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

Version:	Not Applicable (or Unknown)
License:	Licence agreement concerning inclusion of doctoral thesis in the Institutional Repository of the University of Leiden
Downloaded from:	https://hdl.handle.net/1887/43469

Note: To cite this publication please use the final published version (if applicable).

Cover Page

Universiteit Leiden

The handle http://hdl.handle.net/1887/43469 holds various files of this Leiden University dissertation

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

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-

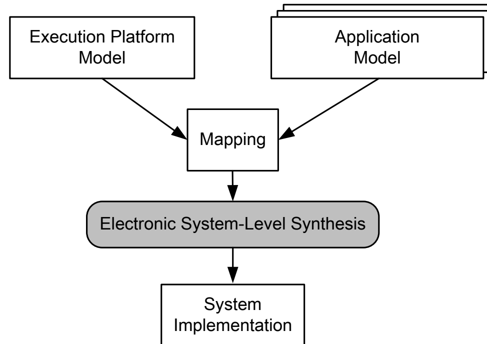


Figure 1.1: *System-level design methodology.*

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.

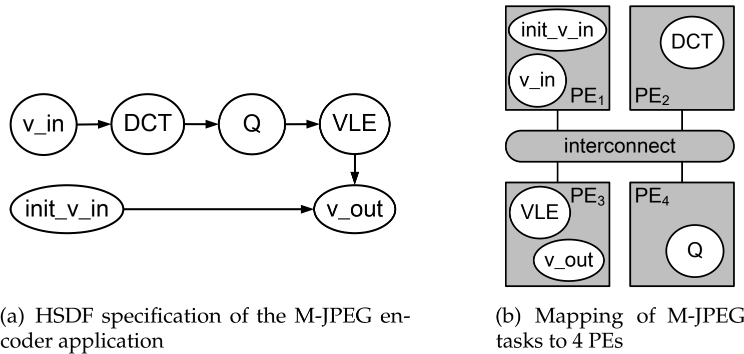
Since embedded systems are now shifting from uniprocessors to multiprocessors, several changes in the design methodology are required. In particular, the two main problems that designers face are: (i) how to model the applications such that the multiple processing resources available in modern execution 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.

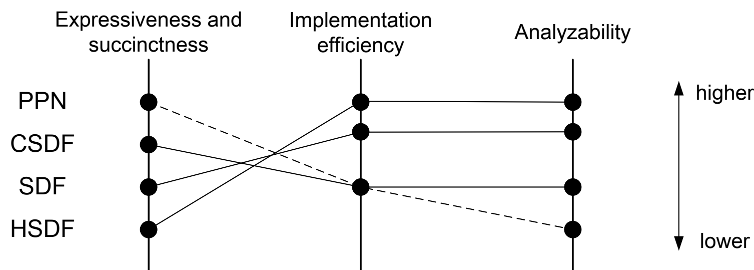


Figure 1.3: *Comparison of the MoCs considered in this thesis.*

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.

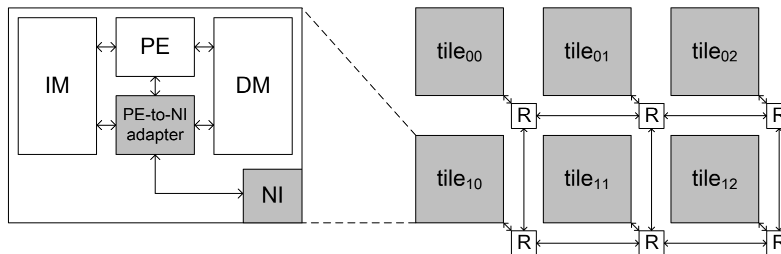


Figure 1.4: *The left part of the figure shows the internal structure of a tile. In each tile, the processing element (PE) performs computations using its data memory (DM) and instruction memory (IM). Communication with the rest of the NoC is handled by the Network Interface (NI) and the adapter between 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.

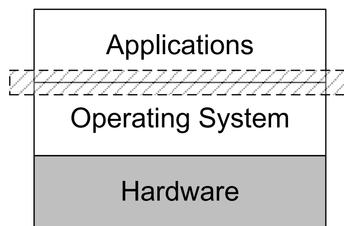


Figure 1.5: *Sketch of the hardware/software stack of a computing system. The dashed area highlights the interface between the application layer and the OS layer.*

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.

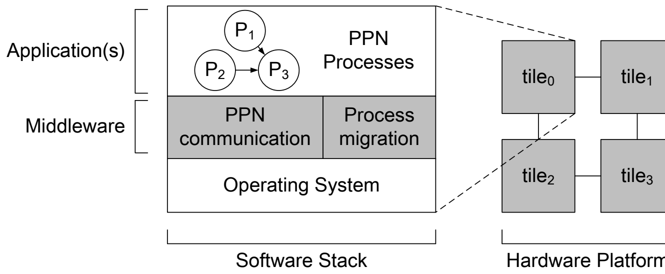


Figure 1.6: *Software stack (left) proposed to achieve adaptivity in BE systems. The middleware layer is denoted by the shaded area. The stack is deployed on each tile of the hardware platform (right).*

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.