Generalized strictly periodic scheduling analysis, resource optimization, and implementation of adaptive streaming applications

Niknam, S.

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

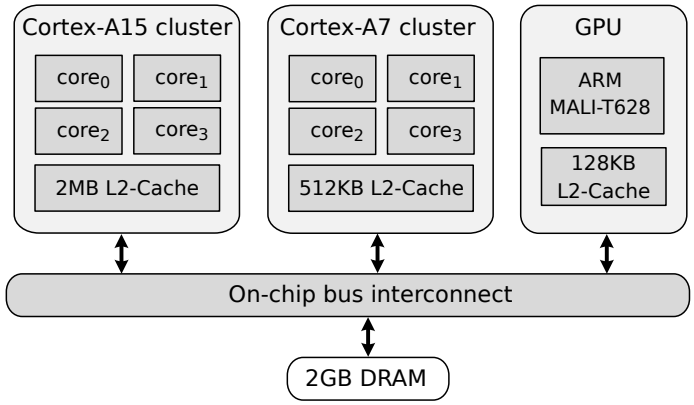

Figure 1.1: *Samsung Exynos 5422 MPSoC [70].*

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.

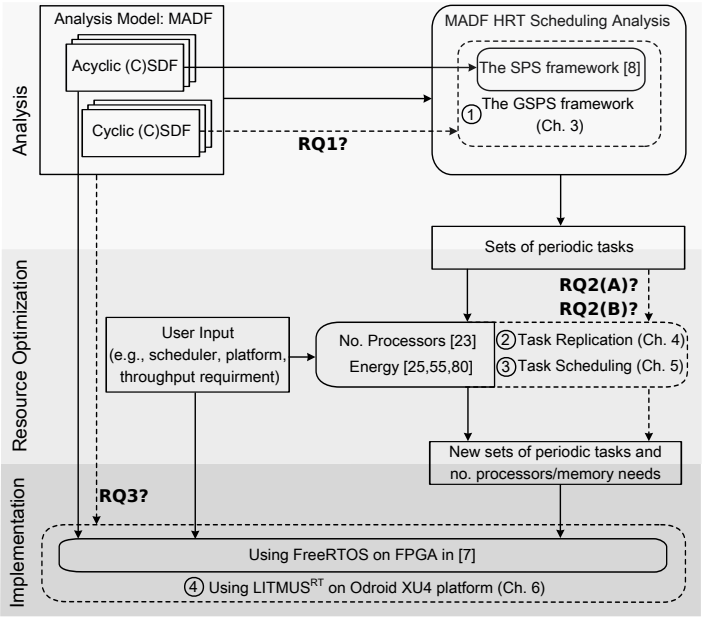

Figure 1.2: *Overview of the research questions and contributions in this thesis using a design flow.*

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.