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 multi-processor 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^{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.

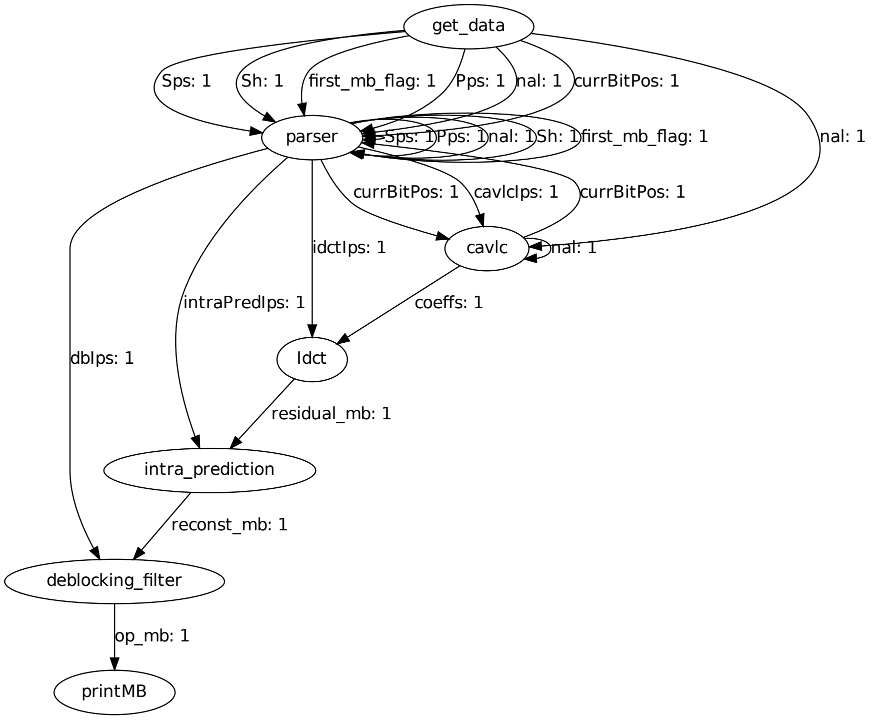

Figure 7.1: *PPN model of the H.264 decoder application.*

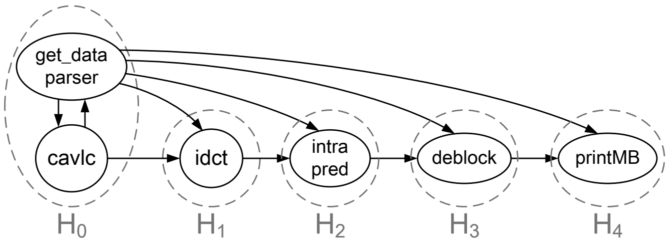

Figure 7.2: *Abstracted PPN specification of the H.264 decoder application. Compared to Figure 7.1, nodes get_data and parser are merged into a single node, H_0.*

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

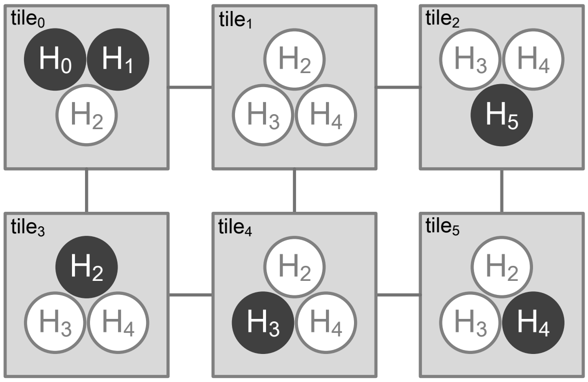

Figure 7.3: *Structure of the execution platform used in our demo and allocation of process replicas to tiles. Process replicas which are active at system startup are filled in dark gray. Input and output interfaces are not shown.*

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^{RT}

The scheduling framework proposed in [BS11, BS12] (shown in Figure 5.1 on page 88) has led to the implementation of Daedalus^{RT} [BZNS12, Bam14]. Daedalus^{RT} combines

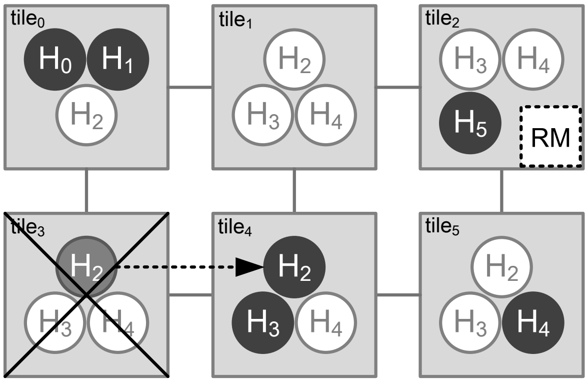

Figure 7.4: *Example of a process migration performed in our demo. The user requires the system to deactivate tile$_3$. Then, the resource manager (RM) which executes on tile$_2$ triggers the migration of process H$_2$ from tile$_3$ to tile$_4$ in order to keep the application running.*

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)

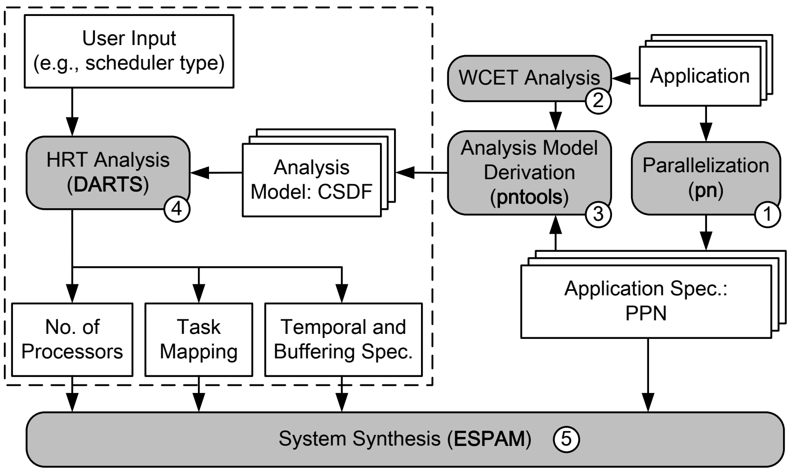

Figure 7.5: *Overview of the Daedalus^{RT} design flow (adapted from [BZNS12]). Steps ①, ② and ③ of the design flow are used to convert the input sequential application to the corresponding PPN specification and CSDF analysis model. Then, based on this analysis model and on the scheduler chosen by the 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.