Semi-partitioned scheduling and task migration in dataflow networks

Cannella, E.

Citation

Cannella, E. (2016, October 11). *Semi-partitioned scheduling and task migration in dataflow networks*. Retrieved from https://hdl.handle.net/1887/43469

Author: Cannella, Emanuele
Title: Semi-partitioned scheduling and task migration in dataflow networks
Issue Date: 2016-10-11

Semi-partitioned Scheduling and Task Migration in Dataflow Networks

Emanuele Cannella

Semi-partitioned Scheduling and Task Migration in Dataflow Networks

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 11 oktober 2016
klokke 10.00 uur

door

Emanuele Cannella
geboren te Udine, Italië
in 1983

Promotor:	Prof. dr. Ed F. Deprettere	Universiteit Leiden
Co-Promotor:	Dr. Todor P. Stefanov	Universiteit Leiden
Promotion Committee:	Prof. dr. Peter Marwedel	Technische Universität Dortmund
	Prof. dr. Luigi Raffo	Università di Cagliari
	Dr. Andy Pimentel	Universiteit van Amsterdam
	Prof. dr. Aske Plaat	Universiteit Leiden
	Prof. dr. Harry Wijshoff	Universiteit Leiden
	Prof. dr. Joost Kok	Universiteit Leiden

Semi-partitioned Scheduling and Task Migration in Dataflow Networks
Emanuele Cannella. -
Dissertation Universiteit Leiden. - With ref. - With summary in Dutch.

Cover designed by Shanshan Yang.

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

Printed by CPI-Koninklijke Wöhrmann – Zutphen.

Al nostro angelo Silvana
e al mitico Franz

Contents

List of Figures

List of Tables

Chapter 1

Introduction

ADVANCES in technology have added new features and functionalities to most of vehicles, homes, industrial facilities, and many other applications that form the basis of our society. For instance, many houses nowadays are endowed with *security camera systems* to monitor the premises of the house itself. These systems are composed of distributed security smart cameras, which are connected to a server where the recorded images are stored. As another example, in recent years several companies are making significant research and development efforts to implement *autonomous driving* cars (e.g. Google [Woo], General Motors [Rho]). As a third example, state-of-the-art operating rooms feature devices that help surgeons to perform minimally-invasive surgeries, less traumatic for patients. During the surgical operation, these devices visualize the organs and tissues on which the surgery is performed, together with the position of surgical instruments, using for instance X-rays. The devices which allow this visualization are termed *live medical imaging devices*.

In all the examples above, computing systems are enclosed into products, buildings, or facilities, to which they provide additional functionalities. These computing systems are called *embedded systems*. For instance, in autonomous driving cars, an embedded system is in charge of planning the motion of the car and the braking and steering actions. As the above examples show, embedded systems are tightly coupled to the environment in which they operate. Moreover, most embedded systems share other characteristics, such as:

- They are designed to implement a well-defined set of functionalities, known at design time;
- They must be dependable, because they often operate in safety-critical environments;
- They must provide hard real-time guarantees, i.e., their output must be correct and also produced within a certain time frame.

These characteristics set embedded systems apart from general purpose systems, such as Personal Computers, which show much greater flexibility in terms of functionality and much lesser emphasis on real-time guarantees and dependability.

Embedded systems can be divided in two categories, depending on the type of functionality that they provide:

- **Control systems** wait for input events (or signals) from the environment and react to these events accordingly. These systems are used, for instance, in industrial automation.
- **Streaming systems** process a continuous, possibly infinite stream of data from the environment. These system find application, for instance, in audio and video processing.

In this thesis, we focus on embedded streaming systems. Examples of streaming applications range from audio/video encoding and decoding (e.g., YouTube), signal processing, computer vision [VAJ$^+$09], medical imaging, navigation systems, security camera systems, and many others.

Complex embedded systems such as the ones that control autonomous driving cars are in fact composed of several sub-systems which communicate among each other. In the autonomous driving cars example, part of these sub-systems belong to the category of streaming systems, and the other part belong to the control category. For instance, autonomous driving cars gather an enormous quantity of data, which comes in the form of continuous *streams*, from cameras and laser sensors mounted on the car itself. These streams of data must be continuously refined and processed in order to perform motion planning (i.e., identify the optimal path and speed that the vehicle should follow) and collision avoidance (i.e., detect and avoid incoming unexpected obstacles). These decisions are made by *streaming* sub-systems, and communicated to other sub-systems (of *control* type), which make the car actually steer, brake, or accelerate. The analyses and techniques presented in this thesis target the sub-systems that belong to the streaming category.

As mentioned above, systems that control autonomous driving cars implement motion planning and collision avoidance algorithms that have extremely high complexity. In addition, these algorithms must produce their output in a short and predictable time, such that the car can react quickly to external events (consider, for instance, a person that suddenly crosses a street in front of the car; the car must stop as soon as possible). The high complexity of the implemented algorithms and the requirement of a short execution time challenge designers to achieve high system performance. This is a requirement shared by many modern embedded systems. In fact, over the years, embedded systems have shown a constant demand for increasing performance.

Until the mid-2000s, most computing systems were implemented as uniprocessor architectures, and the aforementioned demand for increasing performance was addressed by enhancing the computational power of the (single) processor itself [HP07]. However, the performance increase between successive generations of uniprocessors has incurred a major slowdown in the early 2000s [Sut], mainly due to: (i) diminishing returns of novel processor design solutions; (ii) very slow increase between processor generations of clock frequency due to leakage power issues; (iii) growing disparity of speed between processor and memory. Therefore, in order to push system performance even further, chip manufacturers since the mid-2000s have shifted their research and development efforts to **multiprocessor architectures** [HP07]. This is a technology trend that has affected both general purpose and embedded sys-

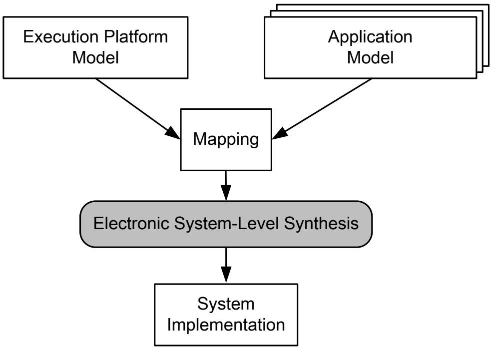


Figure 1.1: *System-level design methodology.*

tem, and is here to stay. Actually, more and more architectures proposed by both research institutes and industry show an increasing count of processing elements (PEs[1]). In fact, nowadays, embedded system designers often integrate in a single chip multiple processors, memories, interconnections, and other hardware peripherals to form a so-called **Multiprocessor System-on-Chip (MPSoC)** [JTW05]. This dissertation describes design methodologies and techniques targeted at such embedded MPSoCs. An early example of such embedded MPSoCs is the Trimedia TM1000 [Tri], where a general purpose microprocessor is combined with several multimedia coprocessors. More recent examples of MPSoCs that find application in the embedded domain include the 64-PEs Adapteva Epiphany [VEMR14], the 72-PEs TILE-Gx72 from EZchip [TIL] and the 256-PEs Kalray MPPA-256 [dDAB$^+$13].

1.1 Trends in Embedded MPSoC Design

In the previous section we have explained the importance of embedded systems in our society and motivated the emergence of MPSoCs in the embedded domain. We have also pointed out that modern embedded MPSoCs demand increasingly higher performance. In addition to this increase in performance, the complexity of modern embedded MPSoCs has also risen.

As the complexity of MPSoCs is constantly increasing, nowadays the design of these systems must be performed at the right abstraction level. In particular, design at *gate-level* and *register-transfer level (RTL)* is no longer effective. A higher abstraction level, namely *system-level*, is necessary to design modern embedded MPSoCs [Hen03]. As represented in Figure 1.1, at system level designers devise a system by specifying the execution platform model, the application model, and the *mapping* of the application to the execution platform. In particular, the **execution platform** model describes the type and number of processors available in the system, and which kind of memories and interconnections are present. The **application** is

---

[1] In this thesis, we use the terms "processor" and "PE" interchangeably.

modeled as a set of tasks that can be distributed to multiple processors. Moreover, the application model describes how these tasks communicate and synchronize among each other. Finally, the **mapping** specifies how the application model is mapped to the execution platform model. For instance, the mapping describes: (i) how tasks are distributed among the processors of the execution platform; (ii) how several tasks, mapped on a single processor, are scheduled; (iii) how communication primitives used in the application model are converted to corresponding execution platform primitives. Then, when application, execution platform and mapping are specified, an Electronic System-Level Synthesis tool (e.g., [NSD08]) generates in an automated way the detailed hardware description (e.g., at RTL) and the software running on each processor of the system.

In general, in order to achieve high performance, the models of the execution platform and the application should be closely related For instance, the Von Neumann architecture matches perfectly with an application specified using a sequential program (e.g., using the C programming language). If the system performance does not meet the requirements, designers have to modify the execution platform, software and/or mapping specification in order to improve the achieved performance level.

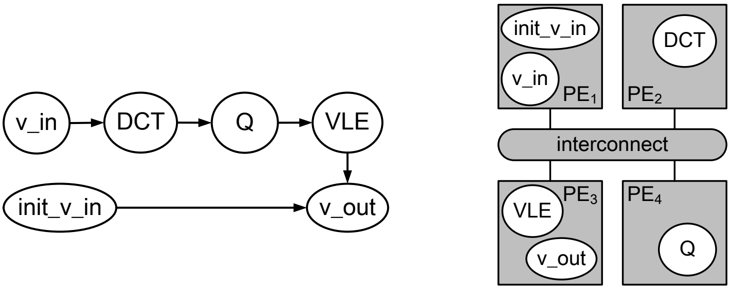
Since embedded systems are now shifting from uniprocessors to multiprocessors, several changes in the design methodology are required. In particular, the two main problems that designers face are: (i) how to model the applications such that the multiple processing resources available in modern execution platforms can be exploited; and (ii) how to connect processors in the execution platform, which is especially complicated as the number of processors in systems is constantly increasing. The current trends to solve these two problem are described in Section 1.1.1 and 1.1.2, respectively.

1.1.1 Programming for Multiprocessors: Models of Computation

As mentioned earlier, in order to achieve the desired performance on MPSoCs, embedded software shall be specified having in mind the parallelism of the execution platform. In particular, applications have to be decomposed in portions that can be executed in parallel. Moreover, designers shall be able to reason about how many processing resources to utilize, and how to distribute the application workload among these resources. Finally, the actual code that will run on the considered execution platform has to be generated, preferably in an automated way given the complexity of modern MPSoCs.

Old-fashioned design flows based only on board support packages and high-level programming interfaces fail to support the aforementioned design activities in a rigorous and efficient way [HHBT09]. The de-facto solution to overcome this issue is to use parallel (or concurrent) **Models of Computation (MoCs)** [Lee99]. Design approaches that exploit MoCs are called **Model-based Designs**. In particular, MoCs support the design process by allowing to:

1. Expose the parallelism available in an application;
2. Perform Design Space Exploration;
3. Generate software for the considered execution platform in an automated way.



(a) HSDF specification of the M-JPEG encoder application

(b) Mapping of M-JPEG tasks to 4 PEs

Figure 1.2: *Example of MoC-based application specification (a) and mapping to a platform with 4 PEs (b).*

All of these activities are instrumental to tackle the design complexity of modern embedded multiprocessor systems. In the following, we describe how these activities are facilitated by MoCs. We conclude this section with an introduction to the MoCs considered in this thesis.

Exposing the available application parallelism

By using a parallel MoC, designers decompose applications into tasks capable of performing computation in parallel. The parallel MoC, in particular, defines the rules by which these tasks communicate and synchronize among each other. By contrast, the actual computation performed by the tasks is specified using a host language, for instance C.

As many MoCs exist, designers choose the MoC most suitable to the considered application domain. In this dissertation we focus on **streaming applications**, which are widespread in the embedded domain. For streaming applications, *dataflow* MoCs are the most appropriate because their semantics allow to express the application parallelism in a natural way. An example of a streaming application specified according to a dataflow MoC is shown in Figure 1.2(a). In particular, the figure shows a Motion JPEG (M-JPEG) encoder application specified using the Homogeneous Synchronous Dataflow (HSDF) MoC [LM87b]. The HSDF MoC is described in greater detail in Section 2.1.1. As shown in Figure 1.2(a), in a dataflow MoC applications are usually specified in the form of directed graphs in which graph nodes represent the tasks of the application (in the example, *v_in*, *Q*, *VLE*, . . .) and graph edges represent inter-task data dependencies.

Performing Design Space Exploration

Dataflow MoCs are not only useful to specify the parallelism available in streaming applications. In fact, they also facilitate the Design Space Exploration (DSE) process

[PEP06]. In a nutshell, DSE consists of evaluating alternative design points until (some) objective criteria are met. Each design point consists of a particular execution platform configuration used to implement the desired functionality, and of a well-defined scheduling of the application onto the considered execution platform.

As the **execution platform configuration** is concerned, several design parameters can be varied, such as the number of processing elements (PEs), the memory subsystems, or the interconnection infrastructure. After the execution platform configuration has been defined, the MoC-based application specification allows designers to specify the **scheduling**[2] **of the application** onto the execution platform in a rigorous way [SB09]. Application scheduling consists in defining *where* and *when* each task of the application is executed. More precisely, scheduling decisions can be divided in:

- **Spatial scheduling (or *mapping*):** it determines the assignment of application tasks to processors.
- **Temporal scheduling:** if more than one tasks are assigned to a PE, it determines the order of execution of the tasks on the PE, and when each task executes such that all precedence constraints are met.

Both spatial and temporal scheduling can be performed at design-time (static approach) or run-time (dynamic approach). For instance, Figure 1.2(b) shows a possible static mapping of the M-JPEG application specified in Figure 1.2(a) to a platform with four PEs. Notice that tasks *VLE* and *v_out* are mapped to PE_3. Since *v_out* is data-dependent from *VLE* (see Figure 1.2(a)), it must always be scheduled on PE_3 after *VLE*.

To summarize, referring to the system-level design methodology shown in Figure 1.1, given a fixed application specification, a point in the design space is determined by defining the specification of the execution platform, together with the mapping of the application tasks to the processors of the execution platform.

Once a point in the design space is defined, as an execution platform configuration and a specific spatial/temporal scheduling of the application, the formal semantics of MoCs allow designers to evaluate it. A design point can be evaluated according to many metrics such as performance, memory requirements, and power consumption. For instance, MoC-based design allows to evaluate system performance using analytical models (e.g. [SB09,GGS+06]) or simulations (e.g. [PEP06]). Based on these evaluations, at the end of the DSE process it is possible to choose a design point that is optimal according to the considered objective criteria.

Generating Code in an Automated Way

Once the spatial and temporal scheduling of the application are defined, the MoC-based application specification allows to generate the code to be run on each processor of the target MPSoC in an automated way [HHBT09,NSD08]. This is because MoCs define without ambiguity the behavior of each task and the communication among them.

---

[2] The interested reader is referred to [Pin16] for an introduction of scheduling problems and techniques that occur in many real-world domains, beyond the embedded system domain considered in this thesis.

MoCs considered in this thesis

In addition to the HSDF MoC used in the example of Figure 1.2(a), other popular examples of dataflow MoCs include **Synchronous Dataflow (SDF)** [LM87b], **Polyhedral Process Networks (PPN)** [VNS07], and **Cyclo-Static Dataflow (CSDF)** [BELP96]. All of these MoCs are described in Chapter 2 of this thesis. These dataflow MoC share three main characteristics. First, they are *determinate* [Kah74], i.e., the order in which the nodes of the graph are scheduled has no influence on the result of the computation. Second, as their expressiveness is concerned, they are *not Turing complete*. This is a limitation required to guarantee the third characteristic: these MoCs are *decidable*. The decidability of a MoC represents the extent to which designers can analyze, at compile-time, properties of an application such as:

- Absence of deadlocks: given a certain set of spatial/temporal scheduling decisions, it can be ensured that the application will never deadlock.
- Boundedness: given a certain set of spatial/temporal scheduling decisions, an upper bound of the required size of buffers used to implement inter-task data dependencies can be derived. As a result, at run-time neither buffer overflow nor underflow can occur.
- Throughput guarantee: the throughput achieved by the system at run-time will never be lower than a certain bound which can be determined analytically.

Due to these convenient characteristics, in this dissertation we assume that applications are specified using one of the MoCs mentioned above. A comparison of the considered MoCs is provided in Figure 1.3, adapted from [SGTB11] and [Zha15]. In the figure, MoCs are compared according to three criteria [SGTB11]: (i) *expressiveness* and *succinctness* indicate which systems can be modeled using the considered MoC and how compact these models are; (ii) *implementation efficiency* evaluates the complexity of the scheduling problem and the (code) size of the resulting schedules; (iii) *analyzability*, as mentioned earlier, refers to the availability of analysis and synthesis algorithms and their computational complexity. As shown in Figure 1.3, there is no "best" MoC among the ones considered in this thesis, because in general expressiveness and analyzability are inversely related. Therefore, in this dissertation we will motivate the choice of one MoC over the others depending on the different addressed problems.

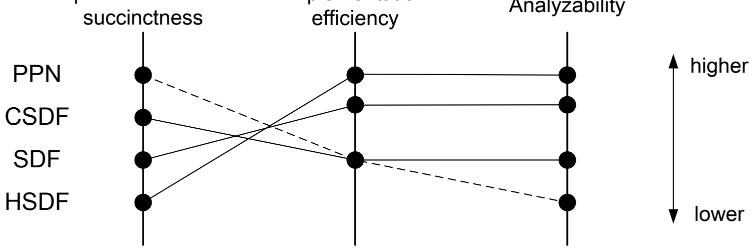


Figure 1.3: *Comparison of the MoCs considered in this thesis.*

Note that, beside the MoCs considered in this thesis, other more expressive dataflow MoCs exist. The interested reader is referred to [BDLT13] and to [SGTB11]. In particular, [SGTB11] provides a MoC comparison more complete than the one given in Figure 1.3. The comparison given in [SGTB11] includes MoCs that can express dynamic application behavior. For instance, such MoCs allow parameters of the application to be changed at run-time, at the expense of a lower analyzability. By contrast, the MoCs considered in this dissertation cannot express such dynamic behavior of applications, favoring a superior analyzability.

1.1.2 Communication Infrastructures: Networks-on-Chip

As mentioned in the beginning of this chapter, since the mid-2000s research communities and industry have shifted their focus from uniprocessor to multiprocessor architectures, for both general purpose and embedded systems. In the beginning, these multiprocessor architectures were composed of a small number of processors, typically two to four (e.g., AMD 64 Athlon X2, Intel Core Duo). Over time this has changed, leading in recent years to architectures with dozens or even hundreds of processors [HDV+11, VEMR14, dDAB+13]. These architectures are referred to as *massively parallel*.

As the number of processors in execution platforms grows, it becomes evident that **traditional on-chip interconnections and memory subsystems are no longer adequate**. For instance, using a shared bus to connect dozens of processors to a global shared memory would result in unacceptable performance degradation due to high contention [BB04, BDM02, AMC+07].

Based on this observation, many research papers since the early 2000s have suggested the use of a scalable communication infrastructure that consists of an on-chip packet-switched network of interconnects, generally known as **Network-on-Chip (NoC)** [BB04, BDM02]. Nowadays, this suggestion has been translated into many commercially available (massively parallel) multiprocessors, that use NoCs as their communication infrastructure [VEMR14, dDAB+13, Ram]. Moreover, as the choice of the memory subsystem is concerned, "traditional" shared memory systems have been replaced by completely distributed memories [VEMR14] or to memories shared only within a cluster, as in [dDAB+13]. Although shared memory systems are still widespread in the embedded domain, they suffer from scalability problems, and NoC interconnection, together with distributed memories, are considered the most potential way to achieve scalability.

In its most common form, a NoC has the regular 2D mesh topology shown in the right part of Figure 1.4 [KJS+02]. The basic building blocks of a NoC-based multiprocessor are *tiles*, *routers* (denoted by *R* in Figure 1.4), and *network links*. We briefly describe these components in the following. Each **tile**, as shown in the left part of Figure 1.4, usually contains: (i) a processing element (*PE*), together with its data memory (*DM*) and instruction memory (*IM*); (ii) a Network Interface (*NI*); (iii) an adapter between the PE and the NI (*PE-to-NI adapter*). The PE is responsible to perform the actual computation of data, whereas the NI allows the bi-directional communication of the tile with the rest of the NoC. Communication over the NoC is performed using **messages**, which represent the unit of communication between tiles. In fact, when a message has to be sent outside the tile, the NI converts it into packets and performs the actual packet transmission from the tile to the router attached to it. Conversely, when a message has the considered tile as destination, the NI performs the reverse actions, i.e., it receives incoming packets and combines them to form the actual received message.

In most NoC implementations, the NI is endowed with **input and output buffers** for messages. Input buffers are used to allow the NI to receive a burst of messages from the NoC, and avoid congesting the network if the PE of the tile cannot process these messages quickly. Similarly, output buffers are useful when the PE produces messages to be sent over the NoC at a fast rate, but the network is congested and cannot immediately accept these messages.

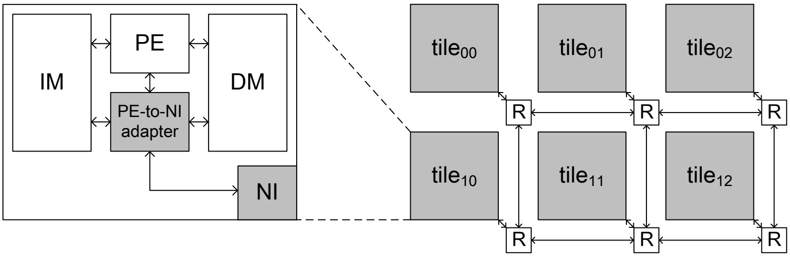


Figure 1.4: *The left part of the figure shows the internal structure of a tile. In each tile, the processing element (PE) performs computations using its data memory (DM) and instruction memory (IM). Communication with the rest of the NoC is handled by the Network Interface (NI) and the adapter between the PE and the NI (PE-to-NI adapter). The right part of the figure depicts the structure of a NoC composed of 6 tiles with a regular 2D mesh topology. Each tile is directly connected only to its corresponding router, and each router is directly connected only to its neighbor routers.*

Within the NoC, **routers** are responsible for dispatching the packets from the source tile (i.e., the tile which sends the message) to the destination tile (i.e., the tile which receives the message), according to defined routing rules. Finally, **network links** allow the communication among tiles and routers, and among different routers of the NoC. They are represented by bidirectional arrows in the right part of Figure 1.4.

Many NoC architectures have been proposed since the early 2000s. Popular examples include ×pipes [BB04], Æthereal [RGR$^+$03], Nostrum [KJS$^+$02], SoCBUS [WL03], Hermes [MCM$^+$04]. From the point of view of the quality of service provided by different NoC architectures, we can distinguish between:

- **Best-Effort NoCs** (e.g., ×pipes): in these NoCs, due to network contention, latency fluctuations for packet delivery can be experienced [BB04] and, in general, no guaranteed latency bound can be given.
- **Guaranteed service NoCs** (e.g., Æthereal): these NoCs can provide, for instance, an upper bound of the latency incurred by messages that traverse the network.

Clearly, for hard real-time applications, in which the timeliness of the computation is as important as its correctness, NoCs with guaranteed services are the preferred design choice. In fact, Chapters 5 and 6 of this thesis, which target hard real-time systems, assume communication infrastructures with guaranteed services. However, for applications with looser timing requirements best-effort NoCs can suffice and lead to other benefits compared to guaranteed service NoCs (e.g., smaller area and cost). Chapters 3 and 4 of this thesis, which are aimed at best-effort systems, consider a best-effort NoC as communication infrastructure.

1.2 Challenges in Embedded MPSoC Design

As described in the beginning of Section 1.1, the design of an embedded MPSoC is a process that entails several steps. The process starts with the definition of the required system functionality (using an application model), together with the specification of the execution platform on which the application will run. After a series of refinement steps, the design process is completed when a detailed description of the system hardware (e.g., at RTL) and of the software running on each processor is obtained.

The design trends described in Section 1.1, namely the widespread adoption of MoC-based design and scalable NoC interconnections, represent emerging design methodologies aimed at achieving high system performance on multiprocessor architectures. Ensuring high system performance, however, is not the only objective of embedded system designers. In the following sections we list other desirable features, specific to embedded systems, that are considered in this dissertation.

1.2.1 System Adaptivity

With the term "system adaptivity" we refer to the ability of the system to adapt to changing conditions imposed by the environment. These conditions are represented by parameters that can be divided in two classes:

1. Parameters belonging to the application. These parameters affect the way in which the application is executed. For instance, the resolution of a video decoding application is commonly represented by two parameters that specify the height and width of frames.
2. Parameters describing the status of the execution platform. For instance, a parameter can specify the number of active processors in the system.

In Chapters 3 and 4 of this thesis, we achieve system adaptivity in response to changes of the second set of parameters, the ones which describe the status of the execution platform. For example, refer again to the system sketched in Figure 1.2(b) on page 5, where the M-JPEG encoder application is mapped to a system composed of four PEs. Notice that each PE of the system is executing one or more tasks of the M-JPEG application. Then, two examples of scenarios that require system adaptivity, and can be handled by the techniques described in Chapters 3 and 4 of this thesis, are the following.

EX1: The system is battery-powered and the battery charge is running low. The user may decide to turn off a certain number of PEs to reduce the energy consumption of the system. This may result in a decreased quality of service of the M-JPEG application (e.g., reduced rate of video encoding). In addition, this scenario requires the system to migrate tasks that are running on PEs that will be switched off to the PEs that will be kept active, i.e., to change task mapping at run-time.

EX2: One or more PEs of the system may become permanently faulty. In this scenario, in order to maintain the functionality of the system, tasks mapped on faulty PEs must be migrated to functional PEs.

Note that system adaptivity can be implemented in a computing system in different ways. Consider the hardware/software stack in Figure 1.5, which abstracts the structure of a computing system. At the top of this stack there is the *application layer*. As mentioned in Section 1.1.1, in the case of embedded multiprocessors, applications are specified using a MoC. Generally, these applications are scheduled on the system by an Operating System (OS), represented by the middle layer in the figure. The OS acts as an interface between the application layer and the *hardware layer*, which lays at the bottom of the stack in Figure 1.5.

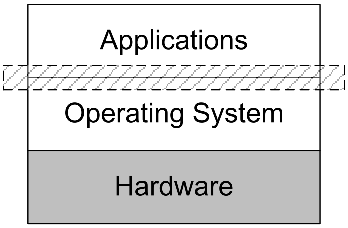


Figure 1.5: *Sketch of the hardware/software stack of a computing system. The dashed area highlights the interface between the application layer and the OS layer.*

In this thesis we consider approaches that provide system adaptivity at the interface between the application layer and the OS layer, as highlighted by the dashed area of Figure 1.5. In this context, approaches that provide system adaptivity already exist in the literature [NVC10, Gab09, BABP06, AACP08, NKG$^+$02]. Most of them allow, to a lesser or greater extent, to change the mapping of the application tasks at run time, therefore they allow **task migration**. As mentioned in the examples *EX1* and *EX2* above, task migration is an essential requirement to guarantee system adaptivity as considered in this dissertation. However, we argue that existing solutions present shortcomings in either the extent to which system adaptivity is supported, or in the scalability of the proposed approaches. In this dissertation we address these shortcomings.

Note that several other research works target system adaptivity by considering that the *parameters* of applications can change. They allow this change of parameters directly at the application level [BB01, SGTB11, BDLT13]. In order to do so, they use *adaptive* MoCs that can model the possibility to change the parameters of applications at run-time [ZNS11, Zha15]. The variation of application parameters at run-time is also referred to as *mode change*. In this thesis, we do not explicitly consider such adaptive MoCs mainly for two reasons. The first reason is that in some cases applications simply do not show inherent (algorithmic) adaptivity, i.e., their parameters are fixed and do not change. The second reason is that we want to encompass the cases in which system adaptivity is required in response to faults detected in the execution platform. These cases cannot be easily expressed by using an adaptive MoC. Moreover, if an adaptive MoC is used to model applications, the techniques presented in Chapters 3 and 4 of this thesis may be used as a way to change the mapping of tasks at run-time, consequently to a mode change.

1.2.2 Timing Requirements

As mentioned earlier, in many cases embedded systems must satisfy timing requirements in their execution. Based on the importance of these timing requirements, embedded systems can be divided in the following categories.

- **Hard Real-time (HRT) Systems:** in these systems, failing to meet timing requirements results in system failure.
- **Soft Real-time (SRT) Systems:** within this category, failing to meet timing requirements results in degraded system performance.
- **Best-Effort (BE) Systems:** in these systems timing requirements are not specified and systems run at the best of their capacity.

Clearly, HRT systems pose a more difficult challenge to embedded designers because they require an additional constraint, timing requirements, to be satisfied. In these systems the result of the computation must be provided within a certain time interval, otherwise it is useless. In some cases, violation of timing constraints may result in catastrophic consequences. Among embedded systems, HRT requirements are extremely common [Mar11]. In the special case of embedded *streaming* systems (the scope of this dissertation), timing requirements that are typically guaranteed are **throughput** and **latency**. Throughput refers to the amount of output tokens that the application can produce in a defined time period. Latency measures the time elapsed between the arrival of an input token in the system and the production of the corresponding output token by the system.

A large variety of scheduling techniques which guarantee HRT behavior for multiprocessor systems have been proposed over the years. In the remainder of this section, we will categorize the most widely adopted techniques to guarantee HRT constraints using the following three classes.

- **Class I** - Scheduling techniques based on direct analysis of the dataflow MoC specification of applications.
- **Class II** - Scheduling techniques based on classical real-time scheduling analysis [DB11, BBB15].
- **Class III** - Scheduling techniques that convert dataflow MoC-specified applications into real-time task sets compatible with classical real-time scheduling analysis.

Each of these classes is briefly introduced in what follows.

Class I - HRT Guarantees by analyzing Dataflow MoC Application Specifications

This class comprises most of the techniques in the literature which guarantee hard real-time behavior of dataflow applications. Relevant examples of this class are the approaches described in [LH89, MB07, GGS$^+$06, SB09]. Techniques belonging to this class require applications to be specified according to a dataflow MoC with high analyzability. For this reason (recall Figure 1.3 on page 7), these approaches do not use the PPN MoC to specify the applications. This is because performing an analysis of timing guarantees directly on a PPN model is rather difficult, if not impossible, due to the complexity of communication patterns which can occur in PPN models[3] [Zha15].

Contrary to PPNs, for HSDF, SDF, and CSDF graphs analysis of timing guarantees is possible. Recall that by using a dataflow MoC an application is represented as a directed graph, where graph nodes represent tasks of the application and graph edges represent inter-task data dependences. Then, several techniques belonging to Class I derive certain hard real-time guarantees by analyzing the properties of the application graph. For instance, [MB07] considers applications specified using an SDF graph [LM87b]. This graph is converted to an equivalent HSDF graph [LM87b], for which the maximum achievable throughput can be determined analytically. In fact, the maximum achievable throughput of an HSDF graph is the inverse of its *Maximum Cycle Mean (MCM)* [Das04]. However, note that this throughput analysis is only applicable to the derived HSDF graph, which is often exponentially larger in size compared to the original SDF graph [LM87b]. To avoid the exponential explosion of the problem size, other techniques avoid the conversion of the original SDF graph to the equivalent HSDF graph [GGS$^+$06]. They, instead, explore the *state-space* of the given SDF graph to calculate the maximum achievable throughput.

Note that all the scheduling techniques belonging to Class I share a significant drawback: in order to provide certain timing properties, a complex design space exploration is needed to determine the minimum number of processors required to schedule the application(s) and the mapping of tasks to these processors. With regard to this drawback, techniques belonging to Class II and III, which are presented in the following, show higher efficiency.

Class II - HRT Guarantees using Classical Real-time Scheduling

The second class of scheduling techniques uses results from classical hard real-time scheduling theory for multiprocessors [DB11, BBB15]. These techniques consider application models which are more restrictive than the dataflow MoC considered in Class I. In fact, scheduling techniques of Class II analyze programs in which tasks conform to a certain *real-time task model*. The most influential example of such task models is the *periodic* real-time task, which was introduced in the seminal work [LL73] of Liu and Layland. In this model, each task is invoked in a strictly periodic way. Each task invocation is called a *job*, and each job must be completed before a certain

---

[3] However, note that there exist techniques which can convert a PPN to its input-output equivalent CSDF graph, see Chapter 3 of [Zha15].

deadline. In the simplest form of a periodic real-time task model, called *Implicit Deadline Periodic* model, the deadline of a job coincides with the release time of the successive job of the same task. Moreover, all tasks in the system are independent among each other, i.e., jobs of a task do not depend on the completion of any jobs of any other tasks in the system. Under all these assumptions, given the *Worst-Case Execution Time (WCET)* of each task in the system, [LL73] proves that a simple *schedulability test* ensures that no deadline will ever be missed under the *Earliest Deadline First (EDF)* scheduling algorithm. In addition, [LL73] proves that EDF is *optimal* for uniprocessor systems, i.e., if the periodic task set can be scheduled by any other scheduling algorithm, then it can also be scheduled by EDF[4].

However, when the scheduling analysis shifts from uniprocessor to multiprocessor systems EDF loses its optimality. In general, scheduling algorithms that consider multiple processors not only have to assign priorities among jobs of different tasks, but they also have to decide *on which processor* each task must be executed (spatial scheduling). This fact adds another dimension to the scheduling problem, making it more complex. Several scheduling algorithms for multiprocessor systems have been proposed in the literature [DB11, BBB15]. Based on the way they allocate tasks to processors, most of the proposed approaches can be classified in either *partitioned* or *global* scheduling algorithms.

Under global scheduling algorithms, all the tasks can migrate among all the processors. Such algorithms can be optimal for multiprocessor systems, which means that they can fully exploit the available computational resources (see for instance [BCPV96]). However, this comes at the cost of high scheduling overheads due to excessive task preemptions and migrations. Partitioned scheduling algorithms, by contrast, incur no migration overhead because each task is statically allocated to a single processor. Moreover, they incur much lower preemption overheads compared to optimal global scheduling algorithms. However, partitioned algorithms are not optimal. This implies that, in general, these algorithms may require twice as many processors to schedule certain sets of tasks compared to an optimal global scheduler [LDG04].

Recently, a third class of algorithms, called *hybrid* scheduling algorithms, has been proposed. Among hybrid scheduling algorithms, **semi-partitioned algorithms** (e.g., [ABD08, AEDC14]) have gained significant attention within the real-time research community. Under semi-partitioned algorithms, most of the tasks are statically allocated to processors and only a subset of the tasks is allowed to migrate among different processors. Migrating tasks follow a migration pattern derived at design-time. Thus, semi-partitioned approaches represent a “middle ground” between partitioned and global scheduling algorithms. In general, semi-partitioned scheduling algorithms require less processors than partitioned algorithms to schedule certain task sets. At the same time, these algorithms do not incur large task migration and preemption overheads like global scheduling algorithms [BBA11].

Note that most of the scheduling techniques that fall into Class II share a common drawback: they assume that the tasks of the applications comply with a rather

---

[4] In other words, no other scheduling algorithm can outperform EDF in terms of schedulability of periodic task sets on uniprocessors.

simple task model, for instance the independent periodic task model. These simple task models are not easily applicable to dataflow applications, in which tasks are dependent among each other. However, the scheduling techniques of Class II provide the following advantages:

- The minimum number of processors needed to schedule a certain task set, and the assignment of tasks to processors, can be derived in a fast analytical way.
- *Temporal isolation*[5] among different applications is guaranteed.
- Applications can be loaded at run-time in the system, provided that the schedulability test pertaining to the adopted scheduling algorithm is satisfied.

Class III - HRT Guarantees by converting Dataflow MoC Application Specifications to Real-time Task Sets

In recent years, several approaches that bridge the gap between dataflow MoCs and real-time task models have been proposed [God98, BS11, BS12, LA10, BTV12]. In these approaches, applications are specified using a dataflow MoC where, as mentioned earlier, tasks have data dependencies. Then, these MoC-based application specifications are converted to a set of tasks which comply with some real-time task model, usually independent among each other. Finally, based on the obtained (independent) real-time task set, the scheduling approaches of Class III apply algorithms from hard real-time scheduling theory [DB11, BBB15] to determine in a fast and analytical way the minimum number of processors that guarantee the required performance and the mapping of tasks to processors.

In this thesis, in particular, we consider the scheduling technique proposed in [BS11, BS12]. The analysis of [BS11, BS12] accepts, as input, applications specified as *acyclic* CSDF graphs [BELP96]. The choice of this kind of MoC to specify the input applications makes the result of [BS11,BS12] applicable to most streaming applications. In fact, it has been shown in [TA10] that around 90% of streaming applications can be modeled as acyclic SDF graphs. Note that acyclic SDF graphs are a subset of the acyclic CSDF graphs considered in [BS11, BS12]. Note also that throughout this thesis, unless otherwise specified, we will assume all considered (C)SDF graphs to be acyclic.

In a nutshell, the core of the analysis in [BS11, BS12] is the conversion of the input application(s) into a set of independent periodic real-time tasks. This conversion is explained in Section 2.3 of this thesis. Then, based on the derived independent periodic real-time task set, *partitioned* scheduling algorithms from multiprocessor real-time scheduling theory (recall Class II of scheduling techniques described earlier) are used to derive the number of processors required to execute the application(s) and guarantee the desired timing requirements.

1.2.3 Cost

Embedded systems are sold in massive quantities, for instance in consumer electronics and cars. In these contexts, keeping the cost of the system competitive is of vital

---

[5] Temporal isolation refers to the ability to start and stop applications in the system without violating the timing requirements of the other applications.

importance. Therefore, an embedded system designer is required to make an effective use of the available hardware resources. When designing an embedded system, two possible scenarios may occur. In the first scenario the hardware platform is yet to be designed and the designer shall **utilize the least amount of resources to implement the required functionality**. In the second scenario, the hardware platform is already given, and the designer shall exploit the available hardware resources efficiently by **implementing as many useful applications as possible on the given hardware**.

As described in Section 1.4.2, one of the contributions of this thesis is aimed at improving the exploitation of hardware resources in HRT embedded streaming systems.

1.2.4 Energy Efficiency

The challenge of power management (and therefore of energy management) of modern computing systems has been explicitly recognized by the International Technology Roadmap for Semiconductors (ITRS) report of 2013 [Int13]. The ITRS report points out, in particular, that among successive technology generations transistor density doubles, while cost-effective heat removal from chips remains almost flat. This means that in the near future chips will become increasingly powerful and rich in terms of hardware resources. However, in these chips, power and energy management will play an increasingly important role due to the limitations in dissipating heat.

Improper power management can affect a computing system in mainly two ways. First, by requiring a huge amount of energy to perform the desired computation. Second, by generating excessive heat, which must be somehow dissipated in order to avoid hardware failures due to high temperatures. These concerns are even more significant in the case of embedded systems. This is because several embedded systems are battery powered and therefore cannot afford to consume huge amounts of energy. Also, many embedded systems operate in safety-critical environments where system failures could lead to catastrophic consequences.

Several energy and power-efficient techniques have been proposed in the literature. These techniques mainly employ two mechanisms to reduce power consumption: Voltage/Frequency Scaling (VFS) and Power Management (PM) [Jha01]. VFS reduces dynamic power consumption by adjusting the voltage and operating frequency of processors. Conversely, PM exploits idle times of processors by switching them to a sleep mode. In multiprocessor systems, VFS and PM techniques can be applied at the granularity of the single PE (so-called *per-core* VFS/PM), of the whole system (so-called *global* VFS/PM), or at an intermediate level, by dividing the chip in clusters (also called *voltage islands*) in which VFS/PM can be applied independently from the rest of the system. Clearly, per-core VFS/PM provides greater flexibility in devising power/energy management techniques and algorithms compared to clustered or global approaches. However, many recent research works (e.g., [DA10,SJPL08,Lee09]) and several industrial examples (e.g., [HDV+11, dDAB+13]) have shown that for massively parallel architectures per-core VFS/PM incurs large hardware overheads and therefore is not a feasible solution.

As mentioned in Section 1.4.2, in this thesis we take this indication into account, therefore we propose an approach that uses a VFS technique to improve the system energy efficiency.

1.3 Problems Addressed in this Thesis

Having summarized the design trends and challenges pertaining to the embedded system domain (in Section 1.1 and Section 1.2), we proceed now to define the problems addressed in this dissertation.

In this thesis we consider two different categories of systems, based on the importance of their timing requirements, as mentioned in Section 1.2.2: **Best-Effort (BE)** and **Hard Real-Time (HRT)** systems. Given the very different nature of BE and HRT systems, we exploit different MoCs and analysis techniques according to the considered category of systems. The problems addressed in the context of BE systems differ, as well, from the problems considered for HRT systems. Research problems concerning BE systems are presented in Section 1.3.1, whereas those regarding HRT systems are presented in the Section 1.3.2.

All the research problems addressed in this thesis, however, consider carefully the trends in embedded multiprocessor design presented in Section 1.1.1 and Section 1.1.2, which we summarize in the following two points.

1. Model-based Design is an instrumental methodology to tackle the complexity of modern embedded multiprocessor systems. This is valid for both BE and HRT embedded systems.
2. For emerging massively parallel architectures, research and industry trends are shifting towards NoC-based interconnections and distributed memories. This choice is necessary to guarantee design scalability.

1.3.1 Best-Effort Systems

In the context of BE systems, this thesis addresses the problem of **providing system adaptivity**, one of the design challenges mentioned in Section 1.2, **by means of dedicated software components**. We consider the following design decisions in order to allow system adaptivity.

- We model applications using the most expressive and succinct MoC mentioned in Section 1.1.1, namely the PPN MoC (recall the comparison among MoCs considered in this thesis, shown in Figure 1.3 on page 7). As mentioned earlier, timing analysis for PPN MoC is very difficult, if not impossible, but this is not a concern for BE systems because no timing requirements are specified.
- We consider a NoC-based architecture, as shown in the right part of Figure 1.4 on page 9, with completely distributed memories. We assume that tiles of the NoC can only access their own memory. Therefore, if tile *b* requires some data produced by tile *a*, this data has to be explicitly sent over the NoC from tile *a* to tile *b*, in the form of a message. This kind of design is found in many NoCs proposed by the research community (e.g., ×pipes [BB04]), to guarantee scalability and minimize hardware cost.

- We aim at providing system adaptivity by allowing to change the mapping of tasks to processors at run-time, i.e., by implementing a mechanism of **task migration**.

Assuming the three design decisions listed above, the problem of providing system adaptivity on NoC-based architectures yields to the following two research questions.

- **Research question 1.** Although the PPN MoC is suitable to be implemented on distributed architectures, the semantics of PPNs and the structure of NoC interconnections do not exactly match. Therefore, in this thesis we provide an answer to this question: **how to implement the semantics of the PPN MoC on NoC-based platforms in an efficient way?**
- **Research question 2.** Assuming that an answer to the research question above is given, **how can we implement a task migration mechanism which respects the PPN MoC semantics and can be deployed to a NoC-based architecture with completely distributed memories?** In particular, in order to reduce the overhead incurred by task migration, we consider task migration using code replication, where the code of the migrating task is copied on *all* the PEs that may execute the task at run-time.

1.3.2 Hard Real-Time Systems

In the context of HRT systems, in this dissertation we use the methodology proposed in [BS11, BS12] as a basis and research driver. As mentioned in Section 1.2.2, the methodology of [BS11,BS12] is particularly appealing because it allows designers to derive analytically the amount of resources (e.g., number of PEs, memories) necessary to execute a set of applications, specified as acyclic CSDF graphs, with guaranteed hard real-time behavior.

So far, in [BS11,BS12] and in all the scheduling methodologies mentioned in Class III of Section 1.2.2, only partitioned or global scheduling algorithms from multiprocessor hard real-time theory [DB11,BBB15] have been considered. Advantages and drawbacks of both of these approaches have been already mentioned in Section 1.2.2, under Class II methodologies. In the context of emerging embedded multiprocessor architectures, where memory is usually distributed, global scheduling algorithms incur an additional drawback which we explain in the following. As mentioned in Section 1.3.1, in order to reduce the overhead of task migration on distributed memory architectures the code of each migrating task is copied to all the PEs that may execute that task at run-time. In the case of global scheduling algorithms this means that the code of *all* tasks must be replicated on *all* PEs, resulting in a huge memory overhead. By contrast, partitioned scheduling algorithms do not incur any memory overhead because all tasks are statically allocated. However, they are not optimal for multiprocessor systems.

Semi-partitioned algorithms represent a middle ground between partitioned and global scheduling algorithms. Under these algorithms, **task migration** is allowed. However, only a few tasks are allowed to migrate and therefore need to replicate their code in distributed memory architectures. Therefore, semi-partitioned algorithms seem to be more applicable to such architectures, compared to global scheduling algorithms, because they do not incur the excessive memory overhead of global approaches mentioned in the paragraph above. In this thesis we focus, especially, on semi-partitioned approaches with *restricted migrations*. In these approaches, migrations can happen at job boundaries only, i.e., when a job is released on a PE, it cannot migrate to another PE until its completion. This is a favorable feature in distributed memory systems, because allowing migrations only at job boundaries reduce the amount of data (state) to be transferred from one processor to the next.

The scheduling methodology of [BS11,BS12] shows that an application, modeled as an acyclic CSDF graph, can be scheduled using a **hard real-time partitioned scheduling algorithm** as a set of real-time periodic tasks. **In this thesis, we extend that scheduling methodology of [BS11,BS12] by allowing semi-partitioned scheduling algorithms with restricted migrations to execute streaming applications using real-time scheduling techniques.** . In particular, in this thesis we provide an answer to the following two questions. **Can semi-partitioned approaches with restricted migrations be exploited to achieve a more efficient utilization of the available hardware resources (see the design challenge in Section 1.2.3)?** And, **can such approaches be used together with VFS techniques to improve the energy efficiency of the system (see the design challenge in Section 1.2.4)?**

1.4 Research Contributions

The contributions of this thesis address the research questions presented in Section 1.3. The common trait of the techniques proposed in this dissertation is the **exploitation of task migration**. Our proposed techniques apply task migration in a specific way depending on the considered category of systems (BE or HRT). In fact, the techniques proposed for BE systems allow task migration to occur at any time, triggered by the user or by the environment (e.g., by a hardware fault). By contrast, techniques aimed at HRT systems perform task migration according to a precise temporal and spatial pattern defined by the adopted semi-partitioned scheduling algorithm. For instance, a task may be allowed to migrate periodically between two processors, alternating the first and the second processor in the execution of successive jobs of the same task. This results in an equal division of the workload of the task among the two considered processors.

The research contributions of this thesis are divided in two parts:

- The **first part** (Chapters 3 and 4) is aimed at best-effort systems. Its contributions are summarized in Section 1.4.1.
- The **second part** (Chapters 5 and 6) is aimed at hard real-time systems. Its contributions are summarized in Section 1.4.2.

1.4.1 Exploiting Task Migration to achieve System Adaptivity in Best-Effort Systems

In the **first part** of this thesis, namely Chapters 3 and 4, we propose the software stack depicted in the left part of Figure 1.6. Within this stack, we introduce an intermediate layer called *middleware*. This layer stays in between the applications, specified as PPN processes, and the underlying OS. **The middleware layer represents the main contribution of the first part of this thesis**, and is aimed at allowing adaptivity in BE systems.

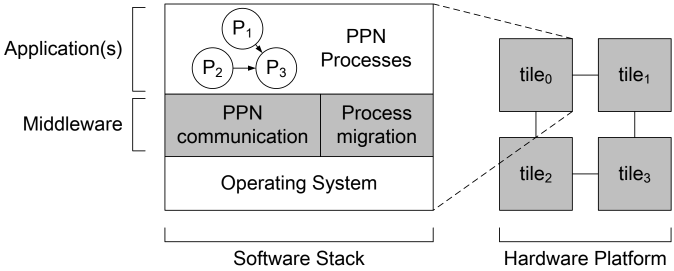


Figure 1.6: *Software stack (left) proposed to achieve adaptivity in BE systems. The middleware layer is denoted by the shaded area. The stack is deployed on each tile of the hardware platform (right).*

As shown in Figure 1.6, the middleware layer comprises two components, which are inter-dependent. The first component, presented in Chapter 3, addresses the problem of *PPN Communication* on NoC-based platforms with distributed memories. More precisely, it converts PPN communication primitives to the corresponding execution platform primitives. We propose and investigate several approaches to efficiently implement PPN Communication on NoCs. The proposed approaches differ in the extent of the required synchronization among tiles of the NoC. However, all of these approaches allow PPN process to communicate regardless of the actual spatial mapping of processes to processors, i.e., they are mapping independent. This is a fundamental requirement in order to maintain the functionality of the system in case of task/process migration(s).

The second component of the middleware layer is proposed in Chapter 4 and implements the *Process Migration*[6] mechanism. We devise a process migration mechanism that complies with the following requirements. The first requirement is that process migration, once triggered, must be completed within a certain known time frame. We refer to such property as *predictability*. The second requirement is that task migration can be triggered in the system at any time. Finally, the third requirement is that the code necessary to allow task migration must be generated in an automated way, without the manual intervention of the designer. The efficiency and applicability of the proposed software stack is shown in a real-life case study in Chapter 4.

---

[6] In this thesis, we will use the terms *task* and *process* interchangeably. In this case, we refer to *process migration* because we allow migration of PPN *processes*.

1.4.2 Exploiting Semi-partitioned Approaches in Hard Real-Time Scheduling of (C)SDF Graphs

In the **second part** of this thesis, namely Chapters 5 and 6, we study the applicability of semi-partitioned approaches in hard real-time scheduling of (C)SDF graphs. Our contributions extend the scheduling framework of [BS11, BS12], which considers only *partitioned* scheduling approaches.

In particular, in Chapter 5 we make the following contributions.

- **Contribution 1.** We extend the framework of [BS11, BS12] such that soft real-time (SRT) scheduling algorithms can be used to schedule the tasks of an application specified as a (C)SDF graph. Under SRT schedulers, tasks can miss their deadlines by a bounded value called *tardiness*. Despite that, our approach can still provide **hard real-time guarantees to the input/output interfaces** of the application with the environment.
- **Contribution 2.** Based on the previous point, we consider the SRT semi-partitioned scheduler EDF-fm [ABD08] (Earliest Deadline First based where tasks can be either fixed or migrating) to schedule the applications. For this semi-partitioned approach, we propose a task allocation heuristic that is aimed at:
  - reducing the minimum number of processors required to schedule the applications compared to a pure partitioned scheduling algorithm;
  - keeping low the memory and latency overhead caused by the SRT scheduler compared to a pure partitioned scheduling algorithm.
- **Contribution 3.** We show on a set of real-life benchmarks that our semi-partitioned approach can lead to significant benefits by reducing the number of processors required to schedule a given application, compared to a partitioned approach, while achieving the same throughput. However, this reduction in number of required processors comes at the cost of increased memory requirements and latency of applications.

In Chapter 6 we show that semi-partitioned approaches can achieve higher energy efficiency compared to partitioned ones. Chapter 6 builds upon Contribution 1 of Chapter 6. In particular, it makes the following contributions.

- **Contribution 1.** We propose a novel SRT semi-partitioned scheduling algorithm with restricted migrations, called EDF-ssl (Earliest Deadline First based semi-partitioned stateless), which is targeted at streaming applications. EDF-ssl is designed to be used in combination with VFS techniques, and exploits the presence of stateless tasks[7] to improve the energy efficiency of the system.
- **Contribution 2.** We use EDF-ssl in combination with a VFS technique, assuming that VFS is supported globally over the considered set of processors (i.e., not per-core) with a discrete set of operating voltage/frequency modes. We derive the conditions that ensure a valid scheduling of the tasks of applications in two cases:

---

[7] A task is called *stateless* if it does not keep an internal state between two successive jobs. A more formal definition of this property for the considered MoCs is given in Section 2.1.

  - First, when we use the lowest frequency which guarantees schedulability and is supported by the system.
  - Second, when we use a periodic frequency switching scheme that preserves schedulability and can achieve higher energy savings.

  In general, our proposed EDF-ssl allows an even distribution of the utilization of tasks among the available processors. In turn, this enables processors to run at a lower frequency, which yields to lower power (and, therefore, energy) consumption.
- **Contribution 3.** We show that, compared to a purely partitioned scheduling approach, our technique achieves the same application throughput with significant energy savings (up to 64%) when applied to real-life streaming applications. These energy savings, however, come at the cost of higher memory requirements and latency of applications.

1.5 Thesis Organization

The remainder of this thesis is organized as follows. **Chapter 2** provides an overview of the MoCs considered in this thesis, some relevant analysis techniques and results from real-time scheduling theory, and the methodology for hard real-time scheduling of (C)SDF graphs proposed in [BS11, BS12]. All of these concepts and techniques are instrumental to understand the contributions of this thesis.

Chapters 3 to 6, which present the contributions of this dissertation, are written in a self-contained way. This means that each of these chapters includes an introduction and related work section specific to the addressed research problem. We summarize the content of each of these chapters in the list below.

- **Chapter 3** describes the first component of the middleware layer (recall Figure 1.6 on page 20) that we propose to achieve system adaptivity in BE systems. The first middleware component implements the communication of PPN processes on NoC-based architectures in an efficient way.
- **Chapter 4** proposes the process migration mechanism for PPNs on NoC-based architectures, which represents the second component of the middleware layer in Figure 1.6.
- **Chapter 5** describes our semi-partitioned scheduling approach for CSDF graphs with HRT constraints.
- **Chapter 6** presents the final contribution of this thesis, which is based on a novel semi-partitioned scheduling algorithm (EDF-ssl) together with a VFS technique aimed at improving the system energy efficiency.

Finally, **Chapter 7** draws some conclusions based on the results of this thesis and suggest possible directions for future work.

Chapter 2

Background

IN this chapter, we introduce the mathematical notations, definitions, concepts, and existing theoretical results that are necessary to understand the contributions of this thesis.

We first provide in Table 2.1 a summary of the mathematical notations used in this thesis.

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
$\hat{x}$	The maximum value of x
mod	The integer modulo operator
${}^{x}V$	An x-partition of a set V (see Definition 2.2.6)

Table 2.1: *Summary of mathematical notation.*

Then, in Section 2.1, we describe in further detail the MoCs used in this dissertation. A preliminary introduction of these MoCs was given earlier in Section 1.1.1. In addition, in Section 2.2 we present results and definitions from hard real-time scheduling theory that are instrumental to understand our contributions in the context of HRT systems (Chapters 5 and 6). Finally, in Section 2.3 we describe the methodology for hard real-time scheduling of CSDF graphs proposed by [BS11, BS12]. In fact, our contributions presented in Chapters 5 and 6 represent an extension of the framework proposed in [BS11, BS12], therefore a thorough introduction to that framework is necessary.

2.1 Dataflow Models of Computation

As mentioned in Section 1.1.1, dataflow MoCs represent a good match for streaming applications because they allow to express the parallelism available in these kind of applications in a natural way. In this thesis we consider the MoCs described in Sections 2.1.1-2.1.3, namely (H)SDF, CSDF, and PPN. In both (C | H)SDF[1] and PPN MoCs applications are specified in the form of directed graphs in which graph nodes represent the tasks (active entities) of the application and graph edges represent inter-task data dependencies. In (C | H)SDF, nodes of the application graph are called *actors*, whereas nodes in a PPN are called *processes*. The actor-based MoCs considered in this thesis, (H)SDF and CSDF, are presented in Sections 2.1.1 and 2.1.2, respectively. These MoCs are used to specify the input applications in the techniques for HRT systems proposed in Chapters 5 and 6. The process-based MoC considered in this thesis, PPN, is described in Section 2.1.3 and is used to specify applications in the approaches proposed for best-effort (BE) systems, which are presented in Chapters 3 and 4.

2.1.1 (Homogeneous) Synchronous Dataflow ((H)SDF)

The **Synchronous Dataflow (SDF)** MoC is introduced in [LM87b]. Under this MoC, an application is modeled as a directed multigraph $G = (A, E)$ where A is the set of actors and E is the set of edges. Actors represent tasks of the application and edges represent inter-task data dependencies. Actors communicate over edges generating a stream of data, which is divided in atomic data objects called *tokens*. An edge $e_u \in E$ represents a first-in first-out (FIFO) buffer and is defined by a tuple $e_u = (A_i, A_j)$. This tuple means that the edge is directed from actor A_i (called *source*) to actor A_j (called *destination*). We define input and output actors of graph G as follows.

Definition 2.1.1. *(Input actor).* An input actor of graph G is an actor that receives the input stream of the application from the environment.

Definition 2.1.2. *(Output actor).* An input actor of graph G is an actor that produces the output stream of the application to the environment.

An execution of an actor $A_i \in A$ is called a **firing** or **invocation**. In this thesis we denote the jth invocation (with $j \in \mathbb{N}_0$) of actor A_i as $A_{i,j}$. Invocation $A_{i,j}$ can begin its execution only if enough input data is present on all its input edges. During one invocation, actor A_i consumes input data from all its input edges, processes this data according to a function f_i, and writes the output data to its output edges. The amount of data read/written from/to each input/output edge is fixed, known at compile-time and it is called consumption/production *rate*. For each $A_i \in A$ we can define a set of *predecessor* and *successor* actors, denoted by $\mathsf{prec}(A_i)$ and $\mathsf{succ}(A_i)$, respectively. These sets are defined as follows.

$$\mathsf{prec}(A_i) = \{A_j \in A : \exists e_u = (A_j, A_i)\} \tag{2.1}$$

$$\mathsf{succ}(A_i) = \{A_j \in A : \exists e_u = (A_i, A_j)\} \tag{2.2}$$

---

[1] For the sake of brevity, we identify CSDF, SDF and HSDF MoCs with the acronym (C | H)SDF.

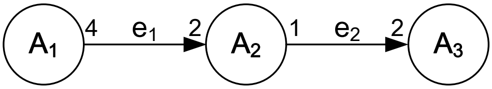


Figure 2.1: *Example of an SDF graph composed of actors A_1, A_2, A_3 and edges e_1, e_2. Numbers over the edges indicate the production/consumption rates of source/destination actors of that edge. For instance, each invocation of actor A_2 consumes 2 tokens from e_1 and produces 1 token to e_2.*

We assume that any input actor A_{in} has no predecessors and any output actor A_{out} has no successors, i.e., $\mathsf{prec}(A_{\text{in}}) = \emptyset$ and $\mathsf{succ}(A_{\text{out}}) = \emptyset$. Moreover, we can define for each $A_i \in A$ a set of *input* and *output* edges, denoted by $\text{inp}(A_i)$ and $\text{out}(A_i)$, respectively.

An example of an SDF graph is shown in Figure 2.1. The numbers over the edges indicate the production/consumption rates of source/destination actors of that edge. Consider edge $e_u = (A_i, A_j)$. The production rate of source actor A_i over edge e_u is denoted by x_i^u. Conversely, the consumption rate of destination actor A_j over edge e_u is denoted by y_j^u. For instance, each invocation of actor A_1 in Figure 2.1 produces 4 tokens to e_1, and each invocation of actor A_2 consumes 2 tokens from the same edge. Therefore, $x_1^1 = 4$ and $y_2^1 = 2$.

A special case of the SDF MoC is the **Homogeneous SDF (HSDF)**, that is an SDF in which all the production/consumption rates of all the actors are equal to one. An example of an HSDF graph is given in Figure 1.2(a) on page 5, where production/consumption rates of actors are omitted because they are all equal to one.

Since streaming applications process continuous streams of data, we are interested in determining a schedule of the SDF graph that can continue indefinitely, using a finite amount of memory to implement the FIFO channels corresponding to the edges of the graph. Such a schedule can be derived at compile-time if the SDF graph is **consistent** and **deadlock-free**.

An SDF graph G is consistent [LM87a] if its *balance equation*, given below, has a positive integer solution.

$$\mathbf{\Gamma_G} \cdot \vec{q} = \vec{0} \tag{2.3}$$

In the expression above, $\mathbf{\Gamma_G} \in \mathbb{Z}^{|E| \times |A|}$ is called **topology matrix** and $\vec{q}$ is called **repetition vector**. The topology matrix is constructed as follows:

$$\Gamma_{uj} = \begin{cases} x_j^u & \text{if actor } A_j \text{ produces on edge } e_u \\ -y_j^u & \text{if actor } A_j \text{ consumes from edge } e_u \\ 0 & \text{otherwise.} \end{cases} \tag{2.4}$$

In particular, the repetition vector with the smallest norm is called **basic repetition vector**. In this thesis, unless otherwise specified, we will utilize the basic repetition vector of a graph to perform our analyses. The meaning of the repetition vector is the following. If every actor A_i of the graph is fired q_i times, where q_i is the ith component of the repetition vector, then the net change of the number of tokens in the FIFO channels is zero.

For the example in Figure 2.1, the topology matrix $\mathbf{\Gamma_G}$ is given below.

$$\mathbf{\Gamma_G} = \begin{bmatrix} 4 & -2 & 0 \\ 0 & 1 & -2 \end{bmatrix}$$

Its (basic) repetition vector $\vec{q_G}$, derived using Equation (2.3), is:

$$\begin{aligned} \vec{q_G} &= \left[q_{A_1}, q_{A_2}, q_{A_3}\right]^T \\ &= [1,2,1]^T \end{aligned}$$

Note that any vector $\vec{q}'$ obtained by multiplying the basic repetition vector $\vec{q_G}$ by a positive natural number is also a repetition vector of G, i.e., it satisfies Equation (2.3). Note also that the existence of a positive integer solution to Equation (2.3) is only a necessary condition to execute an SDF graph indefinitely with a periodic schedule. Another condition that must be satisfied is the absence of deadlocks, which can be verified by constructing a periodic admissible schedule [LM87a] of the graph. An SDF graph that has no deadlock is called *deadlock-free*, or *live*. An important property of the SDF MoC is that the consistency and liveness of an SDF graph can be verified at compile-time. In this thesis we consider only consistent and live SDF graphs.

2.1.2 Cyclo-Static Dataflow (CSDF)

The Cyclo-static Dataflow (CSDF) MoC [BELP96] is a generalization of SDF. Similar to SDF, a CSDF graph $G = (A, E)$ also consists of a set of actors A and a set of edges E. However, contrary to SDF, the behavior of CSDF actors is cyclic, as explained in the following.

Each CSDF actor A_i has a certain number of *phases*, denoted by Ω_i. The execution of each phase φ is associated with a certain function $f_i(\varphi)$. Therefore, we can define an **execution sequence** $[f_i(0), f_i(1), \cdots, f_i(\Omega_i - 1)]$ which links each phase to the corresponding executed function. Moreover, production/consumption rates for each output/input edge are also defined for each phase. Thus, for each actor A_i, the following sequences can be defined.

- **Consumption sequence** for each input edge e_u:

$$[y_i^u(0), y_i^u(1), \cdots, y_i^u(\Omega_i - 1)]$$

- **Production sequence** for each output edge e_u:

$$[x_i^u(0), x_i^u(1), \cdots, x_i^u(\Omega_i - 1)]$$

Notice that the length of all these sequences is Ω_i, the number of phases.

Phases of a CSDF actor A_i are executed in a cyclic fashion. That is, during invocation $A_{i,n}$ (with $n \in \mathbb{N}$) of actor A_i, function $f_i((n) \bmod \Omega_i)$ is executed. Similarly, for each input edge e_u, $y_i^u((n) \bmod \Omega_i)$ tokens are consumed and for each output edge e_u,

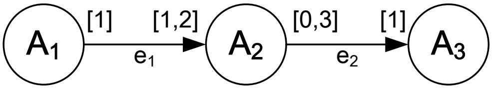


Figure 2.2: *Example of a CSDF graph.*

$x_i^u((n) \bmod \Omega_i)$ tokens are produced. The cumulative number of tokens consumed by invocations $A_{i,0}$ to $A_{i,n}$ of actor A_i from its input edge e_u is denoted by:

$$Y_i^u(n) = \sum_{l=0}^{n} y_i^u(l) \tag{2.5}$$

Similarly, the cumulative number of tokens produced by invocations $A_{i,0}$ to $A_{i,n}$ of actor A_i to its output edge e_u is denoted by:

$$X_i^u(n) = \sum_{l=0}^{n} x_i^u(l) \tag{2.6}$$

Similar to the SDF case, we are interested in finding an indefinite, periodic schedule of a CSDF graph G. As shown in [BELP96], a repetition vector $\vec{q}$ of G is given by:

$$\vec{q} = \mathbf{\Theta} \cdot \vec{r}, \qquad \text{with} \qquad \Theta_{ik} = \begin{cases} \Omega_i & \text{if } i = k \\ 0 & \text{otherwise} \end{cases} \tag{2.7}$$

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

$$\mathbf{\Gamma} \cdot \vec{r} = \vec{0} \tag{2.8}$$

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

$$\Gamma_{uj} = \begin{cases} X_i^u(\Omega_j - 1) & \text{if actor } A_i \text{ produces on edge } e_u \\ -Y_i^u(\Omega_j - 1) & \text{if actor } A_i \text{ consumes from edge } e_u \\ 0 & \text{otherwise.} \end{cases} \tag{2.9}$$

Example 2.1.1. An example of a CSDF graph is shown in Figure 2.2. The graph indicates the production/consumption sequences of the actors over the edges of the graph. For instance, actor A_2 has consumption sequence $[1, 2]$ over e_1 and production sequence $[0, 3]$ over e_2. From Equations (2.7)-(2.9), we derive the repetition vector $\vec{q}$ as shown below.

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

Based on this repetition vector $\vec{q}$, we can derive a static non-preemptive schedule for the CSDF graph in Figure 2.2 that can be repeated forever using bounded buffers. The following schedule possess this property.

$$\text{Schedule 1:} \quad A_1A_1A_1\, A_2A_2\, A_3A_3A_3 = 3A_1\, 2A_2\, 3A_3 \tag{2.10}$$

Note that alternative schedule exists.

For a consistent and live (H)SDF or CSDF graph $G = (A, E)$, given the repetition vector $\vec{q}$ of the graph, we can define the concept of actor iteration and graph iteration as shown below.

Definition 2.1.3. *(Actor iteration).* An **actor iteration** is the invocation of an actor A_i for q_i times.

Definition 2.1.4. *(Graph iteration).* A **graph iteration** is the invocation of every actor A_i for q_i times.

Stateless Actors

In this thesis, we will use the concept of *stateless* and *stateful* actors. This concept is common to both the (H)SDF and CSDF MoCs. Formally, any (C | H)SDF actor is stateless because the relation between tokens consumed and produced during an invocation is defined by a function, as mentioned earlier. Needless to say, a function does not have a state. However, sometimes it is necessary to model actors for which the result of the current invocation is dependent from the data produced in the previous invocations. In the (C | H)SDF MoC, these dependencies from previous invocations are modeled using self-edges. On these self-edges, an invocation can write data tokens that represent the actor "state" and that can be read by successive invocations. A formal definition of stateless actors is given below.

Definition 2.1.5. *(Stateless actor).* A (C | H)SDF actor is called *stateless* if it has no self-edges used to model its state.

Consequently, stateful actors are defined as follows.

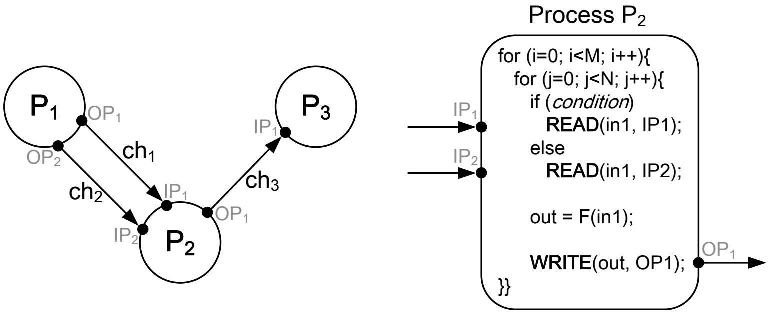
Definition 2.1.6. *(Stateful actor).* A (C | H)SDF actor is called *stateful* if it has self-edges used to model its state.

2.1.3 Polyhedral Process Network (PPN)

The Polyhedral Process Network (PPN) [VNS07] MoC is used in this thesis mainly in the context of best-effort systems (Chapters 3 and 4). The PPN MoC is a special case of the Kahn Process Network (KPN) MoC proposed in [Kah74]. A PPN is a directed multigraph G defined as a tuple $G = (\mathcal{P}, \mathcal{C})$, where:

- $\mathcal{P} = \{P_1, P_2, \cdots, P_{|\mathcal{P}|}\}$ is a set of processes;
- $\mathcal{C} = \{\mathrm{ch}_1, \mathrm{ch}_2, \cdots, \mathrm{ch}_{|\mathcal{C}|}\}$ is a set of FIFO channels.

Processes in $\mathcal{P}$ represent tasks of an application and they communicate among each other using FIFO channels in $\mathcal{C}$. An example of PPN is depicted in Figure 2.3(a). Each PPN process has a set of *input ports* (for instance, process P_2 has input ports IP_1 and IP_2) and a set of *output ports* (P_2 has only one output port, OP_1), through which the process reads and writes data. Channels connected to the input and output ports of a process P are called *input* and *output channels*, and denoted by IC_P and OC_P, respectively.



```
for (i=0; i<M; i++){
  for (j=0; j<N; j++){
    if (condition)
      READ(in1, IP1);
    else
      READ(in1, IP2);

    out = F(in1);

    WRITE(out, OP1);
}}
```

(a) (b)

Figure 2.3: *In sub-figure (a), an example of a PPN topology composed of processes P_1, P_2, P_3 and FIFO channels ch_1, ch_2, ch_3. Processes read/write data tokens from/to channels using input/output ports, which are denoted by dots. Sub-figure (b) shows the internal structure of process P_2 of sub-figure (a). As in all PPN processes, the structure of P_2 is based on nested for-loops.*

Similar to KPN processes, PPN processes are synchronized through the FIFO channels, that is, processes that attempt to read from an empty FIFO will block (*blocking read*). However, contrary to KPNs, in PPNs FIFO buffers have bounded size, therefore processes are also forced to block when trying to write to a full FIFO (*blocking write*).

Note that in PPNs control and synchronization are completely distributed, which allows to change the mapping of processes to PEs at run-time with minor effort. We leverage this advantage in our proposed techniques aimed at achieving system adaptivity in NoC based MPSoCs, see Chapters 3 and 4.

Another restriction with respect to the KPN MoC is that in PPNs processes have a precise structure. As shown in Figure 2.3(b) for process P_2, the execution of a PPN process is defined using nested *for*-loops. Each execution of a PPN process corresponds to a certain value of the *for*-loop iterators. The value of these iterators can be represented as a vector $\vec{I}$, called *iteration vector*. For P_2, the iteration vector is $\vec{I} = [i, j]$.

Each PPN process executes as follows. First, the process reads data from (a subset of) its input ports. The subset of input ports from which data is read depends on the value of the iteration vector. For instance, process P_2 reads data from IP_1 if the *condition*[2] in Figure 2.3(b) is satisfied, otherwise data is read from IP_2. Then, the input data is processed by a function (in P_2, this is represented by the line $out = F(in1)$). This function represents the computational behavior of the process. Finally, the process writes the produced data to (a subset of) its output ports. The subset of output ports to which data is written depends, again, on the value of the iteration vector.

---

[2] Conditions used to determine whether an input/output port has to be read/written can contain any affine relation of the loop iterators, static parameters, and constants.

Note that, similar to the actor-based MoCs presented in Sections 2.1.1 and 2.1.3, the relation between input and output data of a process is defined by a function, which is by definition stateless. In order to model processes for which the result of the current iteration is dependent from the data produced in the previous iterations, one can use self-channels. On these self-channels, an iteration can write data tokens that represent the process "state" and that can be read by successive iterations. This state is therefore stored outside the process itself. However, note that the set of input/output ports which is read/written by the PPN process is derived based on its iteration vector $\vec{I}$. Therefore, the iteration vector is in fact the only state of the PPN process.

Automatic derivation from SANLPs

The restrictions imposed by the PPN MoC compared to the KPN MoC lead to the following important property: any sequential application specified as a Static Affine Nested Loop Program (SANLP) can be automatically converted to an equivalent parallel PPN specification [VNS07]. An SANLP can be defined as follows (from [Mei10]).

Definition 2.1.7. *(Static Affine Nested Loop Program (SANLP)).* An SANLP is a program where each program statement is enclosed by one or more loops and if-statement, and where:

- loops have a constant step size;
- loops have bounds that are affine expressions of the enclosing loop iterators, static program parameters, and constants;
- if-statements have affine conditions in terms of the loop iterators, static program parameters, and constants;
- index expression of array references are affine expressions of the enclosing loop iterators, static program parameters, and constants;
- data flow between statements in the loop is explicit, which prohibits that two statements that contain function calls communicate through shared variables invisible at the SANLP level.

In particular, in this thesis we use the `pn` compiler [VNS07] to automatically convert static affine nested loop programs (SANLPs) to parallel PPN specifications and to determine the buffer sizes that guarantee deadlock-free execution. Although the `pn` compiler imposes some restrictions on the specification of the input application, a large set of streaming applications can be effectively specified as SANLPs. In addition to the case studies considered in Chapters 3 and 4, SANLPs can model applications from various domains, such as image/video processing (JPEG2000, H.264), sound processing (FM radio, MP3), and scientific computation (QR decomposition, stencil, finite-difference time-domain). Moreover, a recent work [TA10] has shown that most of the streaming applications can be specified using the Synchronous Data Flow (SDF) model [LM87b]. The PPN model is more expressive than SDF, thus it can as well be used effectively to model most streaming applications.

2.2 Real-time Scheduling Theory

In this section, we introduce in a formal way the real-time periodic task model and important schedulability results for multiprocessor systems. This task model and analysis techniques are instrumental to the approaches we present in Chapters 5 and 6. Finally, we describe the notation and theoretical results of two semi-partitioned scheduling algorithms which are leveraged in our work: EDF-fm [ABD08] and EDF-os [AEDC14].

2.2.1 Real-time periodic and sporadic task models

Under the real-time **periodic** task model, a **task** is defined by a 4-tuple $\tau_i = (C_i, T_i, S_i, D_i)$, where C_i is the worst-case execution time (WCET) of the task, T_i is the task period, S_i is the start time of the task, and D_i is the deadline of the task. A periodic task τ_i starts at time S_i and is recurrent, with a constant inter-arrival time T_i. That is, a periodic task τ_i is invoked at time instants $r_{i,k} = S_i + kT_i$, for all $k \in \mathbb{N}_0$. Each invocation of τ_i is called a **job**. The kth job of τ_i is denoted by $\tau_{i,k}$. Job $\tau_{i,k}$ must complete its execution before time $d_{i,k} = r_{i,k} + D_i$. In this thesis, we assume that tasks have *implicit deadlines*, i.e., $D_i = T_i$ for each task τ_i. In this case, the absolute deadline of job $\tau_{i,k}$ is $d_{i,k} = S_i + (k+1)T_i$ is coincident with the arrival of job $\tau_{i,k+1}$. We denote the actual completion time of $\tau_{i,k}$ as $z_{i,k}$. We assume that tasks can be preempted at any time. The *demand* of a real-time periodic task is defined as follows.

Definition 2.2.1. *(Demand of a periodic real-time task).* The demand of a periodic real-time task τ_i in the interval $[t_0, t_c)$ is the total time in which jobs of τ_i are executed in $[t_0, t_c)$. This demand is denoted by $\text{dmd}(\tau_i, t_0, t_c)$.

In Section 2.3, we summarize the scheduling technique [BS11, BS12] on which our proposed approaches aimed at HRT systems are based. That scheduling technique considers an input application modeled as an acyclic (C)SDF graph with N actors. Then, this (C)SDF model of the application is converted to a set $\Gamma = \{\tau_1, \tau_2, \cdots, \tau_N\}$ of N real-time periodic tasks. In general, tasks in Γ do not have the same start time, i.e., Γ is an *asynchronous* task set. The **utilization** of task $\tau_i \in \Gamma$ is $u_i = C_i/T_i$ and the total utilization of the task set Γ is $U_\Gamma = \sum_{\tau_i \in \Gamma} u_i$.

The **sporadic** task model is a generalization of the periodic task model. Jobs released by a sporadic task must be separated in time by a minimum inter-arrival interval T_i.

2.2.2 System model

In this thesis, we consider *homogeneous* multiprocessor systems. That is, in the considered systems all the processors are identical and the speed of execution of tasks on processors is the same. In particular, we consider a system composed of a set $\Pi = \{\pi_1, \pi_2, \cdots, \pi_M\}$ of M homogeneous processors.

2.2.3 Multiprocessor Real-Time Scheduling Algorithms

In this section we describe some concepts and results from real-time scheduling analysis which are instrumental to the approaches proposed in this thesis. We focus on scheduling algorithms which handle periodic real-time task sets.

Multiprocessor scheduling algorithms try to solve two problems [DB11]:

1. The *allocation problem*, namely on which processor(s) jobs of a task should execute;
2. The *priority problem*, or when, and in what order with respect to jobs of other tasks, each job should execute.

Based on the way in which scheduling algorithms approach the **allocation problem**, they can be categorized in:

- *No migration.* Each task is allocated to only one processor, and no migration is allowed.
- *Task-level migration.* Different jobs of the same task can execute on different processors. However, each job can only be executed on one processor. These approaches are said to have **restricted migrations**.
- *Job-level migration.* A single job can migrate and be executed on different processors. However, parallel execution of a job is not allowed, i.e., the same job cannot be executed in parallel on two or more processors.

Algorithms that allow any task to migrate, either at task-level or at job-level, are termed **global**. By contrast, the algorithms that do not allow migration at any level are called **partitioned**.

Depending on how scheduling algorithms solve the **priority problem**, they can be classified in:

- *Fixed task priority.* Each task has a single fixed priority applied to all of its jobs.
- *Fixed job priority.* The jobs of a task may have different priorities, but each job has a single static priority. An example of this class is the Earliest Deadline First (EDF) [LL73] scheduling described in Section 2.2.4.
- *Dynamic job priority.* A single job may have different priorities during its execution. An example of this class is Least Laxity First (LLF) scheduling.

We proceed our discussion by introducing some useful definitions from [DB11].

Definition 2.2.2. *(Feasibility of a task set).* A task set is said to be *feasible* with respect to a given system if there exist some scheduling algorithm that can schedule all possible sequences of jobs that may be generated by the task set on that system without missing any deadline.

Definition 2.2.3. *(Optimal scheduling algorithm).* A scheduling algorithm is said to be *optimal* with respect to a system and a task model if it can schedule all of the task sets that comply with the task model and are feasible on the system.

Definition 2.2.4. *(Schedulability of a task and of a task set).* A task τ is referred to as *schedulable* according to a given scheduling algorithm $\mathcal{A}$ if its worst-case response time under $\mathcal{A}$ is less than or equal to its deadline. Similarly, a task set is referred to as *schedulable* under a given scheduling algorithm if all of its tasks are schedulable.

Real-time scheduling theory provides analytical *schedulability tests* to verify the schedulability of a task set Γ under scheduling algorithms $\mathcal{A}$. A schedulability test is termed *sufficient* if all task sets that are deemed schedulable according to the test are in fact schedulable [DB11]. Similarly, a schedulability test is termed *necessary* if all the task sets that are deemed unschedulable according to the test are in fact unschedulable. Finally, a schedulability test that is both sufficient and necessary is called *exact*.

For implicit deadline periodic task sets, an useful performance metric of both uniprocessor and multiprocessor scheduling algorithms is the worst-case utilization bound, as defined below.

Definition 2.2.5. *(Worst-case utilization bound (from [DB11])).* The worst-case utilization bound $U_{\mathcal{A}}$ for a scheduling algorithm $\mathcal{A}$ is the minimum utilization of any implicit deadline task set that is only *just* schedulable under $\mathcal{A}$.

From this definition, it follows that every implicit deadline task set Γ with total utilization $U_\Gamma \leq U_{\mathcal{A}}$ is schedulable under $\mathcal{A}$. Therefore, the condition:

$$U_\Gamma \leq U_{\mathcal{A}} \tag{2.11}$$

can be used as a sufficient (not necessary) schedulability test for task set Γ under scheduling algorithm $\mathcal{A}$.

2.2.4 Uniprocessor Schedulability Analysis

Arguably, the two most popular scheduling algorithms for uniprocessor systems are Earliest Deadline First (EDF) and Rate Monotonic (RM). These two scheduling algorithms are described below.

Earliest Deadline First (EDF)

The EDF scheduling algorithm was proposed in the seminal paper [LL73] of Liu and Layland. Under EDF a task is assigned the highest priority if the deadline of its current job is the nearest. Ties are broken arbitrarily. An exact schedulability test under EDF for implicit deadline periodic task sets is given in the following theorem.

Theorem 2.2.1. *Under EDF, an implicit deadline periodic task set Γ is schedulable on one processor if the total utilization of Γ is less than or equal to one:*

$$U_\Gamma = \sum_{\tau_i \in \Gamma} u_i \leq 1 \tag{2.12}$$

Note that EDF is *optimal* on uniprocessor systems. That is, if a task set is feasible on such a system, it is also schedulable under EDF.

Rate Monotonic (RM)

Under the Rate Monotonic (RM) scheduling algorithm, each task has a fixed priority. In particular, for any two tasks τ_i and τ_j, if the period of τ_i is shorter than the period of τ_j then the priority of τ_i is higher that that of τ_j.

Such a priority assignment is optimal in the sense that no other fixed task priority assignment rule can schedule a task set which cannot be scheduled by RM [LL73]. However, contrary to EDF, RM is in general not optimal on uniprocessors (see Definition 2.2.3) for real-time periodic task sets.

2.2.5 Multiprocessor Schedulability Analysis

As mentioned in Section 1.2.2, the scheduling problem on multiprocessors is much more complex than on uniprocessor systems. In order to find a solution to this problem, a plethora of scheduling algorithms for hard real-time multiprocessor systems have been proposed in the literature [DB11, BBB15]. Each scheduling algorithm has its advantages and its drawbacks compared to the others, and in fact there is no scheduling algorithm that outperforms the rest in all aspects.

Optimal Global Scheduling Algorithms

On a system comprised of M homogeneous processors, hard real-time scheduling algorithms that achieve a worst-case utilization bound of M exploit job-level migrations and dynamic job priority (recall the classification of scheduling algorithms given in Section 2.2.3). Examples of such algorithms include PFAIR [BCPV96] and LLREF [CRJ06]. Under these optimal global scheduling algorithms, an exact schedulability test for an implicit deadline periodic task set Γ on M processors is:

$$U_\Gamma = \sum_{\tau_i \in \Gamma} u_i \leq M \tag{2.13}$$

that is, any implicit deadline periodic task set with total utilization less than or equal to M is schedulable on M processors. Based on the above equation, we can derive the minimum number of processors M_{OPT} required by an optimal scheduling algorithm to schedule an implicit deadline periodic task set Γ:

$$M_{\text{OPT}} = \lceil U_\Gamma \rceil \tag{2.14}$$

Note that other global scheduling algorithms do not achieve optimality, for instance Global EDF (GEDF).

Partitioned Scheduling Algorithms

Unfortunately, optimal global scheduling algorithms entail high migration and preemption overheads. To avoid such overheads, designers often choose partitioned approaches, where no migration is allowed. Partitioned scheduling approaches are composed of two phases, an assignment phase and an execution phase. Under partitioned approaches, as the name suggests, in the **first phase** a *schedulable partition* of the initial task set is created. In general, a partition of a set V is defined as a grouping of its elements into non-empty subsets, in such a way that every element is included in one and only one of the subsets. We provide a definition using mathematical notation below.

Definition 2.2.6. *(Partition of a set).* Let V be a set. An x-partition of V is a set, denoted by ${}^xV$, where:

$$^xV = \{^xV_1, {}^xV_2, \cdots, {}^xV_x\},$$

such that each subset ${}^xV_i \subseteq V$, and:

$$^xV_i \neq \emptyset \quad \forall\, {}^xV_i \quad \text{and} \quad \bigcap_{i=1}^{x} {}^xV_i = \emptyset \quad \text{and} \quad \bigcup_{i=1}^{x} {}^xV_i = V.$$

As mentioned earlier, in the case of partitioned scheduling algorithms, we are interested in obtaining a *schedulable partition* of a task set, which is defined below.

Definition 2.2.7. *(Schedulable partition of a task set).* Let Γ be a set of periodic real-time tasks. A schedulable partition ${}^x\Gamma$ is a partition of Γ that complies with Definition 2.2.6 and guarantees that each subset of ${}^x\Gamma$ is schedulable on one processor under the considered local scheduling algorithm.

In particular, consider a task set Γ and an x-partition ${}^x\Gamma$ of Γ. Assume that each subset ${}^x\Gamma_j \in {}^x\Gamma$ is assigned to a separate processor and it is scheduled by a local uniprocessor scheduler $\mathcal{A}$. Then, using the schedulability test provided by Condition (2.11), we have that Γ is schedulable using $\mathcal{A}$ on each processor if:

$$\sum_{\tau_i \in {}^x\Gamma_j} u_i \leq U_{\mathcal{A}}, \quad \forall\, {}^x\Gamma_j \in {}^x\Gamma \tag{2.15}$$

where $U_{\mathcal{A}}$ is the worst-case utilization bound of $\mathcal{A}$, as defined in Definition 2.2.5. For instance, the worst-case utilization bound of EDF is $U_{\text{EDF}} = 1$ [LL73], therefore we have that Γ is schedulable using Partitioned EDF (PEDF) if:

$$\sum_{\tau_i \in {}^x\Gamma_j} u_i \leq 1, \quad \forall\, {}^x\Gamma_j \in {}^x\Gamma \tag{2.16}$$

Then, in the **second phase**, at run-time, the local (uniprocessor) scheduler $\mathcal{A}$ is used to schedule the subset of the partition which is assigned to each processor.

From the above discussion, it is clear that the first phase of a partitioned scheduling approach is in fact an instance of the classical *bin-packing* problem [Joh73]. In the bin-packing problem, items of different sizes must be packed into the least amount of bins, which have a certain capacity. In partitioned scheduling, in an analog way, tasks with different utilizations must be partitioned into the least amount of processors. The "capacity" of each processor is determined by the worst-case utilization bound

$U_{\mathcal{A}}$ of the local scheduling algorithm. The equivalence of partitioning schemes and the bin-packing problem leads to the following two limitations.

Limitation 1. An optimal solution to the bin-packing problem is one that minimizes the number of bins required to pack the items. Analogously, an optimal partitioning of the set of tasks is one that requires the least amount of processors to assign all tasks, while guaranteeing schedulability on all processors. In both cases, finding an optimal solution is NP-hard [GJ79]. In order to tackle the NP-hardness of the problem, several heuristics have been proposed [Joh74] to find approximate solutions. We provide an overview of the most commonly used heuristics in Section 2.2.6. These heuristics are rather simple and fast, but they do not guarantee the optimality of the provided solution.

Limitation 2. Consider a system composed of M processors. Even if we determine the optimal partitioning of tasks to processors, no partitioned scheduling algorithm can guarantee the worst-case utilization bound of M (recall Equation (2.13)) provided by optimal global algorithms. In general, the worst-case utilization bound of a partitioned scheduling algorithm on M processors can reach at most $(M+1)/2$ [ABD08]. This phenomenon is termed *utilization loss* and implies that partitioned algorithms cannot, in general, exploit the available processing resources in an optimal way. In fact, a partitioned approach may require twice as many processors to schedule certain task sets compared to an optimal global scheduler.

In the rest of this dissertation, we refer to the two above limitations as *bin-packing issues*.

2.2.6 Partitioning Heuristics

Consider a set Γ of N tasks (items) and a set Π of M homogeneous processors (bins). Each processor uses a local scheduler $\mathcal{A}$ with worst-case utilization bound $U_{\mathcal{A}}$, and each task τ_i has utilization u_i. As mentioned in **Limitation 1** of Section 2.2.5, an optimal partitioning of the task set Γ is a partitioning that uses the least amount of processors and satisfies Condition (2.15). Deriving such optimal partitioning, which is an instance of the bin-packing problem, is NP-hard. Given the complexity of this problem, several heuristics have been proposed to solve it. In the following, we summarize some of the most common heuristics used to solve the partitioning problem. These heuristics assign each task $\tau_i \in \Gamma$ to a certain processor $\pi_k \in \Pi$ by considering one task at a time, following a certain sequence. For First-Fit, Best-Fit and Worst-Fit, in particular, the current task to be assigned is determined by following the order of tasks appearance in Γ, e.g., τ_1 is assigned first and τ_N last.

All the partitioned heuristics described in what follows utilize the concept of processor utilization, defined below.

Definition 2.2.8. *(Utilization of a processor).* Let Γ_k denote the set of tasks currently assigned to processor π_k. Then, the utilization σ_k of processor π_k is equal to the sum of the utilizations of the tasks assigned to π_k, i.e.:

$$\sigma_k = \sum_{\tau_i \in \Gamma_k} u_i \tag{2.17}$$

Note that, in the beginning of all partitioned heuristics listed below, $\Gamma_k = \emptyset$ and $\sigma_k = 0$ for each π_k.

- **First-Fit (FF).** A task τ_i is assigned to the lowest-indexed processor π_k that can contain it. That is, the index k of π_k is determined by:

  $$k = \min\{j \mid u_i + \sigma_j \leq U_{\mathcal{A}}\}$$

  If the condition enclosed by the braces is not satisfied by any processor, a new processor is instantiated and τ_i is assigned to it.
- **Best-Fit (BF).** A task τ_i is assigned to a processor π_k such that π_k will have the *minimal* residual utilization ($U_{\mathcal{A}} - \sigma_k$) after the assignment. That is, the index k of π_k is determined by:

  $$k = \min\{j \mid \left(u_i + \sigma_j\right) \text{ is closest to, without exceeding, } U_{\mathcal{A}}\}$$

  If the condition enclosed by the braces is not satisfied by any processor, a new processor is instantiated and τ_i is assigned to it.
- **Worst-Fit (WF).** A task τ_i is assigned to a processor π_k such that π_k will have the *maximal* residual utilization ($U_{\mathcal{A}} - \sigma_k$) after the assignment.

  $$k = \min\{j \mid \left(u_i + \sigma_j\right) \text{ is minimal and does not exceed } U_{\mathcal{A}}\}$$

  If the condition enclosed by the braces is not satisfied by any processor, a new processor is instantiated and τ_i is assigned to it.

Recall that for EDF the worst-case utilization bound $U_{\mathcal{A}}$ is $U_{\mathsf{EDF}} = 1$ (see Equation (2.16)).

As shown in [Joh73], these heuristics achieve better performance if they are preceded by a sorting of the input task set. Usually, the input task set is sorted by decreasing utilization. The approaches composed of a first phase, which sorts the input task set by decreasing utilization, and a second phase, which applies one of the aforementioned heuristics (FF, BF, WF), are termed **First-Fit Decreasing (FFD)**, **Best-Fit Decreasing (BFD)**, and **Worst-Fit Decreasing (WFD)**, respectively.

The performance of a partitioning heuristic can be measured by its *approximation ratio*. Let $OPT(\Gamma)$ be the number of processors needed by an optimal partitioning scheme to assign a certain task set Γ. Consider a certain partitioning heuristic H, which requires $H(\Gamma)$ processors to assign the same task set Γ. Then, the approximation ratio of H, denoted by $\mathcal{R}_H$, ensures that for *any* task set Γ:

$$H(\Gamma) \leq \mathcal{R}_H \cdot OPT(\Gamma) \tag{2.18}$$

The approximation ratios for FF, BF, and FFD are 17/10, 17/10, and 11/9, respectively [CGJ96, GJ79, Yue91].

2.2.7 EDF-fm Semi-partitioned Algorithm

As summarized in Section 2.2.5, global scheduling algorithms can be optimal for multiprocessor systems leading to a full exploitation of the available processors.

However, they incur high migration and preemption overheads, which may limit their applicability. Moreover, they incur significant memory overhead in distributed memory systems, as explained in Section 1.3.2. By contrast, partitioned approaches as PEDF show low preemption overheads and neither migration nor memory overheads. However, they are affected by bin-packing issues and in general may not exploit the available processing resources in an optimal way.

In Chapters 5 and 6 of this thesis we consider semi-partitioned scheduling algorithms, which represent a middle ground between global and partitioned algorithms. As mentioned in Chapter 1, under semi-partitioned algorithms most of the tasks are statically allocated to processors and only a subset of the tasks is allowed to migrate among different processors. Migrating tasks follow a migration pattern derived at design-time. Semi-partitioned algorithms aim at ameliorating the bin-packing issues of partitioned scheduling without incurring the excessive overheads of global scheduling. In particular, in Chapter 5 we exploit the EDF-fm scheduling algorithm [ABD08], which is described in the rest of this section. We recall that the name EDF-fm comes from the fact that the algorithm is based on EDF and allows tasks to be either *fixed* or *migrating*. By contrast, in Chapter 6 we propose a novel scheduling algorithm, EDF-ssl which is based on some concepts and properties of the EDF-os scheduling algorithm [AEDC14] summarized in Section 2.2.8.

As mentioned in Section 1.4.2, EDF-fm can have great benefits for distributed memory MPSoCs. However, it provides only soft real-time guarantees to the scheduled tasks. Since many definitions of soft real-time behavior exist, we provide below the definition of a SRT algorithm adopted in this thesis.

Definition 2.2.9. *(Soft Real-Time (SRT) scheduling algorithm).* A scheduling algorithm is said to be SRT when it allows tasks to miss their deadlines by a bounded value called *tardiness*.

Note that EDF-fm falls into this definition of SRT algorithm. In particular, under EDF-fm we can compute a bound of the tardiness of each task. A definition of tardiness bound is given below.

Definition 2.2.10. *(Tardiness bound).* A task τ_i is said to have a *tardiness bound* Δ_i if each job $\tau_{i,k}$ of τ_i does not miss its deadline $d_{i,k}$ by more than Δ_i. That is, denoting the completion time of job $\tau_{i,k}$ by $z_{i,k}$:

$$z_{i,k} \leq (d_{i,k} + \Delta_i), \forall k \in \mathbb{N}_0$$

We describe now the EDF-fm scheduling algorithm, as presented in [ABD08], in greater detail. In EDF-fm, tasks can be either fixed or migrating. Migrating tasks migrate between exactly two processors, with the restriction that migration can only happen at job boundaries. The EDF-fm approach consists of two phases: the *assignment phase* and the *execution phase*, which are summarized in what follows.

Assignment phase

Consider the following definitions:

Definition 2.2.11. *(Task share).* A task τ_i is said to have a share $s_{i,k}$ on π_k when a part $s_{i,k}$ of its utilization u_i is assigned to π_k.

In turn, the *task fraction* of task τ_i on processor π_k is defined as follows.

Definition 2.2.12. *(Task fraction).* Given $s_{i,k}$, π_k executes a fraction $\varphi_{i,k} = \frac{s_{i,k}}{u_i}$ of τ_i's total execution requirement.

In the assignment phase each task is assigned to either one processor (fixed task) or two processors (migrating task). In particular, the assignment phase assigns tasks in sequence to processors. Since EDF-fm uses EDF as local scheduling algorithm, the capacity of each processor π_k (the maximum utilization that can be assigned to it) is 1 (see Equation (2.12) on page 33), therefore the condition:

$$\sigma_k \leq 1, \; \forall \, \pi_k \in \Pi \tag{2.19}$$

must be satisfied.

In particular, in the assignment phase tasks are assigned to a processor π_k until its capacity is exhausted. Recall that σ_k denotes the total utilization assigned to processor π_k (see Definition 2.2.8). In the case of EDF-fm, σ_k is equal to the sum of *shares* assigned to π_k:

$$\sigma_k \triangleq \sum_{\tau_i \in \Gamma_k} s_{i,k} \tag{2.20}$$

where Γ_k is the set of tasks with non-zero shares on π_k.

If a task τ_i cannot entirely fit on processor π_k, then a share $s_{i,k} = 1 - \sigma_k$ of its utilization is assigned to π_k. This makes sure that, after this assignment, $\sigma_k = 1$, i.e., π_k is fully utilized. The remaining utilization $s_{i,k+1} = (u_i - s_{i,k})$ of τ_i is assigned to the next processor, π_{k+1}. The assignment phase of EDF-fm ensures that at most two migrating tasks are assigned to any processor (see an example in Figure 2.4).

Moreover, on a processor with two migrating tasks (τ_i and τ_j), EDF-fm requires that the sum of the migrating tasks' utilization (denoted by σ_k^{mig}) does not exceed one:

$$\sigma_k^{\text{mig}} = u_i + u_j \leq 1, \tag{2.21}$$

This condition is automatically satisfied if the maximum utilization of any task is limited to 1/2, given the fact that at most two migrating tasks can be assigned to a single processor. However, tasks that exceed this utilization limit can still be scheduled by EDF-fm, provided that Condition (2.21) is respected on all the processors. Note that if no limit on maximum task utilizations is set, EDF-fm is not optimal, because it cannot fully exploit the available processors for all possible input task sets.

Example 2.2.1. Given the task set $\{\tau_1 = (C_1 = 3, T_1 = 10), \tau_2 = (2,5), \tau_3 = (2,5), \tau_4 = (1,2), \tau_5 = (1,2), \tau_6 = (2,5), \tau_7 = (1,2)\}$, the EDF-fm algorithm derives the task assignment shown in Fig. 2.4. For instance, task τ_3 cannot entirely fit onto π_1 in Fig. 2.4, thus its utilization is split between π_1 and π_2 with shares $s_{3,1} = 3/10$ and $s_{3,2} = 1/10$, respectively.

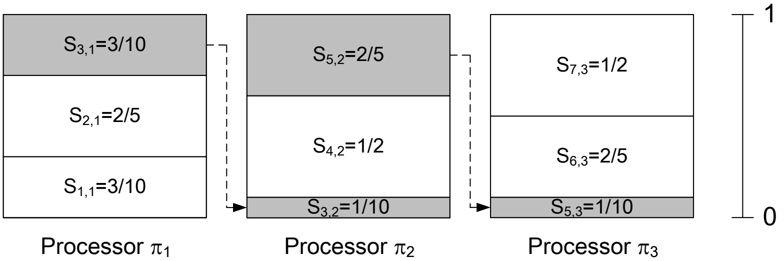


Figure 2.4: *EDF-fm assignment of the task set considered in Example 2.2.1. Tasks τ_1, τ_2, τ_4, τ_6, and τ_7 are fixed, i.e., their whole utilization is assigned to a single processor. For instance, task τ_1 has utilization $u_1 = 3/10$ and the share $s_{1,1}$ of τ_1 on π_1 is equal to its whole utilization, that is, $s_{1,1} = u_1$. By contrast, tasks τ_3 and τ_5 are migrating tasks. Their shares on processors are highlighted with a shaded area. For instance, τ_3 cannot entirely fit onto π_1, thus its utilization is split between π_1 and π_2 with shares $s_{3,1} = 3/10$ and $s_{3,2} = 1/10$, respectively.*

Execution phase

The execution phase employs a simple online scheduling algorithm that is derived from EDF and ensures bounded tardiness with a minimal overhead compared to a canonical EDF scheduler. Let τ_i be a migrating task that migrates between processor π_k and π_{k+1}. Then, jobs belonging to a task τ_i are assigned at run-time such that in the long run the fraction of τ_i's workload executed on π_k (π_{k+1}) is close to $\varphi_{i,k}$ ($\varphi_{i,k+1}$). This result is achieved by leveraging results from PFAIR scheduling [BCPV96]. We recall that EDF-fm allows only *restricted migrations*. As explained in Section 2.2.3, this means that different jobs of the same task can execute on different processors. However, each job can only be executed on one processor.

For instance, according to the share assignment depicted in Figure 2.4, task τ_3 releases its jobs on processors π_1 and π_2 according to the pattern shown in Figure 2.5. Task τ_3 releases a job every period T_3, either to π_1 or to π_2. On average 1 out of 4 jobs of τ_3 are assigned to π_2 and the remaining 3 jobs are assigned to π_1. In the long run (the release pattern continues indefinitely), the number of jobs released on π_1 are three times the number of jobs released on π_2. This is due to the fact that the share $s_{3,1}$ of τ_3 assigned to π_1 is three times larger than the share $s_{3,2}$ of τ_3 assigned to π_2.

Jobs released on a processor are prioritized among each other using a local EDF scheduler. The job release pattern of migrating tasks under EDF-fm, mentioned above, prevents the overloading on processors in the long run. However, it creates *temporary* overloading on processors, which in turn leads to tardiness. In particular, when two migrating tasks, τ_i and τ_j, are assigned to π_k, the tardiness bound under EDF-fm for a fixed task τ_u assigned to the same processor is given by:

$$\Delta(\tau_u) = \frac{C_i \cdot (\varphi_{i,k} + 1) + C_j \cdot (\varphi_{j,k} + 1) - T_u \cdot (1 - \sigma_k)}{1 - s_{i,k} - s_{j,k}} \tag{2.22}$$

where C_i and C_j are the worst-case execution times of τ_i and τ_j (as defined in Section 2.2.1), and T_u is the period of task τ_u. Finally, σ_k is the total utilization assigned

Figure 2.5: *Release pattern of jobs of task τ_3 between processors π_1 and π_2, according to the share assignment of τ_3 in Figure 2.4.*

to π_k, i.e., the sum of fixed tasks' utilization and migrating tasks' shares allocated to π_k (see Equation (2.20)). Note that in Equation (2.22) the tardiness bound of EDF-fm is denoted as $\Delta(\tau_u)$, whereas in Definition 2.2.10 we denote the tardiness bound of a task τ_u as Δ_u. Throughout this thesis, we will use the latter notation if the context makes clear that Δ_u is the tardiness bound of task τ_u.

In contrast with fixed tasks, in EDF-fm migrating tasks do not miss any deadline, therefore their tardiness bound is zero.

2.2.8 EDF-os Semi-partitioned Algorithm

In order to tackle the sub-optimality of the EDF-fm scheduling algorithm, Anderson et al. in [AEDC14] propose EDF-os (EDF-based optimal semi-partitioned scheduling). In what follows, we summarize the features of EDF-os which are leveraged in our EDF-ssl scheduling algorithm presented in Chapter 6.

Similar to EDF-fm, EDF-os is also a SRT scheduling algorithm (see Definition 2.2.9). These two algorithms share some definitions and concepts, but EDF-os introduces modifications to both the assignment and execution phases of EDF-fm to achieve optimality. As in EDF-fm, under EDF-os tasks can be either *fixed* or *migrating*, with migrations only allowed at job boundaries. However, in EDF-os migrating tasks are allowed to migrate among any number of processors, not only between two processors as in EDF-fm. Each task τ_i is assigned a (potentially zero) *share* $s_{i,k}$ of the available utilization of a processor π_k, following Definition 2.2.11.

If task τ_i is migrating, it has non-zero shares on several processors. If τ_i is fixed, it has non-zero shares on a single processor. The **assignment phase** in EDF-os ensures that the cumulative sum of the shares of a task over all the processors equals the task utilization, that is:

$$u_i = \sum_{k=1}^{M} s_{i,k}$$

where M is the total number of processors in the system. Similar to EDF-fm, the total utilization assigned to processor π_k is denoted by σ_k and derived using Equation (2.20).

Also under EDF-os, the total utilization assigned to a processor must always be equal to or lower than the available processor utilization (which is 1). That is, for each processor π_k:

$$\sigma_k \leq 1 \tag{2.23}$$

Condition (2.23) above is ensured by the assignment phase of EDF-os to avoid the over-utilization of any processor in the long run. In fact, Condition (2.23) is identical to Condition (2.12) used in EDF-fm.

In the **execution phase**, EDF-os enforces that, in the long run, the fraction of workload generated by task τ_i on π_k is equal to the task fraction $\varphi_{i,k}$, given by Definition 2.2.12. Similar to EDF-fm, this long-run workload distribution according to task fractions is obtained by leveraging results from Pfair scheduling [BCPV96]. In particular, out of the *first* ν consecutive jobs released by τ_i, EDF-os ensures that the number of jobs released on processor π_k is between $\lfloor \varphi_{i,k} \cdot \nu \rfloor$ and $\lceil \varphi_{i,k} \cdot \nu \rceil$ (Property 1 in [AEDC14]). In turn, out of *any* c consecutive jobs of a migrating task τ_i, the number of jobs released on π_k (indicated as $c_{i,k}$) is bounded by the following expression:

$$c_{i,k} \leq \varphi_{i,k} \cdot c + 2 \tag{2.24}$$

The above expression is given by Property 6 in [AEDC14]. For a more detailed explanation of assignment rules for jobs of migrating tasks, the reader is referred to [ABD08] and [AEDC14].

Tardiness bounds for both fixed and migrating tasks under EDF-os are derived in [AEDC14]. We do not report these bounds because they are not relevant for the contributions of this thesis.

2.3 HRT Scheduling of Acyclic CSDF Graphs [BS11, BS12]

As mentioned in Section 1.2.2 (on page 15, under Class III of scheduling algorithms), several approaches that bridge the gap between dataflow MoCs and real-time task models have been proposed in recent years. In this thesis, in particular, among these approaches we consider the scheduling technique proposed in [BS11, BS12].

Bamakhrama and Stefanov in [BS11, BS12] consider applications specified as acyclic CSDF graphs and show that the set of N actors $A = \{A_1, A_2, \cdots, A_N\}$ of an input CSDF graph G can be converted to a set of N real-time periodic tasks $\Gamma_G = \{\tau_1, \tau_2, \cdots, \tau_N\}$. This conversion allows a designer to apply algorithms from hard real-time scheduling theory to derive in a fast and analytical way the minimum number of processors that guarantee the required performance of an application and the partitioning of tasks to processors.

In particular, for each actor $A_i \in A$ of the input CSDF graph, the analysis in [BS11, BS12] derives the parameters of the corresponding real-time periodic task $\tau_i = (C_i, T_i, S_i)$, where C_i is the WCET of the task, T_i is the task period, S_i is the start time of the task (as described in Section 2.2.1). In the rest of this section, we describe how these parameters (C_i, T_i, S_i) are derived. Then, we show how the size of buffers which implement inter-task communication over edges can be derived. Finally, we summarize the correspondence between the dataflow notation for the input CSDF graph G and the real-time theory notation for the derived periodic task set Γ_G.

WCET of Actors (C_i)

The analysis in [BS11,BS12] begins with the computation of the WCET C_i of each CSDF actor A_i. The value of C_i is derived as follows. First, the WCET $C_{i,k}$ of each phase k of actor A_i is computed. This WCET includes both the worst-case communication and computation time required by phase k of A_i, and is calculated using the following equation:

$$C_{i,k} = C^R \cdot \sum_{e_l \in \mathrm{inp}(A_i)} y_i^l(k) + C^W \cdot \sum_{e_r \in \mathrm{out}(A_i)} x_i^r(k) + C_i^C(k) \tag{2.25}$$

In Equation (2.25), C^R (C^W) represents the worst case time needed to read (write) a single token from (to) an input (output) channel in the considered hardware platform; $y_i^l(k)$ ($x_i^r(k)$) is the number of tokens read (written) by actor A_i from (to) edge e_l (e_r) by phase k of A_i; $\mathrm{inp}(A_i)$ ($\mathrm{out}(A_i)$) is the set of input (output) edges of A_i; and $C_i^C(k)$ is the worst-case computation time of phase k of actor A_i. Note that $C_i^C(k)$ includes also the worst-case overhead incurred by the underlying scheduler (e.g., EDF), following the analysis of [Dev06].

The WCET C_i of actor A_i is derived by finding the maximum value among the WCET $C_{i,k}$ of each phase k of A_i, that is:

$$C_i = \max_{k=1}^{\Omega_i} (C_{i,k}) \tag{2.26}$$

where $C_{i,k}$ is obtained using Equation (2.25).

Given the WCET C_i of each actor A_i in G, we can represent the WCETs of all actors in G using the **WCET vector** $\vec{C}$, where each component C_i of $\vec{C}$ is the WCET of actor A_i.

Minimum Period of Actors (T_i)

Based on the properties of the graph and on the WCET of actors given by Equation (2.26), the minimum period $T_i \in \mathbb{N}$ of actor A_i can be calculated using the following expression:

$$T_i = \frac{Q}{q_i} \left\lceil \frac{\eta}{Q} \right\rceil \tag{2.27}$$

where q_i is the number of repetitions of actor A_i per graph iteration, $\eta = \max_{A_i \in A}\{C_i\, q_i\}$ (recall that C_i is the WCET of A_i), and $Q = \mathrm{lcm}\{q_1, q_2, \cdots, q_N\}$.

The period T_i of actor A_i, obtained using Equation 2.27, is minimum. However, in some cases a designer may want longer periods of actors. These longer periods can be derived by multiplying the minimum period of each actor by a positive integer factor constant among all actors.

Given the period T_i of each actor A_i in G, we can represent the periods of all actors in G using the **period vector** $\vec{T}$, where each component T_i of $\vec{T}$ is the period of actor A_i.

Example 2.3.1. Consider again the CSDF graph shown in Figure 2.2 on page 27. We have already derived in Section 2.1.2 the repetition vector of the graph $\vec{q} = [3, 2, 3]$. Assume that its WCET vector is $\vec{C} = [C_1, C_2, C_3] = [1, 2, 2]$. Then, it follows that $\eta = 6$ and $\vec{T} = [T_1, T_2, T_3] = [2, 3, 2]$.

In general, the derived period vector $\vec{T}$ satisfies the condition:

$$q_1 T_1 = q_2 T_2 = \cdots = q_N T_N = H \tag{2.28}$$

where H is referred to as *iteration period*, and represents the duration needed by the graph to complete one iteration (recall Definition 2.1.4 on page 28).

Earliest Start Times of Actors (S_i)

The earliest start time $S_j \in \mathbb{N}_0$ of an actor A_j is derived using the following expression:

$$S_j = \begin{cases} 0 & \text{if } \mathrm{prec}(A_j) = \emptyset \\ \max_{A_i \in \mathrm{prec}(A_j)} \left(S_{i \to j}\right) & \text{if } \mathrm{prec}(A_j) \neq \emptyset \end{cases} \tag{2.29}$$

where:

$$S_{i \to j} = \min_{t \in [0, S_i + H]} \left\{ t : \mathrm{prd}^{\mathrm{S}}_{[S_i, \max(S_i, t) + k)}(A_i, e_u) \geq \mathrm{cns}^{\mathrm{S}}_{[t, \max(S_i, t) + k]}(A_j, e_u),\ \forall k = 0, 1, \cdots, H \right\} \tag{2.30}$$

where:

- H is the iteration period as defined by Equation (2.28);
- S_i is the start time of a predecessor actor A_i;

and the two functions ($\mathrm{prd}^{\mathrm{S}}$, $\mathrm{cns}^{\mathrm{S}}$) used in Expression 2.30 are defined as follows.

Definition 2.3.1. *(Cumulative production function for start times calculation).* The cumulative production function used to derive start times is denoted by $\mathrm{prd}^{\mathrm{S}}{}_{[t_s, t_f)}(A_i, e_u)$ and represents the total number of tokens produced by actor S_i to edge e_u during the time interval $[t_s, t_f)$.

Definition 2.3.2. *(Cumulative consumption function for start times calculation).* The cumulative consumption function used to derive start times is denoted by $\mathrm{cns}^{\mathrm{S}}{}_{[t_s, t_f]}(A_j, e_u)$ and represents the total number of tokens consumed by actor A_j from edge e_u during the time interval $[t_s, t_f]$.

Note that, for the purpose of computing the start times of actor A_j, the cumulative production function $\mathrm{prd}^{\mathrm{S}}(A_i, e_u)$ assumes that token production happens as **late** as

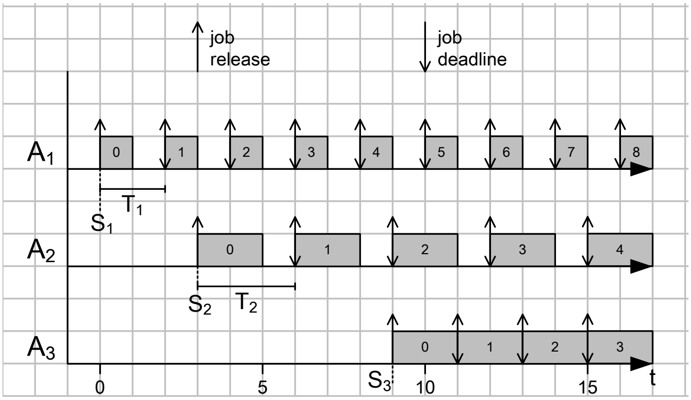


Figure 2.6: *Hard real-time scheduling of the CSDF graph in Figure 2.2 on page 27 derived using the methodology of [BS11, BS12]. For instance, the derived period of A_1 is $T_1 = 2$ and the start time of A_1 is $S_1 = 0$. This means that the first invocation of A_1 is released at time 0, and the successive invocations are released periodically, every 2 time units. The schedule continues indefinitely, only its initial part is shown.*

possible, i.e., at the deadline of each invocation of predecessor actor A_i. Conversely, the cumulative consumption function $\text{cns}^S(A_j, e_u)$ assumes that token consumption happens as **early** as possible, i.e., at the release of each invocation of actor A_j. These assumptions, which make the calculation of actor start times safe, are emphasized by the superscript S in prd^S and cns^S.

Given the start time S_i of each actor A_i in G, we can represent the start times of all actors in G using the **start time vector** $\vec{S}$, where each component S_i of $\vec{S}$ is the start time of actor A_i.

Example 2.3.2. In Example 2.3.1, assuming a WCET vector $\vec{C} = [1, 2, 2]$ we derived the period vector $\vec{T} = [2, 3, 2]$ of the graph G shown in Figure 2.2 on page 27. Then, based on Expression (2.29) we derive the earliest start time vector $\vec{S} = [0, 3, 9]$. Therefore, the real-time periodic task set corresponding to G is completely defined as:

$$\Gamma_G = \{\tau_1 = (C_1 = 1, T_1 = 2, S_1 = 0), \tau_2 = (2, 3, 3), \tau_3 = (2, 2, 9)\}$$

Given the complete specification of the obtained real-time periodic task set, a designer can apply algorithms from hard real-time scheduling theory to derive in a fast and analytical way the minimum number of processors that guarantee the required performance of an application and the partitioning of tasks to processors. For instance, the total utilization U_{Γ_G} is 13/6, therefore even an optimal global scheduling algorithm would require at least 3 processors to schedule Γ_G (see Equation (2.14)).

The periodic schedule of G, resulting from the derived task set Γ_G, is visualized in Figure 2.6. In the figure, notice that the first invocation of each actor A_i is released at the start time S_i, obtained using Expression (2.29). Then, successive invocations of each actor A_i are released periodically, according to the actor's period T_i.

Minimum Buffer Sizes

Given the period T_i and start time S_i of each actor $A_i \in A$, the minimum size b_u of the buffer which implements the communication over edge $e_u = (A_i, A_j)$ is given by:

$$b_u = \max_{k \in [0,1,\cdots,H]} \left\{ \underset{[S_i, \max\{S_i,S_j\}+k]}{\mathrm{prd}^{\mathrm{B}}} (A_i, e_u) - \underset{[S_j, \max(S_i,S_j)+k)}{\mathrm{cns}^{\mathrm{B}}} (A_j, e_u) \right\} \tag{2.31}$$

where:

- H is the iteration period as defined by Equation (2.28);
- S_i and S_j are the start times actors A_i and A_j, respectively;

and the two functions ($\mathrm{prd}^{\mathrm{B}}$, $\mathrm{cns}^{\mathrm{B}}$) used in Expression 2.30 are defined as follows.

Definition 2.3.3. *(Cumulative production function for buffer sizes calculation).* The cumulative production function used to derive buffer sizes is denoted by ${\mathrm{prd}^{\mathrm{B}}}_{[t_s,t_f]}(A_i, e_u)$ and represents the total number of tokens produced by actor S_i to edge e_u during the time interval $[t_s, t_f]$.

Definition 2.3.4. *(Cumulative consumption function for buffer sizes calculation).* The cumulative consumption function used to derive buffer size is denoted by ${\mathrm{cns}^{\mathrm{B}}}_{[t_s,t_f)}(A_j, e_u)$ and represents the total number of tokens consumed by actor A_j from edge e_u during the time interval $[t_s, t_f)$.

Note that, for the purpose of computing the buffer size b_u, the cumulative production function $\mathrm{prd}^{\mathrm{B}}(A_i, e_u)$ assumes that token production happens as **early** as possible, i.e., at the release of each invocation of predecessor actor A_i. Conversely, the cumulative consumption function $\mathrm{cns}^{\mathrm{B}}(A_j, e_u)$ assumes that token consumption happens as **late** as possible, i.e., at the deadline of each invocation of actor A_j. These assumptions, which make the calculation of buffer sizes safe, are emphasized by the superscript B in $\mathrm{prd}^{\mathrm{B}}$ and $\mathrm{cns}^{\mathrm{B}}$.

Correspondence Between Dataflow and Real-Time Theory Notations

The analysis in [BS11, BS12] creates a one-to-one correspondence between actor A_i of the input CSDF graph G and real-time periodic task τ_i of the derived periodic task set Γ_G. In this thesis, we will leverage either the dataflow notation for the (C)SDF graph G (see Sections 2.1.1 and 2.1.2) or the real-time theory notation for the periodic task set Γ_G (see Section 2.2.1), depending on the problem we want to address. For the sake of clarity, Table 2.2 shows the correspondence between these two notations. Recall that we denote the jth invocation of actor A_i as $A_{i,j}$ (with $j \in \mathbb{N}_0$), therefore $A_{i,0}$ represents the *first* invocation of actor A_i. Note that, in Table 2.2, the earliest start time and latest completion time of an invocation $A_{i,j}$ refer to the schedule generated by the methodology of [BS11, BS12].

Note that Chapter 5 and 6 of this thesis extend the methodology of [BS11, BS12]. Therefore, in those chapters the correspondence between dataflow and real-time theory notations shown in Table 2.2 is used extensively.

Dataflow notation for G	**Real-time notation for Γ_G**
Actor A_i	Task τ_i
Invocation $A_{i,j}$ of A_i	Job $\tau_{i,j}$ of τ_i
Earliest start time of $A_{i,0}$	Start time S_i of τ_i
Earliest start time of $A_{i,j}$	Release time $r_{i,j}$ of $\tau_{i,j}$
Latest completion time of $A_{i,j}$	Deadline $d_{i,k}$ of $\tau_{i,j}$

Table 2.2: *Correspondence between dataflow and real-time theory notations resulting from the methodology of [BS11, BS12].*

In addition, both Chapter 5 and 6 use the concept of *stateless* real-time periodic tasks. In general, a task is said to be *stateless* if it complies to the definition below.

Definition 2.3.5. *Stateless task (general).* A task is said to be *stateless* if it does not keep an internal state between two successive jobs.

Using the methodology of [BS11, BS12], we recall that each task τ_i corresponds to actor A_i of the input CSDF graph G. Therefore

Definition 2.3.6. *Stateless task (in [BS11, BS12]).* In the scheduling technique of [BS11, BS12], a task τ_i is said to be *stateless* if it corresponds to a (C)SDF actor A_i which is stateless (i.e., A_i complies to Definition 2.1.5 on page 28).

Chapter 3

PPN Communication on Networks-on-chip

Most of the work presented in this chapter has been published in [CDS11].

IN this chapter and in the following one, Chapter 4, we present techniques which are aimed at achieving system adaptivity[1] in the context of **best-effort systems**. In order to make the system adaptive, our approach provides a mechanism by which application processes can migrate among processors at run-time.

Our approach takes into account the emerging trends in the design of embedded MPSoCs that are described in Sections 1.1.1 and 1.1.2. That is, we base our technique on the following two assumptions:

1. As the methodology used to specify applications is concerned, we consider an approach based on a Model of Computation. In particular, we adopt the PPN MoC, which is presented in Section 2.1.3. In PPNs, memory, control, and synchronization are completely distributed, which allows to change the mapping of processes to PEs at run-time with minor effort.
2. Regarding the choice of communication infrastructures, we assume that PEs in our systems are interconnected by a Network-on-Chip (NoC). Some of the advantages of NoCs are described in Section 1.1.2. In the context of system adaptivity, in particular, we argue that NoCs are appropriate because NoCs are generic, i.e., the same NoC-based platform can be used to run different applications or to run the same application with a different mapping of processes. As mentioned in Section 1.1.2, we consider NoC-based platforms which comprise several processing elements, organized in *tiles*.

From the above discussion, it follows that the PPN MoC and NoC-based interconnections, among other advantages, can favor system adaptivity in embedded MPSoCs.

---

[1] Recall the explanation of our understanding of the term "system adaptivity", given in Section 1.2.1 on page 10.

However, there is a mismatch between the communication primitives allowed in NoC-based execution platforms and the semantics of the PPN MoC. Therefore, in this chapter we investigate and propose several approaches to overcome this mismatch. All of the proposed approaches are aimed at implementing PPN communication on NoCs considering system adaptivity as a driving objective. Moreover, they do not require specific hardware support from the NoC-based platform to realize inter-tile communication among PPN processes. The approaches presented in this chapter represent different possible implementations of the first component of the middleware layer that is proposed in this thesis (see Figure 1.6 on page 20) to achieve system adaptivity on NoC-based MPSoCs.

The remainder of this chapter is organized as follows. Section 3.1 continues the introduction by stating the addressed research problem. A summary of the contributions of this chapter and a list of related work is provided in Sections 3.2 and 3.3, respectively. The proposed and investigated approaches for PPN communication on NoCs are described in detail in Section 3.4. The applications used to evaluate the different approaches are explained in Section 3.5 followed by the performance results in Section 3.6. Note that in the rest of this chapter, for the sake of brevity, we will refer to an "approach for implementing PPN communication on NoCs" as a "PPN communication approach".

3.1 Problem Statement

The main problem addressed in this chapter is the implementation of an efficient approach to implement PPN communication on Network-on-Chip platforms. The first requirement that we consider is that this approach must respect the PPN communication semantics (recall Section 2.1.3). That is, processes must *block on read*, when trying to get data tokens from an empty FIFO, and *block on write*, when trying to write data tokens to a full FIFO. Moreover, we want our communication approach to be application-independent and oriented to system adaptivity.

The communication and synchronization problem when mapping PPNs on a NoC is depicted in Fig. 3.1, showing a producer P and a consumer C connected through a communication FIFO buffer B. We denote the size of buffer B as $size(B)$. Unless otherwise specified, throughout this thesis we will express the size of buffers in number of tokens. If both producer and consumer can directly access the status register of this FIFO buffer, to check if it is empty or full, implementing the PPN semantics is straightforward. However, in this chapter we consider NoC implementations with no direct remote memory access, that is, those NoCs in which a processing element has direct access only to the local memory of its tile. In this scenario, processes P and C, if mapped onto different tiles, cannot access the same piece of memory and they can only exchange tokens through the network. Thus, the FIFO buffer B has to be split on the producer tile and/or on the consumer tile. We denote the buffer allocated on the producer tile and consumer tile as B^P and B^C, respectively. Note that, as will be shown in one of our proposed PPN communication approaches (see Section 3.4.1), it is not necessary that both these buffers actually exist. However, in

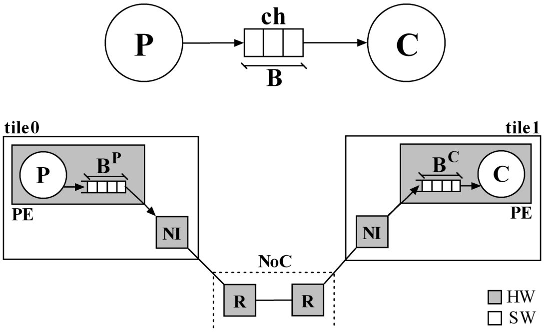


Figure 3.1: *The top part of the figure illustrates a producer-consumer pair which communicates through FIFO buffer B. The bottom part of the figure shows how this pair of processes can be mapped onto a NoC-based platform. The FIFO buffer B has to be split on the producer tile and/or on the consumer tile using two software FIFOs, namely B^P and B^C. Note that the approach presented in this chapter is independent of the considered NoC structure.*

general, if $size(B)$ is the minimum buffer size that guarantees deadlock-free execution of the original PPN graph, the size of B^P and B^C must be necessarily set such that $size(B^P) + size(B^C) \geq size(B)$. In this expression, if either B^P or B^C does not exist, its size is set to zero.

We aim at implementing the PPN semantics without a dedicated support from the underlying hardware architecture that allows checking for the status of the remote FIFO buffers. For instance, in Figure 3.1, process P cannot check the status of B^C, and process C cannot check the status of B^P. Moreover, we do not require support for multiple hardware FIFOs on each NoC tile. Each tile is endowed with only two hardware FIFOs[2], one for incoming messages and one for outgoing messages, both of which reside in the Network Interface (NI). However, we rely on the ability to transfer data, in both directions, from these hardware FIFO buffers to the *software FIFOs* (e.g., B^P and B^C in Figure 3.1) which implement the channels of our PPN and are accessed by the PPN processes.

As the consumer can access the status of B^C, implementing the *blocking read* is trivial because every time C wants to access B^C and this buffer is empty, the consumer just has to wait until tokens arrive from the producer tile. However, since the producer can only access the status of B^P, implementing the *blocking on write* behavior is more difficult. The producer must know that the remote buffer B^C is not full before sending tokens to C over the NoC. Several techniques can be considered to inform the producer about the status of the buffer on the consumer side. We compare the communication approaches that we have investigated in Section 3.4.

As a final requirement of our proposed techniques, we demand that our PPN communication approaches allow processes to be mapped on any tile of the system, without the need to change the actual code structure of the PPN application processes.

---

[2] These hardware FIFOs are not shown in Figure 3.1 to avoid clutter.

An example of such structure is shown in Fig. 2.3(b) on page 29. In particular, we want the communication primitives of PPN processes (*READ*, *WRITE*) to remain generic, without the notion of process mapping or hardware platform primitives. At run-time, these generic PPN primitives are then converted by the PPN communication approaches to corresponding hardware platform primitives which take into account the actual mapping of processes in the system. This is because the mapping of processes in the system can change due to a task migration.

3.2 Contributions

The contribution of this chapter is two-fold. First, we propose different PPN communication approaches that allow applications specified as PPNs to be executed efficiently on NoC-based platforms. The PPN communication approaches support any possible mapping of PPN processes in the system. Second, we ensure that these PPN communication approaches allow the run-time remapping capability of processes among the tiles of the NoC, thus enabling system adaptivity as considered in this thesis.

3.3 Related Work

Kahn Process Networks (KPNs) [Kah74] is a widely studied model of computation used to specify concurrent stream-based applications. The KPN MoC is a superset of the PPN MoC considered in this chapter, therefore in the following paragraphs we list some works, which target KPNs, that are related to the problem addressed in this chapter.

Previous research on the use of KPNs in multiprocessor embedded systems has mainly focused on the design of frameworks which employ that MoC as a model for application specification [NSD08, NKG$^+$02, KKJ$^+$08], and which aim at supporting and optimizing the mapping of KPN processes on the nodes of a reference platform [BHHT10, HSH$^+$09]. In [NSD08, NKG$^+$02], different methods and tools are proposed to generate, in an automated way, KPN application specifications from sequential programs written in C/C++. Design space exploration tools and performance analysis are then usually employed for optimizing the mapping of the generated KPN processes on a reference platform. Then, in the successive design phase, software synthesis for multi-processor systems [KKJ$^+$08, HSH$^+$09] and/or architecture synthesis for FPGA platforms [NSD08] is performed.

The approaches described above, which map applications modeled as KPNs to hardware platforms tailored to the application KPN specification, have a strong coupling between the application and the hardware platform. Running a different application on the generated platform would not be possible or, even if possible, would give bad performance results. In this chapter, we adopt a different approach because we start with the assumption that we have a platform equipped with homogeneous cores well interconnected with a NoC. We provide a PPN API for this platform which implements inter-tile communication among PPN processes. Most importantly, the

PPN processes' code remains the same in all possible mappings of the processes. This is achieved by the proposed PPN communication approaches, that convert the generic PPN communication primitives to corresponding hardware platform primitives that follow the actual mapping of processes in the system.

This approach, where software synthesis relies on the high level APIs provided by the reference platform for facilitating the programming of a multiprocessor system, can be seen in other works in the literature. In fact, the trend from single core design to many core design has forced the research community to consider inter-processor communication issues for transferring data among the cores. One of the emerged message passing communication API is Multicore Association's Communication API (MCAPI) [MCA] that targets the inter-core communication in a multicore chip. MCAPI is the light-weight (low communication latencies and memory footprint) implementation of message passing interface APIs such as Open MPI [ope]. However, these MPI standards do not allow an efficient implementation of KPN (or PPN) semantics [DDF11] because building these semantics on top of their primitives incurs an additional overhead that may be disadvantageous.

The communication and synchronization problem when implementing KPNs on multi-processor platforms without hardware support for FIFO buffers has been considered in [NMSD09] and [HSH$^+$09]. In [NMSD09] the *receiver-initiated* method has been proposed and evaluated for the Cell BE platform. On the same hardware platform, [HSH$^+$09] proposes a different protocol, which makes use of mailboxes and *windowed FIFOs*. The difference with our work presented in this chapter is that we actually compare a number of approaches to implement the KPN semantics, and that we deal with a different kind of platform, with no Direct Memory Access support.

In [DDF11] the active *virtual connector* approach has been proposed and evaluated analytically, whereas our results are obtained by experiments on a real implementation. Moreover, in this chapter we propose yet another approach, namely *virtual connector with variable rate*.

The authors in [NGWK09] address the problem of implementing the KPN semantics on a NoC. However, in their approach the NoC topology is customized to the needs of the application at design time and network end-to-end flow control is used to implement the blocking write feature. In [NGWK09] no run-time task remapping is allowed, because the hardware platform is generated assuming a specific (fixed) mapping of KPN tasks. By contrast, in our work the PPN communication approaches allow run-time remapping of processes and, in turn, system adaptivity.

An approach to guarantee blocking write behavior for KPN processes on NoCs is also used in [Gab09]. In that work, a FIFO buffer that implements a KPN channel is allocated on the tile of the *consumer* process. Then, before sending data tokens, the producer process uses a dedicated operating system communication primitive which guarantees that the remote FIFO buffer is not full. Compared to this kind of protocol, the communication approaches described in this chapter assume a more active behavior of the consumer processes to guarantee the blocking on write behavior. That is, in our approaches the consumer process actively sends back to the producer some messages to inform the producer about the status of the remote FIFO buffer. We actually propose and evaluate three kinds of communication approaches, which re-

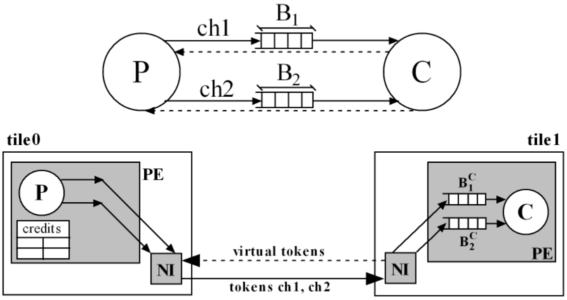


Figure 3.2: *Producer-consumer pair using the virtual connector method. Compared to Figure 3.1, notice that the producer tile does not contain any software FIFO for the considered channels. However, the producer process P uses a credit variable for each channel to keep track of the status of the FIFOs residing on the consumer tile.*

quire the consumer process to be active to a different extent (in terms of the amount of messages sent back to the producer process). The experimental results of Section 3.6.1 show that the communication approach which requires the most proactive behavior of the consumer achieves higher performance compared to the others.

3.4 PPN Communication Approaches

This section presents the different approaches that we have explored for the implementation of PPN processes communication and synchronization on a tiled NoC-based hardware platform. Three PPN communication approaches are proposed and investigated: Virtual Connector approach (VC), Virtual Connector with Variable Rate approach (VRVC), and Request-driven approach (R). Basically, the proposed approaches differ in the frequency of acknowledgment messages sent from the consumer process to the producer process regarding the status of the consumer FIFO buffers.

In all of the approaches described in what follows, system adaptivity is taken into account by using dedicated tables that list, among other information, the current[3] mapping of producer and consumer processes for each channel of the PPN graph. We refer to such tables as *middleware tables*. The current mapping of producer and consumer processes is checked when the PPN primitives (i.e., *READ, WRITE* in Figure 2.3(b) on page 29) are converted to the corresponding hardware platform primitives, such that tokens and synchronization messages are sent to the right tiles in the system. The middleware tables can be updated at run-time, ensuring correct communication in case of remapping of processes.

3.4.1 Virtual Connector approach (VC)

In the Virtual Connector approach, which is depicted in Fig. 3.2, for each channel in the original PPN graph we add a virtual[4] one in the opposite direction. This virtual connector is used for acknowledging the producer about the status of the FIFO buffer on the consumer tile. We adapted this approach, previously proposed in [DDF11], to the needs of our system implementation. In [DDF11], the proposed communication approach is *active*, meaning that it is implemented using separate threads which deal with the PPN communication, while our approach is *static*, with no separate threads dedicated to communication. Although a comparison of the static and active implementations may be worthwhile to do, in this chapter we adopt the static approach with the argument that the scheduling and synchronization of an additional thread dedicated to PPN communication will introduce an additional overhead due to the scheduling and context switching times.

For each channel in the original PPN graph we instantiate a software FIFO buffer on the consumer tile. The size of this buffer is set to the value of the original buffer size in the PPN graph. On the producer tile there are no software FIFOs when using this approach because tokens can be directly sent over the network via the NI. The PPN *blocking write* behavior is implemented by using a credit-system which guarantees that enough locations are free in the FIFO buffers of the consumer processes. Therefore, referring back to Fig. 3.1, in this approach for each channel i, $size(B_i^C) = size(B_i)$ and $size(B_i^P) = 0$.

In our implementation, we store on the producer side a variable for each channel, called *credit*, which represents the number of free slots in the remote FIFO buffer implementing that channel. At startup, the credit is set to the size of the remote FIFO ($credit_i = size(B_i^C)$), because all of its slots are free[5]. For each token sent over the network by the producer, the credit of the corresponding channel is decreased by one. The producer is allowed to send tokens over the network only if the credit is positive, otherwise it blocks. This implements the *blocking write* behavior. At the consumer side, for every token consumed from that channel, a virtual token (VT) is sent back to the producer via the virtual connector. For every virtual token received on the producer tile, the credit of the corresponding channel is increased by one. This way the producer is constantly updated about the status of the remote FIFO buffers.

Read and Write communication primitives

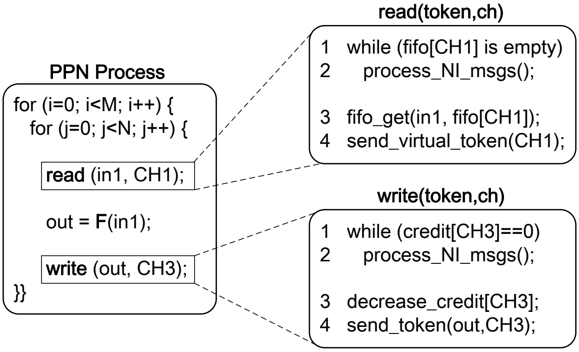
The read and write primitives use an auxiliary function called *process_NI_msgs()*. This function is used in the read primitive when the calling process is blocked on read, and in the write primitive when it is blocked on write. The *process_NI_msgs()* function checks the status of the NI buffer for incoming messages. If the buffer is not empty, it

---

[3] Although a task migration mechanism is not provided in this chapter, our proposed PPN communication approaches must allow the spatial mapping of tasks to change at run-time.

[4] These channels are said to be virtual because they are not used to communicate actual data.

[5] This holds unless there are initial tokens in B^C. In such a case, the value of $credit_i$ is decreased by the number of initial tokens.



```
for (i=0; i<M; i++) {
  for (j=0; j<N; j++) {

    read (in1, CH1);

    out = F(in1);

    write (out, CH3);
}}
```

```
1 while (fifo[CH1] is empty)
2    process_NI_msgs();

3 fifo_get(in1, fifo[CH1]);
4 send_virtual_token(CH1);
```

```
1 while (credit[CH3]==0)
2    process_NI_msgs();

3 decrease_credit[CH3];
4 send_token(out,CH3);
```

Figure 3.3: *Pseudocode of the VC approach. The left part of the figure shows an example of structure of a PPN process. The right side provides the pseudocodes of read and write PPN primitives as implemented in the VC communication approach.*

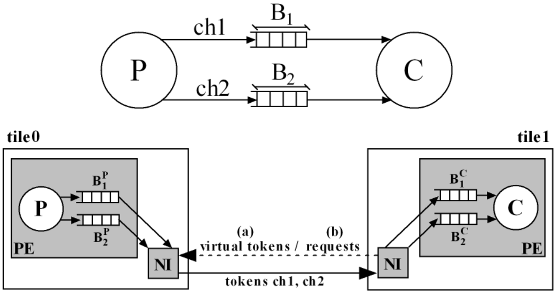


Figure 3.4: *Producer-consumer implementation: when using the VRVC approach, the producer receives back virtual tokens (a); when using the R approach, it receives requests (b).*

processes one message at a time, until all the incoming messages are consumed, in the following way. If the message is an incoming token for channel *i*, it stores the token in the software FIFO which implements channel *i*. If, instead, it is a virtual token for channel *j*, it consumes the token and increases the credit of channel *j*.

Read primitive. In the VC approach, the *read* primitive (used to read a token from channel *ch*) performs the following sequence of actions.

1. It checks if the FIFO buffer corresponding to *ch* contains data tokens (blocking read behavior). If the FIFO is empty, it keeps executing the auxiliary function *process_NI_msgs()* until the FIFO is no longer empty.
2. At this point, the FIFO corresponding to *ch* contains data tokens. Then, the read primitive gets a token from the FIFO.
3. Finally, a virtual token is sent back to the consumer process to acknowledge that a token has been read from the FIFO.

These actions are implemented in the *read* primitive in Fig. 3.3. Lines 1-2 implement the blocking read. If the FIFO buffer corresponding to the calling channel (in the example, *CH1*) is empty, *process_NI_msgs()* is executed until new tokens for that channel reach the NI input buffer. Lines 3 and 4 complete the *read* primitive: the token is transferred from the software FIFO to *in1*, and a virtual token is sent back to the producer side of *CH1*. This is actually performed by putting in the NI outgoing buffer a message representing a virtual token for channel *CH1*.

Write primitive. In the VC approach, the *write* primitive (used to write a token to channel *ch*) performs the following sequence of actions.

1. It checks if the *credit* corresponding to channel *ch* is equal to zero (blocking write). In this case, it keeps executing the auxiliary function *process_NI_msgs()* until the the credit is no longer zero.
2. At this point, the credit corresponding to *ch* is greater than zero. Then, the credit is decreased by one to consider the fact that in the next step a token will be sent, over the NoC, to the consumer.
3. Finally, the token is sent to the consumer over the NoC.

These actions are implemented in the *write* primitive in Fig. 3.3. Lines 1-2 implement the blocking write behavior. If the credit is zero, *process_NI_msgs()* is executed. If virtual tokens for the blocked channel are received, the credit is then increased and this condition unblocks the write to that channel. Lines 3-4 complete the *write* procedure. The credit for the considered channel is decreased, and the token is sent over the network, which is done by putting in the NI outgoing buffer a message representing this token, and then letting the NI to perform the actual transfer over the NoC[6].

3.4.2 Virtual Connector with Variable Rate approach (VRVC)

This approach represents a variant of the *virtual connector* described in Section 3.4.1. The basic idea is that instead of sending one virtual token to the producer for *every* token consumed from channel i, the consumer sends it after n_i consumed tokens, where n_i is a parameter that can be set such that $1 \leq n_i \leq size(B_i)$, where $size(B_i)$ is the buffer size in the original PPN graph. The *credit* variable for channel i will then be increased by n_i for every virtual token received for that channel. This approach leads to a reduced traffic on virtual connectors, which can be beneficial in NoC implementations to avoid congestion of messages.

Since the sending back of virtual tokens does not happen for every consumed token, in some cases the PPN graph properties require to store, also at the producer side, tokens for the channels in order to avoid deadlocks. We provide an explanation of this phenomenon in Example 3.4.1.

Example 3.4.1. Consider the scenario depicted in Figure 3.5, where producer process P is connected to consumer process C through a channel ch, implemented by a FIFO buffer B. Assume that, for channel ch, the parameter n of the VRVC approach is set to $size(B)$, the size of buffer B in the original PPN graph. As mentioned earlier, FIFO buffers are required both on the producer and on the consumer tile. We denote these FIFO buffers as B^P and B^C, respectively. Assume that the size of these buffers are

---

[6] For a brief description of how messages are sent over the NoC, please refer to Section 1.1.2.

$size(B^P) = (size(B) - 1)$ and $size(B^C) = size(B)$. We will eventually show that this assumption is necessary.

We recall that, in the original PPN graph, the size of a certain FIFO buffer B is computed by the `pn` compiler [VNS07] such that deadlocks cannot occur due to a lack of space in B. The size of buffer B derived by `pn` is minimum, that is, at run-time it may happen that B is required to store a number of token equal to its size to avoid a deadlock. Therefore, since FIFO buffer B in the VRVC approach is split over B^P and B^C, in order to avoid deadlocks it is necessary that at any time up to $size(B)$ tokens can be stored over B^P and B^C. Denoting the number of tokens stored in buffer B^P and B^C at time instant t as $tkns(B^P, t)$ and $tkns(B^C, t)$, respectively, we have that, over time, $\big(tkns(B^P, t) + tkns(B^C, t)\big)$ can grow up to $size(B)$.

At run-time, the communication between producer process P and consumer process C may follow the sequence of macro-steps shown in Figure 3.5 and explained below.

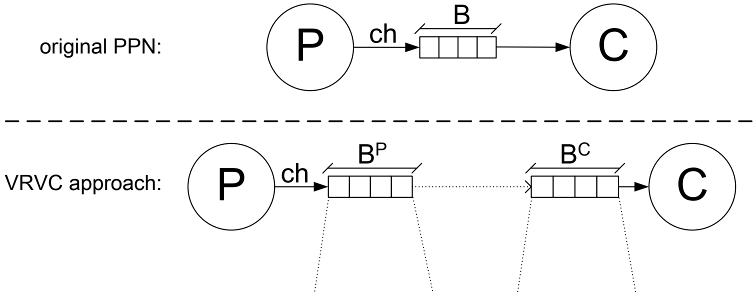
- **Step (1).** This step represents system startup. The credit variable of channel ch is initialized to $size(B)$. The number of tokens stored in B^P and B^C are both zero.
- **Step (2).** After a series of $size(B)$ tokens sent by P, and not consumed by C (recall that, in PPNs, control is completely distributed, so this scenario may occur), the credit variable for channel ch is zero and $tkns(B^C) = size(B)$.
- **Step (3).** Consumer C consumes $size(B) - 1$ tokens, such that, at time instant t_3, $tkns(B^C, t_3) = 1$. However, no virtual token is sent back to P because the parameter n of the VRVC approach is set to $size(B)$ and only $size(B) - 1$ tokens have been consumed.

At the end of Step (3) the total number of tokens stored in B^P and B^C is 1, that is, $\big(tkns(B^P, t_3) + tkns(B^C, t_3)\big) = 1$. However, as mentioned earlier, over time the total number of tokens stored over B^P and B^C should be able to grow up to $size(B)$, to avoid deadlocks. Now, if consumer process C does not consume the last token present in B^C, producer process P cannot send tokens to C. Therefore, P must be able to store up to $size(B) - 1$ tokens in B^P, a scenario which is considered in the next step.

- **Step (4).** This step represents the scenario in which P, at time instant $t_4 > t_3$, has stored $size(B) - 1$ tokens in B^P, and C has only one token left in B^P. Now, since $\big(tkns(B^P, t_4) + tkns(B^C, t_4)\big) = size(B)$, we are sure that process P cannot cause a deadlock due to lack of space in $tkns(B^P)$ and $tkns(B^C)$. Eventually, C will consume the last token left in B^C and send a virtual token back to P, which will increase the credit variable for the corresponding channel and allow new token transfers over the NoC from P to C.

From the scenario described in Example 3.4.1 it follows that FIFO buffers are needed on both the producer and the consumer side. Note that Example 3.4.1 considers a worst-case scenario in the communication between producer and consumer processes. Therefore, the derived size of the buffers, $(size(B) - 1)$ for B^P and $size(B)$ for B^C, is sufficient for all possible scenarios which may arise at run-time.

The pseudocode of the VRVC communication approach is shown in Fig. 3.6. Compared to the VC approach, the behavior of *process_NI_msgs()*, which is used in



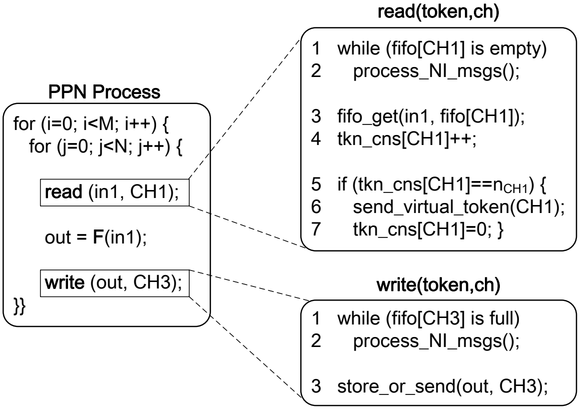
Time	Step	credit(ch)	tkns(B^P)		tkns(B^C)
t_1	(1)	size(B)	0		0
t_2	(2)	0	0		size(B)
t_3	(3)	0	0		1
t_4	(4)	0	size(B)-1		1

Figure 3.5: *Communication sequence of a producer-consumer pair, using the VRVC approach, requiring storing of tokens on both the producer and consumer tile. The table in the lower part of the figure shows, at different steps of the communication sequence, the credit value associated to the considered channel, and the number of tokens stored in the producer FIFO buffer* B^P *and consumer FIFO buffer* B^C.

both the *read* and *write* primitives, changes with regard to the processing of virtual tokens. The first difference is that whenever a virtual token for channel *i* is received, *process_NI_msgs()* consumes it and increases the credit of channel *i* by the parameter n_i. The second difference is that when a virtual token is received and the corresponding FIFO buffer is not empty, as many available tokens as possible are sent to the consumer tile, until the credit for that channel allows so. The credit variable is decreased, accordingly, by the number of tokens sent to the consumer tile.

In the *read* primitive shown in Fig. 3.6, lines 1-2 implement the blocking read behavior, similarly to the VC approach. However, the rest of the primitive is different. In line 3, a token is read from the FIFO buffer which implements channel *CH1*. Line 4 uses an auxiliary variable, *tkns_cns[CH1]*, which keeps track of the number of tokens consumed from *CH1* since the last virtual token sent back (for the corresponding channel) to the producer tile. This auxiliary variable is initialized to zero at startup and is increased for every token consumed by the process from channel *CH1*. In lines 5-7, when this variable reaches the parameter n_{CH1}, a virtual token is sent back to the producer tile and *tkns_cns[CH1]* is reset to zero.

Similarly to the VC approach, in the *write* primitive of VRVC shown in Fig. 3.6, lines 1-2 implement the blocking write behavior. When the control reaches line 3, we are sure that the corresponding FIFO is not full. Then, in the auxiliary function *store_or_send*, the token is either stored in the FIFO buffer corresponding to *CH3* (if the credit variable associated to *CH3* is zero) or sent over the NoC to the consumer



```
for (i=0; i<M; i++) {
  for (j=0; j<N; j++) {

    read (in1, CH1);

    out = F(in1);

    write (out, CH3);
}}
```

```
while (fifo[CH1] is empty)
  process_NI_msgs();

fifo_get(in1, fifo[CH1]);
tkn_cns[CH1]++;

if (tkn_cns[CH1]==$n_{CH1}$) {
  send_virtual_token(CH1);
  tkn_cns[CH1]=0; }
```

```
while (fifo[CH3] is full)
  process_NI_msgs();

store_or_send(out, CH3);
```

Figure 3.6: *Pseudocode of the VRVC approach. The left part of the figure shows an example of structure of a PPN process. The right side of the figure provides the pseudocodes of read and write PPN primitives as implemented in the VRVC communication approach.*

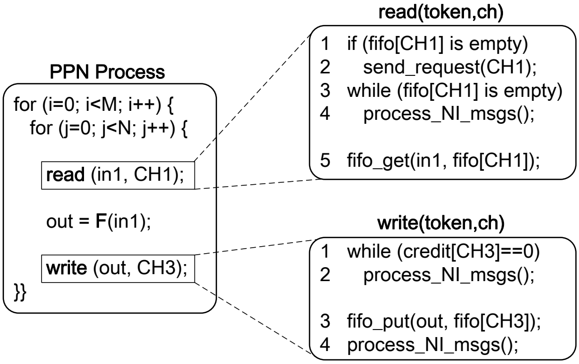
tile (if the credit variable is greater than zero).

3.4.3 Request-driven approach (R)

This method is very similar to the approach used in [NMSD09] for realizing communication among KPN processes on the Cell BE platform [KDH$^+$05]. In the request-driven approach, the transfer of tokens from the producer tile to the consumer tile is *initiated by the consumer*. This means that every time the consumer is blocked on a read at a given FIFO channel, it sends a *request* to the producer to send new tokens for that channel. The producer, after receiving this request, sends *as many tokens* as it has in its software FIFO implementing that channel.

Since also in this case we need to store tokens both on the producer side and on the consumer side, we need software FIFO structures on both sides. The size of these buffers is set, for each channel i, to match the size of the buffer in the original PPN graph (B_i), therefore $\forall i \in \{1, \cdots, |\mathcal{C}|\}$ $size(B_i^P) = size(B_i^C) = size(B_i)$. This condition guarantees deadlock-free execution on the NoC because: (i) the FIFO buffer residing on the producer tile (B^P) is large enough to avoid deadlocks caused by block on write on the producer side; (ii) after a request sent by the consumer process, B^C is large enough to store the maximum amount of tokens stored in B^P. The structure of a producer-consumer pair using the R approach is shown in Fig. 3.4, case (b). Since the consumer buffer of a channel is empty when a request is made, and given that the FIFO buffers for that channel have the same size on both sides, there is always enough space to store tokens sent by the producer as a consequence of the request.

Fig. 3.7 shows the pseudocode of this PPN communication approach. Similarly to the other communication approaches, it makes use of the auxiliary function *process_NI_msgs()* to process incoming messages of tokens or requests. In the R approach, this function is in charge of reacting to a received request message for a channel



```
for (i=0; i<M; i++) {
  for (j=0; j<N; j++) {

    read (in1, CH1);

    out = F(in1);

    write (out, CH3);
}}
```

```
1  if (fifo[CH1] is empty)
2    send_request(CH1);
3  while (fifo[CH1] is empty)
4    process_NI_msgs();

5  fifo_get(in1, fifo[CH1]);
```

```
1  while (credit[CH3]==0)
2    process_NI_msgs();

3  fifo_put(out, fifo[CH3]);
4  process_NI_msgs();
```

Figure 3.7: *Pseudocode of the Request-driven (R) approach. The left part of the figure shows an example of structure of a PPN process. The right side of the figure provides the pseudocodes of read and write PPN primitives as implemented in the request-driven communication approach.*

with the immediate sending of all the tokens contained in the software FIFO that implements that specific channel.

The *blocking on read* behavior is implemented in lines 1-4 of the read primitive in Fig. 3.7. When the software FIFO of the calling channel is empty, a request is sent to the producer tile of that channel, and the processor keeps executing *process_NI_msgs()* until a message of tokens for the calling channel arrives. The *blocking on write* is implemented in lines 1-2 of the write primitive. When the FIFO of the calling channel (in the example, *CH3*) is full, the processor keeps executing *process_NI_msgs()* until a request for that channel arrives. Line 4 of the write primitive allows a faster response to requests for tokens from consumer processes. In fact, if line 2 of the write primitive is not executed, i.e, if the calling channel is not full, *process_NI_msgs()* is anyway executed in line 4, leading to a faster response to token requests.

3.5 Case Studies

We evaluate the three PPN communication approaches presented in Section 3.4 on two applications, specified as PPNs, with extremely different communication/computation characteristics. The reason is that we want to compare the overhead of the PPN communication approaches between two extremes. The application described in Section 3.5.1 represents the first extreme, when the computation/communication ratio is low and the PPN topology is complicated. The case study described in Section 3.5.2, on the other extreme, is computation dominant and with relatively simple PPN topology. We describe briefly the two case studies in order to get a better understanding of the obtained results. In Section 3.5.3 we also provide an overview of the platform that we use to run the experiments.

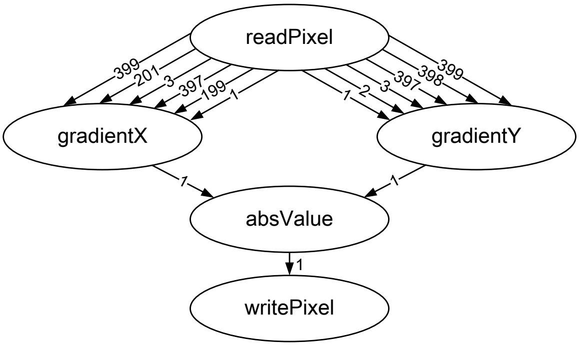


Figure 3.8: *PPN specification of the Sobel filter.*

Table 3.1: *Execution times (in clock cycles) of Sobel functions*

Process	Execution time (c.c.)
readPixel	5
gradientX	31
gradientY	31
absValue	118
writePixel	5

3.5.1 Sobel filter

The Sobel application is an edge-detection algorithm for digital images. Its PPN graph is shown in Fig. 3.8, where the number written over each edge indicate the minimal buffer sizes (expressed in data tokens) needed for that channel, in order to process a 200x122 pixel input image without deadlocks. The PPN processes of this application are very lightweight in terms of computation. The numbers of clock cycles (c.c.) required for one execution of each function are summarized in Table 3.1. The most computationally intensive process is *absValue*, which sums the absolute values of the outputs of the *gradientX* and the *gradientY* processes and normalizes the result. For all of the channels in the graph, the size of exchanged tokens is 4 bytes, and the number of written tokens is 23760. From these metrics it is clear that the Sobel application is largely communication-dominant. Therefore, even before running the actual experiments, we expect this application to perform poorly on NoC-based hardware platforms, where communication is more costly than on platforms with dedicated, point-to-point interconnections. However, we use this example to represent the class of applications in which the communication dominates the computation.

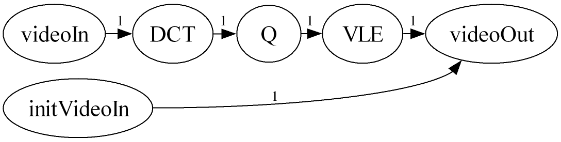


Figure 3.9: *PPN specification of the M-JPEG encoder.*

Table 3.2: *Execution times (in clock cycles) of M-JPEG functions*

Process	Execution time (c.c.)
initVideoIn	18
videoIn	1910
DCT	126386
Q	69238 (avg)
VLE	46688 (avg)
videoOut	1292 (avg)

3.5.2 M-JPEG encoder

The PPN specification of this application is shown in Fig. 3.9. The size of tokens, corresponding to different channels, ranges between 16 and 1024 bytes. All of the channels are written 128 times, except the output of *initVideoIn* which is written only once. The numbers of clock cycles required for the execution of each function of the M-JPEG application are summarized in Table 3.2. This application shows a much simpler communication and synchronization pattern compared to Sobel, and it also has a much higher computation/communication ratio.

3.5.3 Platform setup

The system on which we evaluate our PPN communication approaches is based on a 2x2 mesh of tiles, connected via a custom-built Network-on-Chip. We choose this kind of NoC because, as mentioned in Section 1.1.2, it is the most common and widely studied topology of NoC. However, note that our proposed PPN communication approaches do not depend on the topology of the NoC. Each tile is composed of a MicroBlaze processor, with its program and data memories, and a Network Interface. The platform does not support remote memory access. The system runs at the frequency of 100 MHz.

Each processor has multi-tasking capabilities thanks to the use of the *Xilkernel* operating system, a lightweight, customizable kernel provided by Xilinx. In case of *many-to-one* mapping, i.e. when more than one process are mapped on the same processor, the scheduling is data-driven. This means that a process keeps executing successive iterations until it blocks in reading or writing (recall the PPN process structure of Figure 2.3(b) on page 29). When the process blocks, it yields the processor control to the next process in the ready queue, using round-robin.

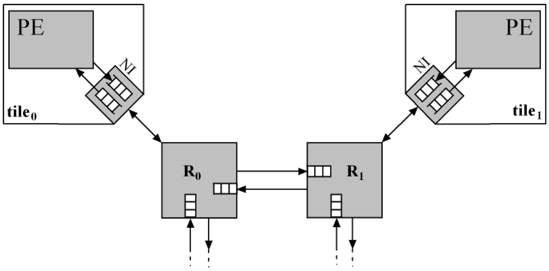


Figure 3.10: *(Part of) the NoC platform structure. The full structure of the adopted NoC structure is a 2x2 mesh of tiles.*

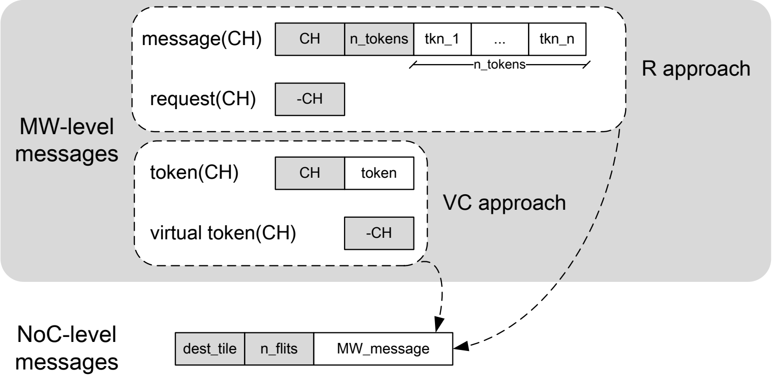
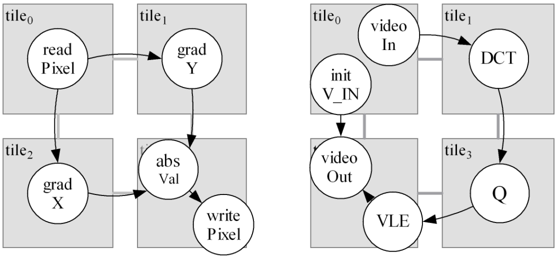


Figure 3.11: *Structure of middleware- and network- level messages.*

As shown in Fig. 3.10, the Network Interface contains only two hardware FIFOs, one for messages which are incoming from the NoC, and one for messages that have to be injected in the NoC. The processor is able to quickly access the status of the incoming hardware FIFO, via a dedicated signal, to see if there are messages to be forwarded from the NI buffer to the software FIFO buffers that implement the channels of the PPN graph. In the opposite direction, when a message has to be sent over the NoC, the processor forwards data from its data memory to the outgoing NI hardware FIFO, then the NI injects the message in the network, with the appropriate header (destination tile and payload size fields). The messages are sent over the NoC using *wormhole switching* (as in [BB04]). As shown in Fig. 3.10, routers (R_0 and R_1) use input buffering to store incoming *flits* (flow control digits, which represent the granules into which messages are split). Moreover, in our implementation the routers use a simple round-robin arbitration policy.

The actual structure of the different kind of messages that are sent over the NoC is represented in Fig. 3.11, for the *VC* and *R* approaches. At NoC-level, the message comprises a NoC header, that indicates the destination tile and the size of the payload, and the payload itself, which we refer to as the middleware (MW)-level message. The



(a) (b)

Figure 3.12: *Fixed mappings for Sobel (a) and M-JPEG (b) to test the different PPN communication approaches.*

structure of MW-level messages depends on the PPN communication approach. In *R*, a request for channel number *i* is implemented as a single flit, with value $-i$. A message used for transferring tokens, instead, has a header composed of two flits (channel number, number of sent tokens) and a payload with the sent tokens. The field that indicates the number of sent tokens (*n_tokens*) is necessary because this number is determined at run-time, when a request for that channel is received. The structure of MW-level messages in *VC* is very similar, the only difference being that there is no need for a *n_tokens* field because in this method sending several tokens in one message is not allowed, i.e. *n_tokens* is always equal to one.

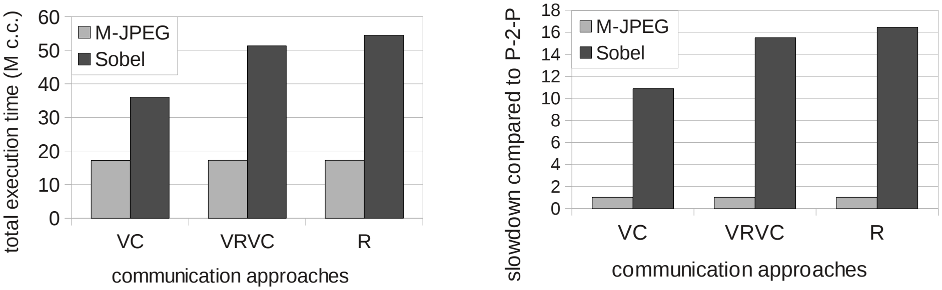
3.6 Experimental Results

The platform described in Section 3.5.3 has been implemented on a Xilinx Virtex-5 FPGA prototyping board. We run the two application case studies of Sections 3.5.1 and 3.5.2, with all the PPN communication approaches proposed in Section 3.4, and obtain the results described below.

3.6.1 Inter-tile communication efficiency

In order to compare the efficiency of inter-tile communication of the different PPN communication approaches, we execute the two case study applications with the fixed mappings shown in Fig. 3.12. We chose these mappings because they expose the maximum amount of inter-tile communication, therefore the obtained results are largely dependent on the efficiency of the considered PPN communication approach.

We found, experimentally, that the parameter n_i of the VRVC approach gives the best performance when set to its maximum value, i.e. when $\forall i \in \{1, \cdots, N_{ch}\}\ n_i = B_i^C$. The performance results, summarized in Fig. 3.13(a), show a large difference of execution time for the Sobel application when using different PPN communication approaches. However, in the M-JPEG case all of the communication approaches yield similar results, due to the much higher computation over communication ratio of



(a) Total execution time for different MW approaches.

(b) Slowdown for different MW approaches.

Figure 3.13: *Sub-figure (a) shows the total execution time of the M-JPEG and Sobel applications obtained with different MW approaches. Notice the large difference of execution time for the Sobel application depending on the used MW approach. By contrast, the execution time for M-JPEG is not affected by the choice of MW approach. Sub-figure (b) compares the performance obtained using our NoC-based platform, together with the proposed MW approaches, with the performance of a customized systems based on point-to-point connections. Notice the large slowdown of the NoC-based implementation for the communication-dominant application, Sobel.*

that application. The VC approach performs much better, compared to the others, in the Sobel application, because its implementation does not require storing of tokens on the producer tile. This leads to a faster communication process, because it avoids the double copy (output variable → software FIFO → NI buffer) that is necessary in the other cases. We argue that the obtained results may change for NoC platforms with Direct Memory Access (DMA) cores, that can benefit more from sending several tokens with one message, as allowed in the VRVC and R approaches.

In order to evaluate the overhead occurred by using the NoC interconnection and our PPN communication approaches, we implemented customized point-to-point systems, for both Sobel and M-JPEG applications, as a baseline reference. In point-to-point systems, generated using the ESPAM tool [NSD08], a dedicated hardware FIFO is instantiated for each channel of the PPN graph. In this way, the hardware platform perfectly matches the PPN MoC semantics. Obviously, customized point-to-point implementations do not allow for system adaptivity, because all the design decisions (e.g.: process mapping) have to be made at design time. It is clear that in our NoC system we sacrifice performance (especially for communication intensive applications) for adaptivity, the ability of managing the system at run-time, and generality, since the system is able to execute any kind of PPN application. The performance slowdown, when comparing the NoC-based systems with the point-to-point systems is shown in Fig. 3.13(b). It is noticeable that while the Sobel application is highly penalized in the execution on our NoC system, the M-JPEG application performs well because of its higher computation/communication ratio and its regular communication pattern. The reason why the PPN communication on the NoC platform is less efficient, compared to customized point-to-point systems, is mainly twofold. The first reason is that in

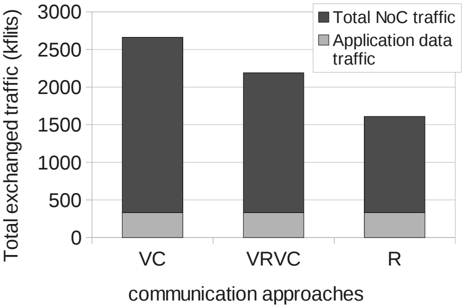


Figure 3.14: *Traffic injected into the NoC by executing Sobel with different MW approaches.*

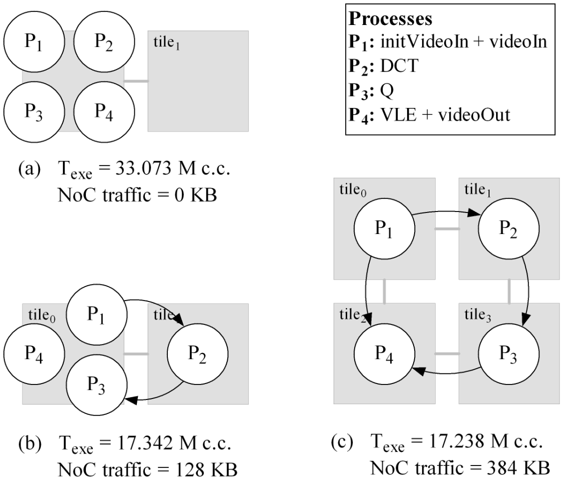
communicating on the NoC, several PPN channels have to share the same physical channel (the NoC link). The second reason is that in the NoC case we have to use software FIFOs on the producer and on the consumer side, which require additional memory copy operations which would be unnecessary in the case of adoption of hardware FIFOs.

Another important metric when executing applications on a NoC is the amount of generated control traffic overhead. In the *VC* case, for instance, this overhead is represented by the NoC-level and MW-level headers, together with all the traffic generated by the virtual tokens. Ideally, a PPN communication should be designed to generate as less control traffic overhead as possible.

Focusing on the Sobel application, since it has the most complex communication pattern, we profiled the amount of traffic injected in the network, depending on the PPN communication approach that is used. The results, depicted in Fig. 3.14, show that the amount of traffic injected in the NoC (that is, including message headers, messages sent over virtual channels, or requests messages) is much larger than the actual application data traffic. This is because the size of tokens in the Sobel application is extremely small, i.e., only 4 bytes. Note also that the *VC* approach injects much more total traffic into the NoC compared to the *R* approach. This large difference can be explained by two factors. The first factor is the overhead of message headers. On the one hand, in the *VC* method, each token travels in the NoC with its own header. On the other hand, in the *R* case, the producer sends as many token as present in its software FIFO, in the same message and therefore with the same header. The second factor is that the traffic on virtual channels in *VC* is much more than the traffic generated by requests in *R*. This is because in the *VC* approach a virtual token is sent back to the producer for every consumed token, while in the *R* approach the requests are made less frequently, just when the consumer is blocked on reading.

3.6.2 System adaptivity support

All the proposed PPN communication approaches (*VC*, *VRVC*, *R*) allow to change at run-time the mapping of PPN processes to tiles of the system. Process remapping is



(a) T_{exe} = 33.073 M c.c.
NoC traffic = 0 KB

(b) T_{exe} = 17.342 M c.c.
NoC traffic = 128 KB

(c) T_{exe} = 17.238 M c.c.
NoC traffic = 384 KB

Figure 3.15: *Execution time and generated traffic as a function of the process mapping. Only inter-tile communication links are depicted.*

allowed because all the PPN communication approaches exploit a *middleware table*, which keeps track of the source and destination tiles of each channel of the PPN. We recall that these tables are used to convert the generic PPN primitives (see for instance *READ, WRITE* in Figure 2.3(b) on page 29) to the corresponding hardware platform communication primitives, where the source tile and destination tile of a channel are precisely specified. When a remapping of processes is performed, the middleware tables are changed accordingly.

Note that the actual mechanism used to perform run-time remapping of processes is proposed and explained in the next chapter, Chapter 4. However, the approaches presented in this chapter ensure that, once the remapping procedure is completed, PPN processes can communicate correctly also in the new spatial mapping.

The possibility of choosing a different mapping at run-time can be exploited by run-time management algorithms, or simply by the user of the system, to trade between performance and other metrics, such as power. For instance, in Fig. 3.15 we show the performance results obtained with different mappings of the M-JPEG application. For each mapping, Fig. 3.15 shows the total execution time (T_{exe}) and total exchanged traffic over the NoC (*NoC traffic*). Mapping (a) in the figure has only one active tile, whereas mapping (b) requires two active tiles. Therefore, we can infer that mapping (a) is more power efficient than mapping (b), also because the former uses no power to communicate over the NoC. However, T_{exe} achieved by mapping (b) is almost half of the one of mapping (a), thus mapping (b) is preferable when the system has to provide high performance. Finally, by comparing mapping (b) and (c) in Fig. 3.15, we see that exploring more than two tiles when mapping the M-JPEG application has only marginal performance benefits.

3.7 Discussion

From the experimental results reported in Section 3.6.1 we can quantitatively compare the three PPN communication approaches proposed and evaluated in this chapter. For each communication approach, we summarize the advantages (+) and disadvantages (−) in the following list.

- Virtual Connector (*VC*):
  - \+ Outperforms all the other approaches (*VRVC*, *R*) for applications with low computation over communication ratio;
  - − Requires a credit-based system, with frequent synchronization between producer and consumer processes.
- Virtual Connector with Variable Rate (*VRVC*):
  - \+ Achieves slightly higher performance than the *R* approach for applications with low computation over communication ratio;
  - − Requires a credit-based system, and synchronization between producer and consumer processes with a frequency which depends on a parameter that can be tuned.
- Request-driven (*R*):
  - \+ Simpler implementation compared to the other approaches, without a credit-based system and with less synchronization points;
  - \+ Achieves performance nearly identical to *VC* and *VRVC* when the computation over communication ratio is high;
  - − Achieves low performance for applications in which the computation over communication ratio is low.

From the above comparison, it follows that if a designer needs to map a PPN application with low computation over communication ratio on a NoC based execution platform and *with a static mapping*, the *VC* approach is preferable.

However, when run-time remapping of processes is necessary, the *VC* approach is less appealing because it requires frequent synchronization between producer and consumer processes. This is especially true when process remapping is needed to achieve fault tolerance, with process migrations that can be triggered at any time by hardware faults. Note that with our proposed middleware we want to address also this kind of scenario. Therefore, in Chapter 4, we propose a process migration mechanism which is based on the *R* approach. In fact, as mentioned above, *R* has two main advantages. First, it has less synchronization points between producer and consumer processes, and it is easy to implement. Second, it achieves identical performance for applications with high computation over communication ratio, the kind of applications which are more likely to be executed on NoC platforms.

Chapter 4

Process Migration Mechanism in a Mapped PPN

Most of the work presented in this chapter has been published in [CDM+12].

IN Chapter 3 we investigated several approaches that allow to execute applications specified as PPNs on Network-on-Chip (NoC) based MPSoCs. The approaches presented in Chapter 3 represent alternative implementations of the first component of our proposed middleware[1], introduced in Section 1.4.1 and depicted in the software stack of Figure 1.6 on page 20. The first component of our middleware, *PPN Communication*, allows PPN processes to communicate in NoC based MPSoCs, regardless of the mapping of processes to the available PEs. This is a necessary property in order to achieve system adaptivity by means of process migration. However, it is also necessary to define how to perform the transition between the current mapping and the next desired one. That is, we have to provide a mechanism to perform process migration. In our approach, this mechanism is implemented by the second component of our proposed middleware shown in Figure 1.6, namely *Process Migration*, which is proposed in this chapter. We recall that the techniques proposed in Chapter 3 and this chapter are aimed at **best-effort systems**.

4.1 Problem Statement

In this chapter, we address the problem of defining and implementing a process migration mechanism, targeted at PPN processes on NoCs, that satisfies the following three requirements:

---

[1] We recall that the proposed middleware is aimed at achieving system adaptivity in embedded NoC based MPSoCs by exploiting process migration.

1. Once the process migration is triggered, it has to be completed within a certain, known time frame. We refer to this property as *predictability*.
2. The process migration can be triggered in the system at any time. We consider this requirement because we want to cover the scenario in which process migration is needed in response to a hardware fault, for which the moment of occurrence is unknown.
3. The code used to allow process migration has to be generated automatically, without the manual intervention of the designer. This is needed to relieve designers from the time-consuming and error-prone task of inserting the code necessary to allow task migration by hand.

4.2 Contributions of this Chapter

We devise and develop a predictable process migration mechanism that allows run-time process remapping among the tiles of the NoC, which is a fundamental requirement for system adaptivity. The peculiarity of our solution is that, leveraging the PPN operational semantics and process structure, the migration can actually start at any point during the execution of the main body[2] of a process without the need of moving a large state. Moreover, an upper bound of the process migration overhead can be found, based on the PPN topology and FIFO buffer sizes. Finally, the code used to allow process migration is minimally invasive with respect to the original code structure and can be generated in a completely automated way.

4.3 Related Work

Run-time resource management is a widely studied topic in general purpose distributed systems scheduling [CJK88]. In particular, process migration mechanisms [Smi88, MDP$^+$00], have been developed and evaluated in this context to enable dynamic load distribution, fault resilience, and improved system administration and data access locality. In recent years, run-time management is gaining popularity and finding applications also in multiprocessor embedded systems. This domain imposes tight constraints, such as cost, power, and predictability, that run-time management and process migration mechanisms must consider carefully. [NVC10] provides a survey of run-time management examples in state-of-the-art academic and industrial MPSoCs, together with a generic description of run-time manager features and design space.

Our work is focused on a specific component of run-time management strategies, namely the process migration mechanism. Papers which specifically address process (or task) migration implementation in MPSoCs can also be found in the literature. The closest to our work is [Gab09], in which the goals of scalability and system adaptivity are achieved through a distributed task migration decision policy over a

---

[2] With the term “main body” we mean that a migration can happen during most of the execution time of a process. This concept will be explained in greater detail in Section 4.5.1.

purely distributed-memory multiprocessor. Similar to our approach, their platform is programmed using a process network MoC. However, in their approach the actual task migration can take place only at fixed points, which correspond to the communication primitive calls. Our approach, instead, enables migration at any point in the execution of the main body of processes. This leads to a faster response time to migration decisions, which is preferable for instance in case of hardware faults.

Other task migration approaches are explained and quantitatively evaluated in [BABP06] and [AACP08]. Dynamic task re-mapping is achieved at user-level or middleware/OS level, respectively. In both these approaches, the user needs to define checkpoints in the code where the migration can take place. This can require a consistent manual effort from the designer which is not needed in our approach. Moreover, a relevant difference with respect to our work is the inter-task communication implementation, which exploits a shared memory system. We argue that our approach, which uses purely distributed memory, can perform better in emerging MPSoC platforms since it provides better scalability.

Finally, the authors in [NKG$^+$02] propose an MPSoC hardware and software architecture template that allows the system to change the mapping of applications' tasks at run-time. Compared to the work presented in this chapter, their approach uses a distributed *shared* memory to implement inter-task communication. That is, in their execution platform a process can read data tokens directly from the memory of a different processor. As mentioned earlier, by contrast, our approach uses completely distributed memories, with the goal of providing better scalability in emerging MPSoCs.

It is worth noting that the process migration mechanism presented in this chapter has been devised and implemented within the MADNESS EU FP7 Project, in close cooperation with DIEE, University of Cagliari, and ALaRI, Faculty of Informatics, University of Lugano. The mentioned migration mechanism, initially presented in [CDM$^+$12], has been used as base infrastructure for other works within the MADNESS project [MTR$^+$12, DCT$^+$13]. In particular, in his PhD dissertation [Der15], Derin has proposed migration techniques that are complementary to the one described in this chapter. In Chapter 5 of [Der15], the author makes the following contributions which are closely related to the process migration mechanism of [CDM$^+$12].

- First, he shows how software self-testing routines, capable of detecting faults, can be coupled with the migration mechanism of [CDM$^+$12].
- Second, he devises a *task migration hardware* module that is included in each tile of the NoC-based MPSoC. In case of a fault, this task migration hardware is in charge of extracting the state of processes from the faulty tile to make it available to the resource manager of the MPSoC.
- Third, he proposes an alternative way of handling faults at the PPN application level. In [CDM$^+$12], when a PPN process is interrupted by a fault, the execution is resumed on another tile by rolling back to the beginning of the interrupted iteration of the PPN process. Chapter 5 of [Der15] provides also a different way of fault recovery, that resumes the execution on a different tile by rolling forward to the PPN process iteration that *follows* the interrupted iteration. This yields to

a simpler task migration hardware implementation, although the application output can be temporarily incorrect.

The remainder of the chapter is organized as follows. In Section 4.4, we summarize the assumptions of our system adaptivity approach and we provide an overview of our proposed process migration mechanism. Then, in Section 4.5, we describe the proposed process migration mechanism in greater detail. Finally, Section 4.6 concludes this chapter reporting the experiments performed to test our process migration mechanism, and the achieved results.

4.4 Proposed Migration Approach

In the following paragraphs we recall some assumptions, related to the structure of the MPSoCs that are considered by our approach, that have an important influence on our proposed process migration mechanism. Then, we provide an overview of the migration mechanism itself.

The starting assumption of our system adaptivity approach, as depicted in the right part of Figure 1.6 on page 20, is that we target an MPSoC composed of tiles, connected by a NoC, with completely distributed memories and no direct remote memory access. This means that the processing element of a tile can only directly access the content of its own local memory. All the communication and synchronization between processes mapped on different tiles can only happen using messages sent over the NoC.

Our approach for realizing system adaptivity consists of deploying the processes of the application(s) modeled as PPNs over the NoC-based MPSoC and allowing their run-time remapping to adapt the system to the changing operating conditions such as variation in quality of service requirements, availability of resources, or power budget constraints. In particular, system adaptivity in our system is supported by using the dedicated middleware highlighted in the software stack in the left part of Figure 1.6 on page 20. For reasons of convenience, we copy Figure 1.6 on page 20 in Figure 4.1.

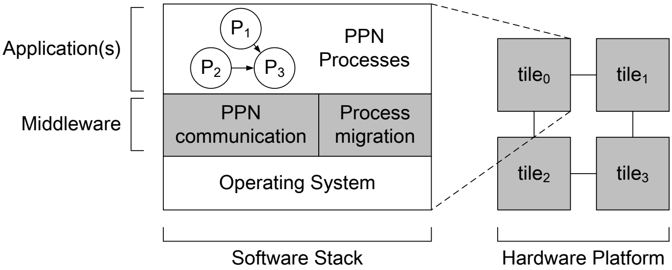


Figure 4.1: *Software stack (left) proposed to achieve adaptivity in BE systems. The middleware layer is denoted by the shaded area. The stack is deployed on each tile of the hardware platform (right).*

At the top of the software stack, applications are specified as a set of PPN processes, which are implemented as separate threads. An example of a thread representing a

PPN process is given in Figure 2.3(b) on page 29. However, in our work the basic structure of PPN processes will be modified to ease the realization of a predictable process migration mechanism, as described in Section 4.5.

At the bottom of the software stack in Figure 1.6 on page 20, the operating system (OS) is responsible for all kinds of process management (process creation, deletion, setting its priority, suspending or resuming it). These features are essential for the run-time management of the system, and in particular to perform process migrations. Moreover, each processor has multi-tasking capabilities thanks to the OS. In case of *many-to-one* mapping, i.e., when more than one process are mapped on the same processing element (PE), the scheduling is data-driven. This means that a process keeps executing successive iterations until it blocks in reading or writing (recall the PPN process structure of Figure 2.3(b) on page 29). When the process blocks, it yields the processor control to the next process in the ready queue in a round-robin fashion.

In between the applications and the operating system, in this thesis we propose the middleware which is highlighted in Figure 1.6 on page 20, which comprises two main components. The first one is *PPN communication*, which realizes the communication and synchronization between processes located in separate tiles, according to the PPN semantics. This middleware component has been already described in Chapter 3. The second component is *Process migration*, which is mainly responsible for the following activities, performed during the migration of processes:

- Coordinates the creation and deletion of processes among different tiles;
- Guarantees the correct transfer of the process' state during process migration.

The second component of our proposed middleware is the main subject of this chapter.

4.5 Process Migration

This section details the proposed mechanism to perform migrations of PPN processes over NoC-based MPSoC systems. It is a fundamental part of the middleware depicted in Figure 1.6 on page 20 because it defines how to perform run-time remapping of processes. This, in turn, allows designers to implement system adaptivity strategies.

The migration mechanism depends on the considered communication approach. As a starting assumption to devise the migration mechanism, we consider the *request-driven (R)* communication approach described in Section 3.4.3. This choice is made because the *R* approach leads to a considerably easier implementation of the migration mechanism since it requires less synchronization points. At the same time, it gives performance comparable to the other approaches for computation-dominant applications, as shown in Section 3.6.1.

We recall that to take into account the run-time remapping of processes over the NoC, each PE stores in its local memory a *middleware table* which is used during the conversion of the generic PPN communication primitives (i.e., *READ, WRITE* in Figure 2.3(b) on page 29) to the corresponding hardware platform primitives, such that the messages are sent to the right tiles in the system. A simple diagram showing the migration of a PPN process is depicted in Figure 4.2. An example of a middleware table generated for the initial mapping in Figure 4.2 is given in Table 4.1. For each

Table 4.1: *Middleware table example*

ch	prod(ch), cons(ch)	map(*prod(ch)*), map(*cons(ch)*)
1	P_1, P_2	$tile_0, tile_1$
2	P_2, P_3	$tile_1, tile_2$

channel of the PPN, the table lists which processes are the producer and consumer of that channel, together with the current mapping of producer and consumer processes in the system. Auxiliary information, for instance requests that are pending when a process migration is triggered, is also saved for each channel.

Mainly two kinds of process migration mechanism are considered in the literature, namely *process replication* and *process recreation*. In process replication, the program code of a migratable process is copied in each tile that may execute it, thereby creating replicas of the process. When a process needs to be migrated from one tile to another, the process is suspended on the first tile and restarted on the second. The state of the process must be copied from the first tile to the second because the process cannot be just restarted from scratch.

The second kind of process migration mechanism is based on the so-called *process recreation*. In this case, if a migration is needed, the process is killed on the initial tile (it runs) and created on another tile by moving both the process code and state. The OS/middleware in this case must support dynamic loading of processes to processors. This way, only one instance of the process code exist at a given time in the system.

The process replication mechanism is less efficient in terms of memory usage, compared to process recreation. Yet, it offers significant advantages such as easier implementation and faster migration procedure. Thus, for our proposed process migration mechanism we choose process replication because we aim at guaranteeing a quick completion of the migration procedure. Moreover, the memory constraint in our system is not critical.

Consider again the simple diagram in Figure 4.2, which shows the migration of a PPN process. Even though it is a simple example, it can be easily generalized for more complex PPN topologies. The diagram highlights the tiles involved in the process migration procedure, which are referred to as:

- the **source tile**, namely the tile which runs the process before the migration takes place;
- the **destination tile**, which is the tile that will execute the process after the migration;
- the **predecessor tile(s)**, which run(s) the predecessor process(es);
- the **successor tile(s)**, which execute(s) the successor process(es).

The structure of PPN processes, modified to allow migration at any point during the execution of the process main bodies, and the proposed process migration mechanism are presented in the following two subsections.

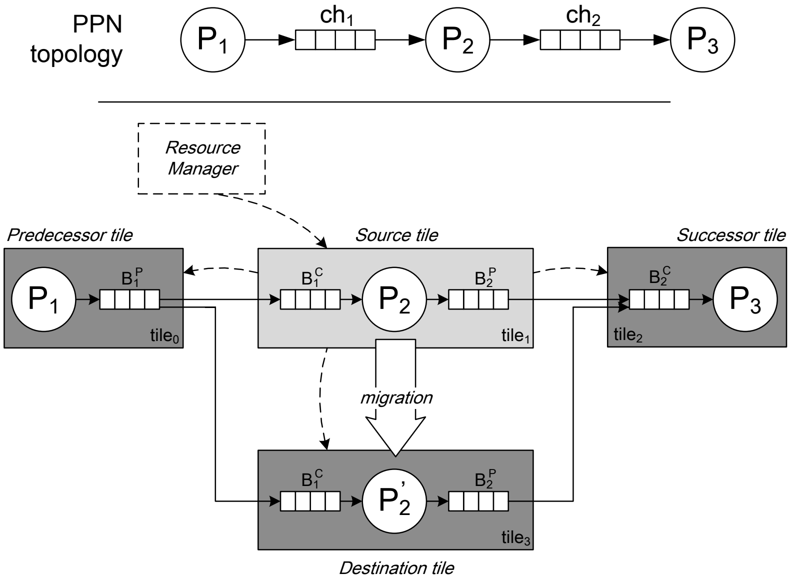


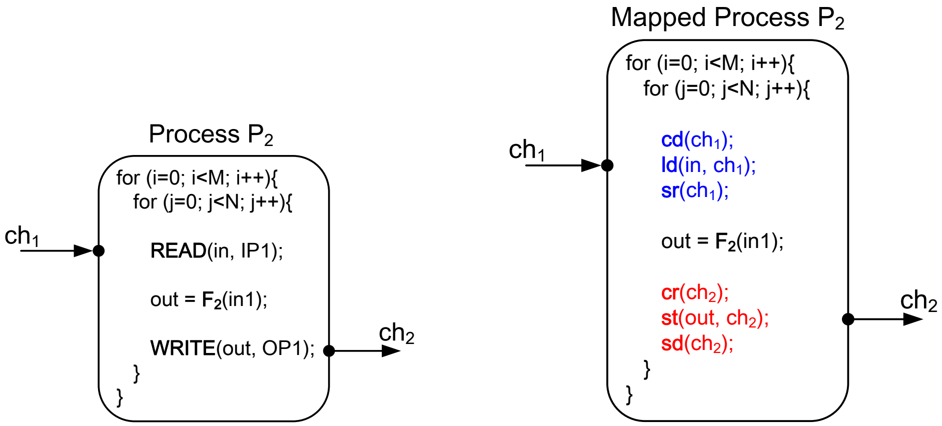
Figure 4.2: *Example of a migration procedure. The PPN topology considered in this example is shown in the top part of the figure. The initial mapping of this PPN is the following: process P_1 on tile$_0$, P_2 on tile$_1$, P_3 on tile tile$_2$. The resource manager (denoted by a dashed box) triggers the migration of P_2 from tile$_1$ to tile$_3$ by sending a specific control message to tile$_1$. This control message is forwarded by tile$_1$ to all the tiles involved in the migration. Control messages are represented by dashed arrows. To perform the actual migration, a few more steps are required. First, process P_2 is suspended on tile$_1$ and its iteration vector is transferred to tile$_3$. Second, the state of the input and output channels of P_2 on tile$_1$ (the content of B_1^C and B_2^P) are also moved to tile$_3$. Finally, the migration procedure is completed by starting the replica of P_2 on tile$_3$. This replicas is denoted by P_2'.*

4.5.1 Migratable PPN process structure

Our goal is to allow the migration to occur at any time during the execution of the process main body, which means that a migration can happen during most of the execution time of a process, as will be explained later in this section. In turn, this improves the latency incurred from the moment that a migration is triggered to its completion. To this end, we extend the NI of a tile with the ability to generate an interrupt for the processing element when a message with a specific tag is received. This extension is made because the detection of migration commands by polling at specific migration points in the code may cause undesired latency in the migration procedure.

In the scenario depicted in Figure 4.2, process P_2 is migrated from tile$_1$ to tile$_3$. The original structure of the code of P_2, as generated by the `pn` compiler, is reported in Figure 4.3(a). This structure of process P_2 hides all the details of the actual mapping of P_2 onto the execution platform.

In particular, to perform such mapping, the PPN communication primitives of P_2



(a) Structure of PPN process P_2. (b) Basic code structure of the mapped process P_2.

Figure 4.3: *Sub-figure (a) shows the structure of PPN process* P_2 *in Figure 4.2. Sub-figure (b) depicts the basic code structure used to map* P_2 *onto the considered execution platform. Notice that PPN communication primitives have been refined into several execution platform primitives.*

must be converted to (a set of) communication primitives of the execution platform. This conversion is a problem already studied in the literature [LvdWD01]. Typical communication and synchronization primitives of an execution platform are the following[3]:

- Check Data (**cd**): Checks if there are available data tokens in the considered FIFO buffer. Otherwise, it stalls the calling process.
- Check Room (**cr**): Checks if there is available space in the considered FIFO buffer. Otherwise, it stalls the calling process.
- Load Data (**ld**): Transfers a token from the considered FIFO buffer to the local space of the process.
- Store Data (**st**): Transfers a token from the local space of the process to the considered FIFO buffer.
- Signal Room (**sr**): After a *ld* operation, it signals that (additional) room is available in the considered FIFO buffer.
- Signal Data (**sd**): After a *st* operation, it signals that (additional) data is available in the considered FIFO buffer.

To derive an efficient mapping of PPN processes to an execution platform, designers have to consider the structure of the execution platform itself. For instance, it is fundamental to know whether the FIFO buffers which implement the channels of a certain process are located in a shared memory or in the local memory of the PE that executes that process.

In our approach, as shown in Figure 4.2, each process can only access the local memory of its tile. The transfer of tokens among tiles of our execution platform is

---

[3] Using the notations of [LvdWD01].

handled by the request-driven middleware approach, as mentioned earlier. Therefore, each process reads and writes tokens only from/to its local memory. In this scenario, the *READ* and *WRITE* PPN communication primitives are typically converted to the aforementioned primitives of the execution platform in the following way.

$$\text{READ} \implies cd \rightarrow ld \rightarrow sr \tag{4.1}$$

$$\text{WRITE} \implies cr \rightarrow st \rightarrow sd \tag{4.2}$$

Using the above conversion of communication primitives, we derive the implementation of PPN process P_2 onto the considered execution platform. The structure of such implementation is shown in Figure 4.3(b). We will refer to the structure shown in Figure 4.3(b) as *basic mapped process structure*. Since we require that migration may happen at any point within the execution of the processes main body, a modification of the process structure is required. In the rest of this section, we will explain why this modification is required and in what it consists.

In order to maintain correct functionality of the application, the state of the whole PPN must be consistent before and after a process migration has occurred. We divide the state of the PPN in two components, as follows:

1. State of PPN processes. As explained in Section 2.1.3, the only internal state of a PPN process is its iteration vector $\vec{I}$, which represents the value of the *for*-loop iterator variables.
2. State of PPN channels. The state of a channel in the PPN is represented by the tokens which are currently stored in the FIFO buffers which implement that communication channel.

Regarding the second component of the PPN state listed above, note that a PPN channel *ch* is actually implemented by two FIFO buffers in the request-driven communication approach which is considered in this chapter. One of these buffers reside on the tile on which the producer of *ch* is mapped, whereas the other resides on the tile in which the consumer of *ch* is mapped (recall Figure 3.4 on page 56). Therefore, when migrating a process P from its source tile to its destination tile, two components of the PPN state have to be migrated:

ST1: The iterator vector of P. For instance, the iterator vector of the process depicted in Figure 4.4 is $\vec{I} = [i, j]$.

ST2: The state of the input and output channels of P residing on the source tile of P. This state is in fact represented by the content of the input and output FIFO buffers connected to P. For example, refer to Figure 4.2, which illustrates the migration of process P_2 from tile$_1$ to tile$_3$. If, when the migration is performed, FIFO buffers B_1^C and B_2^P on tile$_1$ contain tokens, their content has to be moved to the destination tile, tile$_3$, into the corresponding FIFO buffers.

Having defined the state that has to be transferred during a process migration, we comment and describe the migratable PPN process structure shown in Figure 4.4 in the following. We denote this migratable process as P_{mig}. When P_{mig} starts, in line 1 of Figure 4.4, it checks if the *migration* flag is set. If the checking is positive, then this means that a migration has been performed, so the process state is reloaded.

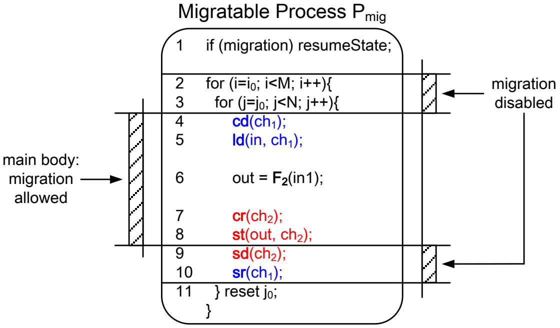


Figure 4.4: *Structure of migratable process P_{mig}. Compared to the basic mapped process structure of Figure 4.3(b), the order of the execution platform communication and synchronization primitives is changed. This allows migrations to be performed at any point within the main body of the P_{mig} (lines 4-8).*

Both state components listed above (*ST1, ST2*) are transferred from the source tile to the destination tile upon migration. If the migration flag is false, then this means that the process P_{mig} starts from scratch, with empty input and output FIFOs and $i_0 = j_0 = 0$.

Lines 2 and 3 differ from the basic mapped process structure in Figure 4.3(b) because the iterators inside the *for* loops do not start from zero in case of migration. Instead, they start from the values i_0 and j_0, which represent the iteration at which the process was interrupted by the migration while running on the source tile. After the first complete execution of the inner *for* loop, starting from j_0, the value of j_0 is set to zero in line 11 such that the next execution of the inner loop starts correctly with $j = 0$.

Moreover, the order in which communication and synchronization primitives are executed in P_{mig} differ from the one used in the basic mapped process structure of Figure 4.3(b). In fact, the execution platform primitives that implement the PPN *READ* primitive of P_{mig} (i.e., *cd*, *ld*, and *sr*) are not executed in a continuous sequence. They are, instead, executed in lines 4, 5, and 10, respectively, with several other operations occurring between *ld* and *sr*.

The reordering of execution platform communication primitives has already been studied in [LvdWD01]. The work in [LvdWD01] defines rules by which the reordering preserves the correctness of the execution of the mapped PPN process. The migratable process structure P_{mig} in Figure 4.4 complies with the rules defined in [LvdWD01].

Recall that the actual *release* of memory locations is performed by the *sr* operation, which consumes the data token from the FIFO by increasing the read pointer. This operation takes place only outside the main body of P_{mig}, in line 10. Then, if a migration is triggered before the *sr* operation, P_{mig} can be correctly resumed on the destination tile since it will read again the same input token, because the read pointer is unchanged. Similarly, the *sd* operation that concludes the *WRITE* primitive is executed at the end of the mapped PPN process, outside its main body. Finalizing the

READ and *WRITE* operations at the end of an iteration allows the process migration to happen anywhere within lines 4-8 correctly. Note that, in case of multiple input or output channels, the *sd* and *sr* operations of all channels are performed together right after the main body of the process, in order to update the state of P_{mig} and of the FIFO buffers in the shortest possible time.

Process migration cannot happen in the migratable process P_{mig} within the lines 9-11 and 2-3 because that will cause an inconsistency in the state of the PPN. This is because lines 9 and 10 can be considered as an update of the output and input FIFOs state, while lines 11, 2 and 3 represent an update of the state of P_{mig}. If, for instance, a migration happens after the FIFO state update but before the iterator set update, the following scenario will occur: (i) the state of the input and output FIFOs connected to P_{mig} are modified as if the current iteration was successfully completed; (ii) P_{mig} restarts the current iteration from the beginning, because the iteration vector was not updated accordingly. This condition will certainly cause a deadlock. Although the process migration cannot happen within lines 2-3 and 9-11, note that these sections represent a minimal part of the process execution, because performing the *sd* and *sr* operations and updating the iterator set is a matter of a few simple instructions. Therefore, disabling the migration within these sections does not increase the migration latency significantly.

The principle behind the proposed migratable process structure is that the state of the PPN must be *consistent* and *up-to-date* when a migration is performed. This allows the PPN to correctly resume its execution, with the migrated process mapped on the destination tile. Leveraging the PPN process structure, our approach does not require the designer to specify the context that has to be transferred upon migration as in [BABP06]. This burden is neither moved to the OS/middleware level as in [AACP08]. Determining the state to be migrated is not needed because the PPN state simply consist of the two components (*ST1*, *ST2*) described above. Moreover, our approach does not need designer-generated checkpoints/migration points. The resource manager in Figure 4.2 can interrupt the process execution at any time during the execution of the process main body. The migrated process will then resume its execution from the beginning of the interrupted iteration. On the one hand, this implies that if the migration is triggered in the middle of the function execution, the time spent in computation since the start of the iteration is lost. On the other hand, this approach leads to a more efficient implementation and predictable migration response time, which we consider more important for our goals.

4.5.2 Process migration mechanism

The migration mechanism requires actions from all the tiles depicted in Figure 4.2. Note that, in the figure, a *resource manager* is in charge of taking the migration decision. How the resource manager makes this decision is out of the scope of this thesis because we focus only on the process migration mechanism itself. Our contributions are in fact complementary to other research works (see [DKF11, AK09, LKwP^{+}10, Gab09, SSHT06]) which provide techniques to determine if a process migration is necessary and/or beneficial and, in that case, the actual destination tile of the process that has to be migrated.

When a migration decision is taken by the resource manager, it initiates the migration by sending a specific control message to the source tile. The source tile then forwards this control message to the destination, predecessor and successor tiles to inform them that the migration procedure has been initiated.

The control messages which notify the involved tiles for the start of the process migration contain the ID of the migrated process and the new mapping of that process. On all of the involved tiles, and on the resource manager, the middleware tables are then updated taking into account the new mapping of the migrated process.

For each of the tiles involved in the migration procedure, the detailed list of required actions are explained below.

Actions on the source tile

The behavior of the source tile depends on whether the tile is functional or faulty.

- In case the source tile is **functional**, the migrating process is stopped on the PE of the tile and the two state components, *ST1* and *ST2* mentioned in Section 4.5.1, are moved to the destination tile. These state components are transferred by means of dedicated messages sent over the NoC. Moreover, the middleware table is updated as described above. The source tile takes also care of propagating the migration decision to the other tiles involved in the migration procedure. This propagation is depicted by the dashed arrows in Figure 4.2.
- In case the source tile is **faulty**, the actions described in the previous point are emulated by a dedicated hardware IP, as proposed in [DCT$^+$13].

Actions on the destination tile

The destination tile receives a specific message for process activation. The migration procedure is handled by creating the required software FIFOs and by activating the replica of the migrated process using the corresponding OS call. Before the process replica is started, the *migration* flag is set to 1 so that the state of the migrated process is resumed (see line 1 in Figure 4.4). This implies that the input and output FIFOs connected to the migrated process are copied, and the iterator set (in the figure, i_0 and j_0) are set such that the execution starts from where it was suspended on the source tile. The middleware table is also updated in the way described above.

Actions on predecessor tile(s)

On these tiles, the only required step is the update of the middleware tables according to the new mapping of the migrated process. This way, new tokens meant for the migrated PPN process will be sent to the destination tile.

A corner case of the communication between the migrated process and its predecessor processes may happen when the migrating process has sent a *request* for new tokens just before the migration command arrives. For instance, it may happen that process P_2 in Figure 4.2 has sent a request for tokens to P_1 just before receiving the migration command from the resource manager. If that request has been served, before the migration command reached the predecessor tile, it means that new tokens are either traversing the NoC or they are already stored in the source tile. The predecessor tile in this case has to send another interrupt-generating message to the source tile, in order to force the forwarding of these data tokens to the destination tile.

Actions on successor tile(s)

Similarly, the successor tiles have to update their middleware tables so that successors of the migrating process will send new requests for data tokens to the destination tile. A particular case in the protocol between successor processes and the migrated process is represented by requests which are sent to the source tile just before the migration command arrives at the source tile. Each successor process checks if its requests have been served before the arrival of the migration command. If this is not the case, the successor tile has to send an interrupt-generating message to the source tile, in order to force the redirection of requests from the source tile to the destination tile.

4.6 Experiments and Results

In this section, we assess the benefits and overhead of the process migration mechanism proposed in this chapter. We perform our experiments on the same hardware platform setup used in Chapter 3, which is described in Section 3.5.3.

The setup of this experiment is shown in the left part of Figure 4.6. We use as a case study the M-JPEG application described in Section 3.5.2. $Tile_1$ initially runs all M-JPEG processes, which are listed in Figure 3.15, in a sequential way. P_1 is derived by merging *initVideoIn* and *videoIn* processes, P_2 and P_3 represent respectively the *DCT* and *Q* processes, and P_4 is obtained by merging the *VLE* and *videoOut* processes. We use the M-JPEG application as a case study because, compared to the Sobel application, M-JPEG processes are coarse-grained with high computation/communication ratio and therefore M-JPEG represents better the kind of applications which are likely to be mapped on a NoC-based MPSoC. The scheduling of the M-JPEG processes on $Tile_1$ before the migration is represented in Figure 4.5. Scheduling charts have been obtained using the GRASP [HvdHBL10] trace visualization tool to plot the information gathered at run-time. The trace shows the periodic scheduling which is obtained when all the processes are mapped on one tile and the scheduling policy is round-robin with yielding (yielding occurs when a process is blocked on reading or writing). The buffer size of each FIFO channel is set to two tokens in this experiment. In this scenario, the process scheduling iterates in the following way. First, P_1 executes two times, until it blocks on writing because its output buffer is full. Then P_2 is scheduled. It completes two iterations, consuming the tokens created by P_1 and producing two tokens for P_3. It then blocks while reading its input FIFO which is empty by then. Similarly, P_3 and P_4 execute twice before blocking on read. This scheduling repeats until the end of the application execution if no migration is performed.

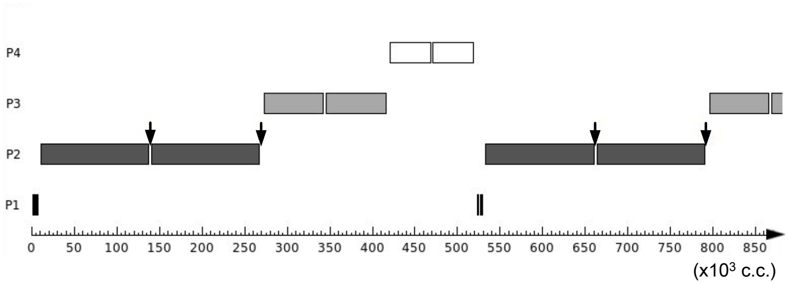


Figure 4.5: *M-JPEG process scheduling when running on a single tile. The scheduling policy is round-robin, with yielding when a process is blocked on reading or writing. The buffer size of each FIFO channel is two. These conditions lead to the periodic schedule (P_1, P_1, P_2, P_2, P_3, P_3, P_4, P_4), which continues indefinitely until the end of the application if no migration is performed.*

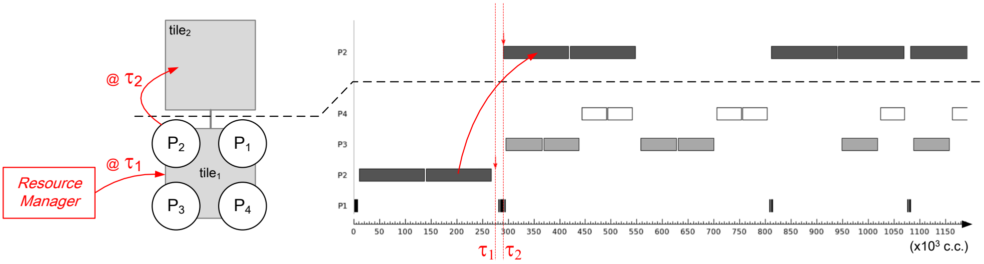


Figure 4.6: *M-JPEG process scheduling while migrating P_2 using the proposed migration mechanism. Until time τ_1, all processes are mapped on tile$_1$. At time τ_1, the resource manager requires process P_2 to migrate from tile$_1$ to tile$_2$. By using our interrupt-driven migration mechanism, the migration request is handled promptly. Process P_2 can be restarted on tile$_2$ within a predictable amount of time, in this case represented by the time interval ($\tau_2 - \tau_1$).*

4.6.1 Process migration benefits and overhead

System adaptivity requires the ability to change the process mapping at runtime in a predictable and efficient way. To illustrate the benefits of our migration approach presented in Section 4.5, we compare our proposed migration mechanism, driven by interrupt-generating control messages, with a migration approach based on fixed migration points.

In the latter case, process migration can take place only at fixed points in the code.

For instance, referring again to Figure 4.5, the arrows over the bars of process P_2 represent the start of an iteration of that process (for the sake of clarity, see line 4 in Figure 4.4). Assume that these points correspond to migration points, namely where the process checks if migration-messages have been sent by the resource manager. Given that the migration request can reach $Tile_1$ at any time, the latency of the actual process migration can vary. In the best case, the migration request reaches the tile right before the migration check point. In the worst case, the migration request arrives just after the migration check point, for instance the one which is reached around clock cycle 275,000 in Figure 4.5. The actual migration would not take place until the next migration check point of P_2, which happens to be after 2 executions of P_3, P_4 and P_1, and one execution of P_2. In this simple case, an upper bound of the process migration response time can be found, based on the process scheduling, which in turn depends on the workload of processes, the buffer sizes and the scheduling policy. In more complex cases, where the scheduling on one tile is affected by the scheduling on other tiles because of data dependencies, even finding an upper bound for the response time practically would not be possible.

By contrast, the interrupt-driven migration mechanism that we propose in Section 4.5 has a predictable behavior. As shown in Figure 4.6, the system has a faster response time to migration requests. At time τ_1, which is the worst case for the fixed point migration strategy discussed above, the resource manager sends a control message which triggers the migration of process P_2 to $Tile_2$. The process can be restarted on the destination tile within a predictable amount of time represented by the difference $(\tau_2 - \tau_1)$. This is the time it takes the source tile and the destination tile to execute the steps described in Section 4.5.2, such as the movement of the iteration vector of P_2 and the content of the FIFO connected to P_2, followed by the activation of P_2 on the destination tile. This migration overhead in time, $(\tau_2 - \tau_1)$, as shown in Figure 4.6, is much smaller than a single execution of the DCT function in process P_2. The migration procedure in this example actually takes less than 12% of a single execution of the DCT process.

Note that an upper bound of the migration procedure overhead can be derived for guaranteed throughput (GT) NoCs. In fact, the migration duration T_{mig} of a process $P \in \mathcal{P}$ can be split in two main components:

$$T_{mig}(P) = T_{stateMig}(stateSize(P)) + T_{procAct} \tag{4.3}$$

$T_{procAct}$ is a constant value which represents the time required to activate the migrated process using OS system calls, to update the middleware table, and complete all the actions described in Section 4.5.2 on the destination tile. $T_{stateMig}$ is the time it takes to transfer the state from the source to the destination tile. Its worst case, for GT NoCs, depends only on the state size. The largest state size occurs when both the input and output FIFO buffers connected to the migrating process P are full. This worst-case value can then be derived from the PPN topology and buffer sizes:

$$\max(stateSize(P)) = \sum_{ch \in IOC_P} size(B(ch)) \tag{4.4}$$

where $IOC_P = IC_P \cup OC_P$ as defined in Section 2.1.3, $size(B(ch))$ is the size of the buffer which represents the channel ch on the source tile. The value $size(B(ch))$ is obtained by multiplying the number of tokens of $B(ch)$ by the token size of a channel ch. An upper bound of the migration time T_{mig} of a process P can be calculated using $\max(stateSize(P))$ in Equation (4.3).

Our interrupt-driven migration mechanism incurs the worst-case overhead when a migration request arrives just before the end of a function execution in a process that has to be migrated. In this case, the migration still takes place in a predictable amount of time but the process execution has to roll back to the beginning of the interrupted iteration. In this scenario, all the time spent in the function execution is wasted.

4.7 Discussion

From the experimental results and analysis provided in Section 4.6.1 we can conclude that the migration mechanism proposed in this chapter complies with the requirements set in the problem statement of Section 4.1. In particular, our proposed migration mechanism possess the following properties:

- It is *predictable*, that is, when a migration is triggered, it will be completed within a certain time frame given by Equation (4.3);
- Thanks to the modified code structure of PPN processes proposed in Section 4.5.1, a migration can be triggered at any time during the process main body.
- Referring again to Section 4.5.1, the code needed to allow the proposed migration mechanism can be generated in a completely automated way.

Finally, note that the experimental results of Section 4.6.1 show that our proposed process migration mechanism is efficient. In fact, the overhead incurred to complete a process migration is experimentally shown to be negligible compared to the overall execution time of the application.

Chapter 5

Semi-partitioned Scheduling of CSDF-modeled Streaming Applications

Most of the work presented in this chapter has been published in [CBS14].

THIS chapter and the following one, Chapter 6, present two methodologies that exploit semi-partitioned scheduling algorithms, in the context of **hard real-time streaming systems**, using the scheduling analysis proposed in [BS11, BS12] as a basis and research driver. In particular, the semi-partitioned approach proposed in this chapter is aimed at reducing the number of processors required to schedule those applications which incur bin-packing issues under the partitioned scheduling approach of [BS11, BS12]. We recall that by bin-packing issues we mean that those applications require, using a partitioned scheduling approach, more processors than the minimum achievable by a global optimal scheduler.

In order to clarify the contributions of this chapter, we depict in Figure 5.1 the scheduling framework proposed in [BS11, BS12]. As input to this framework, the designer provides the application model, in the form of a (C)SDF graph with N actors (see *Analysis Model* in Figure 5.1). Then, Step ① converts the N actors of the input application into N periodic real-time tasks and derives the minimum size of the buffers which implement inter-task communication. Throughout this chapter, we refer to this conversion as *scheduling analysis of [BS11, BS12]*. This conversion is described in Section 2.3, and assumes that a hard real-time (HRT) scheduler will be used to execute the derived task set. In particular, Step ① derives:

- $\boxed{\text{I}}$ The worst-case execution time (WCET) C_i of each task, using Equation (2.26) on page 43.
- $\boxed{\text{II}}$ The period T_i of each task, using Equation (2.27) on page 43.

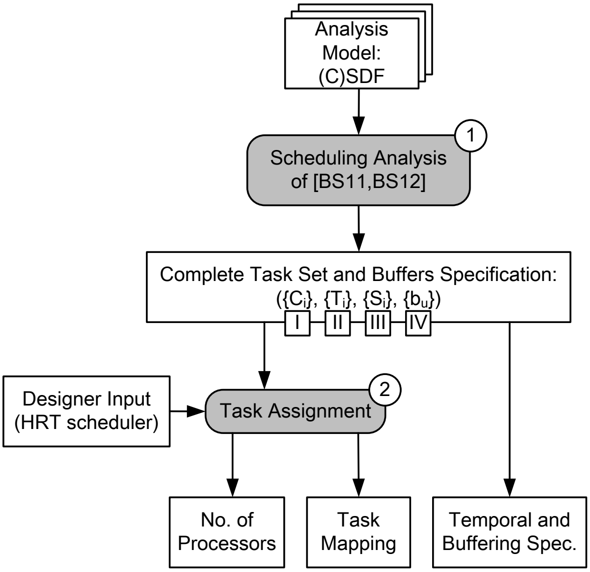


Figure 5.1: *Scheduling framework proposed in [BS11,BS12]. Step ① of the framework converts the input application, modeled as a CSDF graph, to a corresponding set of real-time periodic tasks. The obtained real-time periodic task set is completely specified, i.e., the WCET* $\boxed{I}$*, period* $\boxed{II}$*, and start time* $\boxed{III}$ *of each task are known, together with the required size of buffers* $\boxed{IV}$ *through which the tasks communicate. Then, based on the WCET and period of tasks, and on the designer's choice of HRT scheduling algorithm, Step ② derives the required number of processors and the assignment of task to processors. At this point, the system is completely specified.*

$\boxed{III}$ The start time S_i of each task, using Equation (2.29) on page 44.

$\boxed{IV}$ The size b_u of each buffer, which implements the communication over edge $e_u = (A_i, A_j)$. This size is obtained using Equation (2.31) on page 46.

The next step, Step ② (*Task Assignment*), derives the minimum number of processors required to schedule the application and the assignment of tasks of the application to processors, using the HRT partitioned approach (see Section 2.2.5) chosen by the designer. The assignment is based on the WCET and period of each task (parameters $\boxed{I}$ and $\boxed{II}$ above) derived in Step ①.

At the end of Step ②, the system is completely specified. The system specification consists of the following components.

- **Number of Processors.** It represents the number of processors required to schedule the application (obtained in Step ②).
- **Task Mapping.** It describes the assignment of application's tasks to processors (obtained in Step ②).
- **Temporal and Buffering Specification.** It describes the parameters of the task set, together with the size of buffers which implement inter-task data communication (all of which are obtained in Step ①).

5.1 Proposed Extension of the Scheduling Framework of [BS11, BS12]

So far, the scheduling framework of [BS11, BS12] (see Figure 5.1) considers only **hard real-time *partitioned* scheduling algorithms**. *Partitioned* scheduling algorithms incur neither migration overhead nor memory overhead because each task is statically allocated to a single processor. However, these algorithms are affected by *bin-packing issues* [Joh73], as described in Section 2.2.5. That is, if no limit on the maximum task utilization of a task set is imposed, partitioned algorithms may require twice as many processors compared to an optimal global scheduler [LDG04]. Therefore, for some applications, the number of processors required by partitioned approaches is larger than the number of processors required by optimal global schedulers.

However, also optimal global schedulers present drawbacks. Recall, from Section 2.2.5, that under optimal *global* scheduling algorithms all the tasks can migrate among all the processors. Such algorithms can fully exploit the available computational resources (refer again to Section 2.2.5, and in particular to Expression (2.13) on page 34). However, their optimality comes at the cost of high scheduling overheads due to excessive task preemptions and migrations. Moreover, modern MPSoCs typically have distributed memories in order to avoid the unpredictability of accessing shared resources. Therefore, using a global scheduler on such distributed memory systems implies that the code of each task has to be replicated[1] on all the processors, incurring a large memory overhead.

Semi-partitioned algorithms represent a middle ground between global and partitioned scheduling algorithms. In fact, under semi-partitioned approaches, most of the tasks are statically allocated to processors, and only a subset of the tasks is allowed to migrate among different processors. Migrating tasks follow a migration pattern derived at design-time. By allowing a (usually small) subset of tasks to migrate, semi-partitioned scheduling algorithms can mitigate the bin-packing effects that affect partitioned approaches. As a result, semi-partitioned algorithms require less processors than partitioned algorithms to schedule certain task sets. At the same time, these algorithms do not incur large memory overheads and task migration/preemption overheads like global algorithms. For these reasons, **in this chapter we extend the scheduling framework of [BS11, BS12] in order to support a *semi-partitioned* approach**. In the next section we explain the reasons why, among the various semi-partitioned schedulers, our proposed approach uses EDF-fm [ABD08]. We recall that the name EDF-fm comes from the fact that the algorithm is based on EDF and allows tasks to be either *fixed* or *migrating*.

5.1.1 Choice of the EDF-fm Semi-partitioned Algorithm

Several semi-partitioned scheduling algorithms have been proposed [ABD08, DYGR10, GCS11, KY08, AT06]. These algorithms can be classified based on *when* migrations

---

[1] We assume task migration using code replication, as shown in Chapter 4, because in distributed memory MPSoC systems it guarantees faster completion of the migration procedure.

are allowed to occur. In *restricted-migration* approaches [ABD08, DYGR10, GCS11] migrations can happen at job boundaries only. In *unrestricted-migration* (or *portioned*) approaches, migrations can happen at any time during a job execution. We argue that the restricted-migration class of semi-partitioned schedulers is the most suitable for distributed memory MPSoCs. This is because migrating at job boundaries reduces the amount of data (state) to be transferred from one processor to the next. Moreover, if the task does not keep an internal state between two successive jobs (i.e., it corresponds to a *stateless* dataflow actor, as defined in Definition 2.3.6 on page 47), no state migration is needed.

Within the class of restricted-migration semi-partitioned approaches, EDF-fm [ABD08] is particularly suited to distributed memory systems because in that approach a migrating task is allowed to migrate only between two processors (contrary to [DYGR10, GCS11], in which migrating tasks may span among all the processors). This property reduces substantially the memory overhead caused by replicating the task code. In addition, EDF-fm uses a fast utilization-based schedulability test (contrary to [DYGR10, GCS11]), that can be easily executed at run-time for incoming applications. For the reasons explained above we employ EDF-fm in the work presented in this chapter.

5.1.2 Implications of Using EDF-fm

Although EDF-fm can have great benefits for distributed memory MPSoCs, it provides hard real-time guarantees only for migrating tasks and soft real-time (SRT) guarantees for fixed tasks. Recall that, by Definition 2.2.9 on page 38, this means that fixed tasks can miss their deadlines by a bounded value called *tardiness*. As a consequence, the scheduling analysis of [BS11, BS12] can not be used directly because it assumes that a hard real-time (HRT) scheduler will schedule the derived task set, such that all task deadlines are met. It follows that the scheduling framework depicted in Figure 5.1 has to be modified.

Although our proposed semi-partitioned approach uses a SRT scheduling algorithm, in this chapter we provide a technique which ensures that the input/output interfaces with the environment are not affected by the deadline misses which may occur to the tasks of the application. That is, we can provide HRT guarantees to the input and output interfaces of the application with the environment.

5.2 Problem Statement

The scheduling analysis of [BS11, BS12] shows that an application, modeled as an acyclic CSDF graph, can be scheduled using a **hard real-time partitioned scheduling algorithm** as a set of real-time periodic tasks. In this chapter, we investigate the applicability of the **soft real-time *semi-partitioned* scheduling algorithm EDF-fm** to the scheduling analysis of [BS11, BS12]. In order to do so, we need to modify that scheduling analysis in a way that soft real-time scheduling algorithms can be supported. Overall, our semi-partitioned approach is aimed at reducing the number

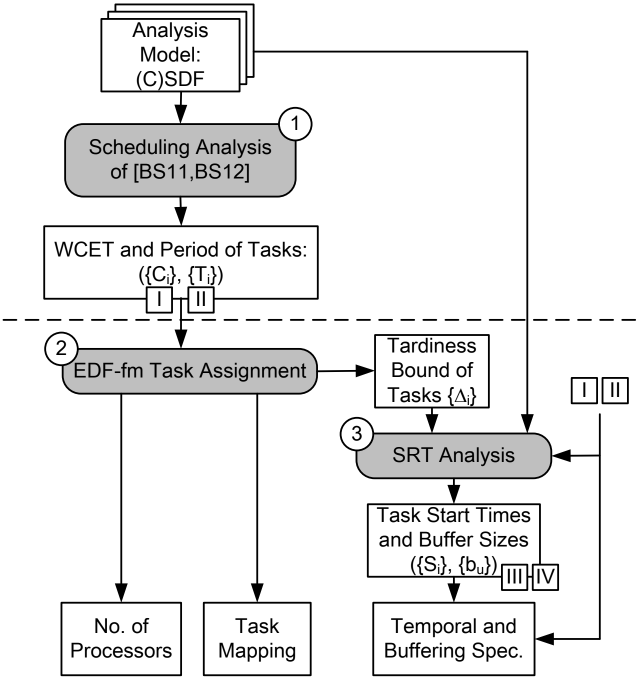


Figure 5.2: *Scheduling framework proposed in this chapter, which assumes that the SRT scheduling algorithm* EDF-fm *will schedule the derived task set, instead of the HRT partitioned schedulers considered in Figure 5.1. The part of the scheduling framework above the dashed line is identical to the scheduling framework in Figure 5.1. However, in Step ① only the WCET* $\boxed{I}$ *and period* $\boxed{II}$ *of each task are derived. This is because the start times* $\boxed{III}$ *of tasks and required size of buffers* $\boxed{IV}$ *through which the tasks communicate can only be derived when the tardiness bound* Δ_i *of each task* τ_i *is known. Step ② of the scheduling framework derives the minimum number of processors, the assignment of tasks to processors, and the tardiness bounds of tasks. Based on these tardiness bounds, Step ③ derives the task start times and buffer sizes. At this point, the system is completely specified.*

of processors required to schedule the applications that incur bin-packing issues under the partitioned scheduling approach of [BS11, BS12]. We recall that by *bin-packing issues* we mean that these applications require more processors under the partitioned scheduler compared to a global optimal scheduler.

5.3 Contributions

In order to address the problem stated in Section 5.2, we propose the scheduling framework shown in Figure 5.2. This scheduling framework is a modification of the one used in [BS11, BS12] and reported in Figure 5.1. The changes in the design flow are necessary in order to use the SRT semi-partitioned scheduler EDF-fm instead of a HRT partitioned approaches, as assumed in [BS11, BS12].

Consider Step ① of the design flow in Figure 5.1. That step converts the input application, modeled as a CSDF graph, into a set of real-time periodic tasks. In particular, it derives the complete specification of the tasks set (WCET $\boxed{\text{I}}$, period $\boxed{\text{II}}$, start time $\boxed{\text{III}}$ of each task) and the size of the buffers $\boxed{\text{IV}}$ which implement inter-task data dependencies. This analysis assumes that the derived periodic task set will be scheduled by a HRT scheduling algorithm, i.e., no task will miss any deadline. Therefore, as shown in Figure 5.1, the scheduling analysis of [BS11,BS12] can derive the complete *Temporal and Buffering Specification* of the system - composed of parameters $\boxed{\text{I}}$, $\boxed{\text{II}}$, $\boxed{\text{III}}$, and $\boxed{\text{IV}}$ - in a single step. In what follows, we will refer to such scheduling analysis, which assumes hard real-time scheduling, as *HRT approach*.

As a **first contribution** of this chapter, we show that this scheduling analysis can be extended to support SRT schedulers, once the tardiness bound[2] Δ_i of each task τ_i is known. In practice, the components of the *Temporal and Buffering Specification* of the system ($\boxed{\text{I}}$, $\boxed{\text{II}}$, $\boxed{\text{III}}$, and $\boxed{\text{IV}}$) cannot be derived in a single step, but in the following three steps of Figure 5.2.

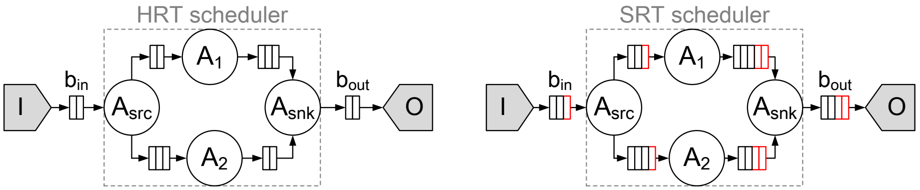
- Step ① derives the WCET and period of tasks (components $\boxed{\text{I}}$ and $\boxed{\text{II}}$), based on the input application model. These components are not affected by the tardiness of tasks.
- Step ② assigns the tasks to processors, according to the rules of the chosen SRT scheduling algorithm (in this case, EDF-fm). The outputs of this step are the required number of processor, the assignment of tasks to processors, and the tardiness bound Δ_i of each task τ_i.
- Step ③, given the value of Δ_i of each task τ_i, derives the earliest task start times and minimum buffer sizes ($\boxed{\text{III}}$ and $\boxed{\text{IV}}$ in Figure 5.2) that guarantee the existence of a valid schedule of the given application. Valid schedule means that, even in presence of task tardiness, tasks can be released periodically and neither buffer underflow nor overflow can occur.

In what follows, we will refer to the scheduling analysis which assumes a SRT scheduler as *SRT approach*. In this chapter, we show that the SRT approach achieves the same throughput of the HRT approach, albeit requiring larger buffers and increased application latency. In Figure 5.3 we compare the HRT and SRT approaches. The mentioned increase in the size of buffers is visualized in Figure 5.3(b) using red color. By appropriately increasing the size of buffers, the analysis presented in this chapter guarantees that the interfaces of the environment with the application (see I and O in Figure 5.3(b)) can execute in a strictly periodic way with neither underflow nor overflow on input and output buffers (see b_{in} and b_{out} in Figure 5.3(b)), also when SRT schedulers are used. This means that the **input/output interfaces are not affected by the deadline misses which may occur to the tasks of the application, i.e., I and O in Figure 5.3(b) can execute with HRT guarantees.**

Then, using the result of the first contribution, we focus on a specific SRT semi-partitioned scheduling algorithm, namely EDF-fm [ABD08]. As mentioned in Section 5.2, we consider EDF-fm instead of the partitioned approaches adopted in [BS11,BS12] because we want to reduce the number of processors required to schedule

---

[2] See Definition 2.2.10 on page 38.



(a) Analysis under HRT schedulers (e.g., [BS11, BS12])

(b) Analysis under SRT schedulers (our approach)

Figure 5.3: *Scheduling framework under HRT (a) and SRT (b) schedulers. Sub-figure (a) represents the scheduling analysis of [BS11, BS12] which considers only HRT schedulers. By contrast, sub-figure (b) depicts the scheduling analysis when SRT schedulers are used. This kind of schedulers allow tasks to miss their deadlines up to a certain value. As shown in this chapter, real-time guarantees can be still provided to the interfaces with the environment (denoted by* I *and* O*). However, the SRT approach requires larger size of buffers (as highlighted in red in sub-figure (b)).*

the applications that incur bin-packing issues under that partitioned approach. As a **second contribution** of this chapter, we propose a novel task assignment heuristic, called FFD-SP (First Fit Decreasing followed by semi-partitioning), that replaces the ones proposed in [ABD08] that are intended for independent task sets. FFD-SP is executed in Step ② in Figure 5.2. We propose this novel task assignment heuristic for the reason described in the following paragraph.

As shown in Figure 5.2, the derivation of task start times and buffer sizes in Step ③ depends on the value of the tardiness bounds of tasks. In general, the more tasks are affected by tardiness, the more task start times need to be postponed. This has a direct effect on the latency of the application. Moreover, as the number of tasks with tardiness increases, so does the size of buffers required to implement inter-task communication. To summarize, the number of tasks affected by tardiness has a direct impact on the overhead in application latency and buffer sizes of the SRT approach compared to the HRT approach. This effect is not considered by the task assignment heuristics proposed in [ABD08] because those heuristics are intended for independent tasks. Our proposed FFD-SP heuristic is aimed at reducing the number of required processors, compared to the HRT approach, while keeping a low buffer size and latency overhead when the EDF-fm algorithm is used.

Finally, as a **third contribution**, we show on a set of real-life benchmarks that our SRT approach can lead to significant benefits by reducing the number of processors required to schedule the applications that incur in bin-packing issues under the HRT approach of [BS11, BS12]. At the same time, both our SRT approach and HRT approaches achieve the same application throughput. However, our experiments show that the SRT approach incurs an increase in memory requirements and application latency. Therefore, our SRT semi-partitioned approach is especially appealing for systems in which the throughput constraint is more important than memory or latency constraints.

5.4 Related Work

To the best of our knowledge, real-time semi-partitioned scheduling algorithms have never been studied when mapping streaming applications with inter-task data dependencies to MPSoCs. In fact, existing semi-partitioned solutions [ABD08, DYGR10, GCS11, KY08, AT06] only consider sets of independent tasks. In the real-time community, however, techniques different from pure partitioning to assign data-dependent application tasks to a multiprocessor platform have already been devised. Existing approaches which are close to our work are [LA09] and [LA10] by Liu and Anderson. These approaches use a global scheduler which, similar to our case, satisfies soft real-time requirements. In particular, [LA09] describes a way to guarantee bounded tardiness of an application specified as a pipeline of tasks under a SRT global scheduler. A strong limitation in [LA09] is that only simple pipeline application topologies are handled, contrary to our approach that can handle more complex topologies like CSDF graphs. In [LA10], the same authors extend their analysis to guarantee bounded task tardiness in more complex application graph topologies, such as Processing Graph Method (PGM) graphs. However, the work in [LA10] does not address the calculation of minimum buffer sizes, which is an important metric to evaluate the practicability of the approach. In contrast, the calculation of buffer sizes is supported by our approach.

5.5 Soft Real-time Scheduling Analysis

In this section, we present the first main contribution of this chapter. Our contribution is based on the scheduling analysis of [BS11, BS12] (see Section 2.3), which converts a CSDF to a set of periodic tasks, assuming that a HRT scheduler is used to schedule the derived task set. We show that such scheduling analysis can be modified in a way that SRT schedulers can be used to execute the derived periodic task set. The SRT scheduler considered in this chapter is the EDF-fm algorithm, whose per-task tardiness bound is given by Equation (2.22) on page 40. Note, however, that the results obtained in this section are valid for any SRT scheduler which provides bounded task tardiness. Our solution extends the analysis of [BS11, BS12] by deriving new earliest start times for each task (Section 5.5.1) and minimum buffer sizes (Section 5.5.2) that can handle task tardiness and still allow a periodic release of each task. Within the design flow proposed in Figure 5.2, this derivation of task start times and buffer sizes is performed in Step ③.

5.5.1 Earliest Start Times in Presence of Tardiness

In order to derive the earliest start times in presence of tardiness, we leverage some concepts which are explained in Chapter 2 of this thesis. We summarize these concepts below.

- Under hard real-time scheduling of acyclic CSDF graphs (Section 2.3), when computing the earliest start times of actors we use the cumulative production/consumption functions defined in Definitions 2.3.1 and 2.3.2 on page 44, namely:

  - $\text{prd}^{S}{}_{[t_s,t_f)}(A_i, e_u)$, which represents the total number of tokens produced by actor S_i to edge e_u during the time interval $[t_s, t_f)$;
  - $\text{cns}^{S}{}_{[t_s,t_f]}(A_j, e_u)$, which represents the total number of tokens consumed by actor A_j from edge e_u during the time interval $[t_s, t_f]$.

- By Definition 2.2.10 on page 38, under a SRT scheduler, a task τ_i does not miss each of its deadlines by more than its tardiness bound Δ_i.

In what follows, we will use the concept of *as late as possible (ALAP) completion schedule* in case of tardiness, which is defined below.

Definition 5.5.1. *(ALAP completion schedule in case of tardiness).* The ALAP completion schedule considers that all invocations $A_{i,j}$ (jobs $\tau_{i,j}$) of an actor A_i (task τ_i) incur the maximum tardiness Δ_i, therefore complete at $z_{i,k} = d_{i,k} + \Delta_i$ (where $d_{i,k}$ represents the absolute deadline of job $\tau_{i,j}$, as defined in Section 2.2.1).

Then, consider that actor A_j has a data dependency from actor A_i through edge e_u. In addition, assume that both A_i and A_j may be affected by tardiness. In order to derive the earliest start time of actor A_j, Equation (2.30) on page 44 has to be modified in order to capture the worst-case schedule of A_i and A_j, as shown in the following proposition.

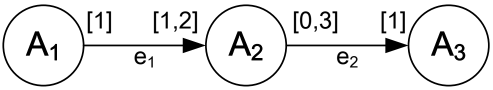
Proposition 5.5.1. *In presence of task tardiness, bounded by Δ_i for source actor A_i and by Δ_j for destination actor A_j, the earliest start time $S_{i\to j}$ of actor A_j due to its dependency from A_i through edge e_u, under a valid schedule, is given by:*

$$S_{i\to j} = \min_{t\in[0,S_i+\Delta_i+H]} \Big\{ t : \underset{[S_i+\Delta_i,\max(S_i+\Delta_i,t)+k)}{\text{prd}^{S}} (A_i, e_u) \geq \underset{[t,\max(S_i+\Delta_i,t)+k]}{\text{cns}^{S}} (A_j, e_u) \quad \forall k = 0, 1, \cdots, H \Big\} \tag{5.1}$$

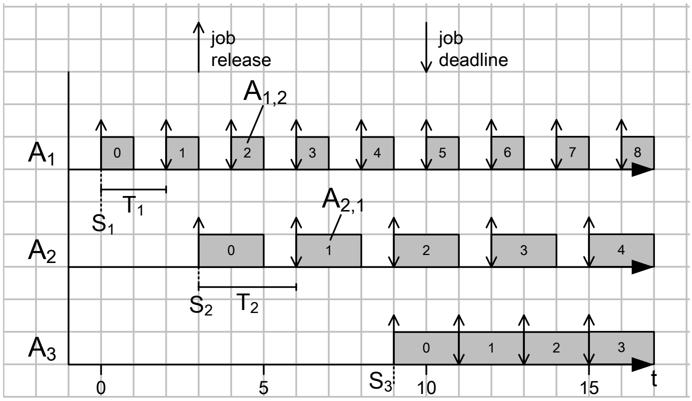
where H is the iteration defined by Equation (2.28).

Proof. If actors A_i and A_j may be affected by tardiness, Equation (2.30) on page 44 can not be applied in its original form to derive earliest actor start times. In order to illustrate this fact, we copy in Figure 5.4(a) the CSDF graph used as an example in Chapter 2. The corresponding hard real-time schedule, derived using the methodology of Section 2.3 in absence of tardiness, is shown in Figure 5.4(b). Note that in Figure 5.4(b) the start times of actors are calculate using Equation (2.29), which in turn exploits Equation (2.30).

Now, for instance, assume that actor A_1 may be affected by tardiness because it is scheduled by a SRT scheduler. Then, if invocation $A_{1,2}$ in Figure 5.4(b) completes later than its deadline, invocation $A_{2,1}$ of A_2 (that depends on the completion of $A_{1,2}$) cannot be released at time $t = 6$. It follows that the start time of actor A_2 has to be changed.



(a) Example of a CSDF graph (extracted from Section 2.1.2).



(b) Real-time periodic task set (extracted from Section 2.3)) obtained from the above CSDF graph.

Figure 5.4: *Example of the conversion of a set of CSDF actors to a real-time periodic task set using the methodology proposed in [BS11, BS12]. As explained in Section 2.3, the three actors of the CSDF graph depicted in sub-figure (a) can be scheduled as three real-time periodic tasks, as shown in sub-figure (b).*

The worst-case scenario of the execution of A_i and A_j to derive the earliest start time $S_{i \to j}$ in case of tardiness occurs when the source actor A_i completes its jobs *as late as possible*, i.e., according to its ALAP completion schedule (see Definition 5.5.1).

As shown in Figure 5.5, the ALAP completion schedule of actor A_i can be represented by a fictitious actor $\tilde{A}_i$, which has the same period as A_i, no tardiness, and start time $\tilde{S}_i = S_i + \Delta_i$. At run-time, any invocation of A_i, even if delayed by the maximum allowed tardiness Δ_i, will never complete later than the corresponding invocation of $\tilde{A}_i$. Notice that invocations $\tilde{A}_{i,k}$ of $\tilde{A}_i$ are strictly periodic, because they incur no tardiness.

By contrast, in the worst-case scenario to determine $S_{i \to j}$, A_j is executed as early as possible, so we assume that all invocations $A_{j,k}$ of A_j are not affected by tardiness. Then, the earliest start time that guarantees the absence of blocking of A_j in its execution, even for the worst-case production and consumption patterns of A_i and A_j, is found by evaluating Equation (2.30) with $\tilde{A}_i$ as source actor and A_j as destination actor. This scenario is captured by Equation (5.1). Note that any completion of an invocation $A_{i,k}$ of A_i earlier than its corresponding worst-case $\tilde{A}_{i,k}$ results in an earlier production of tokens, such that the inequality in Equation (5.1) still holds for all $k \in [0, 1, \cdots, H]$. Similarly, if any of the invocations of A_j is affected by tardiness, the token consumption is executed later and Equation (5.1) guarantees that enough

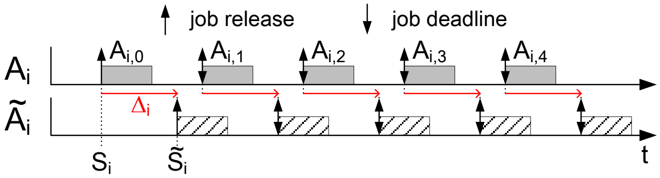


Figure 5.5: *Worst-case scheduling of source actor A_i, with tardiness Δ_i, when deriving the start time $S_{i\to j}$ of destination actor A_j. In the worst case, all invocations of actor A_i incur the maximum tardiness Δ_i. This schedule can be represented by a fictitious actor $\tilde{A}_i$, which has the same period as A_i, no tardiness, and start time $\tilde{S}_i = S_i + \Delta_i$.*

tokens will be available to be read. ■

Note also, from Equation (5.1), that the start time $S_{i\to j}$ of actor A_j due to its dependency from A_i is only affected by the tardiness bound Δ_i of the source actor. In addition, when actor A_j has several predecessors, the start time S_j has to be set to the maximum of the start times $S_{i\to j}$ given by Equation (5.1) considering each predecessor in isolation, as captured by Equation (2.29) on page 44.

5.5.2 Minimum Buffer Sizes in Presence of Tardiness

Similarly to Section 5.5.1, in order to derive minimum buffer sizes we also utilize the concept of tardiness bound under a SRT scheduler, defined in Definition 2.2.10 on page 38. In addition, we use the cumulative production and consumption functions of CSDF actors defined in Definitions 2.3.3 and 2.3.4 on page 46, namely:

- $\text{prd}^{\text{B}}{}_{[t_s,t_f]}(A_i,e_u)$, which represents the total number of tokens produced by actor A_i to edge e_u during the time interval $[t_s,t_f]$;
- $\text{cns}^{\text{B}}{}_{[t_s,t_f)}(A_j,e_u)$, which represents the total number of tokens consumed by actor A_j from edge e_u during the time interval $[t_s,t_f)$.

Then, similar to Section 5.5.1, consider that actor A_j has a data dependency from actor A_i through edge e_u, and both A_i and A_j may be affected by tardiness. Based on the above definitions, and given the actor start times calculated leveraging Proposition 5.5.1, the following proposition captures the worst-case scheduling of A_i and A_j when deriving the minimum buffer sizes in case of task tardiness.

Proposition 5.5.2. *In presence of task tardiness, bounded by Δ_i for source actor A_i and by Δ_j for destination actor A_j, the minimum buffer size b_u of a communication channel e_u connecting A_i and A_j, under a valid schedule, is given by:*

$$b_u(A_i,A_j) = \max_{k\in[0,1,\cdots,H]} \Big\{ \text{prd}^{\text{B}}_{[S_i,\max(S_i,S_j+\Delta_j)+k]}(A_i,e_u) - \text{cns}^{\text{B}}_{[S_j+\Delta_j,\max(S_i,S_j+\Delta_j)+k)}(A_j,e_u) \Big\} \quad (5.2)$$

Proof. To get the minimum buffer size in presence of task tardiness, we consider the worst-case scenario that would result in the maximum buffer requirement for channel e_u. This worst-case scenario occurs when: (i) all the invocations $A_{j,k}$ of the destination actor A_j complete with the maximum tardiness (i.e., A_j is executed according to its ALAP schedule, see Definition 5.5.1); (ii) none of the invocations of the source actor are affected by tardiness.

We can then prove Proposition 5.5.2 with a procedure similar to the one used in the proof of Proposition 5.5.1. We associate the worst-case completion of all the invocations $A_{j,k}$ to a fictitious actor $\tilde{A}_j$. Actor $\tilde{A}_j$ is strictly periodic, with no tardiness, constant period $\tilde{T}_j = T_j$ and start time $\tilde{S}_j = S_j + \Delta_j$. Then, the minimum buffer requirement of the communication channel e_u is found by evaluating Equation (2.31) on page 46 with A_i as source actor and $\tilde{A}_j$ as destination actor. This scenario is captured by Equation (5.2).

Note that any earlier completion of any of the iterations of A_j would not increase the buffer size requirement. This is because an earlier completion of A_j would results in an earlier consumption of tokens from channel e_u. Similarly, any delayed completion of an iteration of A_i would result in a delayed production of tokens to the considered channel. Thus, the derived value of b_u is sufficient. ■

Note that Equations (5.1) and (5.2) can also be used to analyze the interfaces between the external data provider and consumer (I and O in Figure 5.3(b)) and the input and output actors of the application (A_{in}, A_{out}). Compared to the HRT approach shown in Figure 5.3(a), in the SRT approach of Figure 5.3(b) A_{in} and/or A_{out} may experience tardiness. In this case, Equations (5.1) and (5.2) derive delayed start time of the external consumer O and larger buffer sizes of b_{in} and b_{out} such that both I and O can execute strictly periodically with neither buffer overflow nor underflow occurring on b_{in} and b_{out}.

5.6 FFD-SP Task Assignment Heuristic

The analysis provided in Section 5.5 extends the scheduling framework of [BS11, BS12] by calculating different task start times and buffer sizes, depending on the tardiness bounds of tasks. This way, the derived task set can be scheduled by any SRT scheduling algorithm. In this section, we present the second main contribution of this paper, which is focused on *a particular* SRT scheduling algorithm, namely EDF-fm [ABD08]. We recall that the most of the theoretical results regarding the EDF-fm algorithm are summarized in Section 2.2.7 of this thesis.

In our contribution, we propose a task assignment heuristic that does not follow the sequential approach common to all the heuristics proposed in [ABD08]. In fact, as explained in Section 2.2.7, the heuristics in [ABD08] assign tasks to processors in a sequential way, which means that in most cases processors have migrating tasks assigned to them. In turn, this makes most tasks in the system affected by tardiness. Actor tardiness imposes larger buffer sizes (according to Proposition 5.5.2) and postponed start times of successor actors (according to Proposition 5.5.1). Overall, this leads to larger memory requirements and increased application latency. Our proposed heuristic is executed in Step ② of the design flow in Figure 5.2.

In contrast to the heuristics in [ABD08], our proposed heuristic, called FFD-SP, starts to consider semi-partitioning only when the *First-fit Decreasing* (FFD) heuristic [Joh73] (see Section 2.2.6) fails to assign a certain task in the system. The proposed task assignment heuristic accepts as input the number of processors M onto which the task set Γ has to be assigned. Then, the assignment of tasks to processors in FFD-SP proceeds as follows. At first, the set of stateful actors Γ_s is constructed and tried to be assigned to the processors using FFD. Stateful actors are considered first in our heuristics because this way they are fixed to a processor and at run-time there is no need to migrate their state.

Then, FFD-SP tries to assign the remaining (stateless) actors using FFD. Only when FFD fails, semi-partitioning is considered. This way, the number of processors with migrating tasks is likely to be less. Recall from Section 2.2.7 that, under EDF-fm, on processors which runs migrating tasks, fixed tasks are affected by tardiness. Therefore, by reducing the number of processors with migrating tasks, FFD-SP tries to reduce the number of fixed tasks with tardiness. When a task τ has to be semi-partitioned, its utilization is divided into two shares, $s_1(\tau)$ and $s_2(\tau)$. In particular, FFD-SP tries to assign the largest share possible $s_1(\tau)$ from the remaining available utilization on processors; then, it tries to find the *best fit* for the remaining share $s_2(\tau)$, in order to leave larger "chunks" of processor available utilizations to remaining (unallocated) tasks.

Our proposed FFD-SP assignment heuristic is reported in Algorithm 1. As mentioned earlier, at first Algorithm 1 builds Γ_s, the set of stateful actors, which are then assigned using the FFD heuristic (lines 1-4).

Then, considering task $\tau \in (\Gamma - \Gamma_s)$, the algorithm tries to assign task τ to one of the processors using FFD (lines 6-8). If FFD does not succeed, the algorithm tries to divide the utilization of task τ in two shares, $s_1(\tau)$ and $s_2(\tau)$. Traversing the processor list in decreasing order of available utilization, a share $s_1(\tau) = 1 - \sigma(\pi')$ is tried to be mapped on processor π' (lines 9-12). In line 10, the term $\sigma(\pi')$ denotes the total utilization assigned to π'. For the sake of clarity, this notation is slightly different from the one given in Equation (2.20) on page 39. If the assignment of $s_1(\tau)$ is successful, the algorithm attempts to map share $s_2(\tau) = u(\tau) - s_1(\tau)$ by traversing the list of processors in increasing order of available utilization (lines 13-17).

Note that our FFD-SP heuristic may fail to assign tasks to the considered set of processors. In fact, at the first execution of Algorithm 1, the number of processors M is set to M_{OPT}, the number of processors required by an optimal scheduler (see Equation (2.14) on page 34). If the task set cannot be assigned to M processors, M is increased by one and Algorithm 1 is executed again until a successful assignment is found.

The algorithm makes use of the *sp_assign* function to try and assign task shares. As shown in Algorithm 2, this function checks three conditions (see line 1):

1. There must be enough available utilization on the processor to accommodate the share. Similarly to Algorithm 1, the term $\sigma(\pi)$ denotes the total utilization assigned to π.

Algorithm 1: FFD-SP task assignment heuristic.

```
Input: The number of processors $M$, a task set $\Gamma = \{\tau_1, \tau_2, \cdots, \tau_N\}$ of $N$ periodic tasks.
Result: True and an $M$-partition describing the task assignment onto $M$ processors if $\Gamma$ is
        schedulable, False otherwise.
Find $\Gamma_s = \{\tau : \tau \in \Gamma \wedge \tau \text{ is stateful}\}$;
Assign tasks in $\Gamma_s$ using FFD;
if $\Gamma_s$ cannot be assigned then
    return False;
for $\tau \in (\Gamma - \Gamma_s$, sorted in decreasing utilization$)$ do
    Try to assign task $\tau$ using First-Fit heuristic;
    if First-Fit is successful then
        continue;
    for $\pi' \in (\Pi$ sorted in decr. available utilization$)$ do
        $s_1(\tau) = 1 - \sigma(\pi')$;
        Assigned = False;
        if sp_assign($s_1(\tau), \pi'$)==True then
            $s_2(\tau) = u(\tau) - s_1(\tau)$;
            for $\pi'' \in (\Pi$ sorted in incr. available utilization$)$ do
                if sp_assign($s_2(\tau), \pi''$)==True then
                    Assigned = True;
                    break;
            if Assigned==False then
                Revert assignment of $s_1(\tau)$ to $\pi'$;
            else
                break;
    if Assigned==False then
        return False;
Optimize the obtained partition;
return True;
```

Algorithm 2: *sp_assign* function.

```
Input: The share $s$ of task $\tau$ to be assigned, a processor $\pi$.
Result: True if $s$ can be assigned to $\pi$, False otherwise.
if ($\sigma(\pi) + s \leq 1$) and ($\sigma^{mig}_{current}(\pi) + u(\tau) \leq 1$) and ($n_{mig\_tasks} < 2$) then
    Assign $s$ to $\pi$;
    return True;
else
    return False;
```

2. In case another migrating task has already been mapped on processor π, Condition (2.21) in Section 2.2.7 must be satisfied. Note that in Algorithm 2 the term $\sigma^{\text{mig}}_{\text{current}}(\pi)$ denotes the total utilization of migrating tasks assigned to π.
3. The number of migrating tasks $n_{\text{mig\_tasks}}$ assigned to processor π must be less than 2 (as required by the EDF-fm algorithm).

When an M-partition (see Definition 2.2.6 on page 35) has been successfully found, the FFD-SP heuristic tries to *optimize* it (line 24 in Algorithm 1). The optimization consists in re-assigning the migrating task shares, whenever possible, to processors to which less fixed tasks are assigned. This way, less fixed tasks are affected by tardiness, leading to lower application latency and buffer size requirements. Note that in Algorithm 1 the first share of a migrating task $s_1(\tau)$ is set to the largest possible value, given the current available utilization of processors. This in turns makes the second share of each migrating task $s_2(\tau)$ as small as possible, making the process of optimization of the partition more effective.

5.7 Evaluation

We evaluate our semi-partitioned scheduling approach using the StreamIt benchmarks considered in [ZBS13], for which we employ the unfolding technique described in [ZBS13] to derive larger CSDF graphs with improved throughput. Among these benchmarks, seven applications require, under the partitioned FFD allocation scheme, more processors than an optimal scheduler. This set of applications is listed in Table 5.1. In this section we compare the number of required processors, memory requirements, and application latencies obtained with three different allocation/scheduling approaches: (i) Partitioned EDF with FFD heuristic; (ii) Semi-partitioned EDF-fm, with our proposed FFD-SP heuristic; (iii) Semi-partitioned EDF-fm, with the LUF (Lowest Utilization First) heuristic proposed in [ABD08]. These approaches are denoted in Table 5.1 with *FFD*, *FFD-SP*, and *LUF*, respectively.

Note that *all* the approaches in Table 5.1 lead to the same application throughput. This is because the throughput of an application depends on the period of its sink actor, which is unchanged in our analysis even in presence of task tardiness. In addition, we choose to compare the results of the LUF heuristic with our FFD-SP heuristic because, among the heuristics proposed in [ABD08], LUF achieves the smallest number of processors.

The M_{OPT} column in Table 5.1 lists the number of processors required by an optimal scheduler (for instance [BCPV96]) to execute the considered applications. M_{OPT} is obtained using Equation (2.14).

Let us focus on the comparison between the partitioned approach (FFD, described in Section 2.2.6) and our proposed semi-partitioned approach (FFD-SP, proposed in Section 5.6). We note that the FFD approach results in a number of processors (M_{FFD}) which is on average 17.6% greater than the number required by an optimal scheduler (see column $\frac{M_{\text{FFD}}}{M_{\text{OPT}}}$). In contrast, our FFD-SP algorithm requires on average only 2.1% more processors (see column $\frac{M_{\text{SP}}}{M_{\text{OPT}}}$), while maintaining the same throughput. This means that our proposed approach can exploit the available processors more efficiently, getting significantly closer to the results obtained by optimal schedulers (see columns M_{SP} and M_{OPT}). However, this comes at two costs.

The **first cost** is the increase of memory requirements. For each benchmark, column mem_{FFD} reports the memory required by the partitioned approach, expressed in bytes.

Table 5.1: *Comparison of different allocation/scheduling approaches.*

Benchmark	**OPT**	**Partitioned (FFD)**				**Semi-partitioned (FFD-SP)**				**Semi-partitioned (LUF)**		
	$\mathbf{M_{OPT}}$	$\mathbf{M_{FFD}}$	$\frac{\mathbf{M_{FFD}}}{\mathbf{M_{OPT}}}$	$\text{mem}_{\mathbf{FFD}}$[B]	$\mathbf{L_{FFD}}$[c.c.]	$\mathbf{M_{SP}}$	$\frac{\mathbf{M_{SP}}}{\mathbf{M_{OPT}}}$	$\frac{\text{mem}_{\mathbf{SP}}}{\text{mem}_{\mathbf{FFD}}}$	$\frac{\mathbf{L_{SP}}}{\mathbf{L_{FFD}}}$	$\mathbf{M_{LUF}}$	$\frac{\text{mem}_{\mathbf{LUF}}}{\text{mem}_{\mathbf{FFD}}}$	$\frac{\mathbf{L_{LUF}}}{\mathbf{L_{FFD}}}$
FFT	24	30	1.25	144680	192512	26	1.083	1.413	1.483	26	1.485	1.676
Beamformer	26	28	1.077	14492	60912	26	1.0	1.145	1.474	26	1.229	1.606
TDE	20	25	1.25	516282	1127175	20	1.0	1.560	1.396	21	1.722	1.860
DES	26	33	1.269	3381	33088	27	1.038	1.138	1.218	28	1.684	1.862
MPEG2	8	9	1.125	61909	138240	8	1.0	1.290	1.217	9	3.014	3.432
Bitonic	11	13	1.182	2374	2275	11	1.0	1.139	1.185	11	1.413	1.395
Serpent	39	42	1.077	59815	370296	40	1.026	1.012	1.074	39	1.068	1.479
average	-	-	1.176	-	-	-	1.021	1.243	1.292	-	1.659	1.902

It is derived using the following expression:

$$\text{mem}_{\text{FFD}} = \sum_{i=1}^{N} \text{CSS}(\tau_i) + \sum_{i=1}^{|E|} b_u^{HRT} \tag{5.3}$$

where N is the number of tasks, $\text{CSS}(\tau_i)$ is the code and stack size of task τ_i (which represents actor A_i of the input CSDF graph G), E is the set of edges in G, b_u^{HRT} is the size of the buffer that implements the communication over edge e_u. The value of b_u^{HRT} assumes no task tardiness and is obtained using Equation (2.31) on page 46. Compared to FFD, in FFD-SP the memory requirements increase due to both the size of buffers, that have to be enlarged to handle task tardiness, and the code and stack size overhead of task replicas, which are necessary in case of migrating tasks. The memory requirement in FFD-SP is denoted by mem_{SP} and calculated as follows.

$$\text{mem}_{\text{SP}} = \sum_{i=1}^{M_{\text{SP}}} \sum_{\tau_j \in \Gamma_i} \text{CSS}(\tau_j) + \sum_{i=1}^{|E|} b_u^{SRT} \tag{5.4}$$

where M_{SP} is the number of processors required by the semi-partitioned approach, Γ_j is the set of tasks with non-zero shares on processor π_j, and b_u^{SRT} is the size of the buffer that implements the communication over edge e_u, calculated using Equation 5.2. Note that Equation (5.4) differs from Equation (5.3) because in the SP approach a task can have shares on different processors.

In Table 5.1 the overhead of our proposed FFD-SP over FFD, in terms of memory requirements, is expressed by the ratio $\frac{\text{mem}_{\text{SP}}}{\text{mem}_{\text{FFD}}}$. On average, our proposed approach requires 24.3% more memory compared to FFD.

The **second cost** is the increase in applications' latency, due to the postponement of task start times needed to handle task tardiness. Column L_{FFD} shows the applications' latency, expressed in clock cycles, under FFD. These values are derived using the latency analysis described in Section 4.7 of [Bam14]. In order to derive the application latency under FFD-SP, denoted by L_{SP}, we use the same analysis from [Bam14], considering the task start times obtained by our SRT approach (described in Section 5.5.1). Then, we add to that latency value the tardiness (which can be potentially null) of the output actor of the application. The latency increase of our FFD-SP over FFD is on average 29.2% (see column $\frac{L_{\text{SP}}}{L_{\text{FFD}}}$).

Finally, to evaluate the efficiency of our proposed FFD-SP heuristic, we compare its results with LUF. The memory requirements and application latencies under LUF are derived following the same procedures used for FFD-SP. We can see from the last two columns of Table 5.1 that over the considered benchmarks the EDF-fm approach with the LUF heuristic incurs a much larger memory overhead (on average +65.9%, see column $\frac{\text{mem}_{\text{LUF}}}{\text{mem}_{\text{FFD}}}$) and latency increase (on average +90.2%, see column $\frac{L_{\text{LUF}}}{L_{\text{FFD}}}$) compared to FFD. Moreover, we note that for most applications the number of required processors is equal or greater when using LUF (M_{LUF}) compared to our FFD-SP (M_{SP}), with the exception of the *Serpent* application. Only in that example, the LUF outperforms our FFD-SP due to the characteristics of the task set. This means that for most of the benchmarks our FFD-SP heuristic is equally or more efficient than LUF in exploiting the available processing resources.

5.8 Discussion

The theoretical analysis provided in Section 5.5 proves that streaming applications modeled as acyclic CSDF graphs can be scheduled using any soft real-time scheduler, providing hard real-time guarantees on the input/output interfaces between the application and the environment.

Using the theoretical results of Section 5.5, in Section 5.6 we propose a novel heuristic that is aimed at reducing the number of required processors while keeping a low buffer size and latency overhead when the EDF-fm SRT scheduling algorithm is used. Section 5.7 shows that on a set of real-life applications, our approach can reduce the number of processors required to schedule these applications, guaranteeing the same throughput. However, compared to a HRT partitioned approach, our semi-partitioned SRT approach incurs an overhead in terms of memory requirements (on average, 24.3%) and application latency (on average, 29.2%).

Chapter 6

Energy Efficient Semi-Partitioned Scheduling of SDF Graphs

Most of the work presented in this chapter has been published in [CS16].

As mentioned in Section 1.2.4, energy efficiency is one of the emerging challenges of the modern embedded MPSoCs design, for several reasons. For instance, in battery-powered devices, energy efficiency can guarantee longer battery life. In general, energy-efficient design decreases heat dissipation and, in turn, improves system reliability.

To address the energy efficiency challenge many techniques have been proposed in the past decade in the embedded system community. As explained in Section 1.2.4, these techniques exploit Voltage/Frequency Scaling (VFS) of processors and have been applied to both streaming applications and periodic independent real-time tasks sets. These VFS techniques can be classified as *offline* and *online*. Offline VFS uses parameters such as the worst-case execution time (WCET) and the period of tasks to determine, at design-time, appropriate voltage/frequency modes for processors and how to switch among them, if necessary. Online VFS exploits the fact that at run-time some tasks can finish earlier than their WCET and determines, at run-time, the voltage/frequency modes to obtain further energy savings.

In this chapter, we devise an approach to exploit VFS of processors, to minimize the energy consumption of streaming applications with throughput constraints. We do so by reusing the scheduling analysis proposed in Chapter 5, which considers the soft real-time (SRT) EDF-fm scheduling algorithm. However, in this chapter we propose a novel SRT semi-partitioned scheduling algorithm, different from EDF-fm, which allows an even distribution of the utilization of tasks among the available processors. In turn, this enables processors to run at a lower frequency, which yields to lower energy consumption. In particular, our proposed SRT semi-partitioned algorithm is based on *restricted migrations*, for the practical reasons explained in the introduction of Chapter 5.

Although the scheduling algorithm used in our VFS scheduling approach is SRT, our proposed approach can provide HRT guarantees to the input/output interfaces of the application with the environment. This property is ensured by the scheduling analysis proposed in Chapter 5, which is reused in this chapter. Therefore, the results of this chapter can be applied in the context of **hard real-time streaming systems**.

6.1 Problem Statement

To the best of our knowledge, the potential of semi-partitioned scheduling with restricted migrations together with VFS techniques to achieve lower energy consumption has not been completely explored. Therefore, in this chapter, we study the problem of energy minimization when mapping streaming applications with throughput constraints using such semi-partitioned approach. Our technique considers homogeneous multiprocessor systems in which voltage and frequency scaling is supported with a discrete set of operating voltage/frequency modes.

6.2 Contributions

As the main contribution of this chapter, we propose a VFS semi-partitioned scheduling technique aimed at streaming applications with throughput constraints. Our proposed schedling technique is depicted in Figure 6.1. As mentioned earlier, our technique builds upon the results of Chapter 5. In that chapter, we showed that a SRT semi-partitioned scheduler (EDF-fm) can be used to schedule actors of a (C)SDF graph as real-time periodic tasks.

The dependencies of the technique proposed in this chapter with the scheduling analysis of Chapter 5 are highlighted by dashed boxes in Figure 6.1. These boxes include steps that are identical to the scheduling framework in Figure 5.2. In particular, in both figures:

- In Step ①, we use the scheduling analysis of [BS11, BS12] to derive the WCET ($\boxed{\text{I}}$) and period ($\boxed{\text{II}}$) of each task, based on the characteristics of the input application model. Throughout this chapter, we will refer to such derivation as *scheduling analysis of [BS11, BS12]*. Recall, from Chapter 5, that parameters $\boxed{\text{I}}$ and $\boxed{\text{II}}$ do not depend on the (potential) tardiness of tasks.
- In Step ③, we assume that the periodic tasks will be scheduled by a SRT scheduling algorithm, which provides a certain tardiness bound Δ_i of each task τ_i. Then, based on these tardiness bound values, Step ③ derives valid start times of tasks ($\boxed{\text{III}}$), and sizes of the buffers ($\boxed{\text{IV}}$) which implement inter-task communication.

The differences of our VFS semi-partitioned technique shown in Figure 6.1, with regard to semi-partitioned approach of Chapter 5, are the following.

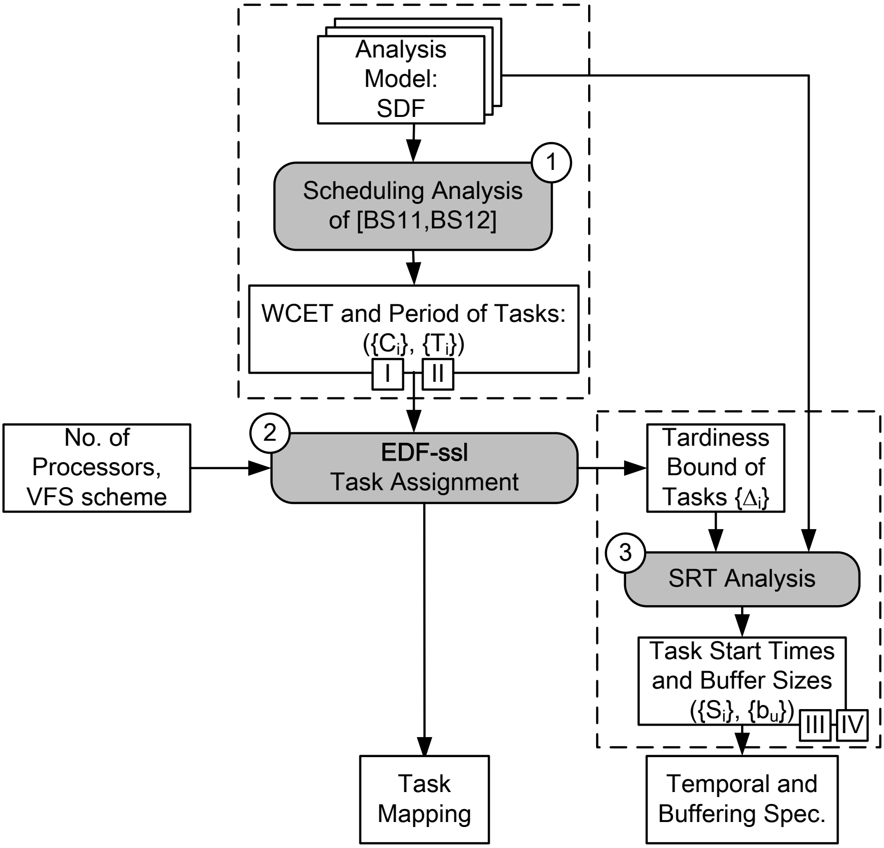


Figure 6.1: *Energy-efficient scheduling technique proposed in this chapter. Our proposed scheduling technique starts with Step ①, where the scheduling analysis of [BS11, BS12] is used to derive the WCET ($\boxed{I}$) and period ($\boxed{II}$) of each task, based on the characteristics of the input application model. Step ② uses as input the derived WCET and period of tasks, together with the considered number of active processors and VFS scheme (parameters provided by the designer). This step derives the Task Mapping and the tardiness bound of each task, based on the* EDF-ssl *scheduling algorithm proposed in this chapter. In turn, given the tardiness bound of tasks, Step ③ derives valid start times of tasks ($\boxed{III}$) and sizes of the buffers $\boxed{IV}$ that implement inter-task communication. Therefore, after Step ③, the complete Temporal and Buffering Specification is known. The steps enclosed in the dashed boxes reuse part of the scheduling framework shown in Figure 5.2 and represent the dependency of this chapter from the theoretical results of Chapter 5.*

1. The scheduling algorithm used to schedule the tasks on the system is different. Chapter 5 uses the EDF-fm SRT semi-partitioned scheduling algorithm. By contrast, in this chapter, we propose a novel semi-partitioned scheduling algorithm, called EDF-ssl (Earliest Deadline First based semi-partitioned stateless), which is targeted at streaming applications where some of the tasks may be

stateless[1]. In the presence of stateless tasks, our proposed EDF-ssl scheduler can be effectively used to achieve higher energy efficiency compared to existing partitioned and semi-partitioned approaches.

2. The scheduling framework shown in Figure 6.1 does not support VFS (i.e., all processors run at the highest frequency), and provides as *output* the minimum number of processors required to schedule the application. By contrast, in Figure 6.1, the number of available processors, and the VFS mode used in the system, are provided by the designer (see inputs of Step ②). This is because, to achieve higher energy efficiency, it may be beneficial to distribute the tasks of the application on a number of processors greater than the minimum required.
3. The scheduling analysis described in Chapter 5 can accept, as input, applications modeled as CSDF graphs (see Analysis Model in Figure 5.2). By contrast, we restrict the VFS scheduling approach presented in this chapter only to SDF graphs, which are a subset of CSDF graphs. This is because our semi-partitioned technique can only be beneficial if there are actors for which successive invocations can be executed in parallel. Actors that possess such property are, by definition, not allowed in the CSDF model of computation. However, they are allowed in the SDF model.

Note that, in order to perform the SRT Analysis of Step ③ in Figure 6.1, the tardiness bound of each task is required. Therefore, in this chapter we derive the tardiness bounds guaranteed by our proposed EDF-ssl scheduler. In particular, we derive these bounds in two cases. First, when using the lowest frequency which guarantees schedulability and is supported by the system. Second, when using a periodic frequency switching scheme that preserves schedulability and can achieve higher energy savings. In general, our EDF-ssl allows an even distribution of the utilization of tasks among the available processors. In turn, this enables processors to run at a lower frequency, which yields to lower power consumption. Moreover, compared to a purely partitioned scheduling approach, our experimental results show that our technique achieves the same application throughput with significant energy savings (up to 64%) when applied to real-life streaming applications. These energy savings, however, come at the cost of higher memory requirements and latency of applications.

6.3 Scope of work

The assumptions and limitations which define the scope of our work are listed in what follows.

6.3.1 Assumptions

Our work is built on some assumptions that we describe and motivate below.

---

[1] See Definition 2.3.6 on page 47.

First, we consider systems with distributed program and data memory. As mentioned in Section 1.1.2, this choice of memory subsystem is needed to ensure predictability of the execution at run-time (since PEs do not have to access shared resources to perform the computation), and scalability.

Second, we consider semi-partitioned scheduling, which is a hybrid between two extremes, *partitioned* and *global* scheduling. As shown in Chapter 5, semi-partitioned scheduling can ameliorate the bin-packing issues of partitioned scheduling when applied to streaming applications. At the same time, semi-partitioned scheduling does not incur the excessive memory and migration overheads of global scheduling.

Third, we assume that the system's communication infrastructure is predictable, i.e., it provides guaranteed communication latency. This assumption is needed because Step ① in Figure 6.1 uses the scheduling analysis of [BS11,BS12] (see Section 2.3) to derive WCET and period of each task, based on the characteristics of the input analysis model. This derivation requires the worst-case communication latency to compute the WCET of a task. The WCET of a task includes the worst-case time needed for the task's computation, the worst-case time needed to perform inter-task data communication on the considered platform and the worst-case overhead of the underlying scheduler, as explained in Section 2.3 (see, in particular, Equation (2.26) on page 43).

6.3.2 Limitations

The research problem addressed in this chapter, described in Section 6.1, is extremely complex. In order to make it more tractable, our approach considers certain limitations. However, we argue that even under these limitations many hardware platforms and applications can be handled by our proposed VFS scheduling technique. In what follows, we list the limitations considered in our proposed approach.

First, we assume that applications are modeled as acyclic SDF graphs. Although this assumption limits the scope of our work, our analysis is still applicable to the majority of streaming applications. In fact, a recent work [TA10] has shown that around 90% of streaming applications can be modeled as acyclic SDF graphs.

Second, we assume that the hardware platform supports a discrete set of operating VFS modes. Moreover, we assume that the operating voltage/frequency mode can only be changed globally over the considered set of processors. Our technique, therefore, finds applicability in hardware platforms that apply the same voltage/frequency mode to all the processors of the system (e.g., the OMAP 4460, as in [ZR13]). Note that our proposed technique does not consider per-core VFS, therefore it may be less beneficial for systems which support this kind of VFS granularity. However, per-core VFS is deemed unlikely to be implemented in next generation of many-core systems, due to excessive hardware overhead [DA10].

Third, our technique uses *offline* VFS because we do not exploit the dynamic slack created at run-time by the earlier completion of some tasks. This choice is motivated by the following two reasons. (i) Online VFS may require VFS transitions for each execution of a task. Since we consider applications in which tasks execute periodically, with very short periods, online VFS would incur significant transitions overhead. For instance, the period of tasks in the applications that we consider can be as low as 100 μs. Since the VFS transition delay overhead of modern embedded systems is in the range of tens of μs [P$^+$13], the overhead of online VFS would be substantial with such short task periods. (ii) Moreover, the existence of a global frequency for the whole voltage island renders *online* VFS less applicable. This is because online VFS would only be effective if *all* cores in the voltage island have dynamic slack at the same time.

6.4 Related work

Several techniques addressing energy minimization for streaming applications have already been proposed in the literature. Among these, the closest to our work are [WLL$^+$11,SDK13,HMGM13]. [WLL$^+$11] considers applications modeled as Directed Acyclic Graphs, applies certain transformation on the initial graph and then generates task schedules using a genetic algorithm, assuming *per-core* VFS. [SDK13] assumes that applications are modeled as SDF graphs, and is composed of an offline and online VFS phases, to achieve energy optimization. As shown in Section 6.7, our approach exploits results from the real-time scheduling theory that allow, in the presence of stateless tasks, to set the global system frequency to the lowest value which guarantees schedulability and is supported by the system. Both [WLL$^+$11] and [SDK13] cannot in general make the system execute at the lowest frequency that supports schedulability because they use pure partitioned assignment of tasks to processors and non-preemptive scheduling. Finally, [HMGM13] considers both per-core and global VFS but assumes applications modeled as Homogeneous SDF graphs, and that the task mapping and the static execution order of tasks is given. By contrast, our approach handles a more expressive MoC and does not assume that the initial task mapping is given.

In addition, several techniques to achieve energy efficiency for systems executing periodic independent real-time tasks have been proposed. Among these techniques, the ones presented in [DA10] and [SJPL08] are closely related to our approach because they consider global VFS. The authors in [DA10] study the problem of energy minimization when executing a periodic workload on homogeneous multiprocessor systems. Their approach, however, considers pure partitioned scheduling. As we show in this chapter, pure partitioned scheduling can not achieve the highest possible energy efficiency. In our approach, instead, we consider semi-partitioned scheduling and we show that this approach yields significant energy savings compared to a pure partitioned one. The authors in [SJPL08] also address the problem of energy minimization under a periodic workload with real-time constraints. However, their approach allows migration of tasks at any time and to any processor. Therefore, their approach considers global scheduling of tasks. As explained earlier, in distributed memory systems global task scheduling entails high overheads, in terms of required memory and number of required preemptions and migrations of tasks. Our approach considers semi-partitioned scheduling in order to reduce such overheads, while obtaining higher energy efficiency than pure partitioned approaches.

Similar to our work, other related approaches exploit task migration to achieve energy efficiency, such as [HXW$^+$10] and [Zhe07]. In [HXW$^+$10], the authors reallocate tasks at run-time to reduce the fragmentation of idle times on processors. This in turn allows the system to exploit the longer idle times by switching the corresponding processors off. As explained earlier, in our approach we do not exploit run-time processor transitions to the off state because such transitions incur high overheads, especially when running dataflow tasks which have short periods.

The approach presented in [Zhe07] is closely related to ours because it leverages a semi-partitioned approach, where tasks migrate with a predictable pattern, to achieve energy efficiency. The author in [Zhe07] presents a heuristic to assign tasks to processors in order to obtain an improved load balancing. When tasks cannot entirely fit on one processor, they are split in two shares which are assigned to two different processors. Our work differs from [Zhe07] in two main aspects. First, we allow tasks with heavy utilization to be divided in more than two shares. This can yield to much higher energy savings compared to the technique proposed in [Zhe07]. Second, we allow job parallelism, i.e., we allow the concurrent execution on different processors of jobs of the same task. This, in turn, contributes to an improved balancing of the load among processors, which allows us to apply voltage and frequency scaling more effectively, as will be shown in Section 6.7. Moreover, the applicability of the analysis proposed in [Zhe07] to task sets with data dependencies, as in our case, is questionable. In fact, the semi-partitioned scheduling algorithm underlying [Zhe07] is identical to the one proposed by Anderson et al. in [ABD08]. As the latter paper shows, under this semi-partitioned scheduling algorithm tasks can miss deadlines by a value called tardiness, even when VFS is not considered. Since in our case tasks communicate data, to guarantee that data dependencies among tasks are respected this tardiness must be analyzed. However, an analysis of task tardiness is not given by [Zhe07].

As mentioned earlier, the approach we propose in our work exploits the concurrent execution on different processors of jobs of the same task. In a similar fashion, related works that exploit parallel execution of a task on different processors to achieve energy efficiency are [W$^+$10] and [Lee09]. In [W$^+$10] the authors exploit the *data* parallelism available in the input application. That is, jobs of an application are divided in sub-jobs which process independent subsets of the input data. These sub-jobs can therefore be executed independently and concurrently on different processors, obtaining a more balanced load on processors, which in turn allows a more effective scaling of voltage and frequency of processors. The approach presented in [W$^+$10], however, incurs a drawback in the case of distributed memory architectures. In fact, the mentioned sub-jobs of the application can be seen as separate instances of the input application, which execute independent chunks of input data. This means that, in distributed memory architectures, the code of the whole application has to be replicated on all the processors which execute these sub-jobs. By contrast, in our approach only certain tasks of the input application have to be replicated (only migrating tasks), which reduces significantly the memory overhead of our approach compared to the one in [W$^+$10]. An approach similar to [W$^+$10] has been proposed by the authors in [Lee09]. The technique presented in [Lee09] also divides computation-intensive tasks to sub-tasks which can be concurrently executed on multiple cores.

As in [W$^+$10], this yields to a more balanced load on processors, and in turn allows the system to run at a lower frequency. Moreover, the authors in [Lee09] consider systems with discrete set of operating frequencies. Similar to our technique, when the lowest frequency which guarantees schedulability is not supported by the system, the analysis in [Lee09] employs a processor frequency switching scheme to obtain this lowest frequency and still meet all deadlines. However, our analysis is different from [Lee09] in several aspects. First, when assigning sub-tasks load to the available processors, [Lee09] considers only *symmetric* distribution of the load of a task to different processors. In contrast, in our proposed approach, as shown in Example 6.7.2 in Section 6.7, in order to obtain optimal energy savings we allow an asymmetric distribution of the load of certain tasks to the available processors. Second, two major differences concern the derivation of the periodic VFS switching scheme that guarantees schedulability. The first difference is that the analysis in [Lee09] does not account for the overheads incurred when performing VFS transitions. By contrast, our analysis take this realistic overhead into account. The second difference is that in [Lee09] such periodic VFS switching scheme is derived in order to meet *all* the deadlines of tasks. This requires the system to perform very frequent VFS transitions, especially when tasks have short periods as in our case. Conversely, in our approach we allow some task deadlines to be missed, by a bounded amount. This allows our approach to perform much fewer VFS transitions. As VFS transitions incur time and energy overhead in realistic systems, our approach guarantees higher effectiveness compared to [Lee09].

The semi-partitioned scheduling that we propose, EDF-ssl, allows only restricted migrations. Notable examples of existing semi-partitioned scheduling algorithms with restricted migrations are EDF-fm [ABD08] and EDF-os [AEDC14], which are described in Sections 2.2.7 and 2.2.8 of this thesis. Our EDF-ssl algorithm inherits some properties from EDF-fm and EDF-os. The closest to our EDF-ssl is EDF-os because it allows migrating tasks to run on two or more processors, not strictly on two as in EDF-fm. The fundamental difference between EDF-os and our proposed EDF-ssl lays in the kind of applications that are considered by these two scheduling algorithms. In EDF-ssl we consider applications in which some of the tasks may be stateless and therefore can execute different jobs of the same task in parallel, if released on different processors. By contrast, EDF-os considers applications modeled as sets of tasks where all tasks are *stateful*. This means that different jobs of the same task cannot be executed concurrently. As explained in detail in Section 6.7, this fact prevents EDF-os from achieving energy-optimal results when streaming applications have stateless tasks with high utilization. This phenomenon is also described in the experimental results section (Section 6.9.3). Similar to our work, analyses of scheduling algorithms that allow jobs within a single task to run concurrently are presented in [EA11, YA14]. However, both these works consider global scheduling algorithms which, as mentioned earlier, entail high overheads especially in distributed memory architectures. In addition, in both [EA11] and [YA14] the potential of exploiting job parallelism to achieve higher energy efficiency is not explored.

6.5 System Model

In this section, we define the system model used in this chapter. As in Chapter 5, we consider a system composed of a set $\Pi = \{\pi_1, \pi_2, \cdots, \pi_M\}$ of M homogeneous processors. In this chapter, however, we assume that processors are endowed with VFS capability. In particular, as explained in the beginning of this chapter, we consider the problem of mapping applications to systems in which all the cores belong to the same voltage/frequency island. This means that any processor in the system either runs at the same "global" frequency and voltage level, or is idle. Each idle processor has no tasks assigned to it and consumes negligible energy. We assume that the system supports only a discrete set $\Phi = \{F_1, F_2, \cdots, F_N\}$ of N operating frequencies, where the maximum frequency is $F_N = F_{\max}$. To ease the explanation of our analysis, based on this maximum frequency $F_{\max}$ we define the normalized system speed as follows.

Definition 6.5.1. *(Normalized speed)*. Given a frequency F at which the system runs, this system is said to run at a *normalized system speed* $\alpha = F/F_{\max}$.

This definition creates a one-to-one correspondence between any frequency at which the considered system runs and its normalized speed. We will exploit this correspondence throughout this chapter. Given the set of supported frequencies Φ, by applying Def. 6.5.1 we obtain a set of supported normalized system speeds $\mathcal{NS} = \{\alpha_1, \alpha_2, \cdots, \alpha_N\}$, where $\alpha_N = \alpha_{\max} = 1$.

6.6 Example of SRT Scheduling of an SDF Graph

In this section, we provide an example of the scheduling technique shown in Figure 6.1. This example will be used in the remainder of this chapter. We recall that the dashed boxes in Figure 6.1 represent the dependencies of the scheduling analysis proposed in this chapter from the theoretical results of Chapter 5. Based on the characteristics of the input application model, the steps contained in the dashed boxes are used to obtain the complete *temporal and buffering specification* of the task set, i.e., the WCET, period, and start time of actors, together with the size of the buffers that implement inter-task communication. Altogether, these steps convert the SDF model of the input application to a set of real-time periodic tasks which can be scheduled by an SRT scheduler.

In particular, Step ① in Figure 6.1 uses the scheduling analysis of [BS11, BS12] (described in Section 2.3) to derive the WCET and period of tasks. More in detail, it uses Equation (2.26) on page 43 to derive the WCET of tasks and Equation (2.29) on page 44 to calculate their periods.

Step ② in Figure 6.1 requires as input the obtained WCET and period of actors. In addition, it assumes that the number of available processors in the system and the VFS scheme are given (for instance, by the designer). Using these inputs, Step ② derives the mapping of tasks to processors and the tardiness bound of each task. In the next sections, we will describe how the task mapping and tardiness bound of tasks are derived under our proposed EDF-ssl scheduler.

Assuming that the tardiness bound of each task is known, Step ③ applies the theoretical results from Chapter 5 to derive the earliest start times of tasks and minimum sizes of the buffers that implement inter-task data communication.

In the example provided below, we describe in greater detail how periods and start times of tasks are derived, in Step ① and Step ③ in Figure 6.1, respectively.

Example 6.6.1. Consider the SDF graph shown in Figure 6.2(a), which has three actors (A_1, A_2, A_3) with WCET indicated between parentheses (C_1=2, C_2=3, C_3=2) and production/consumption rates indicated above the corresponding edges. In Step ① in Figure 6.1, using Equation (2.27) on page 43, we derive the following minimum periods: T_1=T_3=6 and T_2=3, as shown in Figure 6.2(b). Then, suppose that the underlying SRT scheduling algorithm guarantees tardiness bounds Δ_1=1, Δ_2=2 (as indicated in Figure 6.2(a) and visualized in Figure 6.2(b)), whereas Δ_3=0.

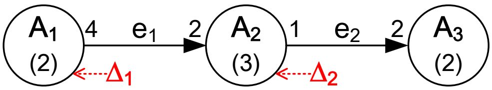
In Step ③ in Figure 6.1, using these tardiness bounds, we apply Proposition 5.5.1 on page 95 and derive the earliest start times S_i shown in Figure 6.2(b). For instance, note that S_2=7 ensures that any invocation of A_2 will always have enough data to read as soon as it is released. This holds even when all the invocations of A_1 incur the largest tardiness Δ_1, i.e., they execute according to the ALAP completion schedule (see Definition 5.5.1 on page 95).

6.7 Proposed Semi-partitioned Algorithm: EDF-ssl

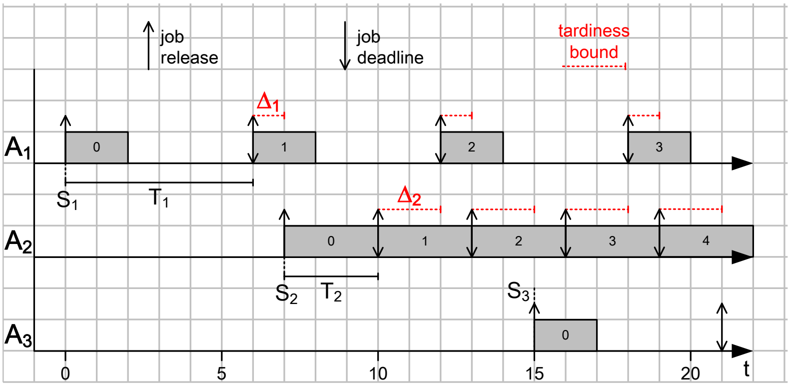
In this section we describe our proposed semi-partitioned scheduler, called EDF-ssl. In EDF-ssl, only stateless tasks (recall Definition 2.3.6 on page 47) are allowed to be migrated. We enforce this condition because migrating the internal state of a stateful task can be prohibitive in a distributed memory system. Note that under EDF-ssl task migrations can only happen at job boundaries. Once a job is released on a certain processor, it cannot migrate to another one. Moreover, EDF-ssl exploits the fact that migrating tasks are stateless by allowing successive jobs to execute in parallel on different processors. For instance, in Figure 6.4(b), jobs $\tau_{1,0}$ and $\tau_{1,1}$ are executed on two different processors and can execute in parallel.

With our EDF-ssl we want to show that, in the presence of stateless tasks, semi-partitioned scheduling can be used to improve energy efficiency, while achieving the same application throughput compared to purely partitioned scheduling. To achieve better energy efficiency it may be beneficial to run processors at voltage/frequency levels lower than the maximum. The following example shows that under certain conditions the classical partitioned VFS techniques (e.g., [AY03]) are not effective. Moreover, existing semi-partitioned approaches do not exploit the presence of some stateless tasks in the considered applications and therefore cannot be applied to achieve energy efficiency, if these stateless tasks have high utilization.

Example 6.7.1. Consider a single stateless task $\tau_1 = (C_1 = 3, T_1 = 3)$. The task utilization is $u_1 = 1$. In this case, existing partitioned VFS techniques can not be



(a) Simple example of an SDF graph. The WCET of actors are indicated between parentheses. The production/consumption rates of actors are indicated above the corresponding edges.



(b) Derived periodic task set and minimum start times. Up arrows represent job releases, down arrows represent task deadlines. The schedule of all actors continues indefinitely.

Figure 6.2: *Example of the approach proposed in Chapter 5 to schedule an SDF graph with a scheduler that provides SRT guarantees. The SDF actors in sub-figure (a) are scheduled as real-time periodic tasks in sub-figure (b). Task periods (T_1, T_2, T_3) are derived using the methodology of [BS11, BS12]. Then, the analysis proposed in Chapter 5 considers the tardiness bounds guaranteed by the SRT scheduler to each task. In this figure, the tardiness bounds of actors τ_1 and τ_2 are Δ_1 and Δ_2, respectively. Using these bounds, the analysis proposed in Chapter 5 derives valid start times (S_1, S_2, S_3) of actors such that all tasks can be released periodically without any buffer underflow.*

effective, because τ_1 can only be assigned to one processor and this processor must run at its highest voltage/frequency level, because $u_1 = 1$. Moreover, even existing semi-partitioned approaches cannot distribute the utilization of τ_1 over more than one processor, as shown in the following. Assume that to improve energy efficiency the utilization of τ_1 has to be split over two cores, π_1 and π_2, running at half of the maximum frequency, i.e., at normalized processors speed $\alpha = 1/2$. Note that under these conditions the schedulability test given by Inequality (2.23) on page 42 has to be changed according to the current normalized processor speed.

We enforce therefore $\sigma_1 \leq \alpha$ and $\sigma_2 \leq \alpha$. The resulting assignment of shares of τ_1 is shown in Figure 6.3.

In this scenario, the problem of EDF-os is that it does not consider job parallelism. This means that job $\tau_{i,k+1}$ of a migrating task τ_i has to wait for the completion of the previous job $\tau_{i,k}$. For instance, in Figure 6.4(a), job $\tau_{1,0}$ is released on π_1 at time 0. Since $\alpha = 1/2$, $\tau_{1,0}$ finishes at time 6. Therefore job $\tau_{1,1}$, although released at time 3 on π_2,

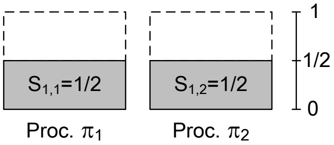
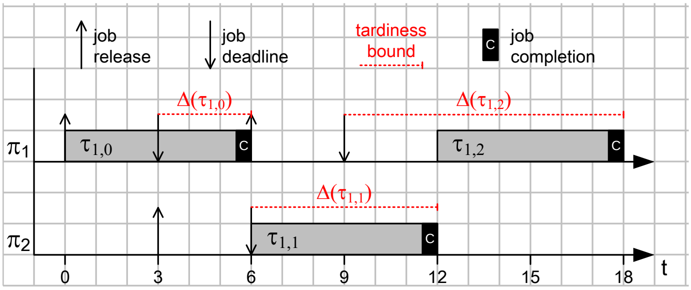
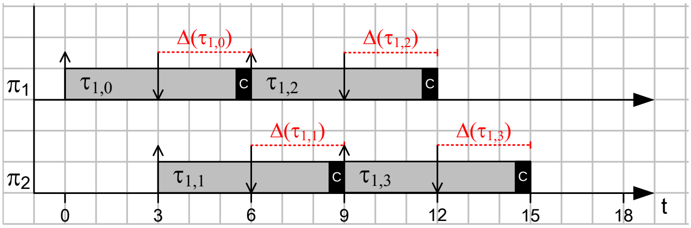


Figure 6.3: *Share assignment considered in Example 6.7.1. The utilization* $u_1 = 1$ *of a single migrating task* τ_1 *is split into two shares* $s_{1,1} = 1/2$ *and* $s_{1,2} = 1/2$*. Shares* $s_{1,1}$ *and* $s_{1,2}$ *are assigned to processors* π_1 *and* π_2*, respectively.*



(a) Job executions according to EDF-os rules. Since EDF-os does not consider job parallelism, the tardiness of successive jobs of task τ_1 grows unboundedly, as shown in red in this figure.



(b) Job executions according to EDF-ssl rules. EDF-ssl does consider job parallelism, allowing jobs released by migrating task τ_1 to execute in parallel. This leads to bounded tardiness for all jobs of τ_1.

Figure 6.4: *Job executions of* $\tau_1 = (C_1 = 3, T_1 = 3)$*, as defined in Example 6.7.1, according to the share assignment of Figure 6.3. Up arrows indicate job releases, down arrows indicate job deadlines. Black rectangles indicate job completion. Although the WCET of* τ_1 *is 3 time units, each job of* τ_1 *takes 6 time units to complete because processors* π_1 *and* π_2 *run at normalized processor speed* $\alpha = 1/2$*.*

has to wait until time 6 to start executing. As shown in Figure 6.4(a), although jobs of τ_1 are assigned alternatively to π_1 and π_2, the tardiness Δ incurred by successive jobs of τ_1 increases unboundedly. Our EDF-ssl avoids this linkage between processors by allowing jobs released by a migrating task to execute in parallel, exploiting the fact that migrating tasks are assumed to be stateless. As depicted in Figure 6.4(b), this leads to bounded tardiness for all jobs of τ_1.

Under our EDF-ssl, necessary (but not sufficient) conditions to guarantee schedulability are the following. First, the total utilization of the task set Γ cannot be higher than the total available utilization on processors: $U_\Gamma \leq \alpha \cdot M$, where M is the number of available processors in the system and assuming that they all run at the same normalized speed $\alpha \leq 1$. Second, α must be greater than the utilization of any stateful task in Γ: $\alpha \geq u_{\mathrm{s,max}}$, where $u_{\mathrm{s,max}}$ is the utilization of the heaviest stateful task in Γ. This is because stateful tasks are fixed, and any processor to which the utilization $u_{\mathrm{s,max}} > \alpha$ is assigned will be overloaded. We merge the above two conditions in the following expression, which provides necessary higher and lower bounds for α:

$$\max\{U_\Gamma / M,\, u_{\mathrm{s,max}}\} \leq \alpha \leq 1 \tag{6.1}$$

We now proceed with a detailed description of our EDF-ssl. As in all semi-partitioned approaches (e.g., [ABD08, AEDC14]), EDF-ssl is composed of two phases, an assignment phase and an execution phase, which are described in Section 6.7.1 and Section 6.7.2, respectively. Tardiness bounds guaranteed under EDF-ssl are derived in Section 6.7.3, for the case of processors running at a fixed normalized speed α. Finally, Section 6.7.4 presents a processor speed switching technique, called "Pulse Width Modulation (PWM) scheme", that provides a certain normalized speed in the long run. Tardiness bounds are derived also for the latter scenario.

6.7.1 Assignment Phase

The assignment phase of EDF-ssl tries to find an assignment of tasks to processors that reduces the number of tasks with tardiness. This is because, as described in Chapter 5, many tasks with tardiness result in high overheads in terms of application latency and buffer sizes.

Note that under EDF-ssl processors can run at a normalized speed α lower than 1. Therefore, to avoid overloading processors in the long run, we modify the schedulability test given by Condition (2.23) on page 42 as follows:

$$\sigma_k \leq \alpha, \ \forall\, \pi_k \in \Pi \tag{6.2}$$

where σ_k is the total share assignment on any processor π_k. Expression (6.2) implies that σ_k cannot exceed the normalized processor speed. Moreover, note that searching a valid assignment makes only sense if Condition (6.1) is satisfied.

The assignment phase of EDF-ssl consists mainly of 3 steps, which we explain below.

First step. In this step, we find the set of stateful tasks Γ_s within the original task set Γ. Then, we use the First-Fit Decreasing Heuristic (FFD) [Joh73] (see Section 2.2.6) to allocate these stateful tasks as *fixed* tasks over the available processors. This means that if $\tau_i \in \Gamma_s$ is assigned to processor π_k, its share on π_k should be equal to the whole task utilization: $s_{i,k} = u_i$. On all the other processors, task τ_i has no shares.

Second step. This step tries to assign all the remaining (stateless) tasks as *fixed* tasks over the remaining available processor utilization, using FFD. The tasks which can not be assigned as fixed are added to a set of tasks Γ_{na} (not assigned), which are assigned in the next step.

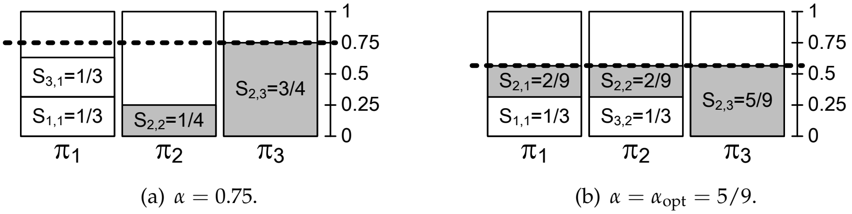
Algorithm 3: Share assignment heuristic.

```
   Input: A set of $M$ processors $\Pi = \{\pi_1, \pi_2, \cdots, \pi_M\}$, their normalized speed $\alpha$,
          a set of $N$ periodic tasks $\Gamma = \{\tau_1, \tau_2, \cdots, \tau_N\}$.
   Result: An $M$-partition describing the share assignment onto $M$ processors if $\Gamma$
           is schedulable, False otherwise.
1  Find $\Gamma_s = \{\tau : \tau \in \Gamma \wedge \tau$ is stateful$\}$;
2  for $\tau_i \in (\Gamma_s,$ sorted by decreasing utilization$)$ do
3      Try to assign $s_{i,k} = u_i$ of task $\tau_i$ on a single $\pi_k$ using FF;
4      if FF fails for all $\pi_k \in \Pi$ then
5          return False;
6  $\Gamma_{\text{na}} = \emptyset$ (the set of unassigned tasks, initially empty)
7  for $\tau_i \in (\Gamma - \Gamma_s,$ sorted by decreasing utilization$)$ do
8      Try to assign $s_{i,k} = u_i$ of task $\tau_i$ on $\pi_k$ using FF;
9      if FF fails for all $\pi_k \in \Pi$ then
10         $\Gamma_{\text{na}} = \Gamma_{\text{na}} \cup \tau_i$;
11 k = M (start share assignment from processor $\pi_M$ to $\pi_1$);
12 for $\tau_i \in \Gamma_{na}$ do
13     $u_{\text{remaining}} = u_i$;
14     while $u_{remaining} > 0$ do
15         $s_{i,k} = \min(u_{\text{remaining}}, (\alpha - \sigma_k))$;
16         $\sigma_k = \sigma_k + s_{i,k}$;
17         $u_{\text{remaining}} = u_{\text{remaining}} - s_{i,k})$;
18         if $\sigma_k = \alpha$ then
19             k = k-1
```

Third step. The final step assigns all the remaining tasks, which could not be allocated as fixed tasks. Considering the processor list in reversed order $\{\pi_M, \pi_{M-1}, \cdots \pi_1\}$, task $\tau_i \in \Gamma_{\text{na}}$ is allocated a share on successive processors, considering the remaining utilization on each processor, in a sequential order. (The remaining utilization on processor π_k is given by $(\alpha - \sigma_k)$). The assignment of task τ_i finishes when the sum of its shares over the processors equals the task utilization u_i. The third step considers the processor list in reversed order as a way to minimize the number of processors, which already have fixed tasks, that are utilized to assign migrating shares. This can lead to a lower number of tasks with tardiness.

The three steps described above are implemented in Algorithm 3. In particular, the first step is represented in lines (1-5), the second step in lines (6-10), the third and final step in lines (11-17).

Example 6.7.2. Consider the SDF graph example in Figure 6.2(a). In Example 6.6.1, we derived the corresponding task set $\Gamma = \{\tau_1 = (2,6), \tau_2 = (3,3), \tau_3 = (2,6)\}$. Tasks τ_1 and τ_2 are stateful, whereas task τ_3 is stateless. The total utilization of the task set is



(a) $\alpha = 0.75$. (b) $\alpha = \alpha_{\text{opt}} = 5/9$.

Figure 6.5: *Share assignments considered in Example 6.7.2. Values of the normalized system speed α are denoted by a dashed line. Migrating tasks are indicated in gray. In sub-figure (a), the normalized system speed α is set to the lowest value that guarantees schedulability and is supported by the system, i.e., $\alpha = 0.75$. In sub-figure (b), we use the periodic frequency switching technique described in Section 6.7.4 to get the normalized speed $\alpha = \alpha_{opt} = 5/9$ in the long run. The technique represented in sub-figure (b) leads to additional energy savings due to an even distribution of tasks shares among processors.*

$U_\Gamma = 1/3 + 1 + 1/3 = 5/3$. Assume that we want to execute this task set on $M = 3$ processors. By Condition (6.1), $\alpha \geq U_\Gamma/M = 5/9$, therefore the lowest α which could provide schedulability is $\alpha_{\text{opt}} = 5/9$. Running the system at this lowest speed α_{opt} minimizes the energy consumption. Now, if the system supports the speed α_{opt}, we can simply set the system speed to that value. In this case, we can derive tardiness bounds using the result in Section 6.7.3, which considers fixed processors speed.

However, suppose that the considered system supports a set of normalized speeds $\mathcal{NS} = \{0.25, 0.5, 0.75, 1\}$. Note that $\alpha_{\text{opt}} \notin \mathcal{NS}$. In this case, we have two choices. *Choice 1)* We set the system speed to the lowest $\alpha \in \mathcal{NS}$ such that $\alpha > \alpha_{\text{opt}}$, condition which could provide schedulability: $\alpha = 0.75$. We can then refer again to Section 6.7.3 to derive tardiness bounds in this scenario. Figure 6.5(a) shows the share assignment of tasks in Γ, when $\alpha = 0.75$ and assuming that input and output actors (τ_1, τ_3) are stateful. *Choice 2)* We use the periodic speed switching technique described in Section 6.7.4 to get the normalized speed α_{opt} in the long run, and we derive the corresponding tardiness bounds. Figure 6.5(b) shows the assignment obtained when $\alpha = \alpha_{\text{opt}} = 5/9$.

6.7.2 Execution Phase

At run-time, EDF-ssl follows the simple rules defined below.

Job releasing rules. Jobs of a fixed task τ_f are released periodically, every T_f, on a single processor. Jobs of a migrating task τ_m are distributed over all the processors on which τ_m has non-zero shares. Our EDF-ssl inherits from EDF-os (and, in turn, from EDF-fm) the job releasing techniques for migrating tasks (see Section 2.2.8). In particular, the job releasing technique uses the concept of *task fraction*, defined in Definition 2.2.12 on page 39, to avoid overloading the processors in the long run. For an example, refer to Figure 2.5 on page 41. That figure shows the job release pattern of tasks τ_3 on processors π_1 and π_2. Since task τ_3 has task fractions $\varphi_{3,1} = 3/4$ on processor π_1, and $\varphi_{3,2} = 1/4$ on processor π_2, in the long run the number of jobs of τ_3 released on π_1 is three times the number of jobs released on π_2. Moreover, Inequality (2.24) on page 42, which provides an upper bound of the number of jobs released on a processor as a function of the migrating task fraction, is still valid. This result will be instrumental to the derivation of tardiness bounds under our EDF-ssl.

Job prioritization rules. Jobs of fixed and migrating tasks released on a certain processor are scheduled using a local EDF scheduler. As shown in Example 6.7.1, under our EDF-ssl when a task migrates from a processor to another one, the job released on the latter processor does not wait until the completion of the job released on the former processor. This is in contrast with what happens under EDF-os. Moreover, contrary to our EDF-ssl, under EDF-os certain tasks are statically prioritized over others.

6.7.3 Tardiness Bounds under Fixed Processor Speed

Given the rules and properties of our EDF-ssl, described in Section 6.7.1 and Section 6.7.2, we now derive its tardiness bounds, which are provided by Theorem 6.7.1 below. Note that due to the way task shares are assigned in the third step of the assignment phase, each processor runs at most two migrating tasks.

Theorem 6.7.1. *Consider a processor π_k running at a fixed normalized speed α. Assume two migrating tasks, τ_i and τ_j, are assigned to π_k. Then, jobs of fixed and migrating tasks released on π_k may incur a tardiness of at most*

$$\Delta^{\pi_k} = \frac{2\,(C_i + C_j)}{\alpha} \tag{6.3}$$

where C_i and C_j are the worst-case execution time of τ_i and τ_j, respectively, and α follows Definition 6.5.1.

Proof. We prove Theorem 6.7.1 by contradiction. We focus on a certain job $\tau_{q,l}$, belonging to either a fixed or a migrating task, assigned to π_k . Let assume that this job incurs a tardiness which exceeds Δ^{π_k}. We define the following time instants to assist the analysis: $\mathbf{t_d}$ is the absolute deadline of job $\tau_{q,l}$; $\mathbf{t_c} = t_d + \Delta^{\pi_k}$; and $\mathbf{t_0}$ is the latest instant before t_c such that no migrating or fixed job released before t_0 with deadline at most t_d is pending at t_0. By definition of t_0, just before t_0 π_k is either idle or executing a job with deadline later than t_d. Moreover, t_0 cannot be later than $r_{q,l}$, the release time of job $\tau_{q,l}$. Note that since we assume that job $\tau_{q,l}$ incurs a tardiness exceeding Δ^{π_k}, it follows that $\tau_{q,l}$ does not finish at or before t_c.

We denote as γ the total set of tasks, fixed and migrating, assigned to π_k. We first determine the demand (see Definition 2.2.1) placed on π_k by γ in the time interval $[t_0, t_c)$. By the definitions of t_0, t_d, and t_c, any job of any task that places a demand in $[t_0, t_c)$ on π_k is released at or after t_0 and has a deadline at or before t_d. Therefore, the demand of any task τ_i in $[t_0, t_c)$ is given by the number of jobs released in this interval multiplied by the job execution time.

The number of jobs released on π_k in $[t_0, t_c)$, by a *fixed* task τ_f, is at most $c = \lfloor \frac{t_d - t_0}{T_f} \rfloor$ because fixed tasks release all of their jobs on π_k. By contrast, a *migrating* task τ_m

releases $c = \lfloor \frac{t_d - t_0}{T_m} \rfloor$ jobs, but only part of them are assigned to π_k. An upper bound of the amount of jobs assigned to π_k, out of every c consecutive jobs, is given by Inequality (2.24) on page 42.

We can now compute the total demand from tasks assigned to π_k. We denote as γ_f and γ_m the fixed and migrating sets of tasks mapped on π_k, respectively. Note that $\gamma_m = \{\tau_i, \tau_j\}$.

Given the total number of released jobs c, from Inequality (2.24) the demand[2] *dmd* from migrating tasks in $[t_0, t_c)$ is upper bounded by:

$$\begin{aligned}\mathrm{dmd}(\gamma_m, t_0, t_c) &\leq \left(\varphi_{i,k}\left\lfloor\frac{t_d - t_0}{T_i}\right\rfloor + 2\right) C_i + \left(\varphi_{j,k}\left\lfloor\frac{t_d - t_0}{T_j}\right\rfloor + 2\right) C_j \\ &\leq (t_d - t_0)\left(\varphi_{i,k}\frac{C_i}{T_i} + \varphi_{j,k}\frac{C_j}{T_j}\right) + 2(C_i + C_j)\end{aligned}$$

Given the definition of task fraction $\varphi_{i,k}$ (see Definition 2.2.12 on page 39), we obtain:

$$\mathrm{dmd}(\gamma_m, t_0, t_c) \leq (t_d - t_0)(s_{i,k} + s_{j,k}) + 2(C_i + C_j) \tag{6.4}$$

At the same time, the demand from fixed tasks in $[t_0, t_c)$ is upper bounded by:

$$\mathrm{dmd}(\gamma_f, t_0, t_c) \leq \sum_{\tau_f \in \gamma_f} \left\lfloor\frac{t_d - t_0}{T_f}\right\rfloor C_f \leq (t_d - t_0) \sum_{\tau_f \in \gamma_f} \frac{C_f}{T_f}$$

From Condition (6.2), we obtain:

$$\mathrm{dmd}(\gamma_f, t_0, t_c) \leq (t_d - t_0)(\alpha - s_{i,k} - s_{j,k}) \tag{6.5}$$

Combining Inequality (6.4) and (6.5), we derive an upper bound for the total demand of fixed and migrating tasks in $[t_0, t_c)$:

$$\mathrm{dmd}(\gamma_f \cup \gamma_m, t_0, t_c) \leq \alpha(t_d - t_0) + 2(C_i + C_j) \tag{6.6}$$

To ease our analysis, we now express the total demand from tasks in clock cycles. In fact, any requirement in processor time can be converted to clock cycles. For instance, for any task τ_a, its worst-case clock cycles requirement is $CC_a = C_a \cdot F_{\max}$. This is because the worst-case execution *time* C_a of τ_a is obtained at the maximum processor frequency, $F_{\max}$ (see definition of $F_{\max}$ in Section 6.5).

Then, from Inequality (6.6) we get:

$$\mathrm{dmd\_cc}(\gamma_f \cup \gamma_m, t_0, t_c) \leq F_{\max}\left(\alpha(t_d - t_0) + 2(C_i + C_j)\right) \tag{6.7}$$

Now, from our initial assumption that the tardiness of job $\tau_{q,l}$ exceeds Δ^{π_k}, it follows that the amount of clock cycles provided by the processor in the interval

---

[2] See Definition 2.2.1.

$[t_0, t_c)$ is less than the total demand from tasks *dmd_cc* in the same time interval. In the considered interval, the total demand from tasks is upper bounded by Inequality (6.7), whereas the amount of clock cycles provided by processor π_k is $\alpha \cdot F_{\max}(t_c - t_0)$, because π_k runs at frequency $\alpha \cdot F_{\max}$. Therefore, we have:

$$\alpha \cdot F_{\max}(t_c - t_0) < F_{\max}\left(\alpha(t_d - t_0) + 2(C_i + C_j)\right) \tag{6.8}$$

Dividing both sides by $\alpha \cdot F_{\max}$:

$$t_c < t_d + 2(C_i + C_j)/\alpha \;\Rightarrow\; t_c < t_d + \Delta^{\pi_k} \tag{6.9}$$

Expression (6.9) contradicts the earlier definition of $t_c = t_d + \Delta^{\pi_k}$, therefore Theorem 6.7.1 holds. ■

Note that the tardiness bound given by Equation (6.3) differs from the tardiness bounds of EDF-os given by Equation (3) and (10) in [AEDC14]. This is caused by the differences in the *execution* phase between the two scheduling algorithm described in Section 6.7.2.

6.7.4 Tardiness Bounds under PWM Scheme

The optimal normalized speed α_{opt}, which can minimize the energy consumption while guaranteeing schedulability, is derived from the lower bound in Expression (6.1). This α_{opt}, however, often may not be supported by the system. Example 6.7.2 shows such a case. Recall that by Def. 6.5.1, α_{opt} corresponds to the optimal frequency F_{opt} that can guarantee schedulability. Although running *constantly* at this optimal frequency F_{opt} may not be supported by the system, it is possible to achieve this optimal frequency value *in the long run*, exploiting a "Pulse Width Modulation" (PWM) scheme, where the system switches periodically between two supported frequencies, F_L and F_H, with $F_L < F_{\text{opt}} < F_H$. In particular, we consider the PWM technique presented in [B$^+$09], which we summarize in the following subsection. Note that other research works have considered the problem of providing the optimal processor speed in processors that only provide a discrete set of frequencies. See, for instance, [IY98]. However, in our work we choose the technique proposed in [B$^+$09] because it is accurate (it considers the overheads incurred during voltage/frequency switching) and it uses the real-time periodic task model (contrary to [IY98]).

PWM Scheme

The PWM scheme presented in [B$^+$09] is aimed at uniprocessor systems with HRT constraints. The execution of the scheme at run-time is sketched in Figure 6.6. The PWM scheme switches periodically between a lower frequency F_L and a higher frequency F_H. The period of the PWM scheme is denoted by P.

The duration of the interval of the low-frequency (high-frequency) mode is Q_L (Q_H). Note that $Q_L + Q_H = P$. Moreover, [B$^+$09] defines $\lambda_L = \frac{Q_L}{P}$ and $\lambda_H = \frac{Q_H}{P}$, the fraction of time spent running at low and high modes, respectively.

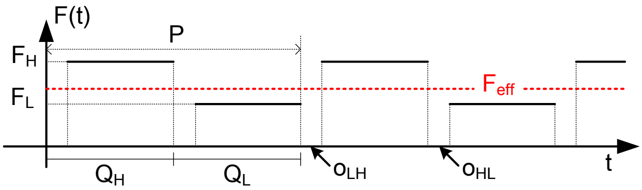


Figure 6.6: *Execution of the PWM scheme. The scheme switches periodically between a lower frequency* F_L *and a higher frequency* F_H*, in order to provide an effective frequency* F_{eff} *in the long run. The period of the PWM scheme is denoted by* P*.*

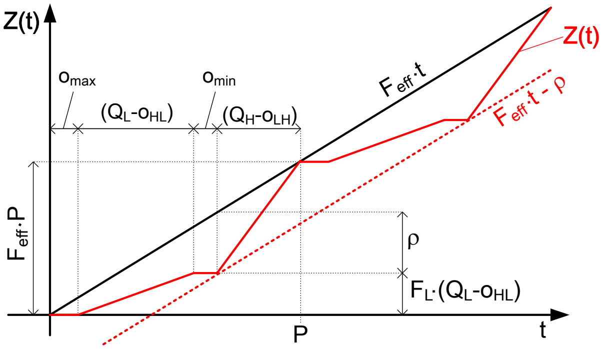


Figure 6.7: *Supply function* $Z(t)$*. This function provides the minimum number of cycles executed by the processor under the PWM scheme in every time interval of length* t*.*

As shown in Figure 6.6, the scheme considers time overheads due to frequency switching. These overheads are denoted by o_{LH} for transitions between lower to higher frequencies, and by o_{HL} for the opposite transitions. In addition, [B$^+$09] denotes the amount of clock cycles lost during frequency transitions as $\Delta_{\mathrm{LH}} = F_L \cdot o_{\mathrm{HL}} + F_H \cdot o_{\mathrm{LH}}$.

Under the above definitions, the effective frequency obtained by running the processor at F_L for Q_L time and F_H for Q_H time is given by expression (8) in [B$^+$09]:

$$F_{\mathrm{eff}} = \lambda_L F_L + \lambda_H F_H - \Delta_{\mathrm{LH}}/P \tag{6.10}$$

To ensure HRT execution on the system, in their analysis the authors leverage the processor supply function $Z(t)$, defined as the *minimum number of cycles that the processor can provide in every interval of length* t. From the parameters of the PWM scheme, $Z(t)$ is depicted with a solid red line in Figure 6.7, with $o_{\max} = \max\{o_{\mathrm{LH}}, o_{\mathrm{HL}}\}$ and $o_{\min} = \min\{o_{\mathrm{LH}}, o_{\mathrm{HL}}\}$. Function $Z(t)$ is zero in $[0, o_{\max}]$; grows linearly with slope F_L in $[o_{\max}, o_{\max} + Q_L - o_{\mathrm{HL}}]$; stays constant in $[o_{\max} + Q_L - o_{\mathrm{HL}}, o_{\max} + Q_L - o_{\mathrm{HL}} + o_{\min}]$; finally, grows with slope F_H until the end of the period P. Note that $Z(t)$ is periodic with period P.

Tardiness Bounds Derivation

In our approach, we leverage the processor supply function $Z(t)$ to derive tardiness bounds for any task running on a processor under our EDF-ssl scheduling algorithm. These tardiness bounds are given by the following theorem.

Theorem 6.7.2. *Consider a processor π_k, on which the PWM scheme described in Section 6.7.4 is applied to obtain an effective frequency F_{eff}. Assume that two migrating tasks, τ_i (with WCET C_i) and τ_j (with WCET C_j), are assigned to π_k. Then, jobs of fixed and migrating tasks released on π_k may incur a tardiness of at most*

$$\Delta^{\pi_k}_{PWM} = \frac{2(C_i + C_j)}{\alpha_{eff}} + \frac{\rho}{F_{eff}} \tag{6.11}$$

with $\rho = (F_{eff} - F_L)Q_L + F_L \cdot o_{HL} + F_{eff} \cdot o_{LH}$ and α_{eff} is derived from F_{eff} using Definition 6.5.1.

Proof. To prove Theorem 6.7.2, we first derive a lower bound for $Z(t)$. We define the following parameter:

$$\rho = max_{t \in \mathbb{R}^+}\{F_{\text{eff}} \cdot t - Z(t)\}$$

which represents the maximum difference between the "optimal" number of cycles, provided in the interval $[0, t]$ by a processor running at F_{eff}, and $Z(t)$. From Figure 6.7 we get:

$$\rho = (F_{\text{eff}} - F_L)Q_L + F_L \cdot o_{\text{HL}} + F_{\text{eff}} \cdot o_{\text{LH}} \tag{6.12}$$

from which we can express a lower bound for $Z(t)$ as:

$$\check{Z}(t) = F_{\text{eff}} \cdot t - \rho \tag{6.13}$$

$\check{Z}(t)$ is depicted in Figure 6.7 with a dashed red line. We can then express $Z(t)$ as $Z(t) = \check{Z}(t) + e(t)$, with $e(t) \geq 0, \forall t \geq 0$.

Now, we follow the proof of Theorem 6.7.1. This time, the instant t_c is defined as $\mathbf{t_c} = t_d + \Delta^{\pi_k}_{\text{PWM}}$, and we assume that a certain job $\tau_{q,l}$ does not complete by time t_c. The definitions of $\mathbf{t_0}$ and $\mathbf{t_d}$ are the same as in the proof of Theorem 6.7.1. The demand from fixed and migrating tasks, expressed in clock cycles, is still bounded by Expression (6.7). However, we have to change the left-hand side of Inequality (6.8) with $Z(t_c - t_0)$, obtaining:

$$F_{\text{eff}} \cdot (t_c - t_0) - \rho + e(t_c - t_0) < F_{\max}\left(\alpha_{\text{eff}}(t_d - t_0) + 2(C_i + C_j)\right) \tag{6.14}$$

Since $F_{\text{eff}} = \alpha_{\text{eff}} \cdot F_{\max}$, dividing both sides by $\alpha_{\text{eff}} \cdot F_{\max}$ we get:

$$(t_c - t_0) - \frac{\rho - e(t_c - t_0)}{F_{\text{eff}}} < (t_d - t_0) + \frac{2(C_i + C_j)}{\alpha_{\text{eff}}}$$

therefore:

$$(t_c - t_d) < \frac{2(C_i + C_j)}{\alpha_{\text{eff}}} + \frac{\rho - e(t_c - t_0)}{F_{\text{eff}}} = \Delta^{\pi_k}_{\text{PWM}} - \frac{e(t_c - t_0)}{F_{\text{eff}}} \tag{6.15}$$

Even with $e(t_c - t_0) = 0$, which represents the worst case, Expression (6.15) contradicts the definition of t_c, therefore Theorem 6.7.2 holds. ■

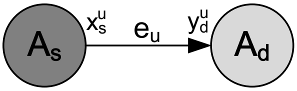


Figure 6.8: *SDF actors A_s (source) and A_d (destination) with dependency over edge e_u. The production rate of A_s over e_u is denoted by x_s^u. The consumption rate of A_d over e_u is denoted by y_d^u.*

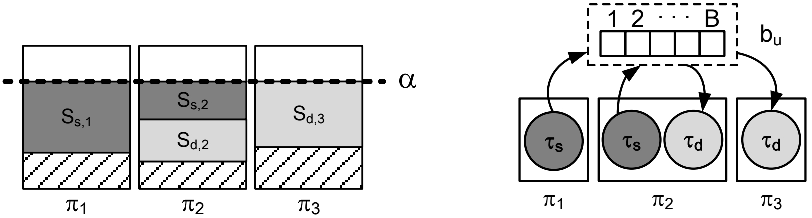
Note that by Equation (6.11) it follows that tardiness can be experienced even on processors with no migrating tasks, given the fact that the term ρ depends only on the parameters of the PWM scheme.

6.8 Start times and buffer sizes under EDF-ssl

As mentioned in Section 6.6, the analysis described in this chapter leverages the results of Chapter 5 to schedule the applications using our EDF-ssl soft real-time scheduler. However, compared to the EDF-fm scheduling algorithm used in Chapter 5, our EDF-ssl is different in certain aspects. In order to maintain the scheduling analysis valid for our proposed EDF-ssl, we must take into account these differences between our EDF-ssl and EDF-fm.

Let us consider the data-dependent actors A_s (source) and A_d (destination) shown in Figure 6.8. We recall that in our analysis A_s and A_d are converted into two periodic tasks τ_s and τ_d using the methodology described in Section 2.3. Assume, for instance, that the system runs at a certain constant normalized speed α, and both τ_s and τ_d are assigned to the processors as migrating tasks, with the share assignment shown in Figure 6.9(a). Shares $s_{s,1}$ and $s_{s,2}$ of τ_s are assigned to π_1 and π_2, whereas shares $s_{d,2}$ and $s_{d,3}$ of τ_d are assigned to π_2 and π_3. In Figure 6.9(a), the dashed areas in each processor represent processor utilization assigned to tasks other than τ_s and τ_d. These other tasks are assumed to be of fixed type (i.e., not migrating). Since π_1 and π_3 run only one migrating tasks, by Equation (6.3) we derive the following tardiness bounds: $\Delta^{\pi_1} = 2C_s/\alpha$, $\Delta^{\pi_2} = 2(C_s + C_d)/\alpha$, $\Delta^{\pi_3} = 2C_d/\alpha$, where C_s and C_d are the WCETs of τ_s and τ_d, respectively. It follows that under our EDF-ssl jobs of the same migrating task have different tardiness bounds, depending on which processor the jobs are released. For instance, jobs of τ_s will incur a tardiness of at most Δ^{π_2} when released on π_2, and Δ^{π_1} when released on π_1, with $\Delta^{\pi_2} > \Delta^{\pi_1}$. By contrast, under EDF-fm used in Chapter 5, jobs of a migrating task experience no tardiness at all, because tardiness can only be experienced by fixed tasks. In addition, under our EDF-ssl jobs of the same migrating task can execute in parallel. This cannot happen under EDF-fm.

In the remainder of this section we define a way to guarantee a correct schedule of τ_s and τ_d, with no buffer underflow or overflow, under our EDF-ssl. As shown in Figure 6.9(b), we assume that processors running communicating tasks have access to a shared memory where data communication buffers are allocated. Note that our approach allows data and instruction memory of all processors to be completely distributed, therefore contention can only occur when accessing the shared communication memory. In Figure 6.9(b), buffer b_u of size B implements the communication



(a) Considered share assignment of τ_s and τ_d.

(b) Scheme of the communication between τ_s and τ_d over b_u, according to the share assignment given in sub-figure (a).

Figure 6.9: *Analysis of the communication between data-dependent actors when both source (τ_s) and destination (τ_d) actors are implemented as migrating tasks. In sub-figure (a), shares $s_{s,1}$ and $s_{s,2}$ of τ_s are assigned to π_1 and π_2, whereas shares $s_{d,2}$ and $s_{d,3}$ of τ_d are assigned to π_2 and π_3. Sub-figure (b) represents the access to the shared communication buffer b_u by jobs of τ_s and τ_d. Under our* **EDF-ssl**, *jobs of τ_s may execute in parallel if released onto different processors and be affected by a different tardiness depending on which processor execute them. The same holds for jobs of τ_d. It follows that jobs of τ_s may write out-of-order to buffer b_u. Similarly, jobs of the destination task τ_d may read from b_u out-of-order. This phenomenon is taken into account by our analysis.*

over edge e_u of Figure 6.8. Our analysis to guarantee a correct scheduling of τ_s and τ_d comprises two parts. First, we guarantee valid start times of τ_s and τ_d and buffer size B by adapting the analysis in Chapter 5 to our EDF-ssl. Second, we define a pattern that τ_s and τ_d use when reading/writing from/to b_u to ensure functional correctness. These two parts are described below.

Part 1 - Valid start times and buffer sizes. As mentioned earlier, under our EDF-ssl jobs of the same migrating task can have different tardiness bounds, if released on different processors. According to Definition 2.2.10 on page 38, the tardiness bound Δ_i of a certain task τ_i must be valid for all its jobs. Therefore, we set the value of Δ_i to the maximum tardiness bound among the processors which are assigned (non-zero) shares of τ_i, as follows:

$$\Delta_i = \begin{cases} \max_{k \mid s_{i,k} > 0}\{\Delta^{\pi_k}\} & \text{under fixed processor speed} \\ \max_{k \mid s_{i,k} > 0}\{\Delta^{\pi_k}_{PWM}\} & \text{under PWM scheme} \end{cases} \tag{6.16}$$

where Δ^{π_k} and $\Delta^{\pi_k}_{PWM}$ are the tardiness bounds calculated for processor π_k under fixed processor speed and under the PWM scheme described in Section 6.7.3 and Section 6.7.4, respectively. For each processor π_k, Δ^{π_k} and $\Delta^{\pi_k}_{PWM}$ are obtained using Equation (6.3) and Equation (6.11), respectively. Finally, in Equation (6.16), $s_{i,k}$ represents the share of τ_i on π_k.

By using the tardiness bound Δ_i expressed by Equation (6.16), we can represent the ALAP completion schedule (see Definition 5.5.1 in Section 5.5.1) of actor A_i (corresponding to task τ_i) as a fictitious actor $\tilde{A}_i$, which has the same period as A_i, no tardiness, and start time $\tilde{S}_i = S_i + \Delta_i$. From Equation (6.16) it follows that at run time

Algorithm 4: Write pattern of job j of source task τ_s.

```
  Input: Number of produced tokens $x_s^u$, job index $j$, buffer size $B$.
1 bgn = $[(x_s^u \cdot j) \mod B] + 1$;
2 end = $(x_s^u \cdot (j+1)) \mod B$;
3 if bgn < end then
4     write $x_s^u$ tokens from $b_u$[bgn] to $b_u$[end]
5 else
6     write (bgn-B+1) tokens from $b_u$[bgn] to $b_u$[B];
7     write remaining tokens from $b_u$[1] to $b_u$[end];
```

any invocation $A_{i,j}$ of actor A_i will never be completed later than the corresponding invocation $\tilde{A}_{i,j}$ of actor $\tilde{A}_i$, regardless of which processor is executing that invocation. Therefore, the analysis for start times and buffer sizes in the presence of tardiness described in Chapter 5 can be applied considering the tardiness bounds given by Equation (6.16) and it is correct for our EDF-ssl.

Part 2 - Reading/writing pattern to/from b_u. Let us focus on the source actor A_s in Figure 6.8, and let assume the share assignment shown in Figure 6.9(a). Under our EDF-ssl, jobs of τ_s, which correspond to invocations of A_s, may execute in parallel if released onto different processors. Moreover, as mentioned earlier, jobs of τ_s may experience different tardiness, depending on which processor the job is released. It follows that jobs of τ_s may write out-of-order to buffer b_u in Figure 6.9(b). This is because job $\tau_{s,k+a}$, for some $a > 0$, may finish before job $\tau_{s,k}$ if they are released on different processors. Similarly, jobs of the destination task τ_d may read from b_u out-of-order.

In the scenario described above, it is clear that b_u is not a First-in First-out (FIFO) buffer. Thus, every job of τ_s/τ_d (invocation of A_s/A_d) must know *where* it has to write/read to/from b_u. **Part 1** of our analysis (described above) ensures that B, the size of b_u, is large enough to guarantee that tokens produced by τ_s will never overwrite locations which contain tokens still not consumed by τ_d.

Then, given x_s^u, the amount of tokens produced on e_u by every job of τ_s, we enforce that job j of τ_s (with $j \in \mathbb{N}_0$) writes tokens to b_u in the memory locations that would be written if the jobs of τ_s wrote in-order to a FIFO buffer of size B implemented as a circular buffer. This writing pattern is implemented in Algorithm 4. Lines 5-7 in the algorithm handle the case in which the x_s^u tokens are "wrapped" in the buffer. Note that by replacing, in Algorithm 4, x_s^u with y_d^u and write operations with read operations we obtain the reading pattern corresponding to job j of destination task τ_d.

Finally, note that under EDF-ssl, as in EDF-os, the job index j is maintained by the scheduling algorithm in order to release a migrating task on the right processor, in order to follow the job releasing rules mentioned in Section 6.7.2. Therefore, the value of j, used in Algorithm 4, is known when a job is executed on a processor.

6.9 Evaluation

In this section, we evaluate the effectiveness of our EDF-ssl semi-partitioned scheduling approach in terms of energy savings. We compare our results with the heuristic-based partitioned approach which guarantees the most balanced distribution of utilization of tasks among the available processors, and therefore the least energy consumption, as shown in [AY03]. The authors in [AY03] also show that the most balanced distributions are derived when Worst Fit Decreasing (WFD) heuristic (see Section 2.2.6) is used to determine the assignment of tasks to processors. Each processor then schedules the tasks assigned to it using a local EDF scheduler. In the rest of this section, we will refer to this partitioned approach with the acronym PAR. Note that under PAR all tasks meet their deadlines. By contrast, our proposed semi-partitioned approach will be denoted in the rest of this section with SP when fixed processor speed is used, and with PWM when the periodic speed switching scheme is adopted. Note that although under our approach tasks may experience tardiness, this has no effect on the guaranteed throughput, which remains constant among all the considered approaches (PAR, SP, PWM). However, task tardiness has an impact on buffer sizes and start times of tasks (and, in turn, on the latency of applications), as described in Chapter 5. Note that although the PAR approach provides HRT guarantees to all tasks in the system, whereas both SP and PWM only provide SRT guarantees, our comparison remains fair. This is because:

- As shown in Chapter 5, also SP and PWM can guarantee HRT behavior at the input/output interfaces with the environment, although some of the tasks of the application may experience tardiness.
- Both SP and PWM, adopting the soft real-time scheduling technique of Chapter 5, guarantee the same throughput as PAR.

These two conditions are sufficient for the kind of applications that we consider, in which throughput constraints are more relevant than application latency and memory overheads. Note also that in our scheduling framework, since actors are released strictly periodically, the application latency is the elapsed time between the start of the first firing of the input actor and the worst-case completion of the first firing of the output actor.

6.9.1 Power Model

As mentioned in Section 6.5, we consider homogeneous multiprocessor systems, in which any core can be either idle or running at a global (normalized) speed α. We assume that the system supports a discrete set of operating voltage/frequency modes. In our experiments, we refer to the operating modes of a modern System-on-Chip, the OMAP 4460, as in [ZR13]. This SoC comprises two ARM Cortex-A9 cores that can operate at $\Phi_{\text{A9}} = \{0.350, 0.700, 0.920, 1.200\}$ GHz, at a supply voltage of $\{0.83, 1.01, 1.11, 1.27\}$ V, respectively. From Φ_{A9} we can derive the set of supported normalized speed:

$$\mathcal{NS}_{\text{A9}} = \{0.292, 0.583, 0.767, 1.0\}$$

We use the power model of a similar dual Cortex-A9 core system, considered in [P$^+$13], which we normalize to a single core:

$$p_{\text{cpu}} = p_{\text{dyn}} + p_{\text{sta}} = (0.223V_{\text{cpu}}^2 F_{\text{cpu}}) + (K_1 V_{\text{cpu}} + K_2) \tag{6.17}$$

where $K_1 = 0.08965$, $K_2 = 0.07635$, V_{cpu} represents the voltage supplied to the CPU in Volts, and F_{cpu} represent the CPU frequency in GHz. Note that the power model given in Expression (6.17) has been validated with actual power measurements in [P$^+$13]. The model comprises dynamic power p_{dyn} and static power p_{sta}, and the value of p_{dyn} assumes that the core is fully utilized. Note also that Expression (6.17) assumes that the processor runs at one of the supported normalized speeds $\alpha_i \in \mathcal{NS}_{\text{A9}}$. From this α_i, we can derive the processor frequency $F_{\text{cpu}} = F_i$ by Definition 6.5.1. Similarly, to a normalized speed α_i corresponds an unique voltage level V_{cpu}. Therefore, the power consumption p_{dyn} and p_{sta} depend uniquely on α_i. We make this relation explicit by using the notation $p_{\text{dyn}}(\alpha_i)$, $p_{\text{sta}}(\alpha_i)$, and $p_{\text{cpu}}(\alpha_i)$.

6.9.2 Energy per Iteration Period

Based on the power model expressed by Equation (6.17), we now proceed by deriving the energy consumption under PAR, SP, and PWM. In particular, we derive the energy consumed by the system during one *iteration period* (H) of the graph (recall Equation (2.28) on page 44). Note that the iteration period of the graph is *the same* and *constant* among PAR, SP, and PWM, because the periods of all tasks do not change depending on the considered scheduling approach. Note also that, *regardless of the application latency*, every task τ_i executes q_i times during one iteration period H (recall, again, Equation (2.28)). We assume that α is sufficient to guarantee schedulability, therefore $\alpha \geq \sigma_k$, for any active processor π_k. In the following, we denote the number of active cores with M_{ON}.

Static energy of (PAR, SP). Both these approaches run at a fixed speed α_i, and the static energy consumed in one iteration period H is given by:

$$E_{\text{sta}}^{H,\text{FIX}} = H \cdot p_{\text{sta}}^{\text{FIX}} = H \cdot M_{\text{ON}} \cdot p_{\text{sta}}(\alpha_i) \tag{6.18}$$

Dynamic energy of (PAR, SP). We derive the dynamic energy consumption in one iteration period H. During one iteration period, each task τ_j executes $q_j = H/T_j$ times. Each worst-case execution takes C_j/α_i time, at dynamic power $p_{\text{dyn}}(\alpha_i)$. Therefore, the dynamic energy consumed by task τ_j during one iteration period H is:

$$E_{\text{dyn}}^{H,\text{FIX}}(\tau_j) = q_j \frac{C_j}{\alpha_i} p_{\text{dyn}}(\alpha_i) \tag{6.19}$$

From Equation (6.19) we derive the dynamic energy consumed in one iteration period H by the whole task set as follows:

$$E_{\text{dyn}}^{H,\text{FIX}} = \sum_{\tau_j \in \Gamma} q_j \frac{C_j}{\alpha_i} p_{\text{dyn}}(\alpha_i) = \frac{p_{\text{dyn}}(\alpha_i)}{\alpha_i} \sum_{\tau_j \in \Gamma} q_j C_j \tag{6.20}$$

Total energy of (PAR, SP). From Equation (6.18) and (6.20) we derive the total energy consumed during one iteration period H under (PAR, SP) by:

$$E_{\text{tot}}^{H,\text{FIX}} = H \cdot M_{\text{ON}} \cdot p_{\text{sta}}(\alpha_i) + \frac{p_{\text{dyn}}(\alpha_i)}{\alpha_i} \sum_{\tau_j \in \Gamma} q_j C_j \tag{6.21}$$

Total energy of PWM. Under PWM, the system switches periodically between normalized speeds α_L and α_H to guarantee a certain α_{eff} in the long run. Therefore, we cannot use Equation (6.21) to model the energy consumption per iteration period under the PWM scheme, because that expression is only valid when the system runs constantly at one of the supported normalized speeds α_i. For the sake of clarity, we will denote $p_{\text{cpu}}(\alpha_L)$ and $p_{\text{cpu}}(\alpha_H)$, obtained from Equation (6.17), with p_L and p_H, respectively. In this scenario, the total power of a single core of the system is provided by expression (9) in [B$^+$09], reported below.

$$p_{\text{cpu}}^{\text{PWM}} = \lambda_L p_L + \lambda_H p_H + E_{\text{SW}}/P \tag{6.22}$$

where $E_{\text{SW}} = e_{\text{LH}} - p_H \cdot o_{\text{LH}} + e_{\text{HL}} - p_L \cdot o_{\text{HL}}$, which represents the energy wasted during two speed transitions. The terms λ_L, λ_H, o_{LH}, o_{HL}, P, are parameters of the PWM scheme defined in Section 6.7.4, whereas e_{LH} and e_{HL} represent the *energy* overhead incurred in the speed transition from α_L to α_H and vice versa. We assume that $e_{\text{LH}} = e_{\text{HL}} = 1\,\mu J$ and $o_{\text{LH}} = o_{\text{HL}} = 10\,\mu s$. Note that these values are compatible with the findings in [P$^+$13], where the time and energy overheads due to frequency switching have been derived using actual measurements. Now, given the number of active cores M_{ON}, we can express the total energy per iteration period H under PWM as:

$$E_{\text{tot}}^{H,\text{PWM}} = H \cdot M_{\text{ON}} \cdot p_{\text{cpu}}^{\text{PWM}} \tag{6.23}$$

Note that Equation (6.23) depends on Equation (6.22), which in turn depends on the parameters of the PWM scheme. In particular, we have to find an appropriate value for the PWM scheme period P. Since we assume that speed changes can only happen at the granularity of the operating system tick (which has period T_{OS}), we enforce P to be a multiple of T_{OS}. From Equation (6.10), we derive the shortest P, multiple of T_{OS}, that makes the overhead-induced clock cycles loss less than $\epsilon = 0.01$ times the desired F_{opt}. Thus, $P \geq \Delta_{\text{LH}}/(\epsilon F_{\text{opt}})$. Given P, we find the shortest Q_H, multiple of T_{OS}, that guarantees an effective frequency F_{eff} greater than or equal to F_{opt} (from Equation (6.10); note that $Q_L = P - Q_H$). At this point, all the parameters of the PWM scheme are known and the total energy consumption per iteration period can be derived using Equation (6.23).

6.9.3 Experimental Results

In Table 6.1, we show the results obtained using the considered approaches (PAR, SP, PWM) on a set of real-life applications (see column App). For each application, column U_Γ reports the cumulative utilization of the corresponding task set. In addition,

Table 6.1: *Comparison of different share allocation/scheduling approaches.*

App	U_Γ	R [tkns/s]	$\hat{M}$	PAR				SP				PWM			
				M^o_{PAR}	TM_{PAR}[kB]	L_{PAR}[ms]	E^H_{PAR}[J]	M^o_{SP}	$\frac{TM_{SP}}{TM_{PAR}}$	$\frac{L_{SP}}{L_{PAR}}$	$\frac{E^H_{SP}}{E^H_{PAR}}$	α_{opt}	$\frac{TM_{PWM}}{TM_{PAR}}$	$\frac{L_{PWM}}{L_{PAR}}$	$\frac{E^H_{PWM}}{E^H_{PAR}}$
DCT	2.43	12000	4	3	22.7	0.667	$9.67 \cdot 10^{-5}$	4	1.37	1.62	0.77	0.61	2.18	3.85	0.66
			8	3	22.7	0.667	$9.67 \cdot 10^{-5}$	5	1.64	2.22	0.64	0.49	3.02	5.95	0.61
			12	3	22.7	0.667	$9.67 \cdot 10^{-5}$	9	3.14	5.25	0.37	0.27	na	na	na
JP2	1.23	333.33	4	2	114	17.5	$1.78 \cdot 10^{-3}$	3	1.42	1.28	0.62	0.41	1.9	3.45	0.51
			8	2	114	17.5	$1.78 \cdot 10^{-3}$	5	2.28	2.13	0.36	0.25	na	na	na
			12	2	114	17.5	$1.78 \cdot 10^{-3}$	5	2.28	2.13	0.36	0.25	na	na	na
MJPEG	1.22	6000	4	2	62.7	0.833	$7.50 \cdot 10^{-5}$	3	1.31	1.42	0.84	0.41	1.94	4.73	0.69
			8	2	62.7	0.833	$7.50 \cdot 10^{-5}$	5	1.8	3.17	0.47	0.24	na	na	na
			12	2	62.7	0.833	$7.50 \cdot 10^{-5}$	5	1.8	3.17	0.47	0.24	na	na	na
MPEG2	6.81	12000	8	8	345	1.50	$2.70 \cdot 10^{-4}$	7	1.44	1.55	0.99	0.97	1.84	2.00	1.00
			12	8	345	1.50	$2.70 \cdot 10^{-4}$	12	2.01	2.21	0.63	0.57	2.45	3.01	0.65
TDE	6.28	3000	8	7	840	9.67	$9.90 \cdot 10^{-4}$	7	1.00	1.00	1.00	0.9	1.42	1.42	0.94
			12	9	840	9.67	$7.60 \cdot 10^{-4}$	11	1.68	1.76	0.83	0.57	1.68	1.78	0.84

column R shows the throughput obtained for each application. For a certain application, given x^{out} which is the number of tokens produced by its output actor at every invocation, the application throughput can be computed as $R = x^{\text{out}}/T_{\text{out}}$ where T_{out} is the period of the output actor. Therefore, the application throughput R is given in tokens per second [tkns/s]. Note that the throughput R and the total utilization U_Γ of each application remain *constant* among all considered allocation/scheduling approaches (PAR, SP, PWM). The reason is that the WCET of each task τ_i, derived using Equation (2.26) on page 43, does not depend on the actual assignment of tasks to processors, because it considers the worst-case communication time among all possible assignments of tasks.

Each row in Table 6.1 corresponds to results obtained considering a system composed of $\hat{M}$ available cores, with $\hat{M} \in \{4, 8, 12\}$. Note that column $\hat{M}$ shows only meaningful values, those which satisfy $\hat{M} \geq \lceil U_\Gamma \rceil$. For each of the considered approaches (PAR, SP, PWM), and for each value of $\hat{M}$, we consider each possible number of active processors M_{ON} in the range $[\lceil U_\Gamma \rceil, \hat{M}]$ and look for the lowest energy consumption, thereby exploring the design space exhaustively. For every value of M_{ON} in that range, we follow a different procedure depending on the considered approach.
In PAR, we simply assign the utilization of tasks to the M_{ON} active cores using the WFD heuristic. Then, if WFD is successful, we derive the lowest $\alpha_i \in \mathcal{NS}_{\text{A9}}$ which guarantees schedulability ($\min_{\alpha_i \in \mathcal{NS}_{\text{A9}}}\{\alpha_i \geq \sigma_k, \forall \text{ active } \pi_k\}$). Knowing M_{ON} and α_i, we determine the total energy per iteration period E^H_{PAR} by Equation (6.21).
In SP, we find the necessary minimum speed $\alpha_{\text{opt}} = U_\Gamma / M_{\text{ON}}$. We round this speed value to the closest greater or equal value in $\mathcal{NS}_{\text{A9}}$, which we denote with α_i. We run Algorithm 3 with this speed value α_i and $M = M_{\text{ON}}$. If Algorithm 3 is successful, we determine the total energy per iteration period E^H_{SP} by Equation (6.21) with the considered α_i and M_{ON}.
In PWM, we calculate $\alpha_{\text{opt}} = U_\Gamma / M_{\text{ON}}$ and we run Algorithm 3 with speed value α_{opt} and $M = M_{\text{ON}}$. If Algorithm 3 is successful, we use $\alpha_H = \min_{\alpha_i \in \mathcal{NS}_{\text{A9}}}\{\alpha_i \geq \alpha_{\text{opt}}\}$ and $\alpha_L = \max_{\alpha_i \in \mathcal{NS}_{\text{A9}}}\{\alpha_i \leq \alpha_{\text{opt}}\}$ and derive the total energy per iteration period E^H_{PWM} by Equation (6.23).

For each valid task share assignment, we derive earliest start times of actors and buffer size requirements by using the formulas derived in Chapter 5 with, for each task τ_l, one of the following tardiness bound values: $\Delta_l = 0$ in PAR; Δ_l obtained by Equation (6.16) in SP and PWM.

At the end of the design space exploration, for PAR and SP, we report in Table 6.1 the values of $M_{\text{ON}} \in [\lceil U_\Gamma \rceil, \hat{M}]$ that yielded to the lowest energy consumption. For PAR (SP), these values are shown in column M^o_{PAR} (M^o_{SP}). Note that the optimal values of M_{ON} for PWM are identical to M^o_{SP}, therefore they are not included.

In the following discussion, we will identify rows in Table 6.1 with the couple $(\text{App}, \hat{M})$. For each of these rows, under PAR, the table shows: the optimal number of active processors M^o_{PAR}; the total memory requirement TM_{PAR} (including code, stack, buffers); the application latency L_{PAR}, calculated using the latency analysis described in Section 4.7 of [Bam14]; the energy consumption E^H_{PAR}. In particular, the total memory requirement in the PAR approach is calculated as follows.

$$\mathrm{TM}_{\mathrm{PAR}} = \sum_{i=1}^{N} \mathrm{CSS}(\tau_i) + \sum_{i=1}^{|E|} b_u^{HRT} \tag{6.24}$$

where N is the number of tasks, $\mathrm{CSS}(\tau_i)$ is the code and stack size of task τ_i (which represents actor A_i of the input CSDF graph G), E is the set of edges in G, b_u^{HRT} is the size of the buffer that implements the communication over edge e_u. The value of b_u^{HRT} assumes no task tardiness and is obtained using Equation (2.31) on page 46.

By contrast, for the semi-partitioned approach SP, the total memory requirement $\mathrm{TM}_{\mathrm{SP}}$ is derived using the following expression.

$$\mathrm{TM}_{\mathrm{SP}} = \sum_{i=1}^{M_{\mathrm{SP}}^{o}} \sum_{\tau_j \in \Gamma_i} \mathrm{CSS}(\tau_j) + \sum_{i=1}^{|E|} b_u^{SRT} \tag{6.25}$$

where M_{SP}^{o} is the number of processors (derived in the design space exploration), Γ_j is the set of tasks with non-zero shares on processor π_j, and b_u^{SRT} is the size of the buffer that implements the communication over edge e_u, calculated using Equation (5.2). Note that Equation (6.25) differs from Equation (6.24) because in the SP approach a task can have shares on different processors. In addition, in order to derive the application latency under SP, denoted by L_{SP}, we use the analysis in Section 4.7 of [Bam14], considering the task start times obtained by our SRT approach. Then, we add to that latency value the tardiness (which can be potentially null) of the output actor of the application.

For the PWM approach, the total memory requirement $\mathrm{TM}_{\mathrm{PWM}}$ and application latency L_{PWM} are derived following the same procedure used for the SP approach.

We see from Table 6.1 that SP consumes significantly lower energy than PAR, see column $E_{\mathrm{SP}}^{H}/E_{\mathrm{PAR}}^{H}$. On average, we obtain an energy saving of 36%. The energy saving goes up to 64%, see row $(\mathrm{JP2}, 8)$. These energy savings, however, come at a cost. The total memory requirements (see column $\mathrm{TM}_{\mathrm{SP}}/\mathrm{TM}_{\mathrm{PAR}}$) and application latencies (see column $L_{\mathrm{SP}}/L_{\mathrm{PAR}}$) are increased. Memory requirements increase due to *i)* more task replicas (with their code and stack memory) needed by the semi-partitioned approach and *ii)* more buffers due to task tardiness. Similarly, application latency increases because task tardiness postpones the start times of the tasks of the application.

The rightmost part of Table 6.1 presents the results under PWM. It shows that this approach can provide higher energy savings compared to SP (compare columns $E_{\mathrm{PWM}}^{H}/E_{\mathrm{PAR}}^{H}$ and $E_{\mathrm{SP}}^{H}/E_{\mathrm{PAR}}^{H}$). The additional energy saving can grow up to 18% (see rows (JP2,4) and (MJPEG,4)) compared to SP. Rows with *na* (not applicable) values indicate that the value of α_{opt} (see the corresponding column) is lower than the minimum speed in $\mathcal{NS}_{\mathrm{A9}}$. Therefore, the PWM scheme is not applicable. Note that in three rows the value of $E_{\mathrm{PWM}}^{H}/E_{\mathrm{PAR}}^{H}$ is higher than $E_{\mathrm{SP}}^{H}/E_{\mathrm{PAR}}^{H}$. This means that, in those cases, PWM is less effective than SP. The largest inefficiency is obtained in row (MPEG2,12). In all these cases, the value of α_{opt} is extremely close to one of the speeds in $\mathcal{NS}_{\mathrm{A9}}$, therefore the energy overhead incurred by the PWM scheme renders

PWM disadvantageous. Finally, note that PWM incurs more total memory and latency overheads compared with SP, see columns $\text{TM}_{\text{PWM}}/\text{TM}_{\text{PAR}}$ and $L_{\text{PWM}}/L_{\text{PAR}}$. This is due to the higher number of task replicas, and higher values of tardiness, incurred under PWM.

Note that our experimental results, summarized in Table 6.1, evaluate our proposed approaches SP and PWM using PAR as a reference point. However, we also made a second comparison, by evaluating our SP and PWM against the results that can be obtained by using the EDF-os scheduling algorithm as a reference. We do not show the results of the comparison against EDF-os in a separate table because that table would be nearly identical to Table 6.1. This is because PAR and EDF-os achieve nearly the same results, for the following reason. Since our designs are aimed at achieving the maximum throughput of the considered applications, the utilization of at least one of the tasks of each application is close to one. In this scenario, as shown in Section 6.7, EDF-os cannot distribute the utilization of such "heavy" tasks on multiple processors, therefore the operating frequency of the system cannot be lowered without compromising the schedulability of the system. Because of this, EDF-os does not outperform the PAR approach in our experiments, with the exceptions of the (MPEG2,8) and (MPEG2,12) cases. In both these two cases, EDF-os requires 7 processors to schedule the tasks set (one processor less than PAR) which results in a slightly reduced total energy of $2.67 \cdot 10^{-4} J$ (compared to $E^{H}_{\text{PAR}} = 2.70 \cdot 10^{-4} J$). Since the difference between PAR and EDF-os involves only the MPEG2 benchmark, and is in fact minimal, we choose not to show explicitly in a separate table the comparison of our proposed SP and PWM against EDF-os to avoid redundancy.

6.10 Discussion

In this chapter, we have proposed EDF-ssl, a soft real-time semi-partitioned scheduling algorithm aimed at reducing the energy consumption of embedded multiprocessor streaming systems with throughput constraints. Our EDF-ssl exploits the presence of some stateless tasks in the application, allowing the execution of different jobs of the same task in parallel, and achieving an even distribution of the utilization of tasks among the available processors. In turn, this enables processors to run at a lower frequency, which yields to lower energy consumption.

As shown in Section 6.9, our semi-partitioned scheduling approach achieves significant energy savings compared to a purely partitioned scheduling approach and an existing semi-partitioned one, EDF-os. The energy savings are on average 36% (and up to 64%) when using the lowest frequency which guarantees schedulability and is supported by the system. By using a periodic frequency switching scheme that preserves schedulability, instead of this lowest supported fixed frequency, an additional energy saving up to 18% is obtained. Although the throughput of applications is unchanged by the proposed semi-partitioned approach, the mentioned energy savings come at the cost of increased memory requirements and latency of applications.

Chapter 7

Summary and Discussion

7.1 Thesis Summary

The improvements in the semiconductor technology and the demand from the industry to provide more and more advanced functionalities to the end user have lead to a sharp increase in the complexity of embedded multiprocessor systems on chip (MPSoCs). In order to exploit the parallelism available in MPSoCs, applications have to be decomposed in portions that can be executed in parallel. The de-facto solution to achieve this decomposition is to use parallel Models of Computation (MoCs) during system design. By using parallel MoCs, applications are divided into tasks (or processes) that can be executed in parallel. Each of these tasks is assigned to a certain processing element (PE) of the system. This assignment of tasks to processor is called *spatial scheduling*, or *task mapping*.

In the first part of this thesis, namely Chapters 3 and 4, we have proposed a *middleware layer* that lays in between the tasks of the applications and the operating system. Our proposed middleware allows to dynamically change the task mapping at runtime, i.e., it allows certain tasks to *migrate* from one PE of the system to another. The goal of our approach is to exploit the ability to migrate certain tasks in order to achieve system adaptivity. The middleware layer presented in Chapters 3 and 4 is aimed at best-effort systems and considers two main assumptions. The first assumption is that the application to be executed on the MPSoC is specified as a Polyhedral Process Network (PPN). The second assumption is that the MPSoC execution platform is based on a Network-on-Chip (NoC) communication infrastructure. Both of these assumptions are beneficial to our goal of achieving system adaptivity by allowing task migration.

In particular, in Chapter 3, we have described the first component of the middleware layer mentioned earlier. This component allows PPN processes to communicate on NoC-based MPSoCs with completely distributed memories. We propose and compare different approaches (referred to as *communication approaches*) to implement communication among PPN processes on NoCs. Our evaluation shows that one of the communication approaches achieves higher performance when mapping communication dominant applications to NoC-based MPSoCs. Most importantly, for our goal of allowing process migration, all of the proposed communication approaches guarantee correct communication among PPN processes even when the mapping of certain processes is changed at run-time.

Chapter 4 describes the second component of our proposed middleware layer. This component is in charge of performing the actual migration of the processes among PEs of the system. That is, it implements the process migration mechanism used by our middleware. Our proposed migration mechanism is based on one of the communication approaches described in Chapter 3, and guarantees the following two important properties.

1. It is *time predictable*, that is, when a migration is triggered, it will be completed within a certain time frame.
2. It allows a migration to be triggered at any time during the execution of a PPN process, except when the status of the input/output FIFO buffers and the iterator set of the process are updated. Note that these updates take negligible time compared to the total execution time of the PPN process.

In the second part of this thesis, namely Chapters 5 and 6, we have targeted hard real-time systems. To this end, we consider applications modeled as Cyclo-static Dataflow (CSDF) graphs and we have proposed two approaches to schedule such applications using a *semi-partitioned* scheduling algorithm. Similar to the approach presented in Chapters 3 and 4, semi-partitioned scheduling algorithms also allow certain tasks to migrate. However, in the approach proposed in Chapters 3 and 4 task migrations can occur at any time, triggered by an input event from the user or from the environment (e.g., a hardware fault). By contrast, under semi-partitioned schedulers task migrations follow a precise temporal and spatial pattern known at design-time.

Chapters 5 and 6 use the scheduling framework proposed in [BS11, BS12] as a basis and research driver. That scheduling framework converts an input application, specified as a CSDF graph, to a set of real-time periodic tasks. Then, by using any partitioned hard real-time scheduling algorithm on the derived task set, a designer can obtain in a fast and analytical way the minimum number of processors that guarantee the required application performance and the mapping of tasks to processors.

The approach proposed in Chapter 5 extends the scheduling framework of [BS11, BS12] by allowing also the *soft real-time, semi-partitioned* scheduling algorithm EDF-fm [ABD08] to schedule the periodic task set derived from the input application model. We recall that, by contrast, in the scheduling framework of [BS11, BS12] only hard real-time, partitioned schedulers are considered. In Chapter 5, we have shown that our semi-partitioned scheduling approach reduces the number of processors required to schedule certain applications, compared to a pure partitioned scheduling approach. However, our proposed semi-partitioned approach incurs an overhead in terms of memory requirements and latency of the application. As an additional contribution of Chapter 5, we have proposed a task allocation heuristic that tries to minimize the mentioned memory and latency overhead incurred by our semi-partitioned approach.

Finally, in Chapter 6, we have proposed a novel soft real-time (SRT) semi-partitioned scheduling algorithm, called EDF-ssl, that can be used instead of the EDF-fm scheduler employed in Chapter 5. EDF-ssl is designed to be used in combination with Voltage/Frequency Scaling (VFS) techniques, and exploits the presence of stateless tasks to achieve an even distribution of the utilization of tasks among the available processors and, in turn, improve the energy efficiency of the system. Our proposed semi-partitioned scheduling achieves the same throughput, at a significantly lower energy consumption, compared to a purely partitioned scheduling approach. However, the mentioned energy savings come at the cost of increased memory requirements and latency of applications.

7.2 Discussion

In Section 7.2.1 and Section 7.2.2, we provide examples of how the techniques presented in this thesis can be applied in practice to the design of embedded multiprocessor systems. In particular, Section 7.2.1 describes how the process migration mechanism proposed in Chapter 4 has been applied to an industrially-relevant case study within the EU FP7 project MADNESS [CGF^+11,MTR^+12,DCT^+13]. In addition, Section 7.2.2 explains how the semi-partitioned scheduling techniques proposed in Chapters 5 and 6 can be integrated within the existing Daedalus$^{\text{RT}}$ [BZNS12, Bam14] system-level design flow. Finally, in Section 7.2.3, we explain why we restricted the contributions of Chapters 3 to 6 to certain application models.

7.2.1 Assessing the migration mechanism in an industrially- relevant case study

In Chapter 4, we have presented our proposed process migration mechanism exploiting a PPN model with rather simple topology as a running example (see the upper part of Figure 4.2 on page 77). The topology of the case study considered in Section 4.6.1, an M-JPEG encoder, is also rather simple.

As a proof that our proposed process migration mechanism can handle more complex PPN topologies, we applied the migration technique presented in Chapter 4 to an industrially relevant case study, an H.264 decoder. The PPN model of this application is shown in Figure 7.1. This proof-of-concept has been carried out within the EU FP7 project MADNESS [CGF^+11,MTR^+12,DCT^+13] and showcased in a live demonstration at the project's booth at the DATE'13 conference [Mac13]. Hereafter, we will refer to the implemented live demonstration as *our demo*.

In order to describe the kind of process migrations allowed in our demo, we first provide an abstraction of the PPN topology shown in Figure 7.1. This abstracted PPN topology is given in Figure 7.2.

By comparing Figure 7.1 and Figure 7.2 we note that, in the latter figure, nodes *get_data* and *parser* have been merged into a single node, denoted by H_0. Each of the other nodes in Figure 7.1 is represented by one unique node in Figure 7.2 and denoted by H_1 to H_5.

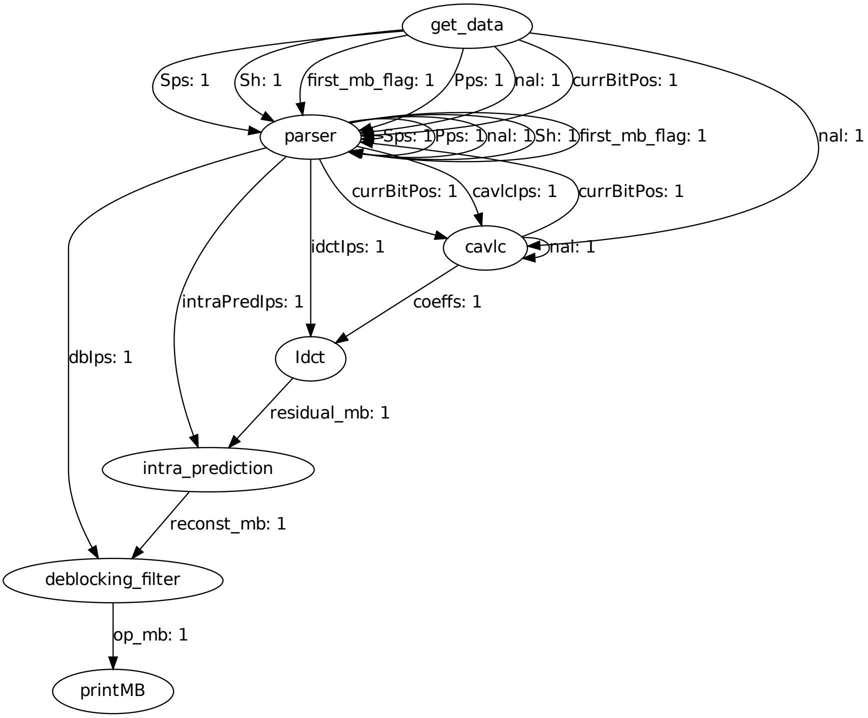


Figure 7.1: *PPN model of the H.264 decoder application.*

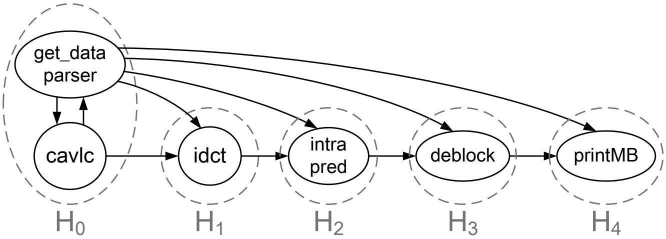


Figure 7.2: *Abstracted PPN specification of the H.264 decoder application. Compared to Figure 7.1, nodes get_data and parser are merged into a single node, H_0.*

The execution platform of our demo is represented in Figure 7.3. It consists of 6 tiles connected by the ×pipes NoC [BB04] and organized as a 2x3 mesh. This execution platform is implemented onto a Virtex-6 FPGA prototyping board.

In addition to the structure of the execution platform, Figure 7.3 shows the mapping of the *replicas* of the PPN processes which comprise the H.264 application. A process can be executed on a tile only if a replica of that process is allocated to that tile. In Figure 7.3, process replicas which are active at system startup are filled in dark

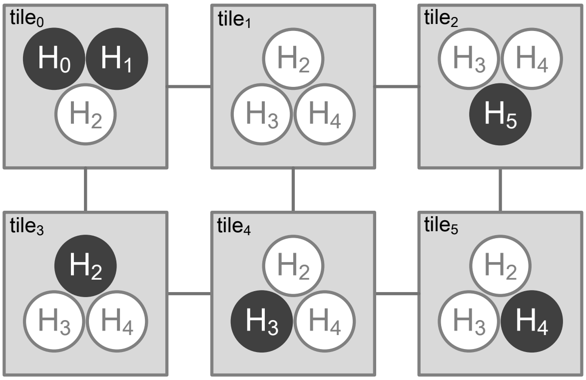


Figure 7.3: *Structure of the execution platform used in our demo and allocation of process replicas to tiles. Process replicas which are active at system startup are filled in dark gray. Input and output interfaces are not shown.*

gray. All the other replicas are inactive, but ready to by activated in case a process migration requires so.

Our demo includes one input and one output interface (which are not shown in Figure 7.3). The input interface allows the user to provide inputs to the system by pushing the buttons available on the FPGA prototyping board. Each button corresponds to one tile of the system. When a button get pressed by the user, the system is requested to disable the corresponding tile. The output interface visualizes the frames generated by the H.264 decoder on an external screen.

Figure 7.4 shows an example of a process migration performed in our demo. In this example, the user requires the system to deactivate tile_3. Then, the resource manager (RM) which is executed on tile_2 triggers the migration of process H_2 from tile_3 to tile_4 in order to keep the application running. The process migration is executed using the mechanism described in Chapter 4 of this thesis. Our demo allows the user to deactivate several tiles, provided that at least one replica of each process of the H.264 decoder application is allocated onto one of the active tiles. In the most resource-constrained scenario, the whole application can be executed by tile_0 and tile_2 alone. However, this results in a much lower frame rate of the application compared to the initial mapping which uses six active tiles.

A simplified setup of the hardware and software implementation of our demo is available to download at:
`http://daedalus.liacs.nl/demos/MADNESS_adaptivity.tar.gz`.
In this prototype, the input and output hardware interfaces of our demo are replaced (and emulated) by software components.

7.2.2 Application of Chapters 5 and 6 to Daedalus$^{\text{RT}}$

The scheduling framework proposed in [BS11, BS12] (shown in Figure 5.1 on page 88) has led to the implementation of Daedalus$^{\text{RT}}$ [BZNS12, Bam14]. Daedalus$^{\text{RT}}$ combines

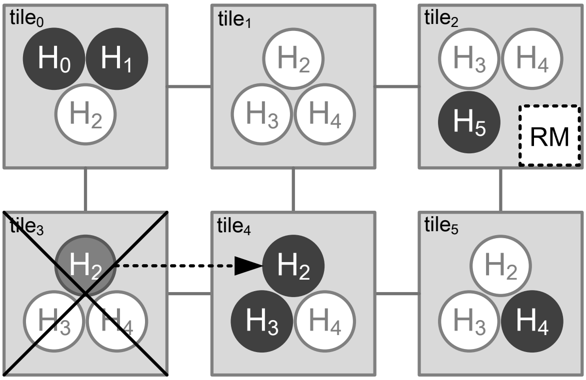


Figure 7.4: *Example of a process migration performed in our demo. The user requires the system to deactivate tile*$_3$*. Then, the resource manager (RM) which executes on tile*$_2$ *triggers the migration of process* H_2 *from tile*$_3$ *to tile*$_4$ *in order to keep the application running.*

the hard real-time scheduling analysis of [BS11, BS12] with the initial Daedalus system-level design flow [NSD08, NTS+08]. The research contributions of Chapters 5 and 6 of this thesis extend the scheduling framework of [BS11, BS12], therefore they can be directly applied to Daedalus^{RT}, as described in this section.

Daedalus^{RT} allows designers to generate a complete hardware and software platform, with guaranteed hard real-time behavior, starting from a sequential application specification. An overview of the Daedalus^{RT} design flow is shown in Figure 7.5. The design process starts by providing the input application(s), written in C/C++ in the form of a Static Affine Nested Loop Program (SANLP) [VNS07] (see the *Application* block in the upper-right part of the figure).

Then, in Step ①, *Parallelization*, the `pn` compiler [VNS07] converts each input SANLP to an equivalent PPN application specification. Each function call of the SANLP is converted to a separate process of the derived PPN. Moreover, if two functions of the SANLP access the same data array through their input/output arguments, `pn` derives data dependencies between the corresponding processes in the PPN.

Based on the derived PPN specification and on the *WCET analysis* (Step ②) of each function of the input SANLP, the *Analysis Model Derivation* (Step ③), performed by `pntools`, derives the analysis model of the application. This model is a CSDF graph, annotated with the WCET of each actor of the graph.

All the parts of Daedalus^{RT} described so far lay outside the dashed box in Figure 7.5 and are used to derive the application specification in the form of a PPN and the analysis model in the form of a CSDF graph. The derivation of these two models is not influenced by the contributions of this thesis.

However, the findings of Chapters 5 and 6 do extend the parts of Figure 7.5 enclosed by the dashed box and in particular the *HRT Analysis* (Step ④) of Daedalus^{RT}. This analysis is performed by the `DARTS` (Dataflow Analysis for Real Time Systems)

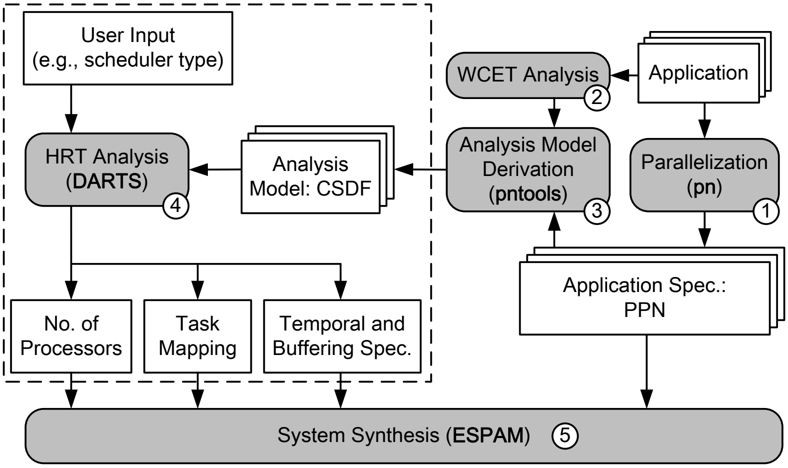


Figure 7.5: *Overview of the Daedalus^{RT} design flow (adapted from [BZNS12]). Steps ①, ② and ③ of the design flow are used to convert the input sequential application to the corresponding PPN specification and CSDF analysis model. Then, based on this analysis model and on the scheduler chosen by the user, Step ④ derives the number of processors required to schedule the application, the task mapping, and the temporal and buffering specification. At the end of Step ④ the system is completely specified and ready for system synthesis, performed in Step ⑤. The parts enclosed by the dashed box have been extended/modified by the contributions of Chapters 5 and 6 of this thesis.*

tool. In fact, the dashed box in Figure 7.5 abstracts the scheduling framework shown in Figure 5.1 on page 88. We briefly recall the operations performed by this scheduling framework in the next paragraph.

Step ④ uses the CSDF model of the application as input. The CSDF graph is then converted to a set of real-time periodic tasks using the scheduling analysis of [BS11, BS12] (described in Section 2.3). Based on the scheduler type selected by the user (see upper-left corner of Figure 7.5), `DARTS` derives: (i) the number of processors required to schedule the input application(s); (ii) the task mapping, which associates each task of the application to the processor responsible for its execution; (iii) the temporal and buffering specification, which consists of parameters that regulate the scheduling of tasks on the system (namely, WCET, period, and start times of tasks), together with the size of the buffers used to implement inter-task communication.

The *System Synthesis* (Step ⑤) finalizes the design flow by generating the RTL specification of the target MPSoC platform, together with the software running on each processor. Step ⑤ is performed by the `ESPAM` tool [NSD08] and uses the following inputs:

- the number of required processors, the task mapping, and the temporal and buffering specification provided by Step ④;
- the PPN application specification derived by Step ①.

Note that the whole Daedalus^{RT} design flow, including the tools `pn`, `pntools`, `DARTS`, and `ESPAM`, is available to download at `http://daedalus.liacs.nl/`

`download.html`.

As mentioned earlier, Step ④ of Daedalus^{RT} uses the the scheduling analysis of [BS11, BS12] and therefore, so far, has considered only hard real-time partitioned scheduling algorithms. The contributions of Chapters 5 and 6 allow designers to exploit soft real-time semi-partitioned scheduling algorithms in the systems generated by Daedalus^{RT}, with the benefits summarized in Section 1.4.2. In order to do so, the parts of Daedalus^{RT} enclosed by the dashed box in Figure 7.5 can be simply replaced by the scheduling frameworks proposed in Figure 5.2 and Figure 6.1, which represent the contributions of Chapters 5 and 6, respectively. Although the contributions of these chapters have been proven to be correct, the schedulers considered in these chapters have still not been implemented in Daedalus^{RT}. That is, the actual deployment of EDF-fm (see Section 2.2.7) and EDF-ssl (see Section 6.7) on the systems generated by Daedalus^{RT} is left as future work.

7.2.3 Application models considered in Chapters 3 to 6

In this section, we explain why Chapters 5 and Chapters 6 consider only applications modeled as *acyclic* (C)SDF graphs, and why Chapters 3 and 4 consider applications modeled as PPNs, instead.

Analysis of Chapters 5 and 6 restricted to acyclic (C)SDF graphs

The restriction on the application models considered in the semi-partitioned scheduling techniques proposed in Chapters 5 and 6 follows naturally from the dependencies of these techniques from the scheduling analysis of [BS11, BS12]. As explained in Section 2.3, such scheduling analysis can only handle applications modeled as *acyclic* (C)SDF graphs. In turn, this restriction applies also to the scheduling techniques proposed in Chapters 5 and 6.

Choice of the PPN MoC in Chapters 3 and 4

Chapters 3 and 4 consider applications modeled using the PPN Model of Computation (MoC) (see Section 2.1.3). We recall that these chapters propose an approach aimed at achieving system adaptivity in the context of **best-effort systems**. In order to achieve system adaptivity, the proposed approach provides a mechanism by which application processes can migrate among processors at run-time. In such a context, PPNs are a suitable MoC. This is because in PPNs, memory, control, and synchronization are completely distributed, which allows to change the mapping of processes to PEs at run-time with low effort.

As explained in Section 2.1.3, any sequential application specified as a Static Affine Nested Loop Program (SANLP) can be automatically converted to an equivalent parallel PPN specification [VNS07]. Moreover, from this specification, it is possible to efficiently generate the code that will run on the actual MPSoC [NSD08] for process execution, communication, and synchronization. For these reasons, the PPN model of the input application is used as *Application Specification* also in Daedalus^{RT} (see one of the inputs of Step ⑤ in Figure 7.5). This application specification is also called *Implementation Model* of the application, that is, the model that is close to the final code to be executed on the MPSoC.

In principle, it would have been possible to base the approach proposed in Chapters 3 and 4 of this thesis on the CSDF MoC (see Section 2.1.2) instead of PPNs. This is because it has been proven that a PPN is equivalent to a CSDF graph where the production/consumption sequences of actors consist of only zeros and ones [DSBS06]. However, we did not choose the CSDF MoC for Chapters 3 and 4 due to the following two reasons.

- First, as mentioned earlier, the PPN MoC works well as an implementation model (see page 16 of [Zha15]), because code can be efficiently generated from it. This is the reason why Daedalus^{RT} uses the PPN MoC as the implementation model of an application and CSDF only as the analysis model, i.e., the model used to perform non-functional analysis (see one of the inputs of Step ④ in Figure 7.5). Since the approach proposed in Chapters 3 and 4 acts at the level of the implementation model of the application, the PPN MoC is a natural foundation for the approach presented in these chapters.
- Second, as mentioned earlier, given any application specified as a SANLP, its equivalent PPN specification can be automatically derived. In turn, this PPN specification could be converted to an equivalent CSDF specification. However, in the approach proposed in Chapters 3 and 4, this additional conversion would not add any benefit. This is because these chapters are aimed at best-effort systems, and do not require a separate analysis model (in the form of a CSDF graph) to perform hard real-time analysis as in Daedalus^{RT}. Moreover, in most cases the PPN application model is more *succint* (or compact) than the equivalent CSDF model. If this CSDF application specification were to be mapped to actual code running on the MPSoC, the lesser compactness of this specification may result in execution time and/or code size overhead for each actor of the application, compared to the code generated from the equivalent PPN specification.

Bibliography

[AACP08] Andrea Acquaviva, Andrea Alimonda, Salvatore Carta, and Michele Pittau. Assessing task migration impact on embedded soft real-time streaming multimedia applications. *EURASIP J. Emb. Sys.*, 2008, 2008.

[ABD08] James H Anderson, Vasile Bud, and UmaMaheswari C Devi. An EDF-based restricted-migration scheduling algorithm for multiprocessor soft real-time systems. *Real-Time Systems*, 38(2):85–131, 2008.

[AEDC14] James H. Anderson, Jeremy P. Erickson, UmaMaheswari C. Devi, and Benjamin N. Casses. Optimal semi-partitioned scheduling in soft real-time systems. In *2014 IEEE 20th International Conference on Embedded and Real-Time Computing Systems and Applications, Chongqing, China, August 20-22, 2014*, pages 1–10, 2014.

[AK09] C. Ababei and R. Katti. Achieving network on chip fault tolerance by adaptive remapping. In *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–4, May 2009.

[AMC$^+$07] Federico Angiolini, Paolo Meloni, Salvatore Carta, Luigi Raffo, and Luca Benini. A layout-aware analysis of networks-on-chip and traditional interconnects for mpsocs. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 26(3):421–434, 2007.

[AT06] Björn Andersson and Eduardo Tovar. Multiprocessor scheduling with few preemptions. In *12th IEEE Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 2006), 16-18 August 2006, Sydney, Australia*, pages 322–334, 2006. URL: `http://dx.doi.org/10.1109/RTCSA.2006.45`, `doi:10.1109/RTCSA.2006.45`.

[AY03] Hakan Aydin and Qi Yang. Energy-aware partitioning for multiprocessor real-time systems. In *17th International Parallel and Distributed Processing Symposium (IPDPS 2003), 22-26 April 2003, Nice, France, CD-ROM/Abstracts Proceedings*, page 113, 2003. URL: `http://dx.doi.org/10.1109/IPDPS.2003.1213225`, `doi:10.1109/IPDPS.2003.1213225`.

[B$^+$09] Enrico Bini et al. Minimizing CPU energy in real-time systems with discrete speed management. *Trans. Embedded Comput. Syst.*, 2009. URL: `http://doi.acm.org/10.1145/1550987.1550994`, `doi:10.1145/1550987.1550994`.

[BABP06] Stefano Bertozzi, Andrea Acquaviva, Davide Bertozzi, and Antonio Poggiali. Supporting task migration in multi-processor systems-on-chip: a feasibility study. In *Proceedings of the conference on Design, automation and test in Europe*, DATE '06, pages 15–20, 2006.

[Bam14] Mohamed A. Bamakhrama. *On Hard Real-Time Scheduling of Cyclo-Static Dataflow and its Application in System-Level Design*. PhD thesis, Leiden University, 2014.

[BB01] Bishnupriya Bhattacharya and Shuvra S Bhattacharyya. Parameterized dataflow modeling for dsp systems. *Signal Processing, IEEE Transactions on*, 49(10):2408–2421, 2001.

[BB04] D. Bertozzi and L. Benini. Xpipes: a network-on-chip architecture for gigascale systems-on-chip. *Circuits and Systems Magazine, IEEE*, 4(2):18–31, September 2004.

[BBA11] Andrea Bastoni, Björn B. Brandenburg, and James H. Anderson. Is semi-partitioned scheduling practical? In *23rd Euromicro Conference on Real-Time Systems, ECRTS 2011, Porto, Portugal, 5-8 July, 2011*, pages 125–135, 2011.

[BBB15] Sanjoy Baruah, Marko Bertogna, and Giorgio Buttazzo. *Multiprocessor Scheduling for Real-Time Systems*. Springer, 2015.

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

[BDLT13] Shuvra S Bhattacharyya, Ed F Deprettere, Rainer Leupers, and Jarmo Takala. *Handbook of signal processing systems*. Springer Science & Business Media, 2013.

[BDM02] L. Benini and G. De Micheli. Networks on chips: a new soc paradigm. *Computer*, 35(1):70–78, Jan 2002.

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

[BHHT10] Iuliana Bacivarov, Wolfgang Haid, Kai Huang, and Lothar Thiele. Methods and Tools for Mapping Process Networks onto Multi-Processor Systems-On-Chip. In Shuvra S. Bhattacharyya, Ed F. Deprettere, Rainer Leupers, and Jarmo Takala, editors, *Handbook of Signal Processing Systems*, pages 1007—1040. Springer, October 2010.

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

[BS12] Mohamed A. Bamakhrama and Todor P. Stefanov. On the hard-real-time scheduling of embedded streaming applications. *Design Automation for Embedded Systems*, 2012. DOI: 10.1007/s10617-012-9086-x. `doi:10.1007/s10617-012-9086-x`.

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

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

[CBS14] Emanuele Cannella, Mohamed Bamakhrama, and Todor Stefanov. System-level scheduling of real-time streaming applications using a semi-partitioned approach. In *Design, Automation & Test in Europe Conference & Exhibition, DATE 2014, Dresden, Germany, March 24-28, 2014*, pages 1–6, 2014. URL: `http://dx.doi.org/10.7873/DATE.2014.376`, `doi:10.7873/DATE.2014.376`.

[CDM$^+$12] Emanuele Cannella, Onur Derin, Paolo Meloni, Giuseppe Tuveri, and Todor Stefanov. Adaptivity support for mpsocs based on process migration in polyhedral process networks. *VLSI Design*, 2012:987209:1–987209:17, 2012.

[CDS11] Emanuele Cannella, Onur Derin, and Todor Stefanov. Middleware approaches for adaptivity of kahn process networks on networks-on-chip. In *2011 Conference on Design and Architectures for Signal and Image Processing, DASIP 2011, Tampere, Finland, November 2-4, 2011*, pages 100–107, 2011. URL: `http://dx.doi.org/10.1109/DASIP.2011.6136862`, `doi:10.1109/DASIP.2011.6136862`.

[CGF$^+$11] Emanuele Cannella, Lorenzo Di Gregorio, Leandro Fiorin, Menno Lindwer, Paolo Meloni, Olaf Neugebauer, and Andy D. Pimentel. Towards an ESL design framework for adaptive and fault-tolerant mpsocs: MADNESS or not? In *9th IEEE Symposium on Embedded Systems for Real-Time Multimedia, ESTIMedia 2011, Taipei, Taiwan, October 13-14, 2011*, pages 120–129, 2011.

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

[CJK88] Thomas L. Casavant, Jon, and G. Kuhl. A taxonomy of scheduling in general-purpose distributed computing systems. *IEEE Transactions on Software Engineering*, 14:141–154, 1988.

[CRJ06] Hyeonjoong Cho, B. Ravindran, and E.D. Jensen. An optimal real-time scheduling algorithm for multiprocessors. In *Real-Time Systems Symposium, 2006. RTSS '06. 27th IEEE International*, pages 101–110, Dec 2006.

[CS16] Emanuele Cannella and Todor P. Stefanov. Energy Efficient Semi-partitioned Scheduling for Embedded Multiprocessor Streaming Systems. *Design Autom. for Emb. Sys.*, 20(3):239–266, 2016.

[DA10] V. Devadas and H. Aydin. Coordinated power management of periodic real-time tasks on chip multiprocessors. In *Green Computing Conference, 2010 International*, pages 61–72, Aug 2010.

[Das04] Ali Dasdan. Experimental analysis of the fastest optimum cycle ratio and mean algorithms. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 9(4):385–418, 2004.

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

[DCT$^+$13] Onur Derin, Emanuele Cannella, Giuseppe Tuveri, Paolo Meloni, Todor Stefanov, Leandro Fiorin, Luigi Raffo, and Mariagiovanna Sami. A system-level approach to adaptivity and fault-tolerance in noc-based mpsocs: The MADNESS project. *Microprocessors and Microsystems - Embedded Hardware Design*, 37(6-7):515–529, 2013.

[dDAB$^+$13] Benoît Dupont de Dinechin, Renaud Ayrignac, P-E Beaucamps, Patrice Couvert, Benoit Ganne, Pierre Guironnet de Massas, François Jacquet, Samuel Jones, Nicolas Morey Chaisemartin, Frédéric Riss, et al. A clustered manycore processor architecture for embedded and accelerated applications. In *High Performance Extreme Computing Conference (HPEC), 2013 IEEE*, pages 1–6. IEEE, 2013.

[DDF11] Onur Derin, Erkan Diken, and Leandro Fiorin. A middleware approach to achieving fault-tolerance of kahn process networks on networks-on-chips. *International Journal of Reconfigurable Computing*, 2011(Article ID 295385):14 pages, February 2011. Selected Papers from the International Workshop on Reconfigurable Communication-centric Systems on Chips (ReCoSoC' 2010). `doi:doi:10.1155/2011/295385`.

[Der15] Onur Derin. *Self-adaptivity of Applications on Network on Chip Multiprocessors: The Case of Fault-tolerant Kahn Process Networks*. PhD thesis, Faculty of Informatics, University of Lugano, May 2015. PhD thesis.

[Dev06] Umamaheswari C Devi. *Soft real-time scheduling on multiprocessors*. PhD thesis, University of North Carolina at Chapel Hill, 2006.

[DKF11] Onur Derin, Deniz Kabakci, and Leandro Fiorin. Online task remapping strategies for fault-tolerant network-on-chip multiprocessors. In *NOCS 2011, Fifth ACM/IEEE International Symposium on Networks-on-Chip, Pittsburgh, Pennsylvania, USA, May 1-4, 2011*, pages 129–136, 2011.

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

[DYGR10] François Dorin, Patrick Meumeu Yomsi, Joël Goossens, and Pascal Richard. Semi-partitioned hard real-time scheduling with restricted migrations upon identical multiprocessor platforms. *CoRR*, abs/1006.2637, 2010. URL: `http://arxiv.org/abs/1006.2637`.

[EA11] Jeremy P. Erickson and James H. Anderson. Response time bounds for G-EDF without intra-task precedence constraints. In *Principles of Distributed Systems - 15th International Conference, OPODIS 2011, Toulouse, France, December 13-16, 2011. Proceedings*, pages 128–142, 2011. URL: `http://dx.doi.org/10.1007/978-3-642-25873-2_10`, `doi:10.1007/978-3-642-25873-2_10`.

[Gab09] Gabriel Marchesan Almeida and Gilles Sassatelli and Pascal Benoit and Nicolas Saint-Jean and Sameer Varyani and Lionel Torres and Michel Robert. An Adaptive Message Passing MPSoC Framework. *International Journal of Reconfigurable Computing*, 2009:20, 2009.

[GCS11] Laurent George, Pierre Courbin, and Yves Sorel. Job vs. portioned partitioning for the earliest deadline first semi-partitioned scheduling. *Journal of Systems Architecture - Embedded Systems Design*, 57(5):518–535, 2011.

[GGS$^+$06] Amir Hossein Ghamarian, MCW Geilen, Sander Stuijk, Twan Basten, AJM Moonen, Marco JG Bekooij, Bart D Theelen, and MohammadReza Mousavi. Throughput analysis of synchronous data flow graphs. In *Application of Concurrency to System Design, 2006. ACSD 2006. Sixth International Conference on*, pages 25–36. IEEE, 2006.

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

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

[HDV$^+$11] Jason Howard, Saurabh Dighe, Sriram R Vangal, Gregory Ruhl, Nitin Borkar, Shailendra Jain, Vasantha Erraguntla, Michael Konow, Michael Riepen, Matthias Gries, et al. A 48-core ia-32 processor in 45 nm cmos using on-die message-passing and dvfs for performance and power scaling. *Solid-State Circuits, IEEE Journal of*, 46(1):173–183, 2011.

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

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

[HMGM13] Pengcheng Huang, Orlando Moreira, Kees Goossens, and Anca Mariana Molnos. Throughput-constrained voltage and frequency scaling for real-time heterogeneous multiprocessors. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing, SAC '13, Coimbra, Portugal, March 18-22, 2013*, pages 1517–1524, 2013. URL: `http://doi.acm.org/10.1145/2480362.2480645`, `doi:10.1145/2480362.2480645`.

[HP07] John L. Hennessy and David A. Patterson. *Computer Architecture, 4th Edition*. Morgan Kaufmann, 2007.

[HSH$^+$09] Wolfgang Haid, Lars Schor, Kai Huang, Iuliana Bacivarov, and Lothar Thiele. Efficient execution of kahn process networks on multi-processor systems using protothreads and windowed fifos. In *Proc. IEEE Workshop on Embedded Systems for Real-Time Multimedia (ESTIMedia)*, pages 35–44, Grenoble, France, 2009. IEEE.

[HvdHBL10] Mike Holenderski, Martijn M.H.P. van den Heuvel, Reinder J. Bril, and Johan J. Lukkien. Grasp: Tracing, visualizing and measuring the behavior of real-time systems. In *International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, July 2010.

[HXW$^+$10] Hongtao Huang, Feng Xia, Jijie Wang, Siyu Lei, and Guowei Wu. Leakage-aware reallocation for periodic real-time tasks on multicore processors. In *Fifth International Conference on Frontier of Computer Science and Technology, FCST 2010, Changchun, Jilin Province, China, August 18-22, 2010*, pages 85–91, 2010. URL: `http://dx.doi.org/10.1109/FCST.2010.105`, `doi:10.1109/FCST.2010.105`.

[Int13] International Technology Roadmap for Semiconductors. 2013 Edition: Executive Summary, 2013. URL: `http://www.itrs.net/` [cited May 14, 2015].

[IY98] Tohru Ishihara and Hiroto Yasuura. Voltage scheduling problem for dynamically variable voltage processors. In *ISLPED 98, August 10-12, Monterey, CA USA*, 1998.

[Jha01] N.K. Jha. Low power system scheduling and synthesis. In *Computer Aided Design, 2001. ICCAD 2001. IEEE/ACM International Conference on*, pages 259–263, Nov 2001.

[Joh73] David S. Johnson. *Near-optimal bin packing algorithms*. PhD thesis, MIT, 1973.

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

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

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

[KDH+05] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the cell multiprocessor. *IBM J. Res. Dev.*, 49(4/5):589–604, July 2005. URL: `http://dl.acm.org/citation.cfm?id=1148882.1148891`.

[KJS+02] Shashi Kumar, Axel Jantsch, J-P Soininen, Martti Forsell, Mikael Millberg, Johny Oberg, Kari Tiensyrja, and Ahmed Hemani. A network on chip architecture and design methodology. In *VLSI, 2002. Proceedings. IEEE Computer Society Annual Symposium on*, pages 105–112. IEEE, 2002.

[KKJ+08] Seongnam Kwon, Yongjoo Kim, Woo-Chul Jeun, Soonhoi Ha, and Yunheung Paek. A retargetable parallel-programming framework for mpsoc. *ACM Trans. Des. Autom. Electron. Syst.*, 13:39:1–39:18, July 2008.

[KY08] Shinpei Kato and Nobuyuki Yamasaki. Portioned edf-based scheduling on multiprocessors. In *Proceedings of the 8th ACM & IEEE International conference on Embedded software, EMSOFT 2008, Atlanta, GA, USA, October 19-24, 2008*, pages 139–148, 2008. URL: `http://doi.acm.org/10.1145/1450058.1450078`, `doi:10.1145/1450058.1450078`.

[LA09] Cong Liu and James H. Anderson. Supporting pipelines in soft real-time multiprocessor systems. In *21st Euromicro Conference on Real-Time Systems, ECRTS 2009, Dublin, Ireland, July 1-3, 2009*, pages 269–278, 2009. URL: `http://dx.doi.org/10.1109/ECRTS.2009.16`, `doi:10.1109/ECRTS.2009.16`.

[LA10] Cong Liu and James H. Anderson. Supporting soft real-time dag-based systems on multiprocessors with no utilization loss. In *Proceedings of the 31st IEEE Real-Time Systems Symposium, RTSS 2010, San Diego, California, USA, November 30 - December 3, 2010*, pages 3–13, 2010.

[LDG04] J. M. López, J. L. Díaz, and D. F. García. Utilization bounds for edf scheduling on real-time multiprocessor systems. *Real-Time Syst.*, 28(1):39–68, October 2004.

[Lee99] E.A. Lee. *Embedded Software: An Agenda for Research*. Memorandum (University of California, Berkeley, Electronics Research Laboratory). Electronics Research Laboratory, College of Engineering, University of California, 1999.

[Lee09] Wan Yeon Lee. Energy-saving dvfs scheduling of multiple periodic real-time tasks on multi-core processors. In *Distributed Simulation and Real Time Applications, 2009. DS-RT '09. 13th IEEE/ACM International Symposium on*, pages 216–223, Oct 2009.

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

[LKwP+10] Chanhee Lee, Hokeun Kim, Hae woo Park, Sungchan Kim, Hyunok Oh, and Soonhoi Ha. A task remapping technique for reliable multi-core embedded systems. In *Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2010 IEEE/ACM/IFIP International Conference on*, pages 307–316, Oct 2010.

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

[LM87a] E. Lee and D.G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *Computers, IEEE Transactions on*, C-36(1):24–35, Jan 1987.

[LM87b] Edward A. Lee and David G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, 1987.

[LvdWD01] Paul Lieverse, Pieter van der Wolf, and Ed F. Deprettere. A trace transformation technique for communication refinement. In *Proceedings of the Ninth International Symposium on Hardware/Software Codesign, CODES 2001, Copenhagen, Denmark, 2001*, pages 134–139, 2001. `doi:10.1145/371636.371703`.

[Mac13] Enrico Macii, editor. *Design, Automation and Test in Europe, DATE 13, Grenoble, France, March 18-22, 2013*. EDA Consortium San Jose, CA, USA / ACM DL, 2013. URL: `http://dl.acm.org/citation.cfm?id=2485288`.

[Mar11] P. Marwedel. *Embedded System Design*. Springer, 2011.

[MB07] Orlando Moreira and Marco Bekooij. Self-timed scheduling analysis for real-time applications. *EURASIP J. Adv. Sig. Proc.*, 2007, 2007.

[MCA] Multicore associations communication api. URL: `http://www.multicore-association.org`.

[MCM+04] Fernando Moraes, Ney Calazans, Aline Mello, Leandro Möller, and Luciano Ost. HERMES: An Infrastructure for Low Area Overhead Packet-switching Networks on Chip. *Integr. VLSI J.*, 38(1):69–93, October 2004.

[MDP+00] Dejan S. Milojicic, Fred Douglis, Yves Paindaveine, Richard Wheeler, and Songnian Zhou. Process migration. *ACM Comput. Surv.*, 32:241–299, September 2000.

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

[MTR+12] Paolo Meloni, Giuseppe Tuveri, Luigi Raffo, Emanuele Cannella, Todor Stefanov, Onur Derin, Leandro Fiorin, and Mariagiovanna Sami. System adaptivity and fault-tolerance in noc-based mpsocs: The MADNESS project approach. In *15th Euromicro Conference on Digital System Design, DSD 2012, Cesme, Izmir, Turkey, September 5-8, 2012*, pages 517–524, 2012. `doi:10.1109/DSD.2012.122`.

[NGWK09] A. B. Nejad, K. Goossens, J. Walters, and B. Kienhuis. Mapping kpn models of streaming applications on a network-on-chip platform. In *ProRISC 2009: Proceedings of the Workshop on Signal Processing, Integrated Systems and Circuits*, November 2009.

[NKG+02] André Nieuwland, Jeffrey Kang, Om Prakash Gangwal, Ramanathan Sethuraman, Natalino G. Busá, Kees Goossens, Rafael Peset Llopis, and Paul E. R. Lippens. C-HEAP: A heterogeneous multi-processor architecture template and scalable and flexible protocol for the design of embedded signal processing systems. *Design Autom. for Emb. Sys.*, 7(3):233–270, 2002.

[NMSD09] Dmitry Nadezhkin, Sjoerd Meijer, Todor Stefanov, and Ed Deprettere. Realizing FIFO Communication When Mapping Kahn Process Networks onto the Cell. In *Proceedings of the 9th International Workshop on Embedded Computer Systems: Architectures, Modeling, and Simulation*, SAMOS '09, pages 308–317, Berlin, Heidelberg, 2009. Springer-Verlag. URL: `http://dx.doi.org/10.1007/978-3-642-03138-0_34`, `doi:http://dx.doi.org/10.1007/978-3-642-03138-0\_34`.

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

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

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

[ope] A high performance message passing library. URL: `http://www.open-mpi.org/`.

[P$^+$13] Sangyoung Park et al. Accurate modeling of the delay and energy overhead of dynamic voltage and frequency scaling in modern microprocessors. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 32(5):695–708, 2013. URL: `http://dx.doi.org/10.1109/TCAD.2012.2235126`, `doi:10.1109/TCAD.2012.2235126`.

[PEP06] Andy D Pimentel, Cagkan Erbas, and Simon Polstra. A systematic approach to exploring embedded system architectures at multiple abstraction levels. *Computers, IEEE Transactions on*, 55(2):99–112, 2006.

[Pin16] Michael L Pinedo. *Scheduling: theory, algorithms, and systems*. Springer Science & Business Media, 2016.

[Ram] Carl Ramey. TILE-Gx100 ManyCore Processor: Acceleration Interfaces and Architecture. URL: `http://www.hotchips.org/wp-content/uploads/hc_archives/hc23/HC23.18.2-security/HC23.18.220-TILE-GX100-Ramey-Tilera-e.pdf` [cited April 30, 2015].

[RGR$^+$03] E. Rijpkema, K.G.W. Goossens, A. Radulescu, J. Dielissen, J. van Meerbergen, P. Wielage, and E. Waterlander. Trade offs in the design of a router with both guaranteed and best-effort services for networks on chip. In *Design, Automation and Test in Europe Conference and Exhibition, 2003*, pages 350–355, 2003.

[Rho] M. Rhodan. GM to Roll Out a Self-Driving Cadillac. URL: `http://time.com/3303212/gm-self-driving-cadillac/` [cited March 11, 2015].

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

[SDK13] Amit Kumar Singh, Anup Das, and Akash Kumar. Energy optimization by exploiting execution slacks in streaming applications on multiprocessor systems. In *The 50th Annual Design Automation Conference 2013, DAC '13, Austin, TX, USA, May 29 - June 07, 2013*, pages 115:1–115:7, 2013. URL: `http://doi.acm.org/10.1145/2463209.2488875`, `doi:10.1145/2463209.2488875`.

[SGTB11] Sander Stuijk, Marc Geilen, Bart Theelen, and Twan Basten. Scenario-aware dataflow: Modeling, analysis and implementation of dynamic applications. In *Embedded Computer Systems (SAMOS), 2011 International Conference on*, pages 404–411. IEEE, 2011.

[SJPL08] Euiseong Seo, Jinkyu Jeong, Seonyeong Park, and Joonwon Lee. Energy efficient scheduling of real-time tasks on multicore processors. *Parallel and Distributed Systems, IEEE Transactions on*, 19(11):1540–1552, Nov 2008.

[Smi88] Jonathan M. Smith. A survey of process migration mechanisms. *SIGOPS Oper. Syst. Rev.*, 22:28–40, July 1988.

[SSHT06] Thilo Streichert, Christian Strengert, Christian Haubelt, and Jürgen Teich. Dynamic task binding for hardware/software reconfigurable networks. In *Proceedings of the 19th Annual Symposium on Integrated Circuits and Systems Design*, SBCCI '06, pages 38–43, 2006.

[Sut] H. Sutter. The Free Lunch Is Over. URL: `http://www.gotw.ca/publications/concurrency-ddj.htm` [cited March 12, 2015].

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

[TIL] TriMedia TM-1000 Datasheet. URL: `http://www.tilera.com/files/drim__TILE-Gx8072_PB041-04_WEB_7683.pdf` [cited June 11, 2015].

[Tri] TriMedia TM-1000 Datasheet. URL: `http://pdf.datasheetcatalog.com/datasheet/philips/TM-1000.pdf` [cited June 11, 2015].

[VAJ+09] Sravanthi Kota Venkata, Ikkjin Ahn, Donghwan Jeon, Anshuman Gupta, Christopher M. Louie, Saturnino Garcia, Serge Belongie, and Michael Bedford Taylor. SD-VBS: the san diego vision benchmark suite. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization, IISWC 2009, October 4-6, 2009, Austin, TX, USA*, pages 55–64, 2009.

[VEMR14] Anish Varghese, Bob Edwards, Gaurav Mitra, and Alistair P. Rendell. Programming the adapteva epiphany 64-core network-on-chip coprocessor. In *Proceedings of the 2014 IEEE International Parallel & Distributed Processing Symposium Workshops*, IPDPSW '14, pages 984–992, Washington, DC, USA, 2014. IEEE Computer Society.

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

[W+10] Yi-Hung Wei et al. Energy-efficient real-time scheduling of multimedia tasks on multi-core processors. In *Proceedings of the 2010 ACM Symposium on Applied Computing (SAC), Sierre, Switzerland, March 22-26, 2010*, pages 258–262, 2010. URL: `http://doi.acm.org/10.1145/1774088.1774142`, `doi:10.1145/1774088.1774142`.

[WL03] D. Wiklund and D. Liu. SoCBUS: switched network on chip for hard real time embedded systems. In *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*, pages 8 pp.–, April 2003.

[WLL+11] Yi Wang, Hui Liu, Duo Liu, Zhiwei Qin, Zili Shao, and Edwin Hsing-Mean Sha. Overhead-aware energy optimization for real-time streaming applications on multiprocessor system-on-chip. *ACM Trans. Design Autom. Electr. Syst.*, 16(2):14, 2011. URL: `http://doi.acm.org/10.1145/1929943.1929946`, `doi:10.1145/1929943.1929946`.

[Woo] Victoria Woollaston. New images show how Google's self-driving cars see the world. URL: `http://www.dailymail.co.uk/sciencetech/article-2317594/` [cited March 11, 2015].

[YA14] Kecheng Yang and James H. Anderson. Optimal gedf-based schedulers that allow intra-task parallelism on heterogeneous multiprocessors. In *12th IEEE Symposium on Embedded Systems for Real-time Multimedia, ESTIMedia 2014, Greater Noida, India, October 16-17, 2014*, pages 30–39, 2014. URL: `http://dx.doi.org/10.1109/ESTIMedia.2014.6962343`, `doi:10.1109/ESTIMedia.2014.6962343`.

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

[ZBS13] Jiali Teddy Zhai, Mohamed Bamakhrama, and Todor Stefanov. Exploiting just-enough parallelism when mapping streaming applications in hard real-time systems. In *The 50th Annual Design Automation Conference 2013, DAC '13, Austin, TX, USA, May 29 - June 07, 2013*, pages 170:1–170:8, 2013. URL: `http://doi.acm.org/10.1145/2463209.2488944`, `doi:10.1145/2463209.2488944`.

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

[Zhe07] Liu Zheng. A task migration constrained energy-efficient scheduling algorithm for multiprocessor real-time systems. In *Wireless Communications, Networking and Mobile Computing, 2007. WiCom 2007. International Conference on*, pages 3055–3058, Sept 2007. `doi:10.1109/WICOM.2007.759`.

[ZNS11] Jiali Teddy Zhai, Hristo Nikolov, and Todor Stefanov. Modeling adaptive streaming applications with parameterized polyhedral process networks. In *Proceedings of the 48th Design Automation Conference*, pages 116–121. ACM, 2011.

[ZR13] Yuhao Zhu and Vijay Janapa Reddi. High-performance and energy-efficient mobile web browsing on big/little systems. In *19th IEEE International Symposium on High Performance Computer Architecture, HPCA 2013, Shenzhen, China, February 23-27, 2013*, pages 13–24, 2013. URL: `http://dx.doi.org/10.1109/HPCA.2013.6522303`, `doi:10.1109/HPCA.2013.6522303`.

List of abbreviations

ALAP	As late as possible
API	Application programming interface
BE	Best-Effort
BF	Best-Fit
BFD	Best-Fit Decreasing
CSDF	Cyclo-Static Dataflow
DCT	Discrete cosine transform
DM	Data Memory
DMA	Direct Memory Access
DPM	Dynamic Power Management
DSE	Design Space Exploration
EDF	Earliest Deadline First
EDF-fm	Earliest Deadline First with fixed and migrating tasks
EDF-ssl	Earliest Deadline First based semi-partitioned stateless
FF	First-Fit
FFD	First-Fit Decreasing
FFD-SP	First-Fit Decreasing followed by semi-partitioning
FIFO	First-in First-out
FPGA	Field-programmable gate array
GEDF	Global EDF
GT	Guaranteed throughput
HRT	Hard Real-Time
HSDF	Homogeneous Synchronous Dataflow
IM	Instruction Memory
IP	Input Port
JPEG	Joint Photographic Experts Group
KPN	Kahn Process Network
LLF	Least Laxity First
MCM	Maximum Cycle Mean
MJPEG	Motion JPEG
MoC	Model of Computation
MPI	Message Passing Interface
MPSoC	Multiprocessor System-on-Chip
MW	Middleware

NI	Network Interface
NoC	Network-on-Chip
NP	Non-deterministic Polynomial-time
OP	Output Port
OS	Operating System
PE	Processing Element
PEDF	Partitioned EDF
PM	Power Management
PPN	Polyhedral Process Network
PWM	Pulse Width Modulation
SANLP	Static Affine Nested Loop Program
SDF	Synchronous Dataflow
SRT	Soft Real-Time
TDMA	Time Division Multiple Access
VFS	Voltage/Frequency Scaling
VLE	Variable-length encoding
WCET	Worst-Case Execution Time
WF	Worst-Fit
WFD	Worst-Fit Decreasing

Samenvatting

In deze dissertatie worden – in de context van ingebedde systemen – bepaalde ontwerp methoden en technieken voorgesteld. Het accent ligt op ingebedde systemen die datastromen bewerken die vanuit de omgeving van het ingebedde systeem worden aangeleverd. De datastromen zijn typisch onbegrensd in lengte. Coderen en decoderen in *real-time* van audio en video datastromen zijn typische voorbeelden. Als bij het bewerken van datastromen een hoge prestatie is vereist kan het nodig zijn om daarvoor meerdere processors in een netwerk op een enkele chip als executieplatform te gebruiken (Multi-Processor System-on-Chip of MPSoC). Daardoor kan parallelisme worden uitgebuit en dus de bewerkingstijd verkort. Het bewerkingsprogramma wordt dan opgesplitst in taken die wel onderling afhankelijk zijn maar toch parallel kunnen worden verwerkt. De verschillende taken worden dan toegekend aan de processors in het platform (ordening in ruimte). De dissertatie geeft technieken voor het dynamisch optimaliseren en adapteren van de toekenning van taken aan processoren. Er zijn twee bijdragen te onderscheiden.

In het eerste deel gaat de aandacht naar systemen waarbij het tijdsgedrag (ordening in tijd) niet is gespecificeerd, anders gezegd systemen met best- mogelijke prestatie. Door het toevoegen van een taak-migratie procedure wordt het mogelijk de toekenning van taken aan processors tijdens de executie te wijzigen met de garantie dat communicatie tussen taken zowel voor als na de migratie correct is. Met deze taak-migratie procedure kan het systeem zich aanpassen aan eventuele veranderende condities die vanuit de omgeving worden opgelegd, waardoor het systeem adaptief wordt. Als bijvoorbeeld een processor faalt, dan kan de taak die aan deze processor was toegekend migreren naar een correct functionerende processor, waardoor het systeem als geheel het falen van een van de procesoren kan overleven.

In het tweede deel is het tijdsgedrag wel gespecificeerd en gaat het om systemen die aan strikte of harde *real-time* condities moeten voldoen. Niet voldoen aan de tijdeisen leidt tot een volledig falend systeem. Gekeken wordt naar de toepasbaarheid van gedeeltelijk gepartitioneerde toekenningen van taken aan processoren. Hierbij worden de meeste taken statisch toegekend aan processors en kan slechts een klein deel van de taken migreren volgens een vooraf bepaald patroon.

In deze dissertatie wordt een benadering voorgesteld die de ontwerper toelaat het aantal processors te verminderen in tegenstelling tot systemen waarin migreren niet kan. De vermindering van het aantal processors gaat ten koste van een geringe toename van het benodigde geheugen. Een tweede voorgestelde benadering is

geschikt voor systemen waarin voedingsspanning en klokfrequentie kunnen worden verlaagd met het doel energie te besparen. In vergelijking met systemen waarin migratie niet mogelijk is, kan met gedeeltelijke partitionering een grotere energie besparing bereikt worden. Ook hier gaat dit gepaard met een geringe toename van het benodigde geheugen.

List of publications

Journal Articles

- **Emanuele Cannella**, Todor Stefanov, "Energy Efficient Semi-Partitioned Scheduling for Embedded Multiprocessor Streaming Systems", *In Design Automation for Embedded Systems (DAEM), vol. 20, number 3, 2016*, pp. 239–266.
- **Emanuele Cannella**, Onur Derin, Paolo Meloni, Giuseppe Tuveri, Todor Stefanov, "Adaptivity Support for MPSoCs Based on Process Migration in Polyhedral Process Networks", *In VLSI Design 2012*, pp. 987209:1-987209:17.
- Jelena Spasic, Di Liu, **Emanuele Cannella**, Todor Stefanov, "On the Improved Hard Real-Time Scheduling of Cyclo-Static Dataflow", *In ACM Transactions on Embedded Computing Systems (TECS), vol. 15, Issue 4, Article 68, Aug 2016*.
- Onur Derin, **Emanuele Cannella**, Giuseppe Tuveri, Paolo Meloni, Todor Stefanov, Leandro Fiorin, Luigi Raffo, Mariagiovanna Sami, "A system-level approach to adaptivity and fault-tolerance in NoC-based MPSoCs: The MADNESS project", *Microprocessors and Microsystems - Embedded Hardware Design, vol. 37, number 6-7, 2013*, pp. 515–529.
- Onur Derin, Prasanth Kuncheerath Ramankutty, Paolo Meloni, **Emanuele Cannella**, "Towards Self-Adaptive KPN Applications on NoC-Based MPSoCs" *In Advances in Software Engineering, 2012*, pp. 172674:1–172674:16, 2012.

Peer-reviewed Conference Proceedings

- **Emanuele Cannella**, Mohamed A. Bamakhrama, and Todor Stefanov, "System-level Scheduling of Real-time Streaming Applications using a Semi-partitioned Approach", *In the Proceedings of the Design, Automation and Test in Europe Conference and Exhibition, (DATE) 2014*, pp. 1–6, Dresden, Germany, 24-28 March 2014.
- **Emanuele Cannella**, Onur Derin, Todor Stefanov, "Middleware approaches for adaptivity of Kahn Process Networks on Networks-on-Chip", *In the Proceedings of the 2011 Conference on Design and Architectures for Signal and Image Processing, (DASIP) 2011*, pp. 100–107, Tampere, Finland, November 2-4, 2011.
- **Emanuele Cannella**, Lorenzo Di Gregorio, Leandro Fiorin, Menno Lindwer, Paolo Meloni, Olaf Neugebauer, Andy D. Pimentel, "Towards an ESL design

framework for adaptive and fault-tolerant MPSoCs: MADNESS or not?”, *In the Proceedings of the 9th IEEE Symposium on Embedded Systems for Real-Time Multimedia, (ESTIMedia) 2011*, pp. 120–129, Taipei, Taiwan, October 13-14, 2011.

- Jelena Spasic, Di Liu, **Emanuele Cannella**, Todor Stefanov, “Improved hard real-time scheduling of CSDF-modeled streaming applications”, *In the Proceedings of the 2015 International Conference on Hardware/Software Codesign and System Synthesis, (CODES+ISSS) 2015*, pp. 65–74, Amsterdam, Netherlands, October 4-9, 2015.
- Giuseppe Tuveri, Simone Secchi, Paolo Meloni, Luigi Raffo, **Emanuele Cannella**, “A runtime adaptive H.264 video-decoding MPSoC platform”, *In the Proceedings of the 2013 Conference on Design and Architectures for Signal and Image Processing, (DASIP) 2013*, pp. 149–156, Cagliari, Italy, October 8-10, 2013.
- Paolo Meloni, Giuseppe Tuveri, Luigi Raffo, **Emanuele Cannella**, Todor Stefanov, Onur Derin, Leandro Fiorin, Mariagiovanna Sami, “System Adaptivity and Fault-Tolerance in NoC-based MPSoCs: The MADNESS Project Approach”, *In the Proceedings of the 15th Euromicro Conference on Digital System Design, (DSD) 2012*, pp. 517–524, Cesme, Izmir, Turkey, September 5-8, 2012.

Curriculum Vitae

Emanuele Cannella was born on August 17, 1983 in Udine, Italy. In 2008, he obtained his MSc degree in Electronic Engineering from University of Udine. His master's thesis project concerned Multiprocessor Systems-on-Chip running embedded streaming applications. He carried out this project as an exchange student at the Computer Engineering laboratory of TU Delft, The Netherlands. Shortly after his graduation, he started to work as a research assistant at University of Udine, in the field of pervasive and distributed computing. In 2010, he joined the Leiden Embedded Research Center at Leiden University as a PhD candidate. His research work, which has led to this thesis, has been funded by the EU FP7 project MADNESS. In July 2015, he joined Lely Industries, where he works as a software engineer focusing on model-driven software engineering approaches to robotic control.

Acknowledgments

First of all, on the professional level, I would like to thank the colleagues I had the pleasure to work with at the Leiden Embedded Research Center (LERC). Hristo Nikolov, Mohammad Al Hissi, Mohamed Bamakhrama, Di Liu, Sjoerd Meijer, Sven van Haastregt, Dmitry Nadezhkin, Sobhan Niknam: it has been a great pleasure to work with you! Thanks to LERC, I have also been lucky to meet Jelena Spasic and Milos Acanski. All the conversations and dinners we had together are great memories for me. Moreover, I won't forget the many times I self-invited myself to have early breakfasts, on weekends, at your place (and you guys were so kind to let me in). In addition, I had great times working (and sharing an accommodation) with Teddy Zhai. Ted, I have always appreciated the discussions we had, regarding research and life in general. I will also not forget that, especially in the beginning of my PhD, every other day you were at my desk to help me or teach me some tricks. Of course, I appreciated even more the time we spent (and will spend) together outside the working hours. With regard to this, a big "thank you" goes to Shan for making our gatherings even more fun.

The research work described in this thesis has been carried out within the EU FP7 project MADNESS. Thanks to this project, I had the chance to meet fellow researchers from all over Europe, with whom I shared many enjoyable moments. My first thought goes to the EOLAB group from University of Cagliari. Luigi, Paolo, Giuseppe, Sebastiano, Simone: thanks a lot for making my visits in Cagliari always fun, despite the challenges we sometimes faced in our project. In addition, I truly enjoyed spending time during and outside office hours with Onur Derin and Roberta Piscitelli.

Outside of the professional context, I had the pleasure to meet many people during my stay in Leiden. I cannot list them all here; however, I'd like to mention especially Andrea and Helene for the happy times we had together. I am very glad that I could make you two meet each other.

Speaking about meeting special persons, I want to thank once more Teddy for making I and my beloved Sing-Cih to meet. Sing-Cih, you are by far the greatest and sweetest thing that has happened in my life since my arrival in the Netherlands. Many many thanks for all the joyful moments we shared, for your continuous support, for your patience and perseverance that has allowed our relationship to stay strong, despite the great physical distance that sets us apart. I really wish we will be able to close this distance soon!

Finally, I would like to thank Sing-Cih's family, and in particular Wang papa, Wang mama, Lisa, Deborah, and Enoch. I enjoy every time I can visit you in Taiwan, you make me really feel at home. Thank you for your kindness and for all your support!

Ringraziamenti

Vorrei ringraziare Tomaso, Jolija, Mykolas e Pietro per le belle serate passate assieme, per i leggendari bbq "scientifici", e per farmi riassaporare le mie radici udinesi in terra d'Olanda. Un grande grazie anche a Fabio, che ho conosciuto poco dopo aver iniziato a lavorare alla Lely, per le risate e i discorsi che hanno spesso arricchito i miei weekend nell'ultimo anno.

Un grazie speciale va a tutti gli amici che mi hanno supportato da lontano. Mi sento molto fortunato ad avere un gruppo di amici che tuttora riesco a capire al volo, nonostante ci si veda molto più raramente che in passato. Marco, Simone, Giorgio e Camilla, Alessio, Manuel, Tiffany, Enrica, Elena, Marsela e Camilla, Tristram: grazie mille per il vostro affetto e per saper illuminare le mie giornate tutte le volte che ritorno in Italia.

Inoltre, una componente fondamentale della mia vita, e mio grande orgoglio, è la mia famiglia. Alberto, Nadia, Andrea, Ludovica, Fiorella, Domenico, Amos, Noemi, Simonetta, Alessandro, Samuele, Susanna, Alessandra, Andrea, Pietro, Stefano, Paola, Nicola, Lisa, Matteo: ognuno di voi mi ha insegnato qualcosa, e rivedervi mi riempie ogni volta il cuore di gioia. Grazie mille anche a Zia Myriam e Zia Mirella per il grande affetto che mi dimostrate sempre!

Infine, era un mio desiderio che questa tesi si aprisse e si chiudesse con un pensiero ai miei fantastici genitori, le due persone che più di tutti hanno contribuito a farmi diventare la persona che sono. Papà, più passano gli anni e più riesco a comprendere i sacrifici che hai dovuto fare per crescere me e i miei fratelli. Per questo, per tutto quello che mi hai insegnato, e per tutto l'amore che mi hai donato, ti sarò sempre riconoscente. Mamma, purtroppo tu non riuscirai a leggere queste parole, ma spero che in qualche modo ti possano raggiungere. Penso che le cure, l'affetto, le gioie, l'amore che mi hai regalato siano qualcosa di inestimabile. Tu sei stata una mamma, ed in generale una persona, assolutamente straordinaria. Ad entrambi voi voglio dire: essere vostro figlio è la mia fortuna ed il mio orgoglio più grande.