{in} = \left\lceil \rho_{in} \right\rceil = \left\lceil t_{wait}\mathcal{R}'_{eff} \right\rceil = \left\lceil Q_H(\mathcal{R}'_H - \mathcal{R}'_{eff}) \right\rceil \tag{5.13}$$

where ρ_{in} is the maximum difference between the number of sampled and needed tokens, as shown in Figure 5.7.

5.7 Experimental Evaluation

In this section, we evaluate the effectiveness of our scheduling approach in terms of energy reduction. We compare our proposed scheduling approach, referred as Switching, in terms of energy reduction with two related approaches: the straightforward approach of always selecting the operating mode whose throughput is the closest higher to the throughput requirement, referred as Higher mode, and the period scaling approach, referred as Scale, explained in Section 5.5.1, which is the way of using the VFS mechanism similar to the related works [74,96] in the context of the SPS framework [8]. In the following, we first explain our experimental setup in Section 5.7.1. Then, we present the experimental results in the Section 5.7.2.

5.7.1 Experimental Setup

Applications

We have performed experiments on a set of six real-life streaming applications collected from the StreamIt benchmark suit [88], the SDF^3 suit [84] and the individual research article [69], where all streaming applications are modeled as CSDF graphs. An overview of all streaming applications is given in Table 5.2. In this table, $|\mathcal{A}|$ denotes the number of actors in a CSDF graph, while $|\mathcal{E}|$ denotes the number of FIFO communication channels among actors.

Table 5.2: *Benchmarks used for evaluation.*

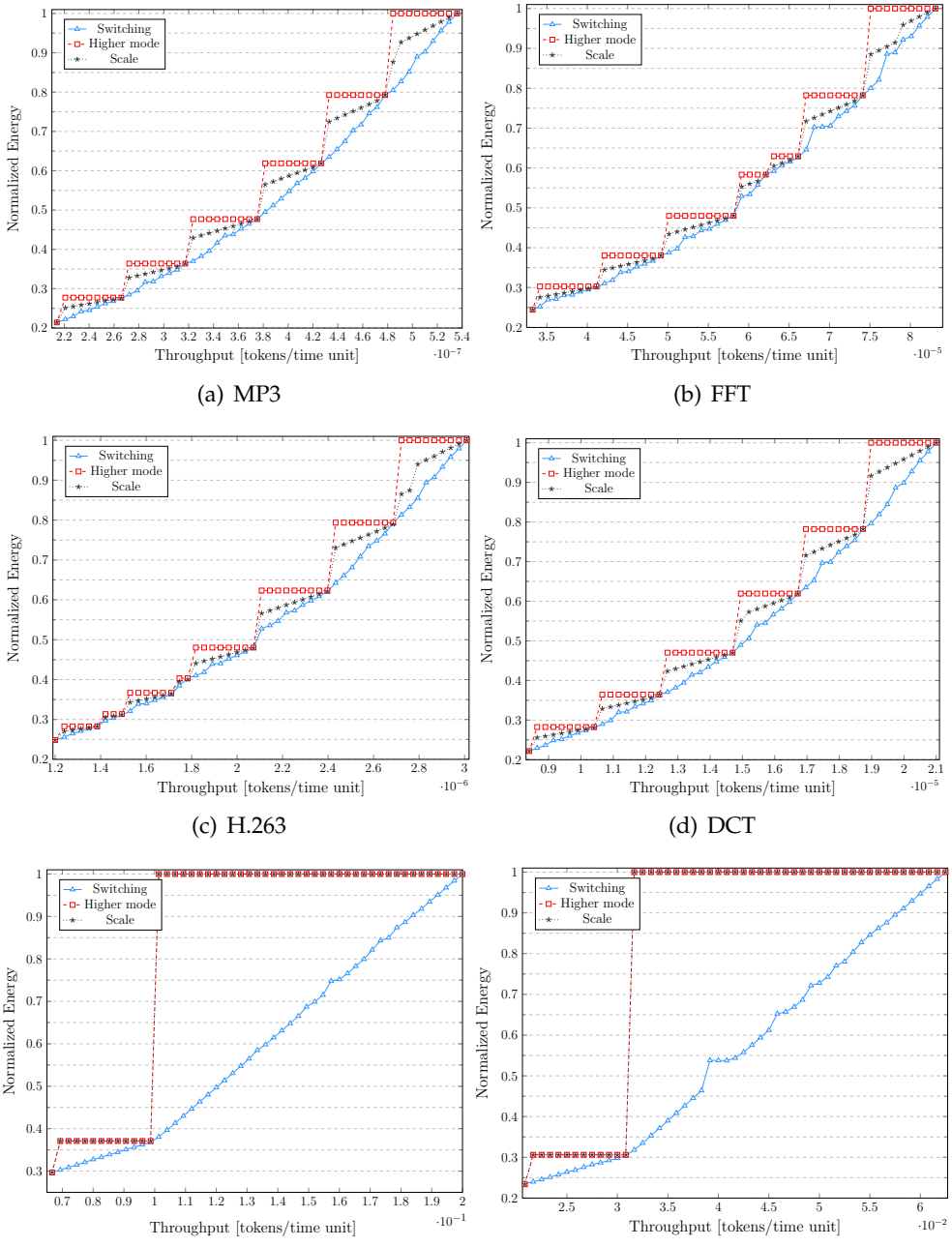
Application	$\lvert\mathcal{A}\rvert$	$\lvert\mathcal{E}\rvert$	Source
Discrete cosine transform (DCT)	8	7	[88]
Fast Fourier transform (FFT)	17	16	[88]
Data modem	6	5	[84]
MP3 audio decoder	14	18	[84]
H.263 video decoder	4	3	[84]
Heart pacemaker	4	3	[69]

Architecture and Power Model

In the experiments, we use the power model presented in Section 5.4.2. In this model, we adopt the power parameters of the Cortex A15 core given in [55], where these parameters have been obtained based on real measurements on the ODROID XU-3 platform [66]. The overhead of the DVFS mechanism is set to values taken from [67], i.e., $10\mu s$ and $1\mu J$ are used for the delay and energy overhead associated with the physical change of the operating frequency in processors, respectively. We evaluate the effectiveness of our scheduling approach on platforms with limited number of processors. To this end, we compute the minimum number of processors needed to schedule each application using FFD-EDF when the maximum achievable throughput under the SPS framework is required.

5.7.2 Experimental Results

All experimental results are shown in Figure 5.8 and Figure 5.9, where the comparison is made for a set $\mathcal{R}_{app}$ of selected application throughputs as throughput requirements. In Figure 5.8, we show the different throughput requirements for the applications on the x-axis and the normalized energy consumption of all three approaches is shown on the y-axis. As can be seen in Figure 5.8, the energy reduction varies considerably among different applications and throughput requirements. When compared to the approach Higher mode, our proposed approach Switching achieves significant energy reduction for all applications. This energy reduction for the Modem, Pacemaker, DCT, MP3, FFT, and H.263 applications can be up to 68.18%, 61.94%, 21.14%, 22.4%, 19.9%, and 19%, respectively. Compared to the approach Scale, our approach Switching can still reduce the energy consumption considerably. This energy reduction for the Modem, Pacemaker, DCT, MP3, FFT, and H.263 applications can be up to 68.18%, 61.94%, 13.1%, 13.78%, 10.7%, and 12.07%, respectively. Among all these applications, the Modem and Pacemaker are the two applica-



(a) MP3 (b) FFT

(c) H.263 (d) DCT

(e) Pacemaker (f) Modem

Figure 5.8: *Normalized energy consumption vs. throughput requirements.*

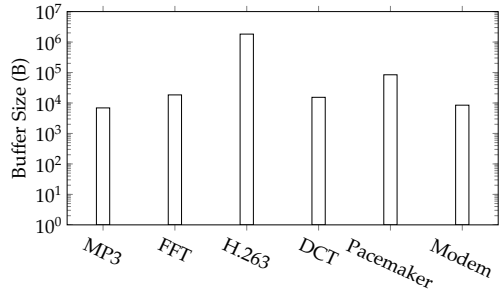


Figure 5.9: *Total buffer sizes needed in our scheduling approach for different applications. Note that the y axis has a logarithmic scale.*

tions for which our approach can obtain the largest energy reduction when compared to the approach Scale. This is mainly because the period of the tasks in Pacemaker and Modem applications are quickly increased by applying the *period scaling* approach, explained in Section 5.5.1. Therefore, a fewer number of operating modes can be determined for these applications and no other application scheduling remains between the operating modes. As a consequence, the same application scheduling as the approach Higher mode is selected in the approach Scale to satisfy the throughput requirement in these applications. This fact can be seen in Figure 5.8 for Pacemaker and Modem applications in which the result of the approach Scale and the approach Higher mode are overlapped on each other.

As can be seen in Figure 5.8, for some throughput requirements no energy reduction is achieved by our approach Switching compared to approach Higher mode and approach Scale. This happens when the throughput requirements match with the throughput of one of the operating modes. In such cases, we simply select the operating mode whose throughput matches with the throughput requirement because mode switching is not needed.

Finally, the memory overhead, discussed in Section 5.6.4, introduced by our scheduling approach, is given in Figure 5.9. In this figure, the x-axis shows the different applications while the y-axis shows the buffer size for each application which is calculated as follows:

$$B_{app} = \max_{\mathcal{R}_i \in \mathcal{R}_{app}} (B^i_{in} + B^i_{out})$$

where B^i_{in} and B^i_{out} are the size of the input and output buffers shown in Figure 5.6, computed by using Equation (5.13) and Equation (5.10), respectively, for a required application throughput $\mathcal{R}_i$. In this regard, the memory overhead for the H.263 application is 1.7 MB whereas for the other applications it is

less than 83 KB. Given such memory overhead and given the size of memory available in modern embedded systems, we can conclude that the memory overhead introduced by our scheduling approach is acceptable.

5.8 Conclusions

In this chapter, we have proposed a novel energy-efficient periodic scheduling approach for streaming applications. This approach can satisfy a system throughput requirement at a long run by periodically switching the application schedule between two selected schedules, referred as operating modes. Contrary to related approaches, our scheduling approach benefits from using multiple voltage and frequency levels at run-time leading to more efficient static slack time utilization while the throughput requirement is still satisfied. The experimental results, on a set of six real-life streaming applications, show that our approach can reduce the energy consumption by up to 68% while satisfying the same throughput requirement when compared to related approaches. However, for some throughput requirements that match with the throughput of one of the operating modes, no energy reduction can be achieved by our approach compared to the related approaches. This is because, in such cases, we can simply select the operating mode which throughput matches with the throughput requirement instead of adopting the mode switching scheme. Finally, although the throughput requirement of the applications is satisfied by our proposed approach, the mentioned energy reductions come at the expense of increased memory requirements.

Chapter 6

Implementation and Execution of Adaptive Streaming Applications

Sobhan Niknam, Peng Wang, Todor Stefanov. "On the Implementation and Execution of Adaptive Streaming Applications Modeled as MADF". *In Proceedings of the 23rd International Workshop on Software and Compilers for Embedded Systems (SCOPES)*, Sankt Goar, Germany, May 25-26, 2020.

Jiali Teddy Zhai, **Sobhan Niknam**, Todor Stefanov. "Modeling, Analysis, and Hard Real-time Scheduling of Adaptive Streaming Applications". *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 37, No. 11, pp. 2636-2648, Nov 2018.

IN this chapter, we present our implementation and execution approach for adaptive streaming applications modeled as MADF graphs, which corresponds to the fourth research contribution, briefly introduced in Section 1.5.4, to address the research question **RQ3**, described in Section 1.4.3. The remainder of the chapter is organized as follows. Section 6.1 introduces, in more details, the problem statement and the addressed research question. It is followed by Section 6.2, which gives a summary of the contributions presented in this chapter. Section 6.3 gives an overview of the related work. Section 6.4 introduces an extra background material, on K-Periodic Schedules, needed for understanding the contributions of this chapter. Section 6.5 presents our extension of the MOO transition protocol (described in Section 2.1.2 and Section 2.4) followed by Section 6.6 presenting our proposed parallel implementation and

execution approach for the MADF MoC. Section 6.7 presents two case studies to demonstrate the practical applicability of our approach, presented in Section 6.6. Finally, Section 6.8 ends the chapter with conclusions.

6.1 Problem Statement

Recall, from Section 1.4.3, that the last phase of the design flow, considered in this thesis and shown in Figure 1.2, is to implement and execute the analyzed application on an MPSoC platform. This phase is an important step towards designing an embedded streaming system where the system should behave at run-time as expected according to the performed analysis at design-time. Concerning static streaming applications, an implementation and execution approach for such applications modeled as CSDF graphs and analyzed by the SPS framework, briefly described in Section 2.3, is presented in [7]. For adaptive streaming applications, modeled and analyzed with the MADF MoC [94], briefly described in Section 2.1.2, however, no attention has been paid so far at this implementation phase. Thus, in this chapter, we investigate the possibility to implement and execute an adaptive streaming application, modeled and analyzed with the MADF MoC, on an MPSoC platform, such that the properties of the analyzed model are preserved.

6.2 Contributions

In order to address the problem described in Section 6.1, in this chapter, we propose a simple, yet efficient, parallel implementation and execution approach for adaptive streaming applications, modeled with the MADF model, that can be easily realized on top of existing operating systems. Moreover, we extend the offset calculation of the MOO transition protocol, briefly described in Section 2.4, for the MADF model in order to enable the utilization of a wider range of schedules, i.e., K-periodic schedules [17], during the model analysis, implementation, and execution depending on the scheduling support provided by the MPSoC and its operating system onto which the streaming application runs.

More specifically, the main contributions of this chapter are as follows:

- We extend the MOO transition protocol employed by the MADF model. This extension enables the applicability of many different schedules to the MADF model, thereby generalizing the MADF model and making

MADF schedule-agnostic as long as K-periodic schedules are considered;

- We propose a generic parallel implementation and execution approach for adaptive streaming applications modeled with MADF that conforms to the analysis model and its operational semantics [94]. We demonstrate our approach on LITMUS^{RT} [22] which is one of the existing real-time extensions of the Linux kernel;
- Finally, to demonstrate the practical applicability of our parallel implementation and execution approach and its conformity to the analysis model, we present a case study (see Section 6.7.1) on a real-life adaptive streaming application. In addition, we present another case study (see Section 6.7.2) on a real-life streaming application to validate our proposed energy-efficient periodic scheduling approach, presented in Chapter 5, which adopts the MOO protocol of the MADF MoC for switching the application schedule, with a practical implementation of this approach by using our generic parallel implementation and execution approach presented in this chapter.

6.3 Related Work

In [60], the MCDF model is presented where the same application graph is used for both analysis and execution on a platform. In such graph, special actors, namely switch and select actors, are used to enable reconfiguration of the graph structure according to an identified mode by a mode controller at run-time. In the MCDF model, every mode is represented as a single-rate SDF graph and the actors are scheduled on each processor according to a precomputed static schedule, called quasi-static order schedule, in which extra switch and select actors are required to model the schedule in the graph. In contrast to MCDF, the MADF model [94], we consider in our work, is more expressive as each mode is represented as a CSDF graph. Moreover, our proposed MOO transition protocol extension and our implementation and execution approach for the MADF model are schedule agnostic and do not require extra switch and select actors. Therefore, our approach enables the utilization of many different schedules than only a static-order schedule, with no need of extra actors.

In [33], the FSM-SADF model is presented as another analysis model for adaptive streaming applications. To implement an application modeled and analyzed with FSM-SADF, two programming models have been proposed in [89, 90]. In [89], the programming model is constructed by merging the SDF

graphs of all scenarios into a single graph which may be larger than the FSM-SADF analysis graph. Then, to enable switching to a new scenario, all actors in all scenarios are constantly kept active while only those actors belonging to the identified new scenario by a detecting actor(s) will be executed after switching. In this way, a single static-order schedule can be used for the application in all scenarios. In contrast to [89], the proposed programming model in [90] uses a similar switch/select actors, as in MCDF [60], in the constructed graph for switching between scenario graphs at run-time. Then, the graph is reconfigured at run-time using the switch/select actors according to the identified scenario by a detecting actor(s) while updating the application's static-order schedule accordingly. However, the proposed programming models in [89,90] need to be derived manually, thereby requiring extra effort by the designer. More importantly, these programming models assume that actors in all scenarios of an application are active all the time. This can result in a huge overhead for applications with a high number of modes, thereby leading to inefficient resource utilization. In contrast to [89,90], our implementation and execution approach does not require derivation of an additional model and enables the utilization of many different schedules rather than only static-order schedule. Moreover, our approach (de)activates actors in different modes at run-time, so we do not need to keep all modes active all the time, thereby avoiding the unnecessary overhead imposed by the approaches in [89,90].

In [47], the task allocation of adaptive streaming applications onto MPSoC platforms under self-timed (ST) scheduling is studied when considering transition delay during mode transitions. In [47], however, the verification of the proposed approach and mode transition mechanism is limited to simulations and no implementation and execution approach is provided. In contrast, in this chapter, we propose a generic parallel implementation and execution approach for applications modeled with MADF which enables the applicability of many different schedules on the application as well as execution of the application on existing operating systems.

6.4 K-Periodic Schedules (K-PS)

In [19], K-periodic schedules (K-PS) of streaming applications modeled as CSDF graphs are introduced, implying that K_i consecutive invocations of an actor $A_i \in \mathcal{A}$ occur periodically in the schedule. For example, when $K_i = q_i$ for every actor $A_i \in \mathcal{A}$, such K-PS is equivalent to a ST schedule [85] where all q_i invocations of the actor A_i in one graph iteration occur in each period and can result in the maximum throughput for a given CSDF graph. On the other

hand, when $K_i = 1$ for every actor $A_i \in \mathcal{A}$, 1-PS is achieved in which only a single invocation of the actor occurs in each period. The SPS schedule [8], briefly described in Section 2.3, is a special case of 1-PS in which the actors are converted to real-time tasks to enable the application of classical hard real-time scheduling algorithms [29], e.g., EDF, to streaming applications modeled as CSDF graphs. Therefore, in general, the K-PS notion covers a wide set of schedules ranging between 1-PS and ST schedules.

6.5 Extension of the MOO Transition Protocol

As explained in Section 2.4, when multiple actors of an application, modeled as an MADF graph, are allocated on the same processor, the processor can be potentially overloaded during mode transitions due to simultaneous execution of actors from different modes. Therefore, a larger offset, than the offset x computed by using Equation (2.4), may be needed by the MOO protocol to delay the starting time of the new mode during a mode transition in order to avoid processor overloading. Then, this offset, represented with δ, is computed under the SPS schedule by using Equation (2.20). As the SPS schedule has the notion of a task utilization, by converting the actors in a CSDF graph to real-time (RT) tasks, the offset δ is computed, according to Equation (2.20), by making the total utilization of the RT tasks allocated on each processor during mode transition instants to not exceed the processor capacity. However, since the K-periodic schedules (K-PS), considered in this chapter and briefly introduced in Section 6.4, have no notion of a task utilization, the offset δ for any K-PS cannot be computed as in Equation (2.20). Therefore, in this section, we extend the MOO transition protocol to compute such an offset for any K-PS.

In fact, to avoid the processor overloading under any K-PS, the schedule interferences of modes (in terms of overlapping iteration period H) during mode transitions must be resolved on each processor. For instance, consider the MADF graph G_1 in Figure 6.1(a), explained in Section 2.1.2, with two operating modes SI^1 and SI^2. Figure 6.2(a) and Figure 6.2(b) show the corresponding CSDF graphs of modes SI^1 and SI^2, respectively. An execution of both modes SI^1 and SI^2 under a K-PS are shown in Figure 6.3(a) and Figure 6.3(b), respectively, as well as an execution of G_1 with two mode transitions and the computed offsets $x^{1\rightarrow 2} = 3$ and $x^{2\rightarrow 1} = 1$, for mode transitions from SI^1 to SI^2 and vice versa, according to Equation (2.4), is illustrated in Figure 6.4(a). Now, let us assume the allocation of all actors of G_1 on an MPSoC platform $\Pi = \{\pi_1, \pi_2, \pi_3, \pi_4\}$ containing four processors that is shown in Figure 6.1(b).

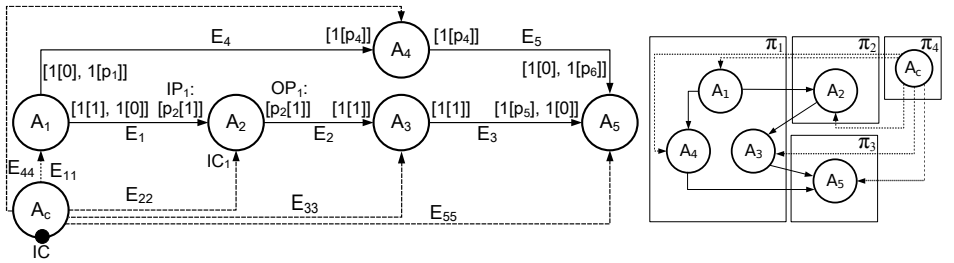
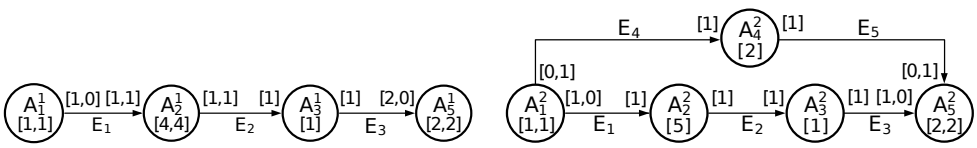


Figure 6.1: *(a) An MADF graph G_1 (taken from Section 2.1.2). (b) The allocation of actors in graph G_1 on four processors.*



(a) CSDF graph G_1^1 of mode SI^1. (b) CSDF graph G_1^2 of mode SI^2.

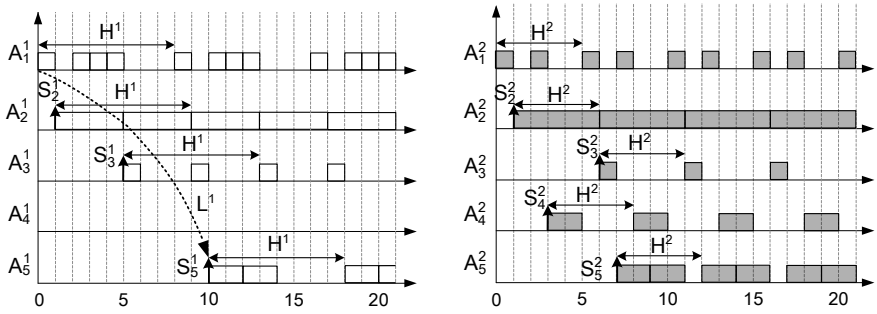
Figure 6.2: *Two modes of graph G_1 in Figure 2.1 (taken from Section 2.1.2 with modified WCET of the actors).*

Then, considering the execution of G_1 in Figure 6.4(a), the schedule interferences on π_1 happen during time periods $[6, 11]$ and $[25, 27]$ for mode transition from SI^2 to SI^1 and vice versa, respectively, while no schedule interference happens on π_2 and π_3. Obviously, to resolve the schedule interferences on π_1, the earliest start time of actors in the new mode should be further offset by the length of the time period in which the schedule interferences happen. Therefore, the extra offsets for mode transitions from SI^2 to SI^1 and vice versa on π_1 are $11 - 6 = 5$ and $27 - 25 = 2$ time units, respectively, thereby resolving the schedule interferences on π_1, as shown in Figure 6.4(b). In this example, $\delta^{2\to1} = x^{2\to1} + 5 = 6$ and $\delta^{1\to2} = x^{1\to2} + 2 = 5$.

Now, considering any K-PS, the offset $\delta^{o\to n}$ can be computed as the **maximum schedule overlap** among all processors when the new mode SI^n starts immediately after the source actor of the old mode SI^o completes its last iteration, as follows:

$$\delta^{o\to n} = \max\{x^{o\to n}, \max_{\substack{\forall\, {}^m\Psi_i^o \in {}^m\Psi^o \wedge {}^m\Psi_i^n \in {}^m\Psi^n \\ {}^m\Psi_i^o \neq \emptyset \wedge {}^m\Psi_i^n \neq \emptyset}} (\max_{A_j^o \in \Psi_i^o} S_j^o - \min_{A_k^n \in {}^m\Psi_i^n} S_k^n)\} \tag{6.1}$$

where ${}^m\Psi = \{{}^m\Psi_1, \ldots, {}^m\Psi_m\}$ is m-partition of all actors on m number of pro-



(a) Mode SI^1 in Figure 6.2(a) (b) Mode SI^2 in Figure 6.2(b)

Figure 6.3: *Execution of both modes SI^1 and SI^2 under a K-PS.*

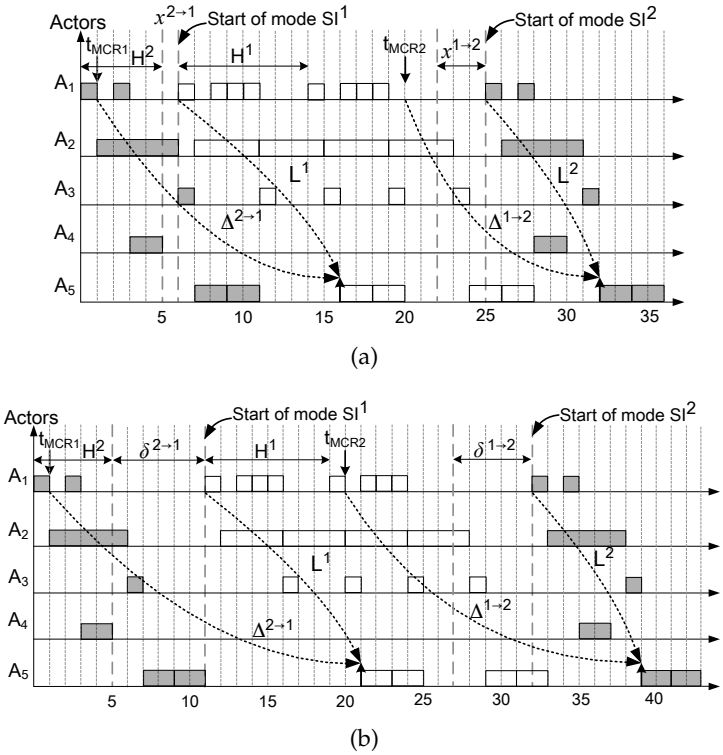


Figure 6.4: *Execution of G_1 with two mode transitions under (a) the MOO protocol, and (b) the extended MOO protocol with the allocation shown in Figure 6.1(b).*

cessors, i.e., ${}^m\Psi_i^o$ and ${}^m\Psi_i^n$ are the sets of actors allocated on the i-th processor (π_i) in the old mode SI^o and the new mode SI^n, respectively. For instance,

consider the allocation of G_1 on the four processors, shown in Figure 6.1(b), and the K-PS of modes SI^1 and SI^2 given in Figure 6.3(a) and 6.3(b), respectively. The offset $\delta^{1\rightarrow 2}$ of the mode transition from SI^1 to SI^2 on each processor is computed using Equation (6.1) as follows: (π_1) $S_3^1 - S_1^2 = 5 - 0 = 5$, (π_2) $S_2^1 - S_2^2 = 1 - 1 = 0$, and (π_3) $S_5^1 - S_5^2 = 10 - 7 = 3$, thereby resulting in the offset $\delta^{1\rightarrow 2} = \max(3, \max(5, 0, 3)) = 5$ for the start time of mode SI^2, as shown in Figure 6.4(b). Similarly, the offset $\delta^{2\rightarrow 1}$ of the mode transition from SI^2 to SI^1 on each processor is computed using Equation (6.1) as follows: (π_1) $S_3^2 - S_1^1 = 6$, (π_2) $S_2^2 - S_2^1 = 0$, and (π_3) $S_5^2 - S_5^1 = -3$, and $\delta^{2\rightarrow 1} = \max(1, \max(6, 0, -3)) = 6$.

6.6 Implementation and Execution Approach for MADF

In this section, we first present our generic parallel implementation and execution approach (Section 6.6.1) for an application modeled as an MADF. Then, in Section 6.6.2, we demonstrate our approach on LITMUS^{RT} [22].

6.6.1 Generic Parallel Implementation and Execution Approach

In this section, we will explain our approach by an illustrative example. Consider the MADF graph G_1 shown Figure 6.1(a). Our implementation consists of three main components: 1) (normal) actors, 2) a control actor, and 3) FIFO channels. We implement the actors as separate threads and the FIFO channels as circular buffers [15] with non-blocking read/write access. Thus, the execution of the threads and the read/write from/to the FIFO channels are controlled explicitly by an operating system supporting and using any K-PS, briefly introduced in Section 6.4. A valid K-PS schedule always ensures the existence of sufficient data tokens to read from all input FIFO channels and sufficient space to write data tokens to all output FIFO channels when an actor executes.

In our implementation, all FIFO channels in the MADF graph of an application are created statically before the start of the application execution to avoid duplication of FIFO channels and unnecessary use of more memory during mode transitions. On the other hand, the threads corresponding to the actors are handled at run-time. This means that when a mode change request (MCR) occurs, in order to switch the application's mode, the executing threads in the old mode are stopped and terminated whereas the threads corresponding to the actors in the requested new mode are created and launched at run-time. In this way, our implementation enables task migration during mode transitions

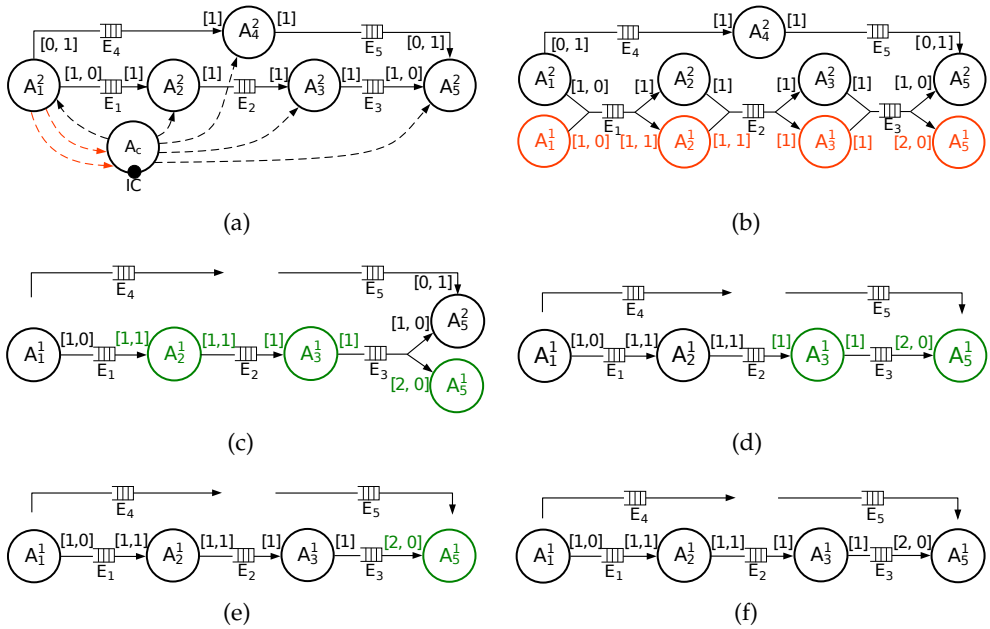


Figure 6.5: *Mode transition of G_1 from mode SI^2 to mode SI^1 (from (a) to (f)). The control actor and the control edges are omitted in figures (b) to (f) to avoid cluttering.*

by using a different task allocation in each application's mode. For instance, the implementation and execution of the mode transition from mode SI^2 to mode SI^1 of G_1, with the given schedule in Figure 6.4(b), is shown in Figure 6.5 and has the following sequence - Figure 6.5(a): The application is in mode SI^2 where the threads corresponding to the actors in this mode run. The threads are connected to the control thread A_c, which runs on a separate processor, through the control FIFO channels (the dashed arrows in Figure 6.5(a)). In our approach, two extra FIFO channels, shown in the red color in Figure 6.5(a), are required, both from the thread of source actor A_1 to control thread A_c in order to notify the control thread in which graph iteration number the source actor is currently running and the time when the thread of the source actor is terminated; Figure 6.5(b): When MCR1 occurs at time instant $t_{\text{MCR1}} = 1$ to switch to mode SI^1, the threads corresponding to the actors in mode SI^1 are created and connected to the corresponding FIFO channels. At this stage the newly-created threads (the red nodes in Figure 6.5(b)) are suspended and they wait to be released. *Note that the mode transitions cannot be performed at any moment. According to the operational semantics of the MADF model, a mode transition is only allowed in a consistent state, that is, after the graph iteration in*

which the MCR occurred, has completed and the graph has returned to its initial state. Therefore, control thread A_c needs to check the current graph iteration number of the source actor A_1^2 and notify all threads at which graph iteration number they have to be terminated; Figure 6.5(c): Next, when the thread of the source actor A_1^2 is terminated at time instant 5 (according to Figure 6.4(b)), which is notified to control thread A_c as well, the control thread signals the suspended threads to be released synchronously $\delta^{2\rightarrow 1} = 6$ time units later at time instant 11 (according to Figure 6.4(b)). At this stage, a mixture of threads in both modes may be running on processors. In the meanwhile, the threads of the actors in the old mode SI^2 are gradually finishing their execution and terminated at the same graph iteration number; Figure 6.5(d)-6.5(f): Since the actors have different start time in the new mode SI^1, as shown in Figure 6.4(b), the threads in mode SI^1 start executing accordingly after the releasing time. The threads which are released but not yet running, are shown in the green color. Then, the released threads in the new mode SI^1 gradually start running and finally, the application is switched to mode SI^1 where all created threads run and the unused channels E_4 and E_5 in this mode are left unconnected to the threads.

6.6.2 Demonstration of Our Approach on LITMUS^{RT}

In this section, we demonstrate how to realize our implementation and execution approach on LITMUS^{RT} [22] as one of the existing real-time (RT) extensions of the Linux kernel. The realizations of a normal actor and the control actor in our approach are given in C++ in Listing 6.1 and 6.2, respectively, in which the bolded primitives belong to LITMUS^{RT}. Note that, any other RT operating system which has similar primitives, e.g., FreeRTOS [72], can be used instead. We also use the standard POSIX Threads (Pthreads) and the corresponding API integrated in Linux to create the threads of the actors.

In Listing 6.1, the RT parameters of an actor, e.g., actor A_2 of graph G_1 shown in Figure 6.1(a), are set up using the data structure `threadInfo` passed to the function as argument in Lines 2-6. Under partitioned scheduling algorithms, e.g., Partitioned EDF, the processor core which the thread should be statically executing on, is set in Line 7. Then, the RT configuration of the thread is sent to the LITMUS^{RT} kernel for validation, in Line 8, in which if it is verified, the thread is admitted as a RT task in LITMUS^{RT}, in Line 9. In Line 10, the RT task is suspended, in order to synchronize the start time of the tasks, until signaled by the *control actor* to begin its execution. Next, the task enters to a `while loop` in Lines 12-31, in which iterates infinitely. At the beginning of each graph iteration, the current time instant is captured and stored in

```
1void Actor_A2(void *threadarg) {
2  threadInfo = (threadInfo *)threadarg; // Get the thread parameters
3  struct rt_task param; // Set up RT parameters
4  param.period = threadInfo.period;
5  param.relative_deadline = threadInfo.relative_deadline;
6  param.phase = threadInfo.start_time;
7  be_migrate_to_domain(threadInfo.processor_core); // For partitioned schedulers
8  set_rt_task_param(gettid(), &param));
9  task_mode(LITMUS_RT_TASK); // The actor is now executing as a RT task
10 wait_for_ts_release(); // The RT task is waiting for a release signal
11 int graph_iteration = 1;
12 while(1) { // Enter to the main body of the task
13    lt_t now = litmus_clock();
14    for(i=1; i<=threadInfo.repetition; i++){
15       lt_sleep_until(now + threadInfo.slot_offset[i]);
16       if(IC_1 is not empty) READ(& terminate, threadInfo.IC_1);
17       if(i == 1 && graph_iteration > terminate){
18          WRITE(& now, threadInfo.OC_trig);
19          task_mode(BACKGROUND_TASK); //Trans. back to non−RT mode
20          return NULL;
21       }
22       if(i == 1) WRITE(& graph_iteration, threadInfo.OC_iter);
23       if(threadInfo.mode == 1){ // Do action according to the task's mode
24          READ(& in1, threadInfo.IP_1);
25          task_function(& in1, & out1);
26          WRITE(& out1, threadInfo.OP_1);
27       }/* Actions according to the other modes */ { . . . }
28       if(i%threadInfo.K == 0) sleep_next_period();
29    }
30    graph_iteration += 1;
31 }
32}
```

Listing 6.1: *C++ code of actor* A_2

variable `now` in Line 13. Then, the task iterates as many repetition times as it has in one graph iteration in a `for loop`, in Lines 14-29. In Line 15, the task sleeps until reaching the start time of its *i*-th invocation, corresponding to the K-PS, from the time instant captured in `now`. After finishing K_i invocations, the task sleeps again, in Line 28, until finishing the current period. In fact, in this line, a kernel-space mechanism is triggered for moving the task from the ready queue to the release queue. Then, LITMUS^{RT} will move the task

```
1 void main(int argc, char **argv) {
2   /* Create FIFO channel E_1 */
3   size_E_1_in_tokens = 4;
4   size_token_E_1= sizeof(token_structure)/sizeof(int);
5   size_fifo_E_1 = size_E_1_in_tokens × size_token_E_1;
6   E_1 = calloc(size_fifo_E_1+2, sizeof(int)); // Allocate memory for E_1
7   /* Create other FIFO channels*/ {···}
8   init_litmus(); // Initialize the interface with the kernel
9   old_mode = 1, new_mode = 1;
10  while(1){
11    switch(new_mode){
12      case 1: /* Create and launch the thread of actor A_2 in mode SI^1 */
13        threadInfo.mode = 1; thread.repetition = 2; threadInfo.processor_core = 1;
14        threadInfo.IP_1 = E_1; /* Connect other FIFO channels to the thread*/ {···}
15        threadInfo.period = 8; threadInfo.relative_deadline = 8;
16        threadInfo.phase = 1; threadInfo.slot_offset = [0, 4];
17        pthread_create(&threadInfo.id, NULL, &Actor_A2, &threadInfo);
18        /* Create and launch the threads of the other actors in mode SI^1 */ {...}
19      case 2: { /* Create and launch the thread of the actors in mode SI^2 */ }
20    }
21    while(rt_task == ready_rt_tasks)
22      read_litmus_stats(&ready_rt_tasks);
23    if(new_mode != old_mode){
24      while(IC_trig is empty);
25      READ(& now, IC_trig);
26    }else now = litmus_clock();
27    release_ts(δ); old_mode = new_mode;
28    do{ READ(& new_mode, IC); } while(new_mode == old_mode)
29    READ(& graph_iteration, IC_iter);
30    t_left = H^o −(litmus_clock() − now − δ)%H^o;
31    if(t_left < t_OV) graph_iteration += ⌈(t_OV − t_left)/H^o⌉;
32    for(all active actor A_i) WRITE(& graph_iteration, OC_i);
33 }
```

Listing 6.2: *C++ code of control actor A_c*

back to the ready queue at the start time of the next period when the task will again be eligible for execution. In Line 16, the state of the input control port $\texttt{IC}_1$ is checked in which if it is not empty, the graph iteration number where the task has to be terminated is read. Then, the termination condition is checked in Line 17. If the condition holds, the mode of the thread is changed to non-RT in Line 19 and the thread is terminated in Line 20. Otherwise, the

task reads from its input FIFO channels, executes its function, and writes the result to the output FIFO channels, in Lines 23-27. Only for the source actor, the latest graph iteration number where the task is currently running and the time instant `now` are written to the output control ports $\texttt{OC}_{\texttt{iter}}$ and $\texttt{OC}_{\texttt{trig}}$, in Lines 22 and 18 highlighted with red color, respectively, which are needed by the control thread, as explained in Section 6.6.1.

In Listing 6.2, realizing control actor A_c, all FIFO channels are created and the needed memory is allocated to them using the standard `calloc()` function, in Lines 3-7. In Line 8, the interface with the LITMUS^{RT} kernel is initialized. In Lines 11-20, the data structure of `threadInfo` is initialized for each actor of the requested new mode and the corresponding threads of the actors in the new mode are created and launched. In Lines 21 and 22, the number of suspended RT tasks is checked which if is equal to the number of the actors in the new mode, they can be signaled to be released simultaneously. Therefore, in Line 27, the global release signal is sent by δ time units after receiving the time instant `now` on the input port $\texttt{IC}_{\texttt{trig}}$ from the thread of the source actor in the old mode in Line 25, implying the termination of the thread and acting as a trigger. Afterwards, the control actor continuously monitors the occurrence of a new MCR in Line 28. If an MCR occurs to a new mode which differs from the current mode, the graph iteration number in which the threads in the current mode need to be terminated is computed in Lines 29-31. The primary graph iteration number is simply the current graph iteration number of the source actor, read from the input port $\texttt{IC}_{\texttt{iter}}$ in Line 29. However, since the control actor has certain timing overhead, represented by $\texttt{t}_{\texttt{OV}}$, the primary graph iteration number needs to be revised corresponding to the time left from the current graph iteration of the source actor $\texttt{t}_{\texttt{left}}$, computed in Line 30, and $\texttt{t}_{\texttt{OV}}$, in Line 31, to ensure that all threads will be terminated in the same graph iteration number. Then, the new graph iteration number is written on the control port of all threads in the current mode in Line 32 to notify them when they have to be terminated.

6.7 Case Studies

In this section, we present two case studies using real-life streaming applications to validate the proposed implementation and execution approach in Section 6.6 as well as the proposed periodic scheduling approach in Chapter 5 by running the applications on actual hardware. We perform these case studies on the ARM big.LITTLE architecture [40], shown in Figure 1.1, including a quad-core Cortex A15 (big) cluster and a quad-core Cortex A7 (LITTLE)

Table 6.1: *Performance results of each individual mode of Vocoder.*

Mode	Analysis [94]		Implementation and execution		Number/Type of processor
	H (ms)	L (ms)	H (ms)	L (ms)	
SI^8	25	21	25	21	1 LITTLE
SI^{16}	25	19	25	19	1 big
SI^{32}	25	33	25	33	2 big
SI^{64}	25	56	25	56	3 big

cluster, that is available on the Odroid-XU4 platform [66]. The Odroid XU4 runs Ubuntu 14.04.1 LTS along with LITMUS^{RT} version 2014.2.

6.7.1 Case Study 1

In this section, we present a case study, using a real- life adaptive streaming application, to demonstrate the practical applicability of our parallel implementation and execution approach for MADF. Moreover, we show that our approach conforms to the MADF analysis model in [94] by measuring the application's performance, in terms of the achieved iteration period, iteration latency, and mode transition delay, and comparing them with the computed ones using the MADF analysis model.

In this case study, we take a real-life adaptive streaming application from the StreamIT benchmark suite [37], called Vocoder, which implements a phase voice encoder and performs pitch transposition of recorded sounds from male to female. We modeled Vocoder using the MADF graph, shown in Figure 6.6, with four modes which captures different workloads. The four modes $\{SI^8, SI^{16}, SI^{32}, SI^{64}\}$ specify different lengths of the discrete Fourier transform (DFT), denoted by $\mathsf{dl} \in \{8, 16, 32, 64\}$. Mode SI^8 ($\mathsf{dl} = 8$) requires the least amount of computation at the cost of the worst voice encoding quality among all DFT lengths. Mode SI^{64} ($\mathsf{dl} = 64$) produces the best quality of voice encoding among all modes, but is computationally intensive. The other two modes SI^{16} and SI^{32} exploit the trade-off between the quality of the encoding and the computational workload. Therefore, the resource manager of an MPSoC can take advantage of this trade-off and adjust the quality of the encoding according to the available resources, such as energy budget and number/type of processors, at run-time.

We measured the WCET of the actors in Figure 6.6 in the four modes on both big and LITTLE processors. Then, since the shortest time granularity visible to LITMUS^{RT}, i.e., the OS clock tick, is 1 millisecond (ms), the WCET of the actors are rounded up to the nearest multiple of the OS clock tick duration. This is necessary to derive the period and start time of the actors

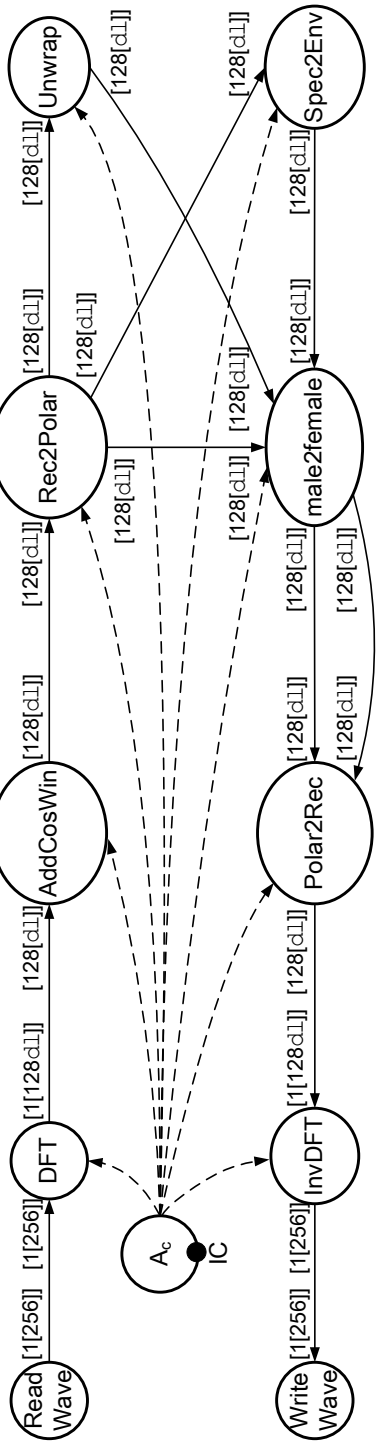


Figure 6.6: *MADF graph of the Vocoder application.*

Table 6.2: *Performance results for all mode transitions of Vocoder (in ms).*

Transition (SI^o to SI^n)	Analysis [94]		Implementation and execution
	$\Delta^{o\to n}_{min}$	$\Delta^{o\to n}_{max}$	$\Delta^{o\to n}$
$SI^8 \to SI^{64}$	146	171	160
$SI^8 \to SI^{32}$	123	148	131
$SI^8 \to SI^{16}$	111	136	122
$SI^{16} \to SI^{64}$	165	190	185
$SI^{16} \to SI^{32}$	142	167	157
$SI^{16} \to SI^8$	112	137	130
$SI^{32} \to SI^{64}$	162	187	168
$SI^{32} \to SI^{16}$	125	150	139
$SI^{32} \to SI^8$	125	150	145
$SI^{64} \to SI^{32}$	160	185	182
$SI^{64} \to SI^{16}$	146	171	162
$SI^{64} \to SI^8$	146	171	152

under any K-PS to be executed by LITMUS^{RT}. Table 6.1 shows the performance results of each individual mode under the self-timed (ST) schedule, which is a particular case of K-PS explained in Section 6.4. In this table, columns 2-3 show the iteration period H and iteration latency L of each individual application mode computed by the analysis model, respectively. The iteration period H indicates the guaranteed production of 256 samples per 25 ms, as a performance requirement, in all modes by sink actor WriteWave. Column 6 shows the number and type of processors required in each mode to guarantee the aforementioned performance requirement. On the other hand, columns 4-5 show the measured iteration period H and iteration latency L of each individual application mode achieved by our implementation and execution approach, respectively. Comparing columns 2-3 with columns 4-5, we see that the performance of Vocoder computed using the MADF analysis model is the same as the measured performance when Vocoder is implemented and executed using our approach. This is because the ST schedule of each mode is implemented in our approach by setting up, in LITMUS^{RT}, the same periods and start times of the actors as in the analysis model. Based on the results, shown in Table 6.1, we can conclude that our implementation and execution approach conforms to the MADF analysis model in terms of H and L for the Vocoder application.

Now, we focus on the performance results related to the mode transition delays for all 12 possible transitions between the four modes of Vocoder. Using the MADF analysis model in [94], the computed minimum and maximum transition delays are shown in columns 2-3 of Table 6.2, respectively. By using

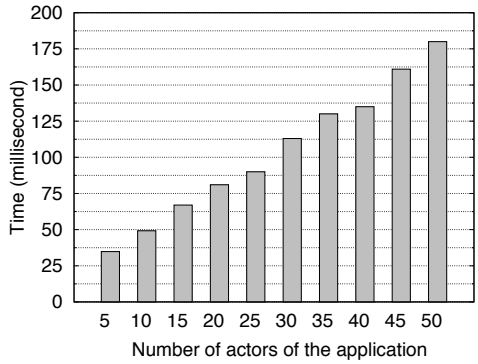


Figure 6.7: *The execution time of control actor A_c for applications with different numbers of actors.*

our implementation and execution approach, however, the measured transition delay depends on the occurrence time of the mode change request (MCR) at run-time, thus the measured transition delay could vary between the computed minimum and maximum values in each transition. For instance, column 4 in Table 6.2 shows the measured transition delay for each transition with a random occurrence time of an MCR, within the iteration period, at run-time. These measured transition delays (column 4) are within the computed bounds using the analysis model (columns 2-3). Therefore, our implementation and execution approach also conforms to the MADF analysis model in terms of mode transition delay $\Delta^{o \to n}$ for the Vocoder application.

Finally, we evaluate the scalability of our proposed implementation and execution approach in terms of the execution time t_{ov} of the control actor for applications with different numbers of actors. Since the most time-consuming and variable part of the control actor is located in Lines 11 to 22 of Listing 6.2, that is the time needed for the threads creation and the threads admission as RT tasks, we only measure the time needed for this part of the control actor. In this regard, the measured time for applications with a varying number of actors is shown in Figure 6.7. In this figure, we can clearly observe that the execution time of the control actor follows a fairly linear scalability when the number of actors in the application increases.

6.7.2 Case Study 2

In this section, we present a case study, using a real-life streaming application, for our energy-efficient periodic scheduling approach presented in Chapter 5. As explained in Chapter 5, this scheduling approach primarily selects a set

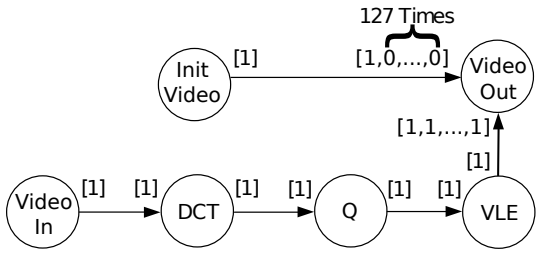


Figure 6.8: *CSDF graph of MJPEG encoder.*

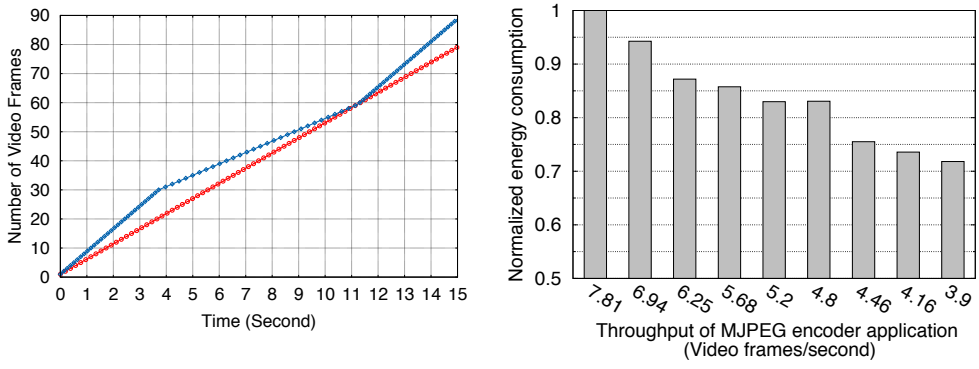
of SPS schedules, as **operating modes**, for an application modeled as a CSDF graph where each mode provides a unique pair of performance and power consumption. Then, it satisfies a given throughput requirement at a long run by switching the application's schedule periodically between modes at run-time. As this scheduling approach is evaluated using only simulations in Chapter 5, this case study aims to validate its applicability on a real hardware platform using our parallel implementation and execution approach presented in Section 6.6. To do so, we only adopt the ARM Cortex A15 cluster with four processors available on the Odroid-XU4 platform. This platform provides the DVFS mechanism per cluster in which the operating frequency of the Cortex-A15 cluster can be varied between 200 MHz to 2 GHz with a step of 100 MHz.

In this case study, we take the Motion JPEG (MJPEG) video encoder application which CSDF graph is shown in Figure 6.8. The specifications of two modes where the SPS schedule is used in each mode of this application, referred as mode SI^1 and mode SI^2, are given in Table 6.3. The iteration period H of these modes, in milliseconds, is given in the second column in Table 6.3. Mode SI^1 has an iteration period of 128 ms which results in the application throughput of $1000/128 = 7.81$ frames/second. Likewise, the iteration period of mode SI^2 is 256 ms which results in the application throughput of $1000/256 = 3.9$ frames/second. In these modes, the operating frequency of the A15 cluster is set to 1.4 GHz and 600 MHz for mode SI^1 and SI^2, respectively, while satisfying their aforementioned application throughput. As a result, these modes have different power consumption which is given in the fourth column in Table 6.3. The WCETs of all actors in these mode are also given in the fifth to tenth columns in Table 6.3. In these modes, we use the partitioned EDF scheduler plugin (PSN-EDF) in LITMUS^{RT} to schedule the actors allocated on each processor separately.

Note that modes SI^1 and SI^2 correspond to two consecutive SPS schedules

Table 6.3: *The specification of modes SI^1 and SI^2 in MJPEG encoder application*

Mode	Iteration Period (ms)	Frequency (GHz)	Power (W)	WCET of actors (ms)					
				Init Video	Video In	DCT	Q	VLE	Video Out
SI^1	128	1.4	2.24	0.003	0.139	0.272	0.136	0.267	0.779
SI^2	256	0.6	1.62	0.004	0.219	0.682	0.251	0.682	1.437



(a) (b)

Figure 6.9: *(a) The video frame production of the MJPEG encoder application over time for the throughput requirement of 5.2 frames/second. (b) Normalized energy consumption of the application for different throughput requirements.*

of the MJPEG encoder application, i.e., no other valid SPS schedule exists between them. So, to satisfy a throughput requirement between 3.9 to 7.81 frames/second, the naive solution is to constantly execute the application in mode SI^1. As a consequence, the application consumes more energy due to producing more frames/second than required. In contrast, our scheduling approach, presented in Chapter 5, can satisfy the throughput requirement at a long run by periodically switching the application execution between mode SI^1 and SI^2. For instance, let us consider the throughput requirement of 5.2 frames/second. Then, Figure 6.9(a) shows the production of video frames over time by the MJPEG encoder application under our proposed scheduling approach. The red line in this figure represents the required number of frames per second according to the throughput requirement whereas the blue curve represents the measured number of produced video frames per second by our scheduling approach implemented and executed on the real hardware platform Odroid XU4. As shown in this figure, the application executes initially in mode SI^1 for about 4 seconds while producing more video frames than required. These excessive frames are accumulated in a buffer to be

consumed when the application executes in mode SI^2 with lower throughput for the next about 7 seconds. After finishing one period of the schedule at about 11 seconds, the application delivers the throughput requirement where the red line and the blue curve in Figure 6.9(a) hit each other. This execution is then repeated indefinitely.

For different throughput requirements, we also measure the energy consumption of the Odroid XU4 platform when running the application using our periodic scheduling approach. To do so, the energy consumption of the Odroid XU4 platform is $E = V \times \int_0^t I(t)dt$, where the current $I(t)$ is obtained by precisely measuring (sampling) the current drawn by the platform during the time interval t of the application execution under the platform operating voltage V. The normalized energy consumption of the platform executing the application with different throughput requirements for a duration of one minute is shown in Figure 6.9(b). This figure clearly shows the effectiveness of our periodic scheduling approach which can reduce the energy consumption by up to 26% compared to the naive scheduling approach, mentioned earlier, where the approach constantly executes in mode SI^1 in order to satisfy any throughput requirement between 3.9 and 7.81 frames per second.

6.8 Conclusions

In this chapter, we proposed a generic parallel implementation and execution approach for adaptive streaming applications modeled with MADF. Our approach can be easily realized on top of existing operating systems and support the utilization of a wider range of schedules. In particular, we demonstrated our approach on $LITMUS^{RT}$ which is one of the existing real-time extensions of the Linux kernel. Finally, we performed a case study using a real-life adaptive streaming application and showed that our approach conforms to the analysis model for both execution of the application in each individual mode and during mode transitions. In addition, we performed another case study using a real-life streaming application to validate the practical applicability of our proposed periodic scheduling approach, presented in Chapter 5, on a real hardware platform by using our generic parallel implementation and execution approach presented in this chapter.

Chapter 7

Summary and Conclusions

STREAMING applications have become prevalent in embedded systems in several application domains, such as image processing, video/audio processing, and digital signal processing. These applications usually have high computational demands and tight timing requirements, such as throughput requirements. To handle the ever-increasing computational demands and satisfy tight timing requirements, Multi-Processor System-on-Chip (MPSoC) has become a standard platform that is widely adopted in the design of embedded streaming systems to benefit from parallel execution. To efficiently exploit the computational capacity of such MPSoCs, however, streaming applications must be expressed primarily in a parallel fashion. To do so, the behavior of streaming applications is usually specified using a parallel Model of Computation (MoC), in which the application is represented as parallel executing and communicating tasks. Although parallel MoCs resolve the problem of explicitly exposing the available parallelism in an application, the design of embedded streaming systems imposes two major challenges: 1) how to execute the application tasks spatially, i.e., task mapping, and temporally, i.e., task scheduling, on an MPSoC platform such that timing requirements are satisfied while making efficient utilization of available resources (e.g, processors, memory, energy, etc.) on the platform, and 2) how to implement and run the mapped and scheduled application tasks on the MPSoC platform. In this thesis, we have addressed several research questions related to the aforementioned challenges in the design of embedded streaming systems. The research questions and the logical connection between them are illustrated in the design flow shown in Figure 1.2. Below, we provide a summary of the presented research work in this thesis along with some conclusions.

To address the first aforementioned challenge in the design of embed-

ded streaming systems, the strictly periodic scheduling (SPS) framework is proposed in [8] which establishes a bridge between the data flow models and the real-time theories, thereby enabling the designers to directly apply the classical hard real-time scheduling theory to applications modeled as *acyclic* CSDF graphs. In Chapter 3, we have extended the SPS framework and have proposed a scheduling framework, namely Generalized Strictly Periodic Scheduling (GSPS), that can handle **cyclic** CSDF graphs. The GSPS framework converts each actor in a cyclic CSDF graph to a real-time periodic task. This conversion enables the utilization of many hard real-time scheduling algorithms that offer properties such as temporal isolation and fast calculation of the number of processors needed to satisfy a throughput requirement. Based on experimental evaluations, using a set of real-life streaming applications, modeled as cyclic CSDF graphs, we conclude that our GSPS framework can deliver an equal or comparable throughput to related scheduling approaches for the majority of the applications, we experimented with. However, enabling the utilization of scheduling algorithms from the classical hard real-time theory on streaming applications by using our GSPS framework comes at the costs of increasing the latency and buffer sizes of the data communication channels for the applications by up to 3.8X and 1.4X when compared with related scheduling approaches.

In Chapter 4, we have addressed the problem of efficiently exploiting the computational capacity of processors when mapping a streaming application, modeled as an acyclic SDF graph, on an MPSoC platform to reduce the number of needed processors under a given throughput requirement. Given the fact that an initial SDF application specification is often not the most suitable one for the given MPSoC platform, we have explored an alternative application specification, using an SDF graph transformation technique, which closely matches the given MPSoC platform. In this regard, in Chapter 4, we have proposed a novel algorithm to find a proper replication factor for each task/actor in an initial SDF application specification such that by distributing the workloads among more parallel task/actor in the obtained transformed graph, the computational capacity of the processors can be efficiently exploited and a smaller number of processors is then required. Based on experimental evaluations, using a set of real-life streaming applications, we conclude that our proposed algorithm can reduce the number of needed processors by up to 7 processors while increasing the memory requirements and application latency by 24.2% and 17.2% on average compared to FFD task mapping heuristic algorithms while satisfying the same throughput requirement. The experimental evaluations also show that our proposed algorithm can still reduce the

number of needed processors by up to 2 processors and considerably improve the memory requirements and application latency by up to 31.43% and 44.09% on average compared to the other related approaches while satisfying the same throughput requirement.

As embedded streaming systems operate very often using stand-alone power supply such as batteries, energy efficiency has become an important design requirement of such embedded streaming systems in order to prolong their operational time without replacing/recharging the batteries. In this regard, in Chapter 5, we have addressed the problem of energy-efficient scheduling of streaming applications, modeled as CSDF graphs, with throughput requirements on MPSoC platforms with voltage and frequency scaling (VFS) capability. In particular, we have proposed a novel periodic scheduling approach which switches the execution of streaming applications periodically between a few energy-efficient schedules, referred as modes, at run-time in order to satisfy a given throughput requirement at a long run. Using such specific switching scheme, we can benefit from adopting a dynamic voltage and frequency scaling (DVFS) mechanism to efficiently exploit available idle time in an application schedule. Based on experimental evaluations, using a set of real-life streaming applications, we conclude that our novel scheduling approach can achieve up to 68% energy reduction compared to related approaches depending on the application while satisfying the given throughput requirement.

Finally, in Chapter 6, we have addressed the second aforementioned challenge in the design of embedded streaming systems, namely, how to implement and run a mapped and scheduled adaptive streaming application, modeled and analyzed with the MADF MoC, on an MPSoC platform such that the properties of the analysis model are preserved. In particular, we have proposed a generic parallel implementation and execution approach for adaptive streaming applications modeled with MADF. Our approach can be easily realized on top of existing operating systems while supporting the utilization of a wider range of schedules. We have demonstrated our approach on LITMUS^{RT} which is one of the existing real-time extensions of the Linux kernel. Based on a case study using a real-life adaptive streaming application, we conclude that our approach is practically applicable on a real hardware platform and conforms to the analysis model. In addition, another case study, using a real-life streaming application, has shown that our proposed energy-efficient periodic scheduling approach presented in Chapter 5, which adopts the MOO protocol of the MADF MoC for switching the application mode, is also practically applicable on a real hardware platform by using our generic

parallel implementation and execution approach presented in Chapter 6.

Bibliography

[1] Embedded System Market. https://www.gminsights.com/industry-analysis/embedded-system-market. [Cited December 17, 2019].

[2] SDFˆ 3. http://www.es.ele.tue.nl/sdf3/download/examples.php. [Cited December 30, 2019].

[3] H. I. Ali, B. Akesson, and L. M. Pinho. Generalized extraction of real-time parameters for homogeneous synchronous dataflow graphs. In *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 701–710. IEEE, 2015.

[4] J. H. Anderson, V. Bud, and U. C. Devi. An EDF-based scheduling algorithm for multiprocessor soft real-time systems. In *17th Euromicro Conference on Real-Time Systems (ECRTS'05)*, pages 199–208. IEEE, 2005.

[5] H. Aydin and Q. Yang. Energy-aware partitioning for multiprocessor real-time systems. In *Proceedings International Parallel and Distributed Processing Symposium*, pages 9–pp. IEEE, 2003.

[6] T. P. Baker and S. K. Baruah. Schedulability analysis of multiprocessor sporadic task systems. In *Handbook of Real-Time and Embedded Systems*, pages 49–66. Chapman and Hall/CRC, 2007.

[7] M. Bamakhrama. *On hard real-time scheduling of cyclo-static dataflow and its application in system-level design*. Leiden Institute of Advanced Computer Science (LIACS), Leiden University, 2014.

[8] 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*, pages 195–204. ACM, 2011.

[9] M. Bamakhrama and T. Stefanov. Managing latency in embedded streaming applications under hard-real-time scheduling. In *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 83–92. ACM, 2012.

[10] M. Bamakhrama and T. Stefanov. On the hard-real-time scheduling of embedded streaming applications. *Design Automation for Embedded Systems*, 17(2):221–249, 2013.

[11] M. Bambagini, M. Marinoni, H. Aydin, and G. Buttazzo. Energy-aware scheduling for real-time systems: A survey. *ACM Transactions on Embedded Computing Systems (TECS)*, 15(1):7, 2016.

[12] 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.

[13] 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(4):301–324, 1990.

[14] A. Bastoni, B. B. Brandenburg, and J. H. Anderson. An empirical comparison of global, partitioned, and clustered multiprocessor EDF schedulers. In *2010 31st IEEE Real-Time Systems Symposium*, pages 14–24. IEEE, 2010.

[15] S. S. Bhattacharyya and E. A. Lee. Memory management for dataflow programming of multirate signal processing algorithms. *IEEE Transactions on Signal Processing*, 42(5):1190–1201, 1994.

[16] G. Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete. Cycle-static dataflow. *IEEE Transactions on signal processing*, 44(2):397–408, 1996.

[17] B. Bodin, A. Munier-Kordon, and B. D. de Dinechin. K-periodic schedules for evaluating the maximum throughput of a synchronous dataflow graph. In *2012 International Conference on Embedded Computer Systems (SAMOS)*, pages 152–159. IEEE, 2012.

[18] B. Bodin, A. Munier-Kordon, and B. D. de Dinechin. Periodic schedules for cyclo-static dataflow. In *The 11th IEEE Symposium on Embedded Systems for Real-time Multimedia*, pages 105–114. IEEE, 2013.

[19] B. Bodin, A. Munier-Kordon, and B. D. de Dinechin. Optimal and fast throughput evaluation of CSDF. In *Proceedings of the 53rd Annual Design Automation Conference*, page 160. ACM, 2016.

[20] A. Burns, R. I. Davis, P. Wang, and F. Zhang. Partitioned EDF scheduling for multiprocessors using a C= D task splitting scheme. *Real-Time Systems*, 48(1):3–33, 2012.

[21] G. C. Buttazzo. *Hard real-time computing systems: predictable scheduling algorithms and applications*, volume 24. Springer Science & Business Media, 2011.

[22] J. M. Calandrino, H. Leontyev, A. Block, U. C. Devi, and J. H. Anderson. Litmusˆ rt: A testbed for empirically comparing real-time multiprocessor schedulers. In *2006 27th IEEE International Real-Time Systems Symposium (RTSS'06)*, pages 111–126. IEEE, 2006.

[23] E. Cannella, M. A. Bamakhrama, and T. Stefanov. System-level scheduling of real-time streaming applications using a semi-partitioned approach. In *2014 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–6. IEEE, 2014.

[24] E. Cannella, O. Derin, P. Meloni, G. Tuveri, and T. Stefanov. Adaptivity support for MPSoCs based on process migration in polyhedral process networks. *VLSI Design*, 2012, 2012.

[25] E. Cannella and T. Stefanov. Energy efficient semi-partitioned scheduling for embedded multiprocessor streaming systems. *Design Automation for Embedded Systems*, 20(3):239–266, 2016.

[26] G. Chen, K. Huang, and A. Knoll. Energy optimization for real-time multiprocessor system-on-chip with optimal DVFS and DPM combination. *ACM Transactions on Embedded Computing Systems (TECS)*, 13(3s):111, 2014.

[27] H. Cho, B. Ravindran, and E. D. Jensen. An optimal real-time scheduling algorithm for multiprocessors. In *2006 27th IEEE International Real-Time Systems Symposium (RTSS'06)*, pages 101–110. IEEE, 2006.

[28] E. G. Coffman, J. M. R. Garey, and D. Johnson. Approximation algorithms for bin packing: A survey. *Approximation algorithms for NP-hard problems*, pages 46–93, 1996.

[29] R. I. Davis and A. Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM computing surveys (CSUR)*, 43(4):35, 2011.

[30] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc. Design of ion-implanted MOSFET's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256–268, 1974.

[31] N. W. Fisher. *The multiprocessor real-time scheduling of general task systems*. PhD thesis, The University of North Carolina at Chapel Hill, 2007.

[32] M. Geilen and S. Stuijk. Worst-case performance analysis of synchronous dataflow scenarios. In *CODES+ISSS*, 2010.

[33] M. Geilen and S. Stuijk. Worst-case performance analysis of synchronous dataflow scenarios. In *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 125–134. ACM, 2010.

[34] A. H. Ghamarian, M. Geilen, T. Basten, B. D. Theelen, M. R. Mousavi, and S. Stuijk. Liveness and boundedness of synchronous data flow graphs. In *2006 Formal Methods in Computer Aided Design*, pages 68–75. IEEE, 2006.

[35] A. H. Ghamarian, M. Geilen, S. Stuijk, T. Basten, B. D. Theelen, M. R. Mousavi, A. Moonen, and M. Bekooij. Throughput analysis of synchronous data flow graphs. In *Sixth International Conference on Application of Concurrency to System Design (ACSD'06)*, pages 25–36. IEEE, 2006.

[36] L. Gide. Embedded/cyber-physical systems ARTEMIS major challenges: 2014-2020. *Draft Addendum to the ARTEMIS-SRA 2011*, 2013.

[37] M. I. Gordon, W. Thies, and S. Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. *ACM SIGOPS Operating Systems Review*, 2006.

[38] M. Grant and S. Boyd. Graph implementations for nonsmooth convex programs. In V. Blondel, S. Boyd, and H. Kimura, editors, *Recent Advances in Learning and Control*, Lecture Notes in Control and Information Sciences, pages 95–110. Springer-Verlag Limited, 2008. `http://stanford.edu/~boyd/graph_dcp.html`.

[39] M. Grant and S. Boyd. CVX: Matlab Software for Disciplined Convex Programming, version 2.1. `http://cvxr.com/cvx`, Mar. 2014.

[40] P. Greenhalgh. Big. little processing with arm cortex-a15 & cortex-a7. *ARM White paper*, 17, 2011.

[41] J. L. Hennessy and D. A. Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

[42] 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*, pages 1517–1524. ACM, 2013.

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

[44] A. Jerraya, H. Tenhunen, and W. Wolf. Multiprocessor systems-on-chips. *IEEE Computer*, 38(7):36–40, July 2005.

[45] D. S. Johnson. *Near-optimal bin packing algorithms*. PhD thesis, Massachusetts Institute of Technology, 1973.

[46] D. S. Johnson and M. R. Garey. *Computers and intractability: A guide to the theory of NP-completeness*. WH Freeman, 1979.

[47] H. Jung, H. Oh, and S. Ha. Multiprocessor scheduling of a multi-mode dataflow graph considering mode transition delay. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 22(2):37, 2017.

[48] A. H. Khan, Z. H. Khan, and Z. Weiguo. Model-based verification and validation of safety-critical embedded real-time systems: formation and tools. In *Embedded and Real Time System Development: A Software Engineering Perspective*, pages 153–183. Springer, 2014.

[49] P. S. Kurtin, J. P. Hausmans, and M. J. Bekooij. Combining offsets with precedence constraints to improve temporal analysis of cyclic real-time streaming applications. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 1–12. IEEE, 2016.

[50] E. Le Sueur and G. Heiser. Dynamic voltage and frequency scaling: The laws of diminishing returns. In *Proceedings of the 2010 international conference on Power aware computing and systems*, pages 1–8, 2010.

[51] E. A. Lee and S. Ha. Scheduling strategies for multiprocessor real-time DSP. In *1989 IEEE Global Telecommunications Conference and Exhibition'Communications Technology for the 1990s and Beyond'*, pages 1279–1283. IEEE, 1989.

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

[53] E. A. Lee and A. Sangiovanni-Vincentelli. Comparing models of computation. In *Proceedings of International Conference on Computer Aided Design*, pages 234–241. IEEE, 1996.

[54] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM (JACM)*, 20(1):46–61, 1973.

[55] D. Liu, J. Spasic, G. Chen, and T. Stefanov. Energy-efficient mapping of real-time streaming applications on cluster heterogeneous mpsocs. In *2015 13th IEEE Symposium on Embedded Systems For Real-time Multimedia (ESTIMedia)*, pages 1–10. IEEE, 2015.

[56] D. Liu, J. Spasic, J. T. Zhai, T. Stefanov, and G. Chen. Resource optimization for CSDF-modeled streaming applications with latency constraints. In *2014 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–6. IEEE, 2014.

[57] P. Marwedel. *Embedded System Design: Embedded Systems, Foundations of Cyber-Physical Systems, and the Internet of Things*. Springer International Publishing: Imprint: Springer, 2018.

[58] P. Marwedel, J. Teich, G. Kouveli, I. Bacivarov, L. Thiele, S. Ha, C. Lee, Q. Xu, and L. Huang. Mapping of applications to MPSoCs. In *Proceedings of the seventh IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 109–118. ACM, 2011.

[59] T. Mitra. Heterogeneous multi-core architectures. *Information and Media Technologies*, 10(3):383–394, 2015.

[60] O. Moreira. Temporal analysis and scheduling of hard real-time radios running on a multi-processor. *ser. PHD Thesis, Technische Universiteit Eindhoven*, 2012.

[61] A. Nelson, O. Moreira, A. Molnos, S. Stuijk, B. T. Nguyen, and K. Goossens. Power minimisation for real-time dataflow applications. In *2011 14th Euromicro Conference on Digital System Design*, pages 117–124. IEEE, 2011.

[62] S. Niknam and T. Stefanov. Energy-efficient scheduling of throughput-constrained streaming applications by periodic mode switching. In *2017 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, pages 203–212. IEEE, 2017.

[63] S. Niknam, P. Wang, and T. Stefanov. Resource Optimization for Real-Time Streaming Applications Using Task Replication. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(11):2755–2767, 2018.

[64] S. Niknam, P. Wang, and T. Stefanov. Hard Real-Time Scheduling of Streaming Applications Modeled as Cyclic CSDF Graphs. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1549–1554. IEEE, 2019.

[65] S. Niknam, P. Wang, and T. Stefanov. On the Implementation and Execution of Adaptive Streaming Applications Modeled as MADF. In *Proceedings of the International Workshop on Software and Compilers for Embedded Systems (SCOPES)*. ACM, 2020.

[66] ODROID. `http://www.hardkernel.com/`. [Cited December 17, 2019].

[67] S. Park, J. Park, D. Shin, Y. Wang, Q. Xie, M. Pedram, and N. Chang. Accurate modeling of the delay and energy overhead of dynamic voltage and frequency scaling in modern microprocessors. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 32(5):695–708, 2013.

[68] 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*, pages 67–72. ACM, 2006.

[69] R. Pellizzoni, P. Meredith, M.-Y. Nam, M. Sun, M. Caccamo, and L. Sha. Handling mixed-criticality in SoC-based real-time embedded systems. In *Proceedings of the seventh ACM international conference on Embedded software*, pages 235–244. ACM, 2009.

[70] M. Processor. Exynos 5 Octa (5422). https://www.samsung.com/semiconductor/minisite/exynos/products/mobileprocessor/exynos-5-octa-5422/. [Cited December 17, 2019].

[71] G. Qu. What is the limit of energy saving by dynamic voltage scaling? In *Proceedings of the 2001 IEEE/ACM international conference on Computer-aided design*, pages 560–563. IEEE Press, 2001.

[72] Real Time Engineers Ltd. The FreeRTOS Project. http://www.freertos.org/. [Cited December 17, 2019].

[73] M. Shafique and S. Garg. Computing in the dark silicon era: Current trends and research challenges. *IEEE Design & Test*, 34(2):8–23, 2016.

[74] A. K. Singh, A. Das, and A. Kumar. Energy optimization by exploiting execution slacks in streaming applications on multiprocessor systems. In *2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–7, 2013.

[75] A. K. Singh, M. Shafique, A. Kumar, and J. Henkel. Mapping on multi/many-core systems: survey of current and emerging trends. In *2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–10. IEEE, 2013.

[76] F. Siyoum, M. Geilen, O. Moreira, R. Nas, and H. Corporaal. Analyzing synchronous dataflow scenarios for dynamic software-defined radio applications. In *2011 International Symposium on System on Chip (SoC)*, pages 14–21. IEEE, 2011.

[77] D. Sopic, A. Aminifar, and D. Atienza. e-glass: A wearable system for real-time detection of epileptic seizures. In *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5. IEEE, 2018.

[78] 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*, pages 65–74. IEEE Press, 2015.

[79] 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 (TECS)*, 15(4):68, 2016.

[80] J. Spasic, D. Liu, and T. Stefanov. Energy-efficient mapping of real-time applications on heterogeneous MPSoCs using task replication. In *2016 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ ISSS)*, pages 1–10. IEEE, 2016.

[81] J. Spasic, D. Liu, and T. Stefanov. Exploiting resource-constrained parallelism in hard real-time streaming applications. In *Proceedings of the 2016 Conference on Design, Automation & Test in Europe*, pages 954–959. EDA Consortium, 2016.

[82] S. Sriram and S. S. Bhattacharyya. Embedded multiprocessors: scheduling and synchronization. 2009.

[83] S. Stuijk, T. Basten, M. Geilen, and H. Corporaal. Multiprocessor resource allocation for throughput-constrained synchronous dataflow graphs. In *2007 44th ACM/IEEE Design Automation Conference*, pages 777–782. IEEE, 2007.

[84] S. Stuijk, M. Geilen, and T. Basten. SDFˆ 3: SDF for free. In *Sixth International Conference on Application of Concurrency to System Design (ACSD'06)*, pages 276–278. IEEE, 2006.

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

[86] S. Stuijk, M. Geilen, B. Theelen, and T. Basten. Scenario-aware dataflow: Modeling, analysis and implementation of dynamic applications. In *2011 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation*, pages 404–411. IEEE, 2011.

[87] B. D. Theelen, M. C. Geilen, S. Stuijk, S. V. Gheorghita, T. Basten, J. P. Voeten, and A. H. Ghamarian. Scenario-aware dataflow. *Technical Report ESR-2008-08*, 2008.

[88] W. Thies and S. Amarasinghe. An empirical characterization of stream programs and its implications for language and compiler design. In *2010 19th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 365–376. IEEE, 2010.

[89] R. Van Kampenhout, S. Stuijk, and K. Goossens. A scenario-aware dataflow programming model. In *2015 Euromicro Conference on Digital System Design*, pages 25–32. IEEE, 2015.

[90] R. Van Kampenhout, S. Stuijk, and K. Goossens. Programming and analysing scenario-aware dataflow on a multi-processor platform. In *Proceedings of the Conference on Design, Automation & Test in Europe*, pages 876–881. European Design and Automation Association, 2017.

[91] M. H. Wiggers, M. J. Bekooij, and G. J. Smit. Efficient computation of buffer capacities for cyclo-static dataflow graphs. In *2007 44th ACM/IEEE Design Automation Conference*, pages 658–663. IEEE, 2007.

[92] K. Yang and J. H. Anderson. Soft real-time semi-partitioned scheduling with restricted migrations on uniform heterogeneous multiprocessors. In *Proceedings of the 22nd International Conference on Real-Time Networks and Systems*, page 215. ACM, 2014.

[93] J. T. Zhai. *Adaptive streaming applications: analysis and implementation models*. PhD thesis, Leiden Embedded Research Center, Faculty of Science (LERC), Leiden Institute of Advanced Computer Science (LIACS), Leiden University, 2015.

[94] J. T. Zhai, S. Niknam, and T. Stefanov. Modeling, analysis, and hard real-time scheduling of adaptive streaming applications. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(11):2636–2648, 2018.

[95] F. Zhang and A. Burns. Schedulability analysis for real-time systems with EDF scheduling. *IEEE Transactions on Computers*, 58(9):1250–1258, 2009.

[96] 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*, pages 119–128. ACM, 2008.

Summary

This thesis focuses on addressing four research problems in designing embedded streaming systems. Embedded streaming systems are those systems that process a stream of input data coming from the environment and generate a stream of output data going into the environment. For many embedded streaming systems, the timing is a critical design requirement, in which the correct behavior depends on both the correctness of output data and on the time at which the data is produced. An embedded streaming system subjected to such a timing requirement is called a real-time system. Some examples of real-time embedded streaming systems can be found in various autonomous mobile systems, such as planes, self-driving cars, and drones.

To handle the tight timing requirements of such real-time embedded streaming systems, modern embedded systems have been equipped with hardware platforms, the so-called Multi-Processor Systems-on-Chip (MPSoC), that contain multiple processors, memories, interconnections, and other hardware peripherals on a single chip, to benefit from parallel execution. To efficiently exploit the computational capacity of an MPSoC platform, a streaming application which is going to be executed on the MPSoC platform must be expressed primarily in a parallel fashion, i.e., the application is represented as a set of parallel executing and communicating tasks. Then, the main challenge is how to schedule the tasks spatially, i.e., task mapping, and temporally, i.e., task scheduling, on the MPSoC platform such that all timing requirements are satisfied while making efficient utilization of available resources (e.g, processors, memory, energy, etc.) on the platform. Another challenge is how to implement and run the mapped and scheduled application tasks on the MPSoC platform. This thesis proposes several techniques to address the aforementioned two challenges.

In the first part of the thesis, the focus is on addressing the first aforementioned challenge in the design of embedded streaming systems. To do so, a scheduling framework is proposed to convert the data-dependent tasks in an application, including cyclic data-dependent tasks, to real-time periodic

tasks. As a result, a variety of hard real-time scheduling algorithms for periodic tasks, from the classical real-time scheduling theory, can be applied to schedule such streaming applications with a certain guaranteed performance, i.e., throughput/latency. These algorithms can perform fast admission control and scheduling decisions for new incoming applications in an MPSoC platform as well as offer properties such as temporal isolation and fast analytical calculation of the minimum number of processors needed to schedule the tasks in the application.

In the second part of the thesis, the focus is on addressing the problem of efficiently exploiting resources on an underlying MPSoC platform when scheduling the tasks of applications on the platform. An algorithm is proposed to transform an initial representation of a streaming application, i.e., an initial application graph, into a functionally equivalent one such that the new representation requires fewer processors while guaranteeing a given throughput requirement. Additionally, this thesis studies the problem of energy-efficient scheduling of streaming applications with throughput requirements on MPSoC platforms with voltage and frequency scaling capability. In this regard, a novel periodic scheduling framework is proposed which allows streaming applications to switch their execution periodically between a few energy-efficient schedules at run-time in order to meet a throughput requirement at long run. Using such periodic switching scheme, system designers can benefit from adopting Dynamic Voltage and Frequency Scaling techniques to exploit available static slack time in the schedule of an application efficiently.

Finally, in the third part of the thesis, the focus is on addressing the second aforementioned challenge in the design of embedded streaming systems. In this regarde, a generic parallel implementation and execution approach for (adaptive) streaming applications is proposed. The proposed approach can be easily realized on top of existing operating systems while supporting the utilization of a wider range of schedules. In particular, a demonstration of the proposed approach on LITMUS^{RT} is provided, which is one of the existing real-time extensions of the Linux kernel.

Samenvatting

Het doel van dit proefschrift is het oplossen van vier onderzoeksproblemen bij het ontwerpen embedded streaming-systemen. Embedded streaming-systemen zijn die systemen die een stroom invoergegevens uit de omgeving verwerken en een stroom van uitvoergegevens genereren voor deze omgeving. Voor velen van deze ingebedde streaming-systemen is de timing een kritische ontwerpvereiste, waarbij correct gedrag afhangt van zowel de juistheid van uitvoergegevens als van het tijdstip waarop de gegevens worden geproduceerd. Een embedded streaming-systeem onderworpen naar zo'n timingvereiste wordt een real-time systeem genoemd. Enkele voorbeelden van real-time embedded streaming-systemen zijn te vinden in verschillende autonome mobiele systemen, zoals vliegtuigen, zelfrijdende auto's en drones.

Om aan de strakke timingvereisten van dergelijke real-time embedded streaming-systemen te kunnen voldoen zijn moderne embedded systemen uitgerust met hardware platforms, de zogenaamde Multi-Processor Systems-on-Chip (MPSoC), die meerdere processors, geheugens, verbindingen en andere hardware-randapparatuur op een enkele chip samenvoegen, om zo te kunnen profiteren van parallelle executie. Om de rekencapaciteit van een MPSoC-platform te kunnen benutten moet een streaming-applicatie, die wordt uitgevoerd op het MPSoC-platform, worden beschreven op een parallelle wijze, d.w.z. de applicatie wordt gedefinieerd als een set van parallele taken die met elkaar communiceren. De belangrijkste uitdaging is om deze taken ruimtelijk te plannen, d.w.z. de afbeelding van taken op processors, en temporeel, d.w.z. de volgorde van de taakplanning, op het MPSoC-platform zodat aan alle timingvereisten wordt voldaan met een efficiÎnt gebruik van de beschikbare middelen (de processors, geheugen, energie, etc.) op het platform. Een andere uitdaging is hoe deze toegewezen en geplande applicatietaken op de MPSoC te implementeren en uit te voeren op het platform. Dit proefschrift stelt verschillende technieken voor om de twee bovengenoemde uitdagingen op te lossen.

In het eerste deel van het proefschrift ligt de focus op de eerste boven-

genoemde uitdaging bij het ontwerpen van embedded streaming-systemen. Hier wordt een methode geintroduceerd om de data-afhankelijke taken in een applicatie, inclusief cyclische data-afhankelijke taken, om te zetten naar real-time periodieke taken. Dit maakt het mogelijk om een verscheidenheid aan harde realtime planning algoritmen voor periodieke taken, van de klassieke real-time planning theorie, toe te passen om dergelijke streamingtoepassingen te plannen met bepaalde gegarandeerde prestaties voor doorvoer en reactietijd. Deze algoritmen kunnen snelle toegangscontrole en planningsbeslissingen uitvoeren voor nieuwe inkomende applicaties in een MPSoC-platform en bieden eigenschappen zoals temporele isolatie en snelle analytische berekening van het minimum aantal processors dat nodig is voor het uitvoeren van de taken in de applicatie.

In het tweede deel van het proefschrift ligt de focus op het efficiÎnt gebruik maken van componenten op een onderliggend MPSoC-platform bij het plannen van de taken van applicaties op het platform. We introduceren een algoritme om een eerste representatie van een streamingapplicatie, d.w.z. een initiÎle applicatie graaf, te transformeren in een functioneel equivalente applicatie graaf die minder processors nodig heeft om de gegeven doorvoervereiste te garanderen. Daarnaast onderzoekt dit proefschrift het probleem van energiezuinige planning van streaming-applicaties met doorvoer vereisten op MPSoC-platforms met spannings- en frequentieschaling mogelijkheden. Hiervoor wordt er een nieuw periodiek planningskader geintroduceerd waarin streaming-applicaties hun uitvoering periodiek kunnen varieren tussen een aantal energiezuinige schema's tijdens runtime om te voldoen aan een doorvoervereiste op de lange termijn. Met behulp van een dergelijke periodieke omschakeling kunnen systeemontwerpers profiteren van het gebruik van dynamische spanning en frequentieschalingstechnieken om de beschikbare extra spelingstijd in het schema van een applicatie efficiÎnt te gebruiken.

Tot slot, in het derde deel van het proefschrift, ligt de focus op de tweede bovengenoemde uitdaging in het ontwerp van embedded streaming-systemen. Hiervoor wordt een generieke parallelle implementatie- en uitvoeringsmethode voor (adaptieve) streaming-applicaties voorgesteld. De voorgestelde methode kan gemakkelijk kunnen worden gerealiseerd bovenop bestaande besturingssystemen en in combinatie met een breder scala aan taakplanningsmethoden. Een demonstratie van de voorgestelde aanpak voor LITMUS^{RT}, een bestaande real-time uitbreidingen van de Linux-kernel, toont de haalbaarheid aan van deze methode.

List of Publications

Journal Articles

- **Sobhan Niknam**, Peng Wang, Todor Stefanov. "Resource Optimization for Real-Time Streaming Applications using Task Replication". *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 37, No. 11, pp. 2636-2648, Nov 2018.

- Teddy Zhai, **Sobhan Niknam**, Todor Stefanov. "Modeling, Analysis, and Hard Real-time Scheduling of Adaptive Streaming Applications". *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 37, No. 11, pp. 2755-2767, Nov 2018.
  (Authors contributed to the paper equally)

Peer-Reviewed Conference Proceedings

- **Sobhan Niknam**, Peng Wang, Todor Stefanov. "On the Implementation and Execution of Adaptive Streaming Applications Modeled as MADF". *In Proceedings of the 23rd International Workshop on Software and Compilers for Embedded Systems (SCOPES)*, Sankt Goar, Germany, May 25-26, 2020.

- Peng Wang, **Sobhan Niknam**, Sheng Ma, Zhiying Wang, Todor Stefanov. "EVC-based Power Gating Approach to Achieve Low-power and High Performance NoC". *In Proceedings of the 22nd Euromicro Conference on Digital System Design (DSD)*, Chalkidiki, Greece, August 28 - 30, 2019.

- Erqian Tang, **Sobhan Niknam**, Todor Stefanov. "Enabling Cognitive Autonomy on Small Drones by Efficient On-board Embedded Computing: An ORB-SLAM2 Case Study". *In Proceedings of the 22nd Euromicro Conference on Digital System Design (DSD)*, Chalkidiki, Greece, August 28 - 30, 2019.

- Peng Wang, **Sobhan Niknam**, Sheng Ma, Zhiying Wang, Todor Stefanov. "A Dynamic Bypass Approach to Realize Power Efficient Network-on-Chip". *In Proceedings of the 21st IEEE International Conference on High Performance Computing and Communications (HPCC)*, Zhangjiajie, Hunan, China, August 10 - 12, 2019.

- Peng Wang, **Sobhan Niknam**, Sheng Ma, Zhiying Wang, Todor Stefanov. "Surf-Bless: A Confined-interference Routing for Power-Efficient Communication in NoCs". *In Proceedings of the 56th ACM/EDAC/IEEE Design Automation Conference (DAC)*, Las Vegas, USA, June 2 - 6, 2019. **Winner of HiPEAC paper award**

- **Sobhan Niknam**, Peng Wang, Todor Stefanov. "Hard Real-Time Scheduling of Streaming Applications Modeled as Cyclic CSDF Graphs". *In Proceedings of the 22nd International Conference on Design, Automation and Test in Europe (DATE)*, Florence, Italy, March 25 - 29, 2019.

- Peng Wang, **Sobhan Niknam**, Zhiying Wang, Todor Stefanov. "A Novel Approach to Reduce Packet Latency Increase caused by Power Gating in Network-on-Chip". *In Proceedings of the 11th International Symposium on Networks-on-Chip (NOCS)*, Seoul, South Korea, October 19 - 20, 2017.

- **Sobhan Niknam**, Todor Stefanov. "Energy-Efficient Scheduling of Throughput-Constrained Streaming Applications by Periodic Mode Switching". *In Proceedings of the 17th IEEE International Conference on Embedded Computer Systems: Architectures, MOdeling, and Simulation (SAMOS)*, Samos, Greece, July 17 - 20, 2017.

- **Sobhan Niknam**, Arghavan Asad, Mahmood Fathy, Amir M. Rahmani. "Energy Efficient 3D Hybrid Processor-Memory Architecture for the Dark Silicon Age". *In Proceedings of the 10th International Symposium on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC)*, Bremen, Germany, Jun 29 - July 1, 2015.

Curriculum Vitae

Sobhan Niknam was born on February 28, 1990 in Tehran, Iran. He obtained his B.Sc. degree in computer engineering from Shahed University, Tehran, Iran, in 2012 and the M.Sc. degree in computer engineering from the Iran University of Science and Technology, Tehran, in 2014. In March 2015, he joined the Leiden Embedded Research Center, part of the Leiden Institute of Advanced Computer Science (LIACS) at Leiden University, as a Ph.D. candidate. His research work, which resulted in this thesis, was funded by NWO under project rCPS3. Besides his work as a researcher, he had been teaching assistant for several courses such as Digital Techniques, Computer Architecture, Operating Systems, and Embedded Systems and Software. Since February 2020, he has been working as a postdoctoral researcher at the University of Amsterdam.

Acknowledgments

Finally, my long academic journey as a PhD student comes to its end. The past five years have been quite an intense and unforgettable experience, full of all sorts of overwhelming emotions - happiness, frustration, anxiety, inspiration, and a lot of hope! Finishing this hard, but the enjoyable journey would not have been possible without the help, guidance, and assistance from many extraordinary people whom I would like to express my gratitude.

First of all, I would like to thank my supervisor, **Dr. Todor Stefanov**, for giving me the chance to pursue my doctoral research at Leiden University and for his support, patience, and effort throughout my PhD study. Thank you, **Todor**, especially for teaching me how to write a good academic paper and spending indefinite time and tremendous efforts on proof-reading my papers and finally my thesis. Secondly, I was very fortunate to be a part of the Leiden Embedded Research Center (LERC) where I had nice colleagues: **Emanuele Cannella**, **Jelena Spasic**, **Di Liu**, **Teddy Zhai**, **Peng Wang**, **Hongchan Shan**, **Erqian Tang**, **Svetlana Minakova**. I really enjoyed working with you. I hope you are all doing well and wish you great success in your current and future endeavors. **Emanuele**, thank you especially for your support at the early stage of my PhD; I never forget about your encouragement and pleasing words about being persistent and not giving up. **Jelena** and **Di**, thank you for your help, exchanging ideas, suggestions about my research, and nice discussions we had. I would like to give my special thanks to **Peng**. I was lucky to have such a wonderful fellow PhD almost from the beginning of my study, who helped by brainstorming, providing feedback, and most importantly being an exceptional friend. We had unforgettable coffee breaks, talking about our daily life and all PhD-related matters, such as our ongoing research and feelings - fear, happiness, failure, and success. It has been a pleasure and privilege to work with you, **Peng**!

Further, during my stay in the Netherlands, I have been lucky to make some good friends, **Seyed Ali Mirsoleimani**, **Hadi Ahmadi Balef**, **Hadi Arjmandi-Tash**, **Seyed Kamal Sani**, **Soroush Rasti**, and many others, whom

I am so grateful for their help in many ways. Without them, I would never feel being like at home in the past five years. A big thanks goes to **Hadi Ahmadi Balef** and his family for the joyfull gatherings and nice trips we have had.

Last but not least, I would like to express my thanks and gratitude to my family, and in particular, my parents, who have believed in me, helped me to pursue my dream and, enabled me to become the person I am today. My thanks also go to my parents-in-law for their understanding and supports. The biggest "thank you" goes to my beloved wife, **Saeedeh**, who sacrifices herself to let me finish my PhD. Thank you for all support, encouragement, and love you have unconditionally given me especially during this extremely difficult time in our lives. Words can not express my gratitude for all what you have done, may God reward you in a thousand folds, **Saeedeh**. My finall thanks go to my little boy, **Amirali**, who has brought joyfull time to our family.

Sobhan Niknam
June, 2020
Leiden, The Netherlands