Dynamic Task Management in MPSoC Platforms

Von der Fakultät für Elektrotechnik und Informationstechnik
der Rheinisch–Westfälischen Technischen Hochschule Aachen
zur Erlangung des akademischen Grades
eines Doktors der Ingenieurwissenschaften
genehmigte Dissertation

vorgelegt von
Diplom–Ingenieur Diandian Zhang
aus Jiangsu, VR China

Berichter:  Universitätsprofessor Dr.-Ing. Gerd Ascheid
Universitätsprofessor Dr. rer. nat. Rainer Leupers

Tag der mündlichen Prüfung: 11.04.2017

Diese Dissertation ist auf den Internetseiten
der Hochschulbibliothek online verfügbar.
Acknowledgements

This thesis is the result of my work as research assistant at the Institute for Integrated Signal Processing Systems (ISS) at the RWTH Aachen University. I am sure that the time at ISS will be one of the most important periods in my life. When I look back at the time, I can still feel the enthusiasm for research, the inspiring discussions and the shining lights in late evenings at the institute. During the time, I met many brilliant people, who are so different in their own way, but all are kind. I had the fortune to work with them and want to take this opportunity to thank them for the wonderful time and their kind help and support.

First, I would like to sincerely thank my advisor Professor Gerd Ascheid for giving me the opportunity to join his team. I deeply appreciate his constant guidance and support in finishing this work, especially his patience and allowing mistakes, which were strong encouragement to me. I would also like to thank Professor Rainer Leupers who gave me many valuable suggestions to my work. In the cooperative projects with his team, I learned many things from him, especially how to concentrate on the key problems and solve them, and he made me become much more confident.

Furthermore, I would like to thank Jeronimo Castrillon. We worked seamlessly together on the OSIP project, which built the base of this work, and I could always get his opinions on the my further work on OSIP. I was always impressed by his fast understanding of problems and his ability of finding good solutions. No wonder that he has become a professor of the department of computer science of the TU Dresden. I am also very grateful to Han Zhang and Li Lu for their valuable experimentations and analysis made for this work. I would also like to thank Stefan Schürmanns for the pleasant teamwork in the project of high-level power estimation for NoCs, which provided me an initial NoC architecture to address the OSIP integration problem into NoC-based systems.

It was my fortune to work with a group of people having different areas of expertise. I could always get my questions answered. Many of them have helped me, even though many of them probably do not remember their help any more. Some of them have become good friends of mine, and some of them will probably only be able to be met occasionally. But here I want to say a sincere thank you to them all, especially to Anupam Chattopadhyay, David Kammler, Martin Witte, Torsten Kempf and Filippo Borlenghi from the architectures group, Jeronimo Castrillon, Stefan Schürmanns, Christoph Schumacher, Hanno Scharwächter, Weihua Sheng, Jianjiang Ceng and Felix Engel from the tools group, Dan Zhang, Xitao Gong and I-Wei Lai from the
algorithms group. I also owe my special thanks to Elisabeth Böttcher, Christoph Vogt, Ute Müller, Michael Rieb, Gabi Reimann, Christoph Zörkler and Tanja Palmen for the years of kind help on the non-technical part of my time at ISS.

Finally, I want to express my deepest thanks to my parents and my wife Yong. They are always there for me. Without their patience and consistent support, it would not have been possible to complete this work.

Diandian Zhang, June 2017
## CONTENTS

2.6 Summary ......................................................... 30

### 3 OSIP-based Systems .............................................. 31

#### 3.1 System Overview ............................................. 31

- Basic Concept ......................................................... 31
- Programming Model and Software Integration .................. 32
- Hardware Integration ................................................. 35

#### 3.2 OSIP Architecture Development ............................ 38

- Design Flow ......................................................... 38
- Application Analysis ............................................... 40
- Processor Architecture ............................................. 43
- OSIP Code Example ................................................. 55

#### 3.3 Results .......................................................... 56

- Area, Timing and Area-Time Product (AT) .......................... 56
- Power, Energy and Area-Time-Energy Product (ATE) ............ 59

#### 3.4 Summary ......................................................... 61

#### 3.5 Discussion ..................................................... 61

- OSIP Address Generation Unit (OSIP-AGU) ....................... 61
- Node Comparator ...................................................... 62

### 4 System-Level Analysis of OSIP Efficiency .................... 63

#### 4.1 System Setup .................................................. 63

#### 4.2 Benchmark Applications ....................................... 64

- Synthetic Application ................................................. 65
- H.264 Video Decoding ................................................. 67

#### 4.3 Efficiency Analysis with an Ideal Communication Architecture .... 68

- Synthetic Application ................................................. 68
- H.264 ................................................................. 70

#### 4.4 Impact of Communication Architecture ...................... 73

- Detailed Analysis of Communication Overhead ................. 74

#### 4.5 Optimized Communication Architecture ....................... 76

- Multi-layer AHB ....................................................... 76
- Cache System ......................................................... 77
## Contents

4.5.3 Write Buffer .................................................. 79
4.5.4 Results ......................................................... 80

### 4.6 Joint Consideration of OSIP Efficiency and Communication Architecture

#### 4.6.1 H.264 ...................................................... 84
#### 4.6.2 Synthetic Application ................................. 84

### 4.7 Summary ....................................................... 89

5 OSIP Support for Efficient Spinlock Control ......................... 91

#### 5.1 Research on Spinlocks ..................................... 92
#### 5.2 Basic Spinlock Control in OSIP-based Systems ............ 94
#### 5.3 Application-Aware Spinlock Control ....................... 95
      5.3.1 Key Concept ............................................. 95
      5.3.2 APIs for Spinlock Reservation ......................... 95
      5.3.3 Enhanced Spinlock Control ............................. 96
      5.3.4 Hardware Overhead .................................... 100

#### 5.4 Case Studies ................................................ 100
      5.4.1 Synthetic Application .................................. 100
      5.4.2 H.264 ................................................... 104

#### 5.5 Joint Impact of OSIP Efficiency and Flexibility ............ 105
      5.5.1 Synthetic Application ................................. 106
      5.5.2 H.264 ................................................... 108

#### 5.6 Summary ....................................................... 110

#### 5.7 Discussion .................................................. 110
      5.7.1 Tool Support for Spinlock Reservation ................ 111
      5.7.2 Scalability Problem ..................................... 111
      5.7.3 Nested Spinlocks ....................................... 112

6 OSIP Integration in NoC-based Systems ................................ 113

#### 6.1 OSIP Integration Problems in NoC-based Systems .......... 114
#### 6.2 Basic Concept of OSIP Integration in NoC-based Systems ..., 114
#### 6.3 Case Study .................................................. 116
      6.3.1 NoC Structure .......................................... 116
      6.3.2 Application – MIMO-OFDM Iterative Receiver ........ 124
6.4 System Implementation ................................. 128
  6.4.1 Data Flow Graph .................................. 129
  6.4.2 VPU Assignment .................................. 129
  6.4.3 Node Mapping .................................... 130
  6.4.4 VPU Subsystem .................................. 131
  6.4.5 OSIP Subsystem .................................. 132
  6.4.6 Preliminary System Analysis .................... 135

6.5 Enhancements for NoC-based Communication Architecture .............. 136
  6.5.1 Increasing link data width .......................... 136
  6.5.2 DMA Support .................................... 136
  6.5.3 Pipelined Execution ............................... 136
  6.5.4 Performance Analysis with Enhanced Communication Architecture .... 139

6.6 Analysis of OSIP Efficiency in NoC-based Systems ....................... 140

6.7 Summary ............................................... 142

6.8 Discussion ............................................ 143
  6.8.1 Proxy Complexity ................................ 143
  6.8.2 Multi-OSIP-System .............................. 145

7 Conclusion and Outlook .................................. 147

Appendix ..................................................... 151

A Packet Types and Communication Protocol of NoC .......................... 151
  A.1 Packet Types ........................................ 151
  A.2 Communication between IP components ......................... 151
    A.2.1 Transmission of Read Request Packet (PCK_R) .............. 152
    A.2.2 Transmission of Read Response Packet (PCK_RR) .......... 153
    A.2.3 Transmission of Write Request Packet (PCK_W) ............ 153
    A.2.4 Transmission of Write Response Packet (PCK_WR) .......... 153
    A.2.5 Transmission of Synchronization Packet (PCK_S) .......... 153
    A.2.6 Constraint ...................................... 153

Glossary .................................................... 157
## CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>List of Figures</td>
<td>161</td>
</tr>
<tr>
<td>List of Tables</td>
<td>165</td>
</tr>
<tr>
<td>Bibliography</td>
<td>167</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

The fast development of the submicron silicon technology enables integrating a huge number of transistors on a single chip. Billion-transistor chips are common today. Well-known examples are Intel Xeon® server processor (4.3B transistors) [158], IBM’s Power8 processor (4.2B transistors) [90] and Nvidia’s Kepler GK110 Graphical Processing Unit (GPU) (7.1B transistors) [147]. According to the International Technology Roadmap for Semiconductors (ITRS), chips with around 50 billion transistors will emerge in the market in 2026 [93].

However, the consistently shrinking silicon technology is reaching its physical limit caused by the size of the silicon atom. The ITRS already predicts a gate length of 5.9 nm in 2026, which is only 25 times as large as the silicon atom diameter of 0.234 nm. Therefore, increasing the computing performance of future chips will be faced with large limitations, if only considering using small silicon technologies for achieving a higher clock speed. In fact, the improvement of clock frequency is much slower than that of chip capacity, as shown in Figure 1.1.

In addition to the gap between performance and chip capacity regarding the clock speed, power and energy consumption is another important topic in today’s chips. With the evolution of the silicon technology, the functions of traditional PCs can be moved into embedded domain. Nowadays, multimedia wireless devices like smartphones and tablets can be found everywhere. They are actually replacing the role of

Figure 1.1: ITRS 2012: Predictions for number of transistors per chip at production and clock frequency
traditional PCs on the electronic market. In these devices, complex applications such as high-resolution image processing, 3-D gaming and wireless baseband processing are very common. It can further be easily imagined that more complex applications will certainly occur in the future. These complex applications on the one hand require high computational performance, on the other hand consume a lot of power and energy, which is critical for handheld devices. Unfortunately, at the deep submicron dimension leakage current increases dramatically, which worsens power and energy consumption and at the same time also causes problems in chip reliability. Therefore, low-power design is also faced with big challenges at small technologies.

There are trends of looking for new materials such as high-κ-dielectrics [24] to replace the conventional silicon dioxide. Research is also going on in "More than Moore" technologies like memristors [175] and spintronics [203] as well as new computing technologies like quantum computers [55, 59] and DNA computers [6, 91]. Apart from new materials and new technologies, Multi-Processor Systems-on-Chip (MPSoCs) provide another means to improve chip performance and power from the system architecture perspective. A big advantage of making improvement in the system architecture is that it is technology-independent. Therefore, they can be applied to different technology improvements orthogonally.

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

As the name implies, multiple processors exist in an MPSoC, which cooperate with each other to achieve common application targets. In comparison to single processor systems, applications are executed in parallel on multiple processors in MPSoCs. This parallel execution is the key of MPSoCs in solving performance, power and energy issues of today’s chips. It is not necessary any more to push the clock frequency to its limit in order to meet application performance requirements. Instead, based on the parallel execution, systems are able to operate at a lower clock frequency with the same or even higher performance. At the same time, low supply voltages are possible with a reduced clock frequency, which largely reduces dynamic power and energy consumption in a quadratic way. For leakage power, the more relaxed timing of circuits enables using low-leakage gates with a higher threshold voltage. Furthermore, it becomes more convenient to dynamically transit some of the processors in the system into power-saving states or even completely switch them off, if they are not being used.

In MPSoCs, processors are also often referred to as Processing Elements (PEs). Based on the PE types used in the system, MPSoCs can be classified into two categories: homogeneous and heterogeneous. In homogeneous systems, all PEs are of the same type. The first MPSoC – the Lucent Daytona [3] processor belongs to this category, which contains four PEs, each consisting of a 32-bit SPARC V8 core and a 64-bit vector coprocessor. Another good example of homogeneous MPSoCs is the ARM MPCore technology [11], which contains a configurable number of ARM11 processors. For heterogeneous MPSoCs, different PE types are employed in the sys-
1.1. Multi-Processor System-on-Chip (MPSoC)

system, such as Reduced Instruction Set Computers (RISCs), Digital Signal Processors (DSPs), Application-Specific Instructions-set Processors (ASIPs), Field-Programmable Gate Arrays (FPGAs) or Application-Specific Integrated Circuits (ASICs). Representative examples for this category are the Open Multimedia Application Platform (OMAP) family [191] from Texas Instruments (TI) and the Cell Broadband Engine from IBM [103]. The latest OMAP processor contains an ARM Cortex-A15, two ARM Cortex-M4 microcontrollers, a mini-C64x DSP, a PowerVR GPU and several other accelerators for image, video and audio processing. The Cell processor consists of a PowerPC and eight specialized coprocessors known as Synergistic Processing Elements (SPEs) for performing floating-point operations.

Both homogeneous and heterogeneous MPSoCs have their advantages and disadvantages. For homogeneous MPSoCs, the homogeneity of PEs makes the system architecture easily scalable, from both the hardware and the software perspective. Especially, programming such a system is relatively simple, which becomes very important for today’s large systems. Other advantages include convenient system debugging due to the same software tool set, higher fault-tolerance with multiple instances of the same PE type and low system development costs. The biggest disadvantage of homogeneous systems is their low efficiency in area, energy and power consumption. The PEs are mostly RISC-based for the programmability reason. Therefore, they cannot easily meet high performance requirement of complex applications without applying a large number of them, which reduces efficiency.

In contrast, the different types of PEs in heterogeneous MPSoCs provide a better possibility of making trade-offs between programmability and efficiency. A qualitative comparison between different PE types with respect to flexibility (mainly due to the programmability of PEs), performance and power dissipation is made in Figure 1.2. With different PE types available, fast PEs like ASICs or ASIPs can be chosen for performing performance-critical tasks, providing higher efficiency. For the tasks which require high flexibility instead of high computational performance, General Purpose Processors (GPPs) such as RISCs are preferred. However, this variety of PE types raises big challenges in programming such systems. Typical questions that need to be answered are: where to make suitable task partitions for different PEs, how to set up a unified programming environment with different software tool-sets and how to debug the system?

In general, MPSoC design contains a wide spectrum of research topics, ranging from the hardware architecture development to the system programming, and from the high-level design space exploration to the low-level system implementation. This thesis focuses on task management in MPSoCs in the embedded domain. Independent of the system types, whether homogeneous or heterogeneous, efficient task management always plays an essential role in achieving the design goals, such as achieving high performance, low power and energy consumption, or meeting real-time application constraints.
1.2 Task Management in MPSoCs

To be executed in parallel in the system, applications are partitioned into a number of tasks. The goal of task management in MPSoCs is to coordinate PEs to run tasks in such a way, that applications are functionally correctly executed and at the same time the application requirements (performance, real time, power etc.) are met. It includes simple basic task operations such as task creation, task deletion, task suspension and more complex task operations – task scheduling and task mapping. The scheduling decides the order of the task executions, while the mapping decides on which PEs the tasks are executed. Therefore, they are also often referred to as temporal and spatial mapping in the literature, respectively. In addition, synchronizations between tasks are also typical operations in task management. For a task manager, task scheduling, mapping and synchronization are the most computationally critical operations.

Task management in MPSoCs can be classified into static and dynamic. In static task management, task scheduling and mapping are fixed and applications are also fixed for the system. Generally, no new applications are allowed to enter the system during system execution. However, the behavior of today’s systems is mostly dynamic. Therefore, dynamic task management is nowadays more widely applied, which makes task scheduling and mapping decisions at runtime. This, however, requires an efficient but still flexible task manager.

From the organization perspective, task management can be classified into centralized and distributed. In the centralized organization, a single global task manager
is responsible for the whole system. In contrast, in the distributed task management multiple local task managers exist in the system. Nevertheless, efficient task managers are always needed, regardless of the task management organization.

In the literature, the implementation of dynamic task management in the embedded domain mostly either follows software approaches using RISCs or applies dedicated hardwired solutions based on ASICs. In the former approaches, task management is traditionally a part of an Operating System (OS). Typical examples are Linux mobile OSes with Symmetric Multiprocessing (SMP) support. These approaches often suffer from low efficiency in managing tasks for large MPSoCs and task management can easily become the system bottleneck. In contrast, in the latter approaches, task management is often separated from the OS, so that it can be accelerated by native hardware. Therefore, these approaches can provide high efficiency. However, they lack in flexibility, which makes them difficult to adapt to different system scenarios.

In this thesis, a centralized ASIP-based approach is presented, which combines the pros of both implementation types, namely the flexibility of RISCs and the efficiency of ASICs, and shows its unique advantages in the task management.

1.3 Contributions of this Thesis

The contributions of this thesis are summarized as follows:

- An ASIP has been developed, which is specialized for dynamic task management in MPSoCs. The ASIP is called OSIP, standing for Operating System application-specific Instruction-set Processor.

- The advantages of using an ASIP as the task manager of MPSoCs have been presented from both the efficiency and the flexibility perspective. New applications in system control have been identified for ASIPs in addition to their traditional usages in improving data processing.

- An extensive analysis has been provided for the joint impact between the efficiency of the task manager and communication architectures on system performance.

- An advanced spinlock control mechanism for MPSoCs has been proposed, which is based on the so-called spinlock reservation.

- A proof-of-concept of integrating OSIP into large-scale MPSoCs has been provided, in which distributed communication architectures based on the Networks-on-Chip (NoCs) communication paradigm are used.

1.4 Outline

In this chapter, the background and importance of MPSoCs are introduced. For large MPSoCs, efficient dynamic task management is highly demanded in order to utilize
the PEs of the system in an effective way, whether considering performance or power and energy consumption. At the same time, task management should still be flexible enough to make the system adaptable. The requirement of both efficiency and flexibility calls for an ASIP-based implementation of the task management, which is the main objective of this work. The remainder of this thesis is organized as follows.

Chapter 2 introduces related works on task management in MPSoCs. In addition, two state-of-the-art communication paradigms in MPSoCs — buses and NoCs — are introduced. Furthermore, a virtual platform, which is used as the evaluation environment in this work is described.

In Chapter 3, OSIP-based systems are introduced. The basic concept of these systems is explained from the perspective of the software and hardware integration. The main part of this chapter shows the development flow of the OSIP architecture and special hardware features to efficiently support dynamic task management. Hardware results of the area, maximum clock frequency and power of the OSIP architecture are reported.

In Chapter 4, the efficiency of OSIP is evaluated based on the comparisons with a RISC-based architecture and a hypothetical extremely fast ASIC implementation in a multi-ARM-processor system. This chapter also analyzes the joint effect of the task manager and the communication architecture on the system performance. The latter has a big impact on the utilization of the efficiency of the former.

Chapter 5 analyzes OSIP from the flexibility point of view. An advanced spinlock control mechanism is enabled by using the programmability of OSIP, which effectively reduces high randomness of spinlocks by introducing user-defined control algorithms in OSIP. The analysis results also show that the efficiency of a task manager is important for utilizing its flexibility.

Chapter 6 discusses the integration of OSIP into NoC-based systems to support large MPSoCs. OSIP is a central task manager, while NoCs are a distributed communication paradigm, which have big advantages in future large MPSoCs due to its scalability. This contradiction makes the integration difficult. In this chapter, a proxy-based solution is proposed and a proof-of-concept is made, demonstrating how to easily integrate OSIP in such NoC-based systems.

Chapter 7 summarizes this thesis and gives an outlook to possible extensions to OSIP-based systems.
Chapter 2

Related Work

Efficient task management plays an essential role in MPSoCs in maximizing system performance, meeting real-time constraints and minimizing power and energy consumption. In the literature, numerous works about task management in MPSoCs have been presented from various aspects. In this chapter, a short survey of the major related works is made, mainly considering the aspects from the dynamism, the organization and the implementation of the task management. As will be shown in later chapters of this thesis, communication architectures also have big impact on the efficiency of task management. So two mainstream communication architectures based on on-chip buses and NoCs are also briefly introduced in the chapter. Throughout this work, the MPSoCs are modeled using a virtual platform environment called Platform Architect to assess the proposed ASIP-based approach for the implementation of the task manager. Using a virtual platform has big advantages in fast system architecture exploration and convenient in-depth system-level performance analysis. Therefore, an introduction to the system-level exploration as well as to the Platform Architect is also made.

The organization of this chapter is as follows. First, related works on static and dynamic task management are introduced. Then, centralized and distributed task management are discussed. Afterwards, different types of implementations of the task manager are shown, which are either hardware-based or software-based. Then, the mainstream communication architectures and system-level exploration using virtual platforms are introduced. Finally, a short summary is given.

2.1 Static and Dynamic Task Management

Static and dynamic task management differ in the decision time of scheduling and mapping. While the former makes scheduling and mapping decisions at design time of a system, the latter makes the decisions at runtime.

2.1.1 Static Task Management

In static task management, both the applications and the underlying hardware platform are known to the designer at design time. The application information is often represented using Data Flow Graphs (DFGs) [125]. Depending on the optimization goals of the task management, specific models are typically created for the platform. Both the applications and the platform models are then taken as the inputs to an
optimization algorithm, the outputs of which are scheduled tasks and their spatial mapping to the PEs.

The accuracy of a platform model plays an important role in achieving good quality of task management. It can be a performance model, a power model, a thermal model or a system lifetime model. A performance model presents the execution time of tasks on different PEs of a system, in which the communication overhead is also often considered. A power model and a thermal model reflect the power consumption and thermal characteristics of system components during task execution, respectively. A system lifetime model describes the possibility of component failures caused by hardware wear-out. It is often applied for the optimization of system reliability. If multiple optimization goals are targeted in the task management, multiple platform models can be combined during the optimization.

Optimization algorithms of the task management are another important factor influencing its quality. Finding an optimal solution for task scheduling and mapping is NP-Complete [198]. However, as the optimization is performed at design time, complex algorithms such as Integer Linear Programming (ILP) [166], Genetic Algorithms (GA) [67], Tabu Search (TS) [65,66] and Simulated Annealing (SA) [102,200] can be applied to find optimal or close to optimal solutions. For example, in [155] and [96], ILP is used to optimize the energy consumption of the system and the execution time of the application, respectively. The authors of [70] use ILP for improving the power consumption while considering real-time constraints. Generic algorithms can be found in [112] and [206], which target the optimization of the application execution time and energy consumption in NoCs, respectively. Tabu search is used in [119] for task mapping in NoCs to guarantee the packet latency. Examples of using simulated annealing can be found in [148] and [121]. While the former targets improving the execution time, the latter also considers the energy consumption in addition. Other heuristics like Ant Colony Optimization (ACO) [56] and Constraints Programming (CP) [124] can be found in [26,75] and [157], respectively.

There are also customized optimization algorithms for task management presented in the literature. For example, in [207] two scheduling algorithms Scheduling1D and Scheduling2D are proposed for linear and general task graphs of streaming applications, respectively, optimizing the throughput, response time and energy consumption. In [36], the optimization algorithm is partitioned into four steps (task scheduling, processor mapping, data mapping and packet routing) to reduce the algorithm complexity. In [35], a heuristic called URSEM is applied to maximize the throughput of streaming applications by iteratively performing unrolling and re-timing the stream formats, targeting systems with scratchpad memories.

In Table 2.1 some research works in the literature on static task management are listed, showing a wide optimization spectrum. As the applications and the hardware platform are known at design time, static task management has a global view of the system. Enhanced by complex optimization algorithms in addition, static task management is able to achieve highly optimized task scheduling and mapping results. On the other hand, the pre-condition of having system information at design time makes static task management unsuitable for systems with dynamic behavior,
which is its major disadvantage. In fact, nowadays systems are becoming more and more dynamic, especially in the embedded domain. Simple examples are dynamic power management in MPSoCs for saving power and energy or downloading new applications in smartphones.

### 2.1.2 Semi-static Task Management

Semi-static task management is an enhancement to static task management to provide certain support for dynamic system behavior. In the literature, it is sometimes also called *quasi-static* or *pseudo-dynamic* task management. In this task management strategy, the target applications and the hardware platform are also known at design time. But, in contrast to a single fixed task scheduling and mapping configuration in static management, a set of task scheduling and mapping configurations are determined for multiple scenarios at design time in this management. Then at runtime, depending on the actual scenario the system is in, one proper configuration is selected out of the configuration set.

The scenarios can be distinguished between *use-case scenarios* and *system scenarios*. In use-case scenarios, task management focuses on different use cases of the system. Considering smartphones as an example, the use cases of a smartphone can be the place where it is used (e.g., urban or rural), or the different applications and their combinations running on it (e.g., making a call, playing MP3 or games). System scenarios are a complementary to use-case scenarios, which in addition consider the dynamic behavior of the applications and system resource requirements to reduce the system cost. For example, a video decoding normally requires different computational resources for different video frames. In this case, decoding the frames that require similar computational resources is considered as the same system scenario and those requiring different computational resources result in different system scenarios. Examples of using use-case scenarios for semi-static management are given in [211] and [174], and [63] and [210] are examples of using system scenarios.

In semi-static management big challenges are raised by a large number of scenarios, which grows exponentially with the number of the applications, different system environments as well as the internal application modes. Apart from the large exploration effort at design time for calculating an optimal solution for every scenario, the large number of scenarios would require a huge memory space to store the solutions. Furthermore, it would take a long time to select the optimal solution out of the large solution space for a given scenario at runtime. Therefore, various proposals are made in the literature to reduce the optimum solution space (Pareto configurations). In [211], a step-wise filtering is applied to reduce the Pareto configuration set according to the number of resources for each application. To reduce the selection overhead at runtime, the configurations are further sorted. In [176], a heuristic is used to determine the Pareto configurations in an iterative way for Synchronous Data Flow Graph (SDFG) based applications. It has a complexity of $O(hn^3)$, $n$ being the number of actors in the application graph and $h$ being the maximum hop distances between the actors, which makes the approach well scalable. In [152], simulations
and an analytical model are combined to prune the design space and accelerate the exploration time at the design phase. A framework called NASA is provided in [97] to allow the designer to configure and tailor different search algorithms for various design space exploration dimensions.

More related work on semi-static task management can be found in Table 2.1. In general, semi-static task management is much more flexible than the pure static management. It enables the system to effectively cooperate with changing scenarios for achieving optimal design goals. However, additional memory spaces are needed to store the pre-defined task scheduling and mapping configurations, which can often be large, depending on the system and application dimensions. Furthermore, the prerequisite that the applications and scenarios must be well known at the design time for the exploration largely limits the flexibility of this strategy by nature. It will be difficult to handle unknown system situations, e.g., adding new applications at runtime.

2.1.3 Dynamic Task Management

In dynamic task management, task scheduling and mapping configurations are calculated completely at runtime. It takes the actual system information as input, e.g., the PE status, the task information, the utilization of system resources like communication bandwidth and decides at what time which task should be executed on which PE. The strategy of runtime calculation of scheduling and mapping configurations is the key difference of the dynamic management from the semi-static management. The latter performs a runtime selection of pre-calculated configurations. With this strategy, dynamic task management well overcomes the limitations of static and semi-static task management that the applications and scenarios must be well known at design time. Unpredictable or unexpected changes at system runtime caused by e.g. new applications or defect PEs can be effectively tackled by re-scheduling and re-mapping of the application tasks. For today’s systems, which often present highly dynamic behavior, dynamic task management is especially important.

A list of researches in the literature on dynamic task management is also given in Table 2.1. Similar to static and semi-static task management, dynamic management can be applied for different optimization goals like performance, power, energy, etc., or combinations of them. For example, in [82], the application execution time is reduced by re-mapping the tasks based on the online detected task load of the PEs. In [107], guarantee of performance requirement is targeted by applying admission control to new applications and regulating (suspending and recovering) the running applications based on monitoring their progress. The work in [42] optimizes the communication energy in a NoC-based system. The task management of this system takes both the application characteristic and the user behavior into consideration. A Global-scheduling-based Reliability-Aware Power Management (G-RAPM) is applied in [153], which addresses possible transient faults during Dynamic Voltage-Frequency Scaling (DVFS). The task scheduling at runtime takes in addition the task recovery time caused by possible failures into account and improves both the en-
energy consumption and the system reliability. The system reliability problem is also addressed in [74] and [46] (further extended in [47]). In the former, the wear information of the system components is included in the dynamic mapping to increase the system lifetime, while in the latter, temperature hotspots and gradients are reduced by migrating tasks onto cool cores.

Since the scheduling and mapping decisions are made at runtime, complex heuristics such as simulated annealing or genetic algorithms are not suitable for dynamic management due to their large latency. Rather, relative simpler heuristics are preferred to enable the system to react to changing scenarios in time. Typical scheduling algorithms are, e.g., first-come first-served (FCFS), priority-based and fair queuing. These algorithms define the policies, which task should be executed as the next. For mapping, round-robin (RR) and priority-based are widely used algorithms. They decide which PE should execute the next task. Different PEs can be assigned with different priorities depending on the PE types and design optimization goals. A fast PE like ASIP will most probably be assigned with a higher priority than a slow PE based on RISC if the design targets performance optimization. Due to the limitations in applying highly complex heuristic optimization algorithms, dynamic task management is unlikely able to achieve the same optimization level as static/semi-static management, if the applications are known at design time. Also, compared to static management, dynamic management introduces additional runtime computational overhead. But it can efficiently meet the inherent challenges faced by static/semi-static management in handling unpredictable system, application and user behavior.

**Table 2.1:** List of task management references in the literature

<table>
<thead>
<tr>
<th>Reference</th>
<th>Strategy</th>
<th>Optimization Goal</th>
<th>Organization</th>
</tr>
</thead>
<tbody>
<tr>
<td>[112]</td>
<td>Static</td>
<td>Execution time</td>
<td>–</td>
</tr>
<tr>
<td>[173]</td>
<td>Static</td>
<td>Energy</td>
<td>–</td>
</tr>
<tr>
<td>[70]</td>
<td>Static</td>
<td>Power, realtime</td>
<td>–</td>
</tr>
<tr>
<td>[207]</td>
<td>Static</td>
<td>Throughput, response time, energy</td>
<td>–</td>
</tr>
<tr>
<td>[206]</td>
<td>Static</td>
<td>Execution time, energy</td>
<td>–</td>
</tr>
<tr>
<td>[119]</td>
<td>Static</td>
<td>Reliability, latency, energy</td>
<td>–</td>
</tr>
<tr>
<td>[148]</td>
<td>Static</td>
<td>Execution time</td>
<td>–</td>
</tr>
<tr>
<td>[157]</td>
<td>Static</td>
<td>Execution time</td>
<td>–</td>
</tr>
<tr>
<td>[160]</td>
<td>Static</td>
<td>Execution time</td>
<td>–</td>
</tr>
<tr>
<td>[26]</td>
<td>Static</td>
<td>Throughput</td>
<td>–</td>
</tr>
<tr>
<td>[114]</td>
<td>Static</td>
<td>Throughput, latency, resource utilization</td>
<td>–</td>
</tr>
<tr>
<td>[137]</td>
<td>Static</td>
<td>Energy</td>
<td>–</td>
</tr>
<tr>
<td>[155]</td>
<td>Static</td>
<td>Energy</td>
<td>–</td>
</tr>
</tbody>
</table>

Continued on next page
Table 2.1 – continued from previous page

<table>
<thead>
<tr>
<th>Reference</th>
<th>Strategy</th>
<th>Optimization Goals</th>
<th>Organization</th>
</tr>
</thead>
<tbody>
<tr>
<td>[36]</td>
<td>Static</td>
<td>Energy</td>
<td>–</td>
</tr>
<tr>
<td>[83] [84]</td>
<td>Static</td>
<td>Energy with performance constraints</td>
<td>–</td>
</tr>
<tr>
<td>[121]</td>
<td>Static</td>
<td>Energy, execution time</td>
<td>–</td>
</tr>
<tr>
<td>[162]</td>
<td>Static</td>
<td>Performance, communication latency</td>
<td>–</td>
</tr>
<tr>
<td>[17]</td>
<td>Static</td>
<td>Execution time, power</td>
<td>–</td>
</tr>
<tr>
<td>[131]</td>
<td>Static</td>
<td>Lifetime, cost</td>
<td>–</td>
</tr>
<tr>
<td>[195]</td>
<td>Static</td>
<td>Temperature</td>
<td>–</td>
</tr>
<tr>
<td>[194]</td>
<td>Static</td>
<td>Execution time</td>
<td>–</td>
</tr>
<tr>
<td>[40]</td>
<td>Static</td>
<td>Execution time</td>
<td>–</td>
</tr>
<tr>
<td>[35]</td>
<td>Static</td>
<td>Execution time</td>
<td>–</td>
</tr>
<tr>
<td>[32]</td>
<td>Static</td>
<td>Execution time</td>
<td>–</td>
</tr>
<tr>
<td>[120]</td>
<td>Static</td>
<td>Execution time, realtime</td>
<td>–</td>
</tr>
<tr>
<td>[96]</td>
<td>Static</td>
<td>Execution time</td>
<td>–</td>
</tr>
<tr>
<td>[216]</td>
<td>Static</td>
<td>Reliability</td>
<td>–</td>
</tr>
<tr>
<td>[75]</td>
<td>Static</td>
<td>Reliability</td>
<td>–</td>
</tr>
<tr>
<td>[123]</td>
<td>Semi-static</td>
<td>QoS, power</td>
<td>Centralized</td>
</tr>
<tr>
<td>[211]</td>
<td>Semi-static</td>
<td>QoS, power</td>
<td>Centralized</td>
</tr>
<tr>
<td>[208]</td>
<td>Semi-static</td>
<td>Execution time</td>
<td>Centralized</td>
</tr>
<tr>
<td>[52]</td>
<td>Semi-static</td>
<td>Reliability</td>
<td>Centralized</td>
</tr>
<tr>
<td>[176]</td>
<td>Semi-static</td>
<td>Throughput</td>
<td>Centralized</td>
</tr>
<tr>
<td>[210]</td>
<td>Semi-static</td>
<td>QoS, energy</td>
<td>Centralized</td>
</tr>
<tr>
<td>[63]</td>
<td>Semi-static</td>
<td>Energy, resource utilization</td>
<td>Centralized</td>
</tr>
<tr>
<td>[165]</td>
<td>Semi-static</td>
<td>Power</td>
<td>Centralized</td>
</tr>
<tr>
<td>[109]</td>
<td>Semi-static</td>
<td>Execution time</td>
<td>Centralized</td>
</tr>
<tr>
<td>[86]</td>
<td>Semi-static</td>
<td>Performance yield</td>
<td>Centralized</td>
</tr>
<tr>
<td>[111]</td>
<td>Semi-static</td>
<td>Reliability</td>
<td>Centralized</td>
</tr>
<tr>
<td>[163]</td>
<td>Semi-static</td>
<td>Reliability</td>
<td>Hierarchical</td>
</tr>
<tr>
<td>[54]</td>
<td>Semi-static</td>
<td>Reliability</td>
<td>Centralized</td>
</tr>
<tr>
<td>[87]</td>
<td>Semi-static</td>
<td>Reliability, energy</td>
<td>Centralized</td>
</tr>
<tr>
<td>[177]</td>
<td>Semi-static</td>
<td>Throughput, energy</td>
<td>Centralized</td>
</tr>
<tr>
<td>[174]</td>
<td>Semi-static</td>
<td>Execution time, solution quality</td>
<td>Centralized</td>
</tr>
</tbody>
</table>

Continued on next page
2.1. Static and Dynamic Task Management

Table 2.1 – continued from previous page

<table>
<thead>
<tr>
<th>Reference</th>
<th>Strategy</th>
<th>Optimization Goals</th>
<th>Organization</th>
</tr>
</thead>
<tbody>
<tr>
<td>[82]</td>
<td>Dynamic</td>
<td>Execution time</td>
<td>Centralized</td>
</tr>
<tr>
<td>[136]</td>
<td>Dynamic</td>
<td>Performance, resource utilization</td>
<td>Centralized</td>
</tr>
<tr>
<td>[128]</td>
<td>Dynamic</td>
<td>Energy</td>
<td>Centralized</td>
</tr>
<tr>
<td>[153]</td>
<td>Dynamic</td>
<td>Energy, reliability</td>
<td>Centralized</td>
</tr>
<tr>
<td>[43]</td>
<td>Dynamic</td>
<td>Energy, reliability, throughput</td>
<td>Centralized</td>
</tr>
<tr>
<td>[46] [47]</td>
<td>Dynamic</td>
<td>Temperature, reliability</td>
<td>Centralized</td>
</tr>
<tr>
<td>[218]</td>
<td>Dynamic</td>
<td>Communication cost</td>
<td>Distributed</td>
</tr>
<tr>
<td>[192]</td>
<td>Dynamic</td>
<td>Performance</td>
<td>Centralized</td>
</tr>
<tr>
<td>[201]</td>
<td>Dynamic</td>
<td>Performance yield</td>
<td>Centralized</td>
</tr>
<tr>
<td>[4]</td>
<td>Dynamic</td>
<td>Execution time, QoS</td>
<td>Centralized</td>
</tr>
<tr>
<td>[85]</td>
<td>Dynamic</td>
<td>Performance, communication cost</td>
<td>Centralized</td>
</tr>
<tr>
<td>[37]</td>
<td>Dynamic</td>
<td>Execution time, resource utilization</td>
<td>Centralized</td>
</tr>
<tr>
<td>[144]</td>
<td>Dynamic</td>
<td>Mapping time, quality</td>
<td>Centralized</td>
</tr>
<tr>
<td>[53]</td>
<td>Dynamic</td>
<td>Communication cost, latency, energy</td>
<td>Centralized</td>
</tr>
<tr>
<td>[181] [180]</td>
<td>Dynamic</td>
<td>QoS, energy</td>
<td>Centralized</td>
</tr>
<tr>
<td>[190]</td>
<td>Dynamic</td>
<td>Energy</td>
<td>Centralized</td>
</tr>
<tr>
<td>[164]</td>
<td>Dynamic</td>
<td>Power</td>
<td>Centralized</td>
</tr>
<tr>
<td>[178]</td>
<td>Dynamic</td>
<td>Energy, communication cost</td>
<td>Centralized</td>
</tr>
<tr>
<td>[74]</td>
<td>Dynamic</td>
<td>Reliability</td>
<td>Centralized</td>
</tr>
<tr>
<td>[104]</td>
<td>Dynamic</td>
<td>Execution time</td>
<td>Distributed</td>
</tr>
<tr>
<td>[57]</td>
<td>Dynamic</td>
<td>Temperature</td>
<td>Distributed</td>
</tr>
<tr>
<td>[193]</td>
<td>Dynamic</td>
<td>Execution time, energy</td>
<td>Centralized</td>
</tr>
<tr>
<td>[107]</td>
<td>Dynamic</td>
<td>QoS</td>
<td>Centralized</td>
</tr>
<tr>
<td>[170]</td>
<td>Dynamic</td>
<td>QoS</td>
<td>Distributed</td>
</tr>
<tr>
<td>[61]</td>
<td>Dynamic</td>
<td>Generic RTOS Framework</td>
<td>Distributed</td>
</tr>
<tr>
<td>[29]</td>
<td>Dynamic</td>
<td>Execution time</td>
<td>Distributed</td>
</tr>
<tr>
<td>[209]</td>
<td>Dynamic</td>
<td>Latency, power</td>
<td>Distributed</td>
</tr>
<tr>
<td>[62]</td>
<td>Dynamic</td>
<td>Temperature</td>
<td>Distributed</td>
</tr>
<tr>
<td>[41]</td>
<td>Dynamic</td>
<td>Energy</td>
<td>Centralized</td>
</tr>
<tr>
<td>[42]</td>
<td>Dynamic</td>
<td>Energy</td>
<td>Centralized</td>
</tr>
</tbody>
</table>

Continued on next page
### 2.2 Centralized and Distributed Task Management

From the point of view of the structural organization, task management in MPSoCs can be performed either in a central**ized** or *distributed* way. The organization is especially important for the systems that apply runtime scheduling and mapping, i.e. systems with dynamic or semi-static task management. In these systems, the organization has a big impact on two types of overhead which are essential for the system optimization: the communication overhead of collecting local or global runtime system information and distributing scheduling and mapping decisions to PEs, and the overhead of making runtime scheduling and mapping decisions. For static task management, the organization of the management is less relevant as these overheads do not exist since the complete system execution is already pre-defined at design time.

#### 2.2.1 Centralized Task Management

In centralized task management, one single PE is used in the system to monitor the system states and manages runtime task execution. Most work in the literature with semi-static management follows this approach due to a central storage of the scheduling and mapping configurations. One exception is proposed in [163], in which the task managers are organized in a hierarchical way targeting many-core systems. For dynamic task management, centralized approaches are also widely used. Classical bus-based systems prefer to these approaches due to the central communication architecture. Good examples are the early generations of the OMAP processors from Texas Instruments and the Snapdragon processors [154] from Qualcomm. In NoC-based systems, centralized approaches can also be often found. Examples are given in [53, 81, 145, 180, 181]. In [181] and [180], an ARM processor is used to map the tasks of the Digital Radio Mondiale (DRM) application in a NoC-based heterogeneous system at runtime, targeting the minimization of energy consumption of the system, while still providing the required Quality of Service (QoS). In [81], the task mapping algorithm is also implemented on an ARM processor to minimize the energy consumption for streaming applications. In that work, a coarse-grained reconfigurable hardware called MONTIUM [78] is also one object of the task management. In [145], fine-grained reconfiguration hardware is considered in the task management. A central OS running on a StrongARM processor tracks the utilization of the computation and communication resources and makes decisions accordingly. In [53], one PE of the system is chosen as the Manager Processor (MP) to control the system resource,
configuration and task manipulations (scheduling, binding, mapping and migration) with the goal of optimizing the network channel congestion.

A big advantage of managing tasks in a centralized way is that the task manager has a global view of the whole running system, such as task execution states, resource utilization and communication congestion. This enables the task manager to make efficient decisions with respect to a global optimization and largely reduces the risks of falling into locally optimal solutions. Furthermore, the implementation of such systems with a central task manager is rather intuitive and has a relatively low design complexity, both from the hardware and the software architecture perspective. For the user, programming such systems is also relatively straightforward, mainly focusing on the communication between the central manager and the rest of the system.

The major drawback of centralized approaches is their limited scalability, which can be critical, if applied to large systems such as many-core systems. In these large systems, a central task manager has to control the task execution over a large number of PEs, which can introduce extremely high workload to the manager. If the manager cannot provide a high scheduling and mapping efficiency to cope with the high workload, it can easily become the bottleneck of the system and consequently degrade the system performance. In addition, to make global optimal decisions on the task management, the amount of the system information collected by the central manager is high in large systems. This generates a lot of communication traffic to the manager, which can also make the communication near the manager become the system bottleneck.

### 2.2.2 Distributed Task Management

The scalability problem of centralized approaches can be well overcome by distributed task management, in which multiple PEs instead of one single PE are used to perform task management. These PEs are typically distributively located over the system from the topology view. Distributed task management can be further classified into fully distributed, hierarchical and cluster-based.

In the fully distributed case, each PE performs local task management. An example is presented in [62], in which a local core only communicates and exchanges tasks with its nearest neighbors with respect to reducing thermal hot spots and large temperature gradients on the chip.

An alternative to the fully distributed case are the hierarchical approaches, as proposed in [218] and [209], in which the task management is realized in two steps. The first step is performed in a global way, which is meant to find a good initial solution for the task mapping or migration by utilizing global system information statistics. In the second step, distributed algorithms are executed on local PEs to further improve the task mapping or migration results. A similar approach is proposed in [170], which mainly focuses on the task admission control. A central admission controller is used to calculate the credits or rates of different actors (tasks) for each individual PE and distributes this information to them. After receiving the credits or the task rate information, each individual PE determines the task execution locally. In com-
comparison to the work presented in [218] and [209], this one exhibits more centralized characteristics.

Very often, task management in large systems is also organized using clusters [5, 29, 61, 104]. The clusters do not necessarily have a fixed size. Instead, they can be resized and organized virtually at runtime. Within each local cluster, one PE is chosen as the manager. As the clusters are built at runtime, the managers can, but not necessarily have to be bound to certain fixed PEs. In addition to local managers, global managers might also exist in the systems. In [5], global managers are applied, which have global information of all clusters. Therefore, when a task mapping request arrives at a global manager, it is able to choose the most suitable cluster to execute the task (probably with task migration), or even resizes the clusters, if the task can not be mapped to any of the existing clusters. Similarly, a global manager is applied in the system of [29], in which the position of each local manager is fixed and the clusters are initially equally sized. During runtime, the cluster sizes can vary. In case that more computational power is needed, a local manager can borrow processing resources from other clusters over the global manager. In contrast to the two systems above, the system presented in [104] does not have a global manager. Each local cluster manager negotiates processing resources with its neighboring managers and identifies its local domain for an application. During the time of building the local domain, the role of the task manager can be transferred from one PE to another within the cluster. Generally, due to the virtual clusters and variable local resource managers, these cluster-based approaches are more suitable for homogeneous systems.

Distributed task management has a unique advantage over centralized task management in the scalability, which makes it more suitable in supporting large systems with many cores. At the same time, its nature also implies that it mostly only uses local system information for making decisions. Because of the lack of global information, there is a high possibility that only sub-optimal solutions are found. For this reason, in many distributed approaches, as shown above, a global manager is still used to guide the task management in local domains. Furthermore, in local domains, mostly one PE is selected to manage each domain in a kind of centralized way. Therefore, despite the difference of the centralized and distributed approaches, an efficient implementation of the task manager is usually highly desired, no matter whether it is used globally or locally.

2.3 Task Management Implementations

The efficiency of a task management implementation largely influences the efficiency of the system. It is especially important for dynamic task management, for which scheduling and mapping have to be calculated at runtime. A slow task manager in such systems would on the one hand result in a long decision time, which can be critical for the systems with high performance requirements such as real-time constraints. On the other hand, the task manager would not be able to react to changing system states in time, thus leading to sub-optimal scheduling and mapping decisions. In con-
contrast to dynamic task management, the efficiency requirement of the task manager in static and semi-static task management is relatively low because of the pre-calculated scheduling and mapping configurations at design time. In this section, a discussion about different implementations of the task manager is made, which range from RISC implementations to ASIC implementations, and the focus is laid on dynamic task management.

### 2.3.1 RISC-based Implementations

Most of the task managers in the literature have a RISC-based implementation. They apply RISC processors like ARM processors (e.g., in [81, 136, 144, 180, 181]), MicroBlaze processors (e.g., in [41, 42, 42, 193]) or SPARC processors (e.g., in [46, 47, 82]) to execute scheduling and mapping algorithms. There are also researches, in which the task manager is not really implemented, but given as an abstract generic processor model [53, 164] or based on SystemC threads [178], or uses a formal model [107, 170]. These implementations can most probably also be considered as RISC-based.

In the mainstream commercial devices, RISC architectures are also the most popular choice for the implementation of the task manager. For example, the master of the OMAP processors is based on the ARM architectures. While on the OMAP 1 platform, only one ARM926EJ-S processor controls one C55x DSP, in the latest OMAP systems, a MPCore technology based master (ARM Cortex-A15 with four parallel cores) is used to control a bunch of slave components including two ARM Cortex-M4 microcontrollers, a mini-C64x DSP, a PowerVR GPU and several other hardware accelerators for image, video and audio processing. Similar to the OMAP processors, the Qualcomm Snapdragon processors apply a single ARM11 processor as the master in its first generation, and later on a quad-core Krait processor (ARM instruction-set based) in the latest generation. The Samsung Exynos application processor family [159] also follows the same strategy. In its early generation, only one ARM processor is used to control the system. In the later generations multi-core processors based on the MPCore technology or even eight cores (a quad-core A15 and a quad-core A7), configured as the ARM big.LITTLE architecture [72], are used. In the big.LITTLE architecture, the quad-core A15 is larger and has higher performance, while the quad-core A7 is smaller but more power-efficient. Depending on the system load, the task management can be internally switched between both quad-cores. Another commercial example is made by the Cell Broadband Engine, which was jointly developed by IBM, Sony and Toshiba. In this system, one PowerPC is used as the master of the system, which runs the operating system and coordinates eight specialized coprocessors known as Synergistic Processing Elements (SPEs).

RISC-based implementations of a task manager have very high flexibility in handling tasks. Different algorithms can be executed by the manager and easily extended and upgraded. As there is no hardware support for the task management algorithms, this sort of implementations are considered as pure software implementations. Conceptually, they are well suited to dynamic task management, in which flexibility is needed due to the system dynamism. Furthermore, the convention of running an OS
RISC processors for managing tasks makes the system able to be used and controlled in a convenient way for the user. However, the main disadvantage of using RISC processors as the task manager is their low efficiency, which can raise serious performance issues for the systems with high performance requirements. In the literature, several to several tens of milliseconds are often reported for the task scheduling and mapping [81, 164, 180, 181, 190]. As a comparison, the round-trip of processing a slot of one radio frame according to the LTE standard is allowed to only have a latency of 4 ms at maximum. For such systems, RISC-based implementations can become inapplicable without further optimizations in the algorithms.

2.3.2 ASIC-based Implementations

Due to the inefficiency of pure software implementations for the task manager based on RISC processors, especially for large MPSoCs, researches are made to consider shifting the implementation from software to hardware in order to improve the efficiency. Several ASIC-based implementations are proposed in the literature, which provide native hardware support for the OS functionality either in uni-core platforms [105, 140, 142, 146] or in MPSoCs [113, 115, 141, 150, 169]. In the following, the ASIC support for the MPSoC task management is further introduced.

**HOSK** The Hardware Operating System Kernel (HOSK) [150] is a hardware coprocessor, which performs task scheduling, inter-processor communication and context switching in multiprocessor systems. It consists of three components: a main controller, a thread manager and a context manager. The main controller receives the service requests from the PEs of the system and controls the other two components. The thread manager schedules the threads and provides inter-processor communication services based on hardware semaphores. A single thread queue is implemented in the thread manager, which supports up to 128 threads. In the context manager, the context of the scheduled and suspended threads is maintained. However, to make the context manager to work properly, a dedicated context controller needs to be implemented at the side of each PE to cooperate with the context manager. This makes the integration of HOSK difficult into the systems using off-the-shelf processors. HOSK has low context switch overhead. Less than 1% overhead is reported for 1 kcycle threads, if using a wide 544-bit context bus.

**CoreManager** The CoreManager [113, 169] is a hardware scheduler designed to support task management in systems targeting wireless communication applications such as LTE and WiMAX. It detects and solves task dependency at runtime and schedules ready tasks to the PEs. The PEs can be instruction-set processors or ASICS. A programming model is provided along with CoreManager, supporting a Model of Computation (MoC) similar to SDFGs. High scheduling efficiency is reported for CoreManager (60 cycles to schedule a task in average), which however is at the cost of high area overhead.
2.3. Task Management Implementations

**SystemWeaver** The approach of SystemWeaver [115] focuses on the issue of task scheduling and mapping on heterogeneous MPSoCs, supported by a programming model. It has a slightly higher flexibility than HOSK and CoreManager by allowing the user to compose different basic scheduling and mapping primitives so as to construct complex algorithms. The construction of the algorithms is built upon task and PE queues, over which a hardware state machine is executed to make scheduling and mapping decisions. However, its flexibility is still rather limited and the usage is rather difficult due to its design complexity.

**HW-RTOS** The HardWare Real Time Operating System (HW-RTOS) [34] is a dedicated hardware implementation of the standard OS layer. In [141], it is integrated into Symmetric Multiprocessing (SMP) architectures to support task synchronization and scheduling in multiprocessor systems. In the system, a HW-RTOS instance is created for each PE, and the multiple HW-RTOS instances are synchronized using a common Lock Unit to protect critical shared memory accesses. A task queue is implemented in HW-RTOS, and the task scheduling is based on the round-robin policy. As a case study, a dual-ARM system is analyzed. A major problem of this architecture is its scalability. To support a large number of PEs, the area increases significantly due to multiple HW-RTOS instances.

The ASIC-based implementations for dynamic task management largely increases the scheduling and mapping efficiency in comparison to the RISC-based implementations. On the other hand, the flexibility of these implementations is low. They can only support limited scheduling and mapping algorithms or narrow application domains, or have difficulties in integration into common systems due to the hardware specialty. Furthermore, the additional hardware logic introduces area overhead, which is often not negligible.

### 2.3.3 ASIP-based Implementations

While dynamic task management requires both flexibility and efficiency in scheduling and mapping, and both RISC-based and ASIC-based implementations only partly fulfill the requirements (RISCs for flexibility and ASICs for efficiency), new implementations need to be found that can fill the gap between RISCs and ASICs. As shown in Figure 1.2 of Chapter 1, the uniqueness of ASIP in combining programmability and native hardware support for performance-critical operations provides a promising solution for solving this problem. This motivates an ASIP-implementation for dynamic task management.

In the literature, ASIP designs are widely used for intensive data processing, such as baseband signal processing [171], multimedia signal processing [77], or cryptographic operations [99], etc. For supporting control-centric applications, in this context, managing tasks as a system controller, using ASIPs is still not very popular. Two architectures have been found so far in the literature, which follow the ASIP concept
for this purpose. The one is the ASIP-CoreManager presented in [15] and the other one is the OSIP architecture presented in [31].

**ASIP-CoreManager** The ASIP-CoreManager presented in [15] is an ASIP-like version of the hardwired CoreManager in [113, 169]. Its development was motivated by the analysis on the performance and scalability of an RISC-based (ARM926) implementation for the same functionality of the hardwired CoreManager (see [14]). This ASIP architecture is an extension of a basic Tensilica Xtensa LX4 core [28], featuring Very Large Instruction Words (VLIW) and Single Instruction Multiple Data operations (SIMD). A set of customized instructions are introduced, under which many are used for solving task dependencies or enhancing normal arithmetic and logic instructions. In comparison to the original Xtensa LX4 core, the ASIP-CoreManager doubles the area, but can reduce the execution time by up to 97% when checking task dependencies.

**OSIP** In comparison to CoreManager and ASIP-CoreManager, which support a limited set of applications based on SDF, another ASIP-based implementation called OSIP is presented in [31], which provides more generic support for task management. OSIP stands for Operating System application-specific Instruction-set Processor. It has a Harvard architecture with 6 pipeline stages, containing special instructions to handle the operations related to lists, which are a common data structure used in operating systems. A programming model and hardware interfaces are provided along with OSIP, which enable an easy integration of OSIP into different MPSoCs. During runtime, OSIP communicates with all PEs of the system, which can be homogeneous or heterogeneous, and schedules and distributes tasks based on user-defined algorithms. In summary, the OSIP-based task management is ASIP-based and centralized dynamic task management.

In [31], a preliminary analysis of OSIP efficiency in task management is made. This thesis provides a much more extensive analysis of OSIP efficiency as well as its flexibility from the system perspective in order to highlight the applicability and advantages of using ASIPs as the task manager in general. Furthermore, a proof-of-concept of integrating OSIP, which is a central task manager, into large MPSoCs connected by NoCs, which are distributed architectures, is provided. The results presented in this thesis have been published in part by the author in [31, 213–215].

### 2.4 Communication Architectures

Using an ASIP for managing tasks combines the advantages of RISCs and ASICs. However, evaluating the performance of an ASIP for this kind of control-centric applications is not a simple task. It largely differs from the evaluation for the ASIPs used for data processing. For the latter, the performance evaluation can be reasonably done in a standalone way, i.e., isolated from the other components of the system. This is because data-centric processing normally has a relatively small performance vari-
2.4. Communication Architectures

ance, which enables the designer easily to judge whether an ASIP implementation is suitable for given applications or not. For example, in the domain of signal processing, measuring the latency and throughput of executing certain algorithm kernels is a typical way of evaluating the ASIP performance.

In contrast, the performance of a control-centric ASIP has to be evaluated from the system-level, since the ASIP in principle controls the complete system. So, the evaluation has to be jointly investigated with other system aspects, such as the system size, task sizes (execution time of tasks), etc. Among them, the communication architecture plays a specially important role in evaluating the efficiency of an ASIP as the task manager. As both the communication architecture and the task manager are shared resources, they compete with each other when determining the overall system performance.

The two mainstream communication architecture paradigms of today’s MPSoCs are bus-based and NoC-based. In the following, some basics of both communication paradigms are introduced.

2.4.1 On-Chip Bus

Buses are a traditional on-chip communication architecture, which is nowadays still widely used in modern system designs. Representative bus examples are the ARM Advanced Microcontroller Bus Architecture (AMBA) series [10], IBM CoreConnect [89], STBus of STMicroelectronics [184] and Sonics SMART Interconnect [182].

On-chip buses are a centralized communication architecture, connecting all system components, as shown in Figure 2.1. It typically has two types of sockets: master sockets and slave sockets. The master sockets are used to connect the system components that initiate requests to the bus, e.g., PEs. These components are considered as the masters of the bus. Analogously, the slave sockets are connected to the slaves of the bus, e.g., memories and peripherals. The slaves react to the bus requests sent from the master sockets. There are also system components such as Direct Memory Access (DMA) controllers, which are both the masters and the slaves of the bus at the same time.

The basic bus signals can be divided into three groups: data signals, address signals and control signals. The data signals transfer the real data payload exchanged between the masters and slaves. The address signals are used to identify which slave component is accessed. The identification of the accessed slave component is made by a decoder inside the bus. The control signals defines bus access types, e.g., write/read signals, burst accesses, data masking, etc. Being a shared resource, it is very likely that multiple masters access the bus at the same time. In order to ensure that only one master gains the access to a slave at a time, arbiters are used in the bus. Different arbitration schemes can be implemented in the bus, ranging from simple schemes like static priority-based and round-robin to complexes ones like dynamic priority-based or fair-among-equal.

To cope with the increasingly complex systems of today, the bus complexity increases steadily as well. Taking the AMBA series as an example. In the Advanced
High-Performance Bus (AHB) of AMBA 2.0, multiple masters can be connected to the bus, and pipelined read or write operations and simple split transactions are supported. In AMBA 3.0, the Advanced Extensible Interface (AXI) bus incorporates five separate channels (read address, read data, write address, write data and write response), which enable simultaneous read and write transactions and complex out-of-order transactions. The AXI Coherency Extension (ACE) of AMBA 4.0 further adds three new channels (coherent address, coherent response and coherent data) to efficiently support cache coherency protocols.

Having been successfully applied in small-scale systems, bus-based communication architectures, however, have a major disadvantage in supporting system scaling due to long bus wires. In today’s deep submicron silicon technology, long wires crossing the whole chip can introduce severe problems in signal integrity due to significantly increased signal noise caused by crosstalk and electromagnetic interface, etc. This makes bus-based communication architectures conceptually not suitable for large-scale systems like many-core systems, which can contain tens or even hundreds of cores.

### 2.4.2 Network-on-Chip (NoC)

Networks-on-Chip are a newer communication architecture paradigm which is able to efficiently cope with the scalability problem in large-scale systems [19]. It borrows the idea of the layered communication approach from the computer networks and introduces packets into on-chip communication.

Representative industrial NoCs are the Intel TeraFLOPS NoC [199], the Phillips Æthereal NoC [68,69], the Octagon [100] and its extension Spidergon STNoC [44,45] of STMicroelectronics, the Sonics SGN [182] and the Arteris FlexNoC [16,110]. From the academic domain, numerous NoC architectures also have been proposed, which include the HERMES NoC [135,149], the MANGO network [21,22], the Nostrum
In NoC-based systems, system components or subsystems are often referred to as nodes. The communication between the nodes takes place through one or several routers, as depicted in Figure 2.2. The data from the node that initiates the communication are forwarded from one router to the next, until the target node of the communication is reached. By this, the long communication wires existing in buses are practically segmented by the routers, which is the main reason for the high scalability of the NoC-based communication architecture paradigm.

The communication data between the nodes are organized in packets, the format of which is known to the routers. In addition to the actual data payload, a packet contains the information needed for the routing, such as the coordinates of the communication node pair and the packet size, etc. In the data payload, semantic information such as packet type (read, write, response, etc.) can be included, so that the target node is able to understand the purpose of the packet. More advanced information such as credits can also be added into packets for better management of resources.

According to [19], a simplified ISO-OSI reference model can be applied to the communication stack of NoCs, which from the bottom up consists of the physical layer, the data link layer, the network layer, the transport layer and the software layer. Routers and the physical connections between the routers implement the lowest three layers of the stack. To abstract the implementation detail at these low layers for establishing end-to-end communications between source and destination nodes, network interfaces (NIs) are inserted between the nodes and the underlying communication infrastructure, which implement the transport layer. At the source node side, the routing-specific information is encoded into data packets. At the destination node
side, the NI extracts the payload data including the semantics from the packet for the
destination node for further processing. At the software layer of the NoC commu-
nication stack, the operating system, being either centralized or distributed, and the
applications are located. Especially, a programming model is often needed to simplify
the accesses to the network and to enable an efficient control of the network resources.

Designing a NoC is a complex task. Many design aspects have to be considered
at the different layers, among which NoC topologies, routing schemes and algorithms
and the flow control are especially important.

2.4.2.1 Topology

The topology of a NoC defines the way how the nodes of the NoC are connected with
each other. Depending on the performance and cost requirements of the application
and the system, different topologies can be considered. The most common topology
is mesh, in which the connection of the NoC nodes is constructed as a grid. The topo-
logy of the NoC presented in Figure 2.2 is an example of the mesh topology. A slight
extension to the mesh topology is torus, in which connections are added between the
nodes at the opposite edges of the network to reduce the communication latencies
between these nodes. Other well-known topologies are, e.g., ring (a special case of
torus), octagon, spidergon, fat tree and butterfly. There are also irregular topologies,
which are mostly dedicated designed for specific applications. For example, unneces-
sary nodes of a regular topology are removed from the network to reduce the cost, or
a combination between different NoC topologies or even between a NoC and a shared
bus can be considered.

2.4.2.2 Routing Scheme

Routing schemes determine the paths, following which the packets travel from the
source nodes to the destination nodes. In general, routing paths of the packets can
either be pre-calculated before the source nodes send them, or be calculated at the
time when the packets arrive at different routers. The former is referred to as source
routing, in which routing paths are completely encoded into the packets and routers
only need to follow the paths. Source routing simplifies the router architecture, but
the data payload is reduced due to the additional path information in the packets.
In the latter case, each router itself determines the routing direction. The routing
algorithm in the routers is implemented using either dedicated hardware logic or the
so-called routing tables which can be configured.

When packets are transferred between the routers, different switching mecha-
nisms can be applied, such as store and forward, virtual cut through and wormhole switch-
ing. In the modern NoC design, the wormhole switching is the most commonly used
one due to its low buffer requirement in the router architecture. However, it can
introduce cyclic data graphs into the network, hence potentially create deadlocks.

There are different ways to cope with deadlocks during routing. One way is
adding virtual channels (VCs) into the routers and multiplexing the VCs to access
the physical channels of the network. By ordering the assignment to the VCs, cyclic
graphs can be prevented, which is mathematically proven in [51]. In addition, VCs generally help improve the performance of networks, shown in [50]. Other widely used schemes for avoiding deadlocks in the network are the XY routing scheme and the turn model [64]. In the XY routing, packets are first routed along the x direction of a 2-D network, then along the y dimension till the destination node is reached. In the turn model such as west-first routing, north-last routing and negative-first routing, some special turns of the packets are forbidden during the routing in order to avoid cycles. For example, in the west-first routing, the turns from south to west and from north to west are not allowed.

2.4.2.3 Flow Control

In NoCs, the flow control decides how data are transferred over the network, which needs to take several aspects into consideration. These aspects include e.g., deadlocks, network congestion, virtual channel and buffer utilization. The units, on which the network control operates, are the so-called flow control units (flits). Flits are the basic elements to build up a packet, containing the necessary information for the flow control, such as routing-related information, priorities and semantics information. Commonly, a packet uses three types of flits, which starts with a head flit, followed by multiple data flits, and ends with a tail flit. A flit can be further divided into one or multiple physical units (phits), which are the basic data units transferred through the physical links between the routers. So, the sizes of phits always correspond to the data link width of the routers, while the flit size can be larger than the data link width.

The flow control can operate at different network layers. At the data-link layer, synchronization mechanisms like using hand-shake signals can be applied to ensure the correctness of the flit transmission. At the network layer, routing schemes and algorithms are important for establishing efficient and reliable routing paths through the network, e.g., avoiding deadlocks as discussed above, and/or providing QoS, etc. At the transport layer, end-to-end flow control and congestion control services can be implemented at the NIs to prevent message-dependent deadlocks [79, 139] and to reduce network contention.

Compared to shared buses, NoCs have a unique advantage in the scalability, which makes NoCs be a more promising solution to address the communication issues in large-scale systems. At the same time, challenges also exist for NoCs. Some of the main challenges are presented in the following. More comprehensive surveys of open research problems in designing NoCs can be found in [122] and [151].

- **Complexity**: NoCs are generally more complex than shared buses. In large systems, a large number of routers and NIs are required to connect the system components. A router is usually implemented as a crossbar, enhanced with VCs for improving performance and helping solve deadlock issues. Sophisticated arbiters are often applied for the arbitration between the VCs and between the router ports. For achieving a better clock frequency, pipelining a router is a
standard way. All of these make a router architecture quite complex. Upon that, the architecture of NIs can also be very complex if specific flow control mechanisms are needed. In total, NoCs have much larger area than buses for the same system dimension, which at the same time also implies higher power consumption, both in the static power (especially in the small technologies below 65 nm) and in the dynamic power due to the high circuit switching distributed along the routing paths.

- **Latency:** Due to the parallel point-to-point data communications between the routers, NoCs provide higher bandwidth than traditional buses, hence improve the throughput. However, latency becomes a critical issue in NoCs. First, when a packet is routed from a source node to a destination node, it has to travel along the routing path over multiple intermediate routers. Second, the pipeline stages and the arbitration inside the routers also contribute to the latency. Third, packetizing and de-packetizing the data at the source and destination NI introduce again additional latency. Therefore, for the real-time applications, the latency problem should be carefully handled.

- **Design:** In today’s design for bus-based systems, buses are mostly well standardized and can be used as an off-the-shelf component. The most effort that the designer needs to spend on the integration of a bus into the system is the bus configuration. In case that the bus interfaces do not match the interfaces of the system components, adding adapters between them is sufficient. In comparison to the standardized buses, using a NoC to connect a system often requires many customizations or a new design from scratch. Considering the many design aspects and parameters of NoCs, the design effort which covers from high-level design space exploration and specification down to low-level verification and physical implementation is significantly larger than for buses.

In this section, two mainstream communication architecture paradigms – buses and NoCs, are shortly introduced. Both have their advantages and disadvantages. Choosing which one to use in the system largely depends on the system size and the target applications. Generally for small systems, buses would be more suitable due to their simplicity, and NoCs are preferred in large systems due to the scalability.

### 2.5 System-Level Exploration

Today’s embedded systems are highly complex, from both the hardware and the software perspective, which results in a large design space. To meet design requirements and optimization goals, system-level explorations are an inevitable design step to find the most optimized design points. During the system-level exploration, the following questions usually need to be addressed:

- Where is the hardware/software partitioning of the system made?
2.5. System-Level Exploration

- How many PEs and what kinds of PEs are required in the system?
- How is the memory hierarchy organized?
- How does the communication architecture look like?
- What are the task scheduling and mapping strategy and policies?

To answer these questions, experiments and evaluations for different system design points considering both the hardware and software architecture are desired. For large systems, it can be very beneficial if the experiments and evaluations can be performed fast due to the large design space. However, at the early design stage of a system, the hardware prototypes of the system are mostly still not available, which prevents the designer from developing the software, hence delays the exploration process. A widely used way to deal with this problem is using high-level simulation environments, which are also known as virtual platforms.

2.5.1 Virtual Platform

A virtual platform is a software simulation platform, which mimics the hardware behavior of a system in software. As the platform is software-based, it makes the experiments for the system exploration more convenient and enables an efficient hardware/software co-design. In the virtual platform, once the initial simulation models of the hardware components are available, the software development can already take place. At the time the software is being developed, the hardware components can be further modified, optimized and refined in parallel. This parallel development of the hardware and software significantly reduces the development time of the system. Furthermore, the designer is provided with better opportunities to analyze, profile and debug the system, thanks to the software modeling of the system. The internal states of the system can be much more easily traced in the software that in a hardware prototype. Furthermore, a virtual platform typically also provides a user-friendly graphical interface to assist these purposes.

In a virtual platform, both PEs and communication architectures as well as memory hierarchies can be modeled and simulated at different abstraction levels, depending on the required simulation accuracy. For example, a programmable processor can be modeled with a synthesizable Register-Transfer Level (RTL) description, sensitive to every signal event. Or a cycle-accurate Instruction-Set Simulator (ISS) can be applied to the processor, guaranteeing the correct behavior of the processor for each clock cycle. An ISS can also be modeled at the instruction-accurate level, in which the simulator only needs to make sure that correct results are produced after the execution of each instruction. Source-level simulation [183] is an even higher abstraction model, in which the software is executed as the native machine code, but instrumented with timing annotations. In [60] and [106], a hybrid-simulation technology is proposed, which combines ISS with native C simulation. The authors of [33] developed a generic High-level Virtual Platform (HVP) for supporting software development at early design stage, which also allows mixed use of ISS and native C
simulation. Similar to the modeling of PEs, communication architectures and memory hierarchies can also be modeled at different abstraction levels, from RTL over cycle-accurate level to Transaction-Level (TL).

In general, the higher an abstraction level is, the faster the simulation becomes and the lower accuracy the simulation has. In order to increase the simulation speed and at the same time still provide sufficient accuracy, co-simulations are often applied in the system, in which the components of a system are modeled at different simulation abstraction levels. While the high-level simulations are more used for the functional modeling of non-performance-critical components, the simulations at a low abstraction level are typically used for the components that are performance-critical and need in-depth analyses. It is the designer’s responsibility to decide for which system parts, at which design stages and at which granularity to apply the co-simulation during the system development.

Virtual platforms exist both in the industry and from the academic domain. Representative examples are Synopsys Platform Architect [189], Intel CoFluent Technology [92], ARM SoC Designer [12] and Open Virtual Platforms (OVP) [1]. For this work, the Synopsys Platform Architect is used for creating, simulating and evaluating the systems.

2.5.2 Platform Architect

The Synopsys Platform Architect is a SystemC-based virtual platform, aiming at an efficient hardware/software co-design for SoCs. It provides an IP library containing a rich number of widely used IP blocks such as processors (e.g., ARM processors), buses (e.g., AHB and AXI buses) and bridges, memories and many others like Ethernet, USB, HDMI, etc. For supporting easy exploration with co-simulation, different types of transactors for the interface translation between the models at different abstraction levels are also provided. With simple drag and drop operations, the designer can easily set up a basic working system. The designer also has the flexibility to develop his own system components at different abstraction levels and integrate them into the system. A number of system evaluation and debugging tools further support the designer to make fast as well as detailed analyses.

To accelerate the exploration of hardware/software partitioning for MPSoCs and enable an early start of the software development, Platform Architect introduces the so-called MultiCore Optimization (MCO) technology based on Virtual Processing Units (VPUs) [101].

Virtual Processing Unit (VPU)  To exploit the parallel execution of multiple PEs in MPSoCs, an application is commonly divided into a set of tasks, which are connected with each other using communication channels. The tasks are mapped onto the PEs, while the communication channels are mapped onto the communication architecture.

At the early design stage of the system, it is often desired that the analysis and exploration of different mapping strategies are fast but still accurate enough. However, at this stage many hardware components such as PEs are mostly still not available,
which makes the mapping impossible. Even if some PE models exist, e.g., by means of simulators, running tasks on the simulators can take a long time. This slows down the process of early design space exploration.

Virtual Processing Units (VPUs) provide a means to address this problem. As indicated by the name, a VPU is not a real PE, but it can virtually represent different PE types like processors or ASICs using SystemC threads. In other words, VPUs are able to execute tasks like normal PEs and enable the mapping of tasks onto them at a high abstraction level. In order to be mapped onto VPUs, tasks are also modeled using SystemC threads. By annotating timing information into tasks, which can be obtained, e.g., by educated guess or reasonable estimates based on the underlying PE types, different mapping strategies can be tried and evaluated in a relatively accurate way. As both VPUs and the tasks in the VPUs are SystemC-based, the simulation of the system is in principle based on the native machine code, therefore very fast. This enables fast system exploration. Even further, when searching for proper mappings to meet the target constraints, it is sometimes less relevant, how the tasks are really implemented, if the timing estimates for the tasks are good enough. So, even dummy tasks can be employed in the system, running on the VPUs, as long as the data traffic with respect to the input and output data of the tasks is precise, which is important for the communication architecture exploration. In this way, the software development can take place in parallel to the system design space exploration, which accelerates the design process even more.

To schedule multiple tasks running on the same VPU, which is a typical case if a PE is a programmable processor, a generic OS is provided in VPUs as the task manager. Several basic scheduling algorithms like round-robin and priority-based are already pre-implemented in the generic OS to handle the task queues in the VPUs, and the tasks can be present in different states (ready, running, suspended, etc.) as in a conventional OS. If needed, the designer can also implement his own scheduling algorithms. In addition, interrupts are supported by the generic OS for easy integration of VPUs.

In this work, system-level exploration is used for modeling different systems to analyze the efficiency and flexibility of using OSIP as the task manager from the system perspective. The systems are varied in the number of PEs and different communication architectures in order to show what kind of impact these system configurations have on the OSIP performance. In some systems, VPUs with dummy tasks are used to hide the simulations of complex algorithm kernels, while accurate timing annotations and task communication patterns are provided. This is intended to speed up the system simulation by omitting less relevant information for the OSIP performance analysis.
2.6 Summary

This chapter gives a short survey about related work on task management in MPSoCs, which can be categorized into static, semi-static and dynamic task management. As the system behavior nowadays exhibits high dynamism, dynamic task management is especially important. To provide efficient but still flexible management, using an ASIP as the task manager is motivated in this chapter. In addition, this chapter also makes a short introduction to the today’s mainstream communication architectures, which are also important for task management. Finally, the system-level exploration environment used by this work is introduced.
Chapter 3

OSIP-based Systems

As shown in the previous chapter, efficient dynamic task management is important in today’s MPSoCs. It is at the same time also challenging, if flexibility of the implementation for task management is desired. From the user perspective, programming large MPSoCs, especially heterogeneous systems, is another big challenge. A programming model is typically needed, with which the user can conveniently define tasks and configure task scheduling and mapping as well as task synchronizations. A good programming model should on the one hand simplify the programming by hiding the underlying hardware details from the user. On the other hand, it should also be able to provide the user with the flexibility of constructing different task scheduling and mapping algorithms for different applications and design goals. In this chapter, an approach based on an ASIP task manager called OSIP is introduced, which comes along with a well-defined programming model and efficiently addresses the above-mentioned challenges.

The chapter first starts with an overview of OSIP-based systems, introducing their basic concept and the system construction from both the software and the hardware perspective. Then, the core component of these systems – OSIP, is described in detail together with its architectural development flow. Hardware analyses in terms of area, timing, power and energy as well as a preliminary performance analysis are presented afterwards. Finally, a summary is given and discussions on possible architectural extensions are made.

3.1 System Overview

In this section, an overview of OSIP-based systems is given. After a short explanation of the basic concept of the systems, the details of the software and hardware integration are introduced.

3.1.1 Basic Concept

The OSIP-based systems address the efficiency and flexibility problem in the task management by using an ASIP called OSIP as the task manager. OSIP is an abbreviation for OS ASIP. It is a special programmable processor customized for OS operations and can be used as an off-the-shelf IP block for integration into MPSoCs.

In order to functionally involve OSIP into operation, a set of light-weight software APIs are provided. Through the APIs, the information specified by the user during programming can be translated into the low-level information, which is interpretable
by OSIP. The low-level information gives instructions to OSIP and issues OSIP to react to the instructions. These APIs build up the programming model of the OSIP-based systems.

Figure 3.1 illustrates the basic concept of OSIP-based systems. On the side of PEs, the user specifies system and task information based on the programming model. With the information, tasks are internally formed and stored in the shared memory. In addition, the information needed for the task management such as task descriptors and PE descriptors is transferred to OSIP in a defined format. The tasks in the shared memory and the task descriptors in OSIP have a one-to-one mapping. It means that with the information contained in a task descriptor, the corresponding task can be uniquely identified, which is important for the PEs to find and execute the tasks scheduled by OSIP. The transformation from the user-defined information to the low-level information, and the detailed hardware communication between the PEs and the memory as well as OSIP are completely transparent to the user, which largely eases programming.

On the OSIP side, special instructions are developed in the processor architecture to efficiently process the information and requests originated from the PEs. Furthermore, the user has the full freedom to customize the scheduling and mapping algorithms according to different system scenarios. Together with the information provided by the PEs, OSIP makes scheduling and mapping decisions and responds to the PE requests. The decision result can include complex task-related information needed by the PEs or be just a simple status, indicating whether the requests from the PEs to OSIP are successfully completed.

![Figure 3.1: Basic concept of OSIP-based systems](image)

3.1.2 Programming Model and Software Integration

As introduced in Section 3.1.1, the programming model of OSIP based-systems consists of a set of light-weight software APIs, supporting high-level services and abstracting away the low-level communication primitives. Together with local operating systems, it builds up an abstract software layer between the user application and the underlying hardware, as shown in Figure 3.2. The local operating systems are orthog-
3.1. System Overview

![Figure 3.2: OSIP software layer](image)

**Listing 3.1:** OSIP API example of creating PE classes

```c
1 #define NUM_PE_CLASSES 2
2 PEClassParam peClassParams[NUM_PE_CLASSES];
3 SetPEClassParam(&peClassParams[0], numOfPEs0, &peIDList0, mappingAlgo0);
4 SetPEClassParam(&peClassParams[1], numOfPEs1, &peIDList1, mappingAlgo1);
5 CreatePEClasses(&peClassParams, NUM_PE_CLASSES);
```

...onal to the OSIP APIs. While the former one is responsible for the task management on local PEs, the latter one is for the system-wide task management crossing multiple PEs.

The APIs can further be divided into two main categories: *configuration-related* and *task-related*. With the configuration-related APIs, the user provides the system information to OSIP such as the number of PEs and defines PE classes. The API-support for defining PE classes is a conceptual feature of OSIP-based systems for performing dynamic mapping. By defining classes, a task is not directly mapped to a dedicated PE instance, but to a PE class. In this way, a task can be mapped onto any PE in the corresponding class, which improves the utilization of PEs and greatly reduces the programming effort when extending the system. The PE classes can, e.g., be determined according to the PE types such as RISC, DSP, hardware accelerator, etc. In Figure 3.2, the superscripts of PEs represent the different processing classes and the subscripts represent the individual PEs inside each class. Other configurations such as task scheduling hierarchies and policies are also sent to OSIP using this category of APIs.

Listing 3.1 gives an example of creating PE classes. In the example, two classes are created using the API `CreatePEClasses()` (line 5). The PE instances in each class are specified by PE IDs, which are given through an ID list (`peIDList0` and `peIDList1` in line 3 and 4, respectively). At the same time, the mapping policies (`mappingAlgo0` and `mappingAlgo1`) in both PEs classes are defined.
The task-related APIs are responsible for the operations performed on tasks, e.g., task creation, suspension and deletion, etc. An example of creating a task is given in Listing 3.2, which uses the API `CreateTask()` (line 3). The most relevant information needed for creating a task is specified in line 2. It includes the information needed by the task execution (the function pointer of the task (`funcPtr`) and the task parameters (`taskParams`)). It also includes the information needed for the task scheduling such as the task priority (`prio`), the task list (`listID`), into which the task is to be added, and the synchronization information (`dependentOn`). The synchronization information specifies whether and how a synchronization needs to be resolved, before the task becomes ready for execution. All information about the task is stored in the memory, while the scheduling information is additionally sent to OSIP.

The software APIs contribute to one part of the software integration of OSIP-based systems from the programming perspective. The contribution to the other part of the software integration is made by a firmware running on OSIP from the task scheduling and mapping perspective, illustrated on the right side of Figure 3.2.

The firmware provides a set of low-level functions, which effectively exploit the special hardware features of OSIP like dedicated memory accesses and task-level comparisons. It also contains several basic scheduling algorithms such as round-robin, FIFO, priority-based and fair queue. On top of the provided firmware, the user has the full flexibility of implementing his own code to extend or add new scheduling features, which is supported by a C-compiler. Two exemplary firmware function signatures are given in Listing 3.3. The first example inserts a task into a task list based on a given insertion policy. The second example is to solve possible task dependencies in a task list, based on a given synchronization type. In the examples, `OSIP_DT` is a special data structure storing task and list information in OSIP, which will be described in detail later in Section 3.2.2.1.

### Listing 3.2: OSIP API example of task creation

```c
1 TaskInfoContainer infoContainer;
2 InitTaskInfo(&infoContainer, funcPtr, &taskParams, prio,
   listID, dependentOn, ...);
3 CreateTask(&infoContainer);
```

### Listing 3.3: OSIP firmware examples

```c
1 typedef OSIP_DT DT;
2 InsertTask(DT *pList, DT *pTask, POLICY insertPolicy);
3 SyncTask(DT *pTask, DT *pList, TYPE syncType);
```
3.1.3 Hardware Integration

A typical hardware integration of OSIP-based systems is depicted in Figure 3.3. The system consists of several PEs, a shared memory, peripherals and OSIP, which are connected with each other through some interconnect, e.g., a bus. OSIP further comprises three components, a register interface (REG_IF), an interrupt interface (IRQ_IF) and the OSIP core. While the OSIP core is the actual ASIP that performs task scheduling and mapping, the two generic interfaces of OSIP enable easy integration of OSIP into the system as a hardware IP. The main features of the two interfaces are described in the following.

- **Register Interface**: The register interface makes OSIP able to be addressed as a standard memory-mapped I/O. In this interface, the essential information required for the system-wide task scheduling and mapping is exchanged and maintained.

As illustrated in the figure, the registers can be divided into two groups. The first group contains several OSIP-core-related registers, which interact with the OSIP core. The most important registers in this group are a command (Cmd) register and several argument (Arg) registers. Through the command register, the PEs send instructions to the OSIP core, informing what the core should perform with the information contained in the argument registers. The commands can be used for configuring the system, handling tasks or accessing/setting specific system information.

Some exemplary commands are given in Listing 3.4. In the first example, the command is called SYNC_TASK, which is used to synchronize pending tasks that have dependency of other tasks. In this command, the information in Arg0
Listing 3.4: Exemplary OSIP commands

<table>
<thead>
<tr>
<th>Command example 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cmd : SYNC_TASK</td>
</tr>
<tr>
<td>Arg0: pendingTaskList</td>
</tr>
<tr>
<td>Arg1: syncTaskRef</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Command example 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cmd : CREATE_SCHEDULE_NODE</td>
</tr>
<tr>
<td>Arg0: parentNode</td>
</tr>
<tr>
<td>Arg1: schedulePolicy</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Command example 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cmd : GET_NUM_OF_PENDING_TASK_LISTS</td>
</tr>
</tbody>
</table>

is interpreted as the list of the pending tasks that are considered for possible synchronizations. Arg1 is the task, to which the tasks in Arg0 are synchronized. More arguments are needed for this command, but they are omitted in this example. In the second example, the command CREATE_SCHEDULE_NODE is a system configuration command. It creates a scheduling node with a specific scheduling policy (Arg1) in a scheduling hierarchy under a given parent scheduling node (Arg0). The third command does not have any arguments. It simply gets the number of total pending task lists in the system.

OSIP also returns information to the PEs through the argument registers after it finishes command execution. To ensure that the PEs fetch the return information at the proper time, a status register is introduced in the interface, indicating the OSIP status whether OSIP is still handling the command (busy state) or the command is already done (idle state).

In order to prevent accidental mixing up of the command and arguments sent by different PEs, e.g., the Cmd register containing the SYNC_TASK command sent by PE_A while the Arg registers containing the arguments sent by PE_B for the command CREATE_SCHEDULE_NODE, a spinlock register (OSIP-lock) is used in the interface to guarantee that only one PE can access these registers, consequently the OSIP core at a time. Only after a successful acquisition of the OSIP-lock, a PE is allowed to send a command and the corresponding arguments to OSIP. The other PEs have to poll OSIP-lock until it is freed by the owning PE, and the owning PE frees OSIP-lock only after the command is finished by OSIP. A pseudo code of implementing this on the PEs is given in Listing 3.5.

The second group of registers are a set of spinlock registers, which are used to protect shared resources, such as shared variables in the memory or peripherals. These registers can only be accessed by the PEs and have no interaction with the OSIP-core. Therefore, they are OSIP-core-unrelated. So, they can in principle also
be separated from the OSIP register interface as an individual hardware block. In Chapter 5, more details and investigation on these spinlock registers will be given.

- **Interrupt Interface:** Through this interface, interrupts are generated from OSIP to PEs using interrupt lines. Each PE is connected to one dedicated interrupt line, and the current OSIP architecture supports up to 32 PEs. The interrupts are typically for triggering task execution on the PEs or preempting task execution. They are only generated when OSIP executes a command. Whether to interrupt and which PE to interrupt depends on the system status, e.g., whether there are tasks ready for execution or whether there are PEs currently available for executing the tasks, etc.

In general, the register interface is a slave interface and the interrupt interface is a master interface. The behavior in both interfaces is simple and very generic, which enables easy integration of OSIP. As the operations in the interrupt interface are only activated when a command is being executed, i.e., after the OSIP core reacts to the command from the slave register interface, the whole OSIP behaves more like a slave component in the system from this perspective.

In both interfaces, little flexibility is required due to simple and regular operations such as standard register accesses and address decoding. In fact, the simplicity of the interface operations is also important for ensuring fast information exchange between the PEs and OSIP, e.g., for accessing the spinlock registers. Therefore, an ASIC implementation is naturally used for the interfaces. For different systems, the two interfaces might need to be slightly adapted, e.g., to the number of PEs or the number of spinlocks required by the target applications. However, the adaption is simple and straightforward, requiring only very low design overhead.
3.2 OSIP Architecture Development

In the previous section, an overview of OSIP-based systems is given, especially from the system integration point of view. In this section, the main component of the OSIP-based systems — the OSIP core — and its development are introduced.

The section starts with an introduction to the general design flow of the OSIP core. Then, the application running on the OSIP core, namely the task scheduling, mapping and synchronization, is analyzed and characterized in order to provide hints for the development of the OSIP core architecture. Following the analysis, the OSIP core architecture is proposed and the main architectural features to support task management are presented. As introduced in the previous section, OSIP behaves mainly as a slave component in the system, which becomes active, only when there is a new command received at the register interface. The details about this synchronization between the register interface and the OSIP core are also described. Finally, an exemplary program is given, showing how to start and control the OSIP core to react to the commands or how to enter the idle state.

In the remaining part of this thesis, OSIP is implicitly referred to as the OSIP core for simplifying the text description, if not explicitly stated otherwise.

3.2.1 Design Flow

OSIP is developed with the Synopsys Processor Designer [143]. In the Processor Designer, a processor architecture is described in an Architecture Description Language (ADL) called LISA [80, 161]. From the LISA description, both the software tools such as assembler, linker, compiler and simulator and the hardware descriptions in VHDL or Verilog HDL at RTL can be automatically generated by the tool suite provided by the Processor Designer. The automatic process of this design environment significantly reduces the processor development cycles.

The OSIP design flow is depicted in Figure 3.4, in which three main steps are followed. Along with the three steps, the processor architecture is explored and refined.

In the first step, a complete system containing multiple ARM926EJ-S processors [13], a shared memory, several peripherals, an AHB bus and a SystemC model of OSIP is created. In the SystemC model, the functional behavior of OSIP is implemented in the plain C language and further enhanced by a SystemC wrapper for the integration into the system. This functional behavior covers a wide range of scheduling and mapping algorithms such as FIFO, round-robin, priority-based and fair queuing, etc. It interprets the low-level information originated from the PEs and processes the information correspondingly with the scheduling and mapping algorithms.

The goal of creating this SystemC model is to provide a functionally correct C-implementation of the OSIP behavior, which serves as the base for the further architecture development. Therefore, no timing information is included in the model. When simulating the system for different applications, the inputs to the OSIP model (commands and arguments from the register interface) and the outputs generated from the OSIP model (return values to the register interface and interrupt signals to
the PEs) are recorded. The recorded inputs and outputs are used in the later ASIP development as the driving stimuli to the processor and the reference results, respectively.

In the second step, the C-implementation obtained from the first step is ported to a template processor provided in the Processor Designer. The template processor is called LTRISC. It has a generic RISC architecture and is used in this step for detailed profiling of the C-implementation at a cycle-accurate basis by processing the inputs recorded in the first step. The profiling information contains e.g., the execution hotspots in the OSIP behavior and memory access frequencies, etc.

In the last step, the final OSIP, namely the ASIP, is developed. The development employs on the one hand LTRISC as the base architecture. On the other hand, it utilizes the profiling information delivered in the second step to guide the architectural optimizations, such as extending the instruction-set by adding more specific instructions for scheduling algorithms, optimizing the memory accesses and enhancing the pipeline control, etc.

Along with the development steps, three different OSIP implementations are generated, which are untimed SystemC-based, RISC-based and ASIP-based, respectively. These three implementations will be compared throughout this work. For integrating LTRISC and OSIP into the system, Processor-Support Packages (PSPs) are generated out of their LISA descriptions using the Processor Designer. Both LTRISC and OSIP are cycle-accurate. The following annotations are made for the three implementations:

- **UT-OSIP**: It stands for the untimed SystemC implementation. As it is untimed, UT-OSIP is able to perform task management within zero latency. Therefore,
Chapter 3. OSIP-based Systems

this implementation can be regarded as a hypothetical extremely fast ASIC architecture. However, as an ASIC, the behavior of this implementation cannot be extended or modified.

- **LT-OSIP:** It stands for the implementation based on LTRISC, which is however further extended with a pre-fetch pipeline stage for synchronous program memory access and additional instructions for accessing the interfaces. In comparison to UT-OSIP, this implementation is very flexible, meaning that the task management running on it can be changed and adapted to different applications and system scenarios. However, its flexibility is at the cost of lower performance.

- **OSIP:** It is the final ASIP implementation supporting the task management. With specific hardware features in the architecture, while still containing an instruction-set for programming, OSIP is meant to combine the efficiency of UT-OSIP and the flexibility of LT-OSIP. The analysis for this will be performed throughout this thesis.

### 3.2.2 Application Analysis

In order to obtain the hints for the ASIP development targeting task management, its C-implementation is analyzed. The analysis is made at two levels: a high-level analysis of the data structure and a low-level profiling of the C-implementation. In the high-level analysis, the focus is laid on the way how the information, especially the tasks are organized and maintained in OSIP and in which data structure the tasks are presented. In the low-level analysis, cycle-accurate simulations are made on LT-OSIP to obtain detailed application profiling results such as the frequency of memory accesses, the control flow overhead, etc.

#### 3.2.2.1 Data Structure

In the C-implementation, the complete system information is organized hierarchically as a tree-like structure. Each information node of the tree can be a task node, a PE node, a scheduling or mapping node. Note that a task node does not contain the complete information of a task, such as the function pointer and the task parameters, but a reference to the location in the shared memory, from which the task information can be found. As explained for the example given in Listing 3.2 in Section 3.1.2, creating a task, more precisely, creating a task node in OSIP actually means the registration of a task in OSIP with necessary scheduling information. Similarly, fetching a task from OSIP does not fetch the complete task information directly, but the above-mentioned reference in the shared memory.

The tree-structure enables a hierarchical scheduling and mapping, which is similar to the approach introduced in [71]. It has the benefits of easy construction and extension of the scheduling and mapping structures for different applications and system configurations.
In Figure 3.5, an exemplary scheduling hierarchy is given. This hierarchy has three levels, one level for the task nodes (the leaf nodes of the tree) and two levels for the scheduling nodes (the root node and intermediate nodes). In each scheduling node, a scheduling policy is defined to determine the best task candidate from the underlying task nodes and sub-trees. For example, the best candidate among node 2, node 3, and node 4 is determined based on the FIFO policy defined in node 8. And the best candidate for all task nodes of the whole hierarchy is determined among node 1, node 5, the winner in sub-tree of node 8 and the winner in sub-tree of node 9, following the fair queuing policy.

The mapping hierarchy is very similar to the scheduling hierarchy, but with the PE nodes being the leaf nodes of the tree and the mapping nodes being the root or the intermediate nodes. A mapping node defines the rule which one of the underlying PEs should be selected for executing a chosen task.

In summary, in the scheduling hierarchy a winner task, i.e., the best task candidate that shall be executed as the next, is determined, and in the mapping hierarchy a winner PE, i.e., the best PE candidate, is determined to execute the winner task. The switching from the task scheduling to the task mapping is achieved by merging the root scheduling and mapping node of both hierarchies. Note that multiple independent scheduling and mapping hierarchies can exist in parallel in a system.

These hierarchies are implemented using doubly linked lists, which have the advantage of easy maintenance of the data structure, when adding or removing nodes. The lists are created not only along the hierarchy, but also for the nodes at the same the hierarchy level, e.g., for connecting task nodes or connecting a task node to a scheduling node, which is however not shown in the figure for clarity.

In comparison to the standard list implementation, in which the nodes are linked using C pointers, they are linked with the node indices. Each node is assigned with a unique index. From the hardware perspective, these index-based lists have two main advantages against the pointer-based lists. First, less memory space is needed for the index-based nodes than for the pointer-based. The size of a pointer typically corresponds to the processor architecture. For example, the pointers of a 32-bit processor
usually have a size of 32 bits. In contrast, the bits needed for an index depend on the maximum number of the supported nodes. To support 65536 nodes in the system, which are already large enough for most of the today’s applications, only 16 bits are needed for the index. Second, the indexing of the nodes enables efficient hardware support for the operations performed in the lists, which will be shown later.

The linked lists through indices are made possible by: a) fixing the same size for all node types (task node, PE node, scheduling node and mapping node); b) allocating consecutively a static array for the nodes. In the actual C-implementation, each node has a size of eight 32-bit words, serving as the basic data type that is considered during the scheduling and mapping by OSIP. This basic data type is further referred to as OSIP_DT. But internally, for different node types, the construction of OSIP_DT is different, largely depending on which information should be included in the nodes.

### 3.2.2.2 Profiling

In addition to the high-level analysis for the data structure of the C-implementation, cycle-accurate simulations are run on LT-OSIP to profile the C-implementation from the instruction-level. A set of applications, ranging from synthetic programs to real-life applications like H.264 video decoding are simulated. The profiling results are given in Figure 3.6.

![Figure 3.6: Instruction-level profiling of C-implementation](image)

The following observations can be made from the profiling results. First, arithmetic operations, memory accesses and control flows (branch/jump operations) can be roughly considered as evenly distributed, shown by the pie-diagram in the figure. Even the largest group of the operations — the arithmetic operations — does not dominate the execution time.

Second, there are only few cases, in which a large number of arithmetic instructions are executed consecutively (see the left bar diagram of the figure). 46% of the
consecutive executions contain only one or two arithmetic instructions. This results from the fact that the arithmetic instructions are frequently separated by the control and/or memory access instructions. This is determined by the characteristic of this kind of OS-like applications. During task scheduling and mapping, no highly complex data processing exists, unlike the signal processing from the multimedia or wireless communication domain. Instead, examination of the system information like checking the status of a task or a PE, and simple arithmetic instructions like comparing task priorities are the most common operations needed for making scheduling and mapping decisions. This application characteristic on the one hand requires frequent memory accesses, on the other hand involves high control overhead for making decisions based on the information obtained from the memory accesses. Therefore, the ASIP development for OSIP largely differs from the typical data-centric ASIP development. Instead, it is control-centric. This makes the development challenging, for which efficient handling of the control and memory accesses is extremely important.

Third, a further profiling of the control and memory accesses in the two bar diagrams on the right side of the figure shows that large execution overhead of these operations is caused by the pipeline hazards, including both the data and the control hazards. A large number of pipeline stall and flush operations, and additional nop instructions are presented. This again highlights the necessity of efficient handling of the control and memory accesses, or even reducing them generally by using native hardware support.

3.2.3 Processor Architecture

This section introduces the main architectural features of OSIP. First, an overview of the pipeline architecture of OSIP is given. Then, special instructions are introduced, considering the profiling results presented in the previous section. In addition, instructions for accessing the interfaces are introduced. Finally, the synchronization mechanism between the OSIP core and the register interface is described.

3.2.3.1 Pipeline Structure

OSIP is a 32-bit load-store architecture, which is extended from LT-OSIP. It has 6 pipeline stages: pre-fetch (PFE), fetch (FE), decode (DC), execution (EX), memory (MEM) and writeback (WB). A program memory and a data memory are connected to the pipeline. Both are single-port Synchronous Static Random Access Memories (SSRAMs). A register file with 16 32-bit registers is implemented.

An overview of the pipeline structure is given in Figure 3.7. In the PFE-stage, it is first decided whether the complete pipeline is suspended or activated by checking the OSIP internal state and the request from the register interface. In case that the pipeline is activated, the Program Counter (PC) is generated, and the instruction is fetched in the FE-stage. In the DC-stage, the instructions are decoded and the execution

---

1 The original LT-OSIP processor does not have a pre-fetch stage. It has an asynchronous program memory.
of unconditional branch instructions (jump, call) also takes place here. Also, the
command and arguments are read from the register interface at this stage. In the EX-
stage, arithmetic and logic operations as well as conditional branches are executed,
and the register interface and the interrupt interface are written. Furthermore, the
memory access address is prepared, for which two different ways are distinguished.
The one is for accessing the data of the OSIP_DT nodes of the scheduling and mapping
hierarchies. The other one is for accessing the normal data. Memory accesses take
place either in the MEM-stage or in the WB-stage. In addition, OSIP_DT nodes are
compared in the WB-stage and the register file is written there.

3.2.3.2 Customized Instructions

To efficiently support the control and memory operations performed for the list-based
scheduling and mapping hierarchy, special instructions are implemented.

Enhancing control: The OSIP application is highly control-centric. In many cases,
the control is needed to check the information of the nodes for making further de-
cisions. It checks e.g., the type of a node, the state of a task node, or whether the
task list of a scheduler node is empty, etc. This kind of node information is typically
not long, can be enumerated with maximum 4 bits. If following a typical branch im-
plementation, three instructions would normally be needed. First, the status or the
type, with which the information of a node is going to be compared, is loaded into
a general purpose register. Then, this register is compared with another one, which
contains the fetched node status or type, and the result can be stored in an internal
status register of the processor or again in a general purpose register. In the third
instruction, a branch is executed based on the comparison result.

In the OSIP architecture, these three instructions are merged into one single in-
struction.

- Compare & Branch

\[
\text{enh\_b cond } R[idx_1], \text{imm}4, \text{ BRANCH\_ADDR}
\]

\[
\text{enh\_b cond } R[idx_1], R[idx_2], \text{BRANCH\_ADDR}
\]

These two instructions compare a register value \(R[idx_1]\) against a 4-bit immedi-
ate \(\text{imm}4\), more generally, against another register value \(R[idx_2]\) and make the
branch decision right after the comparison. \text{cond} specifies typical comparison
conditions such as equal, not equal, larger than, etc.

In the pipeline, the comparison and branch decision are both made in the EX-
stage. If the branch is taken, the instructions in the PFE- and FE-stage are
flushed, while the DC-stage is still further executed. Therefore, this instruction
has one delay slot.
Figure 3.7: Pipeline structure of the OSIP Core
Accessing OSIP_DT: As the basic data type of the nodes in the scheduling and mapping hierarchy, OSIP_DT is very frequently accessed to get the information of the nodes from the memory. Thus, accelerating the accesses to OSIP_DT plays an essential role in improving the scheduling and mapping performance. As introduced in Section 3.2.2.1, all nodes are stored in a statically allocated array, each having a unique index and a size of 8 32-bit words. A generic way of calculating a word address inside a node is given by the following equation:

\[
\text{word\_address} = \text{array\_base} + (\text{node\_index} \ll 3) + \text{word\_offset}
\]

In the equation, the base address of the static array first needs to be determined before the calculation of the word address. This normally requires one or two instructions, depending on whether the globally allocated memory address for the array can fit into the coding space of the immediate specified in a MOV instruction. Then, the shift operation and additions are further executed. In LT-OSIP, it takes five instructions to calculate a word address in an OSIP_DT, before the word is accessed from the memory.

To speed up this address calculation, an internal register can be introduced to store the base address of the OSIP_DT array, because only one such array exists in the program. The internal register just needs to be initialized once at the beginning of the program, then can be directly added with the address offset to obtain the word address when accessing an OSIP_DT. However, this is at the cost of hardware overhead in terms of area (a special configuration instruction and a 32-bit adder are needed) and would also possibly influence the timing at the logic interface to the memory.

Therefore, a further step is made in the final OSIP implementation, which simply removes the base address from the calculation by assuming the base address to be at 0x0 by default (see Figure 3.7). To achieve this, it is sufficient to link the data segment to an address right after the OSIP_DT array, when compiling the program. The start address of the data segment defines the size of the OSIP_DT array. The resulting structure of the OSIP Address Generation Unit (OSIP-AGU) for OSIP_DT is presented in Figure 3.8, which is simple and has a very low hardware overhead.

Using the OSIP-AGU for accessing OSIP_DT, the OSIP_DT array is not visible to the C-compiler anymore. Therefore, the nodes of the scheduling and mapping hierarchy cannot be accessed by normal load-store instructions, but only by the special instructions that use the OSIP-AGU. The special instructions are listed below, which accelerate the data accesses to the OSIP_DT node information.

- **Load/Store OSIP_DT**

  \[
  \begin{align*}
  &\text{sp\_load} \ R[\text{value}], R[idx], W, W_{\text{MASK}} \\
  &\text{sp\_store} \ R[\text{value}], R[idx], W, W_{\text{MASK}}
  \end{align*}
  \]
These instructions load/store information \( R[value] \) from/into a given node numbered with index \( R[idx] \). \( W \) specifies the word offset inside the node; \( W_MASK \) masks the word value information at the granularity of a halfbyte.

In the pipeline implementation, the memory access type (whether or not accessing an OSIP_DT) decides which address (from OSIP-AGU or from the address generation for loading/storing normal data) should be taken, which is shown in the EX-stage of Figure 3.9. The normal data address generation is done by adding a base address and an address offset. The access type is further used in the MEM-stage to translate the word mask to the memory address mask in case of writing an OSIP_DT or as a multiplexing signal in case of reading an OSIP_DT. In other cases, the word mask is ignored. In the WB-stage, the result is written into the register file in case of loading data.
• **Update OSIP_DT & Continue**

```
update R[idx2], R[idx1], W, W_MASK, R[value]
update R[idx2], R[idx1], W, W_MASK, simm4
```

These instructions update the information of the $W$-th word of the node numbered with index $R[idx1]$ at the position masked by $W\_MASK$. The masking is at the granularity of a halfword and the to be updated information is increased by a given value, which can be specified by using a register ($R[value]$) or a signed 4-bit immediate ($simm4$). At the same time, the information of non-masked halfword is loaded into $R[idx2]$.

The update instructions combine several instructions into one in order to reduce execution cycles. In LT-OSIP, apart from the instructions used for the address generation, at least two memory load instructions (for loading the lower and higher halfword separately), one arithmetic instruction (for the increment) and one store instruction (for updating the memory) are needed to implement the same functionality of the update instructions.

With these instructions, an efficiency way is provided to update the information along a list. An example is given in Listing 3.6, in which a cyclic list is updated. In the example, the lower halfword of word 4 inside a node is decremented. Meanwhile, the next node index in the list, which lies in the higher halfword of word 4, is pre-fetched for the next iteration (line 2 and 5). The loop stops when all nodes in the list have been visited (line 4), i.e., the head node is reached again.

**Listing 3.6:** Update a list

```
1 // R[it]: iterator; R[head]: head of list
2 update R[it], R[head], w=4, hw=0, -1
3 _loop_start:
4   enh_b eq R[it], R[head], _loop_end
5 update R[it], R[it], w=4, hw=0, -1
6 b _loop_start
7 _loop_end:
```

Updating the information along a list is a major purpose of developing these instructions. This, however, requires a proper data alignment in OSIP_DT, namely, the to be updated information and the next node index in the list should be located in the same word. The programmability of OSIP allows an easy adaption of the information organization inside OSIP_DT to achieve this, if needed, as long as the total size of OSIP_DT is not changed. Certainly, the use of these instructions is not necessarily limited to updating lists.

There are also situations that no other operations than updating the node information need to be performed. For these situations, $R[0]$ can be used at the place
of \( R[\text{idx}_2] \) in the instruction. \( R[0] \) is a zero register which always has the value 0 and cannot be modified.

The pipeline implementation of these instructions is shown in Figure 3.10. In the EX- and MEM-stage, it shares part of the implementation with the \textit{sp\_load} instruction. The difference exists in the memory access unit, which writes two values to the pipeline registers. The one contains the information that needs to be updated, the other one containing the value that should be written back to the register file. The control signal access type (as in Figure 3.9) is omitted from the figure for clarity. The actual information update takes place in the WB-stage, in which a 16-bit addition of the loaded information and the update value is performed. Further, another memory access is issued in this stage to store the sum back into the memory at the same address generated at the EX-stage together with the word mask.

\[
\begin{array}{c}
\text{EX} & \text{MEM} & \text{WB} \\
\text{Operands preparation} & \text{Memory access} & \text{writeback} \\
\text{Word mask} & \text{Load value} & \text{Node index} \\
\text{Update value} & \text{Update (16-bit adder)} & \text{& store} \\
\end{array}
\]

Figure 3.10: Update OSIP_DT

**Scheduling and mapping:** Naturally, as the task manager, OSIP has to perform many scheduling and mapping operations. The efficiency of scheduling and mapping undoubtedly influences the system performance significantly. Thus, it is important to speed up these operations.

In OSIP-based systems, multiple scheduling and mapping algorithms are provided in order to support different applications and systems. Furthermore, an algorithm can be further refined with different rules. For example, in a priority-based algorithm, the rules can be \textit{greater than}, \textit{greater than or equal to} and \textit{equal to}, etc. So, in total there are a number of comparison rules which need to be considered when comparing the nodes in the scheduling and mapping hierarchy. In a typical software implementation, a \textit{switch-case} statement is normally used to first identify the comparison rule. After identifying the rule, the corresponding information of the nodes is
fetched and compared. This kind of implementation has high control overhead due to a large number of branches. To reduce this overhead, a hardware implementation for the node comparison is introduced in OSIP, which supports a range of frequently used comparison rules and compares the node information on-the-fly. Its structure is illustrated in Figure 3.11.

In the hardware implementation, a group of typical priority-based comparison rules are included. In addition, two compound comparison rules in favor of the fair queuing algorithm are supported. There is also a special hardware-supported comparison rule for supporting FIFO/round-robin. As for FIFO/round-robin, no node information actually needs to be compared (indicated by dashed lines in Figure 3.11), the rule itself determines the comparison result. Note that these hardware-supported comparisons do not implement a complete scheduling or mapping algorithm, but provide a fast way to execute the key steps of the algorithm, namely comparing nodes in the scheduling and mapping hierarchy. Upon this hardware implementation, a special instruction is created.

- **Compare Nodes**

  \[
  \text{cmp\_node} \ R[\text{result}], R[\text{rule}], R[\text{idx}_1], R[\text{idx}_2] 
  \]

  The instruction compares the nodes \( R[\text{idx}_1] \) and \( R[\text{idx}_2] \) based on the specified scheduling or mapping rule \( R[\text{rule}] \). The comparison result \( R[\text{result}] \) shows if \( R[\text{idx}_1] \) wins against \( R[\text{idx}_2] \). The node comparison can be a task–task, a task–scheduler/mapper or a task–PE comparison. During the comparison, the information of both nodes is automatically extracted according to the specified rule. In this sense, this instruction performs a sort of task-level comparison.

To efficiently support the list-based data structure in OSIP, the Compare Nodes instruction is further extended.
3.2. OSIP Architecture Development

Listing 3.7: Find the winner of a list

```assembly
1    or      R[curr_best], R[head], 0  // initialization
2    sp_load R[it], R[head], w=6, hw=0  // load next node
3    _start:
4    enh_b eq R[it], R[head], _end  // stop criterion
5    cmp_node_e R[result], R[rule], R[curr_best], R[it]
6    sp_load R[it], R[it], w=6, hw=0  // load next node
7    b _start
8    _end:
```

- **Compare Nodes & Continue**

  \[
  \text{cmp_node_e } R[\text{result}], R[\text{rule}], R[\text{idx}_1], R[\text{idx}_2]
  \]

  The instruction compares the nodes and keeps the winner node index updated. In the syntax shown above, nodes \(R[\text{idx}_1]\) and \(R[\text{idx}_2]\) are compared and the comparison result is written into \(R[\text{result}]\). Meanwhile, \(R[\text{idx}_1]\) is updated with \(R[\text{idx}_2]\), if node \(R[\text{idx}_2]\) wins.

  This instruction is very useful for finding the best node throughout a list. An example is given in Listing 3.7. In the example, the list iterator \(R[it]\) contains the current node index as the candidate node. It is compared with the node \(R[\text{curr_best}]\), which is the current winner among the nodes before \(R[it]\). After the comparison, the winner node index between \(R[\text{curr_best}]\) and \(R[it]\) is stored in \(R[\text{curr_best}]\), so that the information of \(R[\text{curr_best}]\) is always up-to-date. The value of \(R[\text{result}]\) in this example is ignored.

  The pipeline implementation for both node comparison instructions are given in Figure 3.12. Multi-cycles memory accesses are introduced in the MEM-stage for obtaining the information from both nodes for comparison. The requests to the memory and reading data from the memory are interleaved, such that at every cycle one value can be fetched from the memory. A second OSIP-AGU is applied here for generating memory addresses in the MEM-stage. Depending on the comparison rule, up to four cycles can be required for the memory accesses. The node comparison is made in the WB-stage, the result of which is written back to the register file. In case of the \text{cmp_node_e} instruction, the comparison result is further used to select the winner node index from the two input node indices, which is written back to the register file as well.

**Accessing Interfaces:** In addition to the special instructions for improving scheduling and mapping, dedicated instructions are also needed to access both interfaces of OSIP, such that OSIP is able to communicate with the whole system.
• **Set Interrupt**

\[ \text{interrupt } R[\text{pe_id}] \]

This instruction generates an interrupt to the PE specified by the register \( R[\text{pe_id}] \) in order to trigger it for task execution. The interrupt is generated from the EX-stage of the pipeline to the interrupt interface, the latter then raises the interrupt signal to the PE if the interrupt of the PE is not masked. This instruction supports up to 32 PEs. For an MPSoC using a central task manager, this number is already very large.

• **Read/Write OSIP Arguments**

\[ \text{read_argument } R[\text{idx}], \text{IF}_R[\text{arg_idx}] \]
\[ \text{write_argument } \text{IF}_R[\text{arg_idx}], R[\text{idx}] \]

The first instruction reads the value of an argument register (\( \text{IF}_R[\text{arg_idx}] \)) from the register interface into a general purpose register (\( R[\text{idx}] \)), and the second writes an argument register. \( \text{arg_idx} \) has three bits, supporting up to eight 32-bit argument registers, corresponding to the size of OSIP_DT.

• **Read OSIP Command**

\[ \text{read_command } R[\text{idx}] \]

This instruction reads the command register from the register interface into a general purpose register (\( R[\text{idx}] \)).
• **Set Idle Status:**

\[ \text{set\_idle} \]

This instruction sets the status register at the register interface to *idle*, indicating that OSIP has finished the current command and is ready for executing the next one. This instruction has no argument. The setting of the status takes place at the DC-stage, and flushes the following instructions at the PFE- and FE-stage from the pipeline.

Note that there is no explicit instruction to set the status register to *busy*. Instead, the *busy* state is internally generated in the PFE-stage, which depends on the current processor state and the input from the register interface to the OSIP core. Details about this are given later in Section 3.2.3.3.

**Accessing Command Handler Decoder:** OSIP is a component which only reacts to the requests from the PEs. When a new command arrives, OSIP first needs to identify the corresponding command handler through a command handler decoder (see the program memory structure in Figure 3.7), which follows a similar principle as an interrupt vector table. So the entry address of the handler decoder must be available at the arrival of the command. In OSIP, this entry address is stored in an internal register, which enables a hardware-assisted PC determination. A dedicated instruction is implemented for this purpose, which is shown below.

• **Get Decoder Address:**

\[ \text{get\_decoder\_address} \]

The instruction determines the entry point to the command handler decoder. It does not have an explicit argument, but uses its instruction address as an internal argument. By incrementing this address, it automatically calculates the address of the following instruction. The calculated address is then stored into an internal register called *Decoder_PC* (DPC). From the DPC, the command handler decoder can be reached, which however should be ensured by the programmer. In this work, this instruction is placed before a *CALL* instruction, which calls the handler decoder. When OSIP switches from *idle* to *busy* at the arrival of a new command, the PC is updated with DPC. More details about this are given in Section 3.2.3.3.

In addition to getting the entry point to the handler decoder, this instruction also sets the processor state to *idle* for the initialization purpose.

### 3.2.3.3 Interaction with Register Interface

When the OSIP core is at the busy state, i.e., executing a command, the processor behaves in a standard way. At every cycle, a new PC is generated and a request is
sent to the program memory (PMEM) to fetch the instruction. The PC is selected between the current PC register and two other branch PCs, as shown in Figure 3.13.

The difference from the standard execution of the processor occurs when the OSIP core finishes a command (switching from busy to idle state) or when OSIP receives a new command (switching from idle to busy state).

When a command is finished, the command handler calls the instruction set_idle to change OSIP state to idle, indicating that OSIP is now ready for accepting a new command. At the same time, the state that the current command is finished is stored in an internal register (cmd_done). This state disables the read requests to PMEM and stalls the remaining pipeline stages. It means that at the idle state, the OSIP core is practically suspended.

The OSIP core keeps on staying in the idle state, until it receives a new command request signal from the register interface. This signal on the one hand sets the state register at the interface to busy. On the other hand, it clears the internal state cmd_done to activate the pipeline and the accesses to PMEM again. In addition, the information that a new command arrives selects the internally stored entry point to the handler decoder, i.e. DPC, as the current PC. The timing between resetting the cmd_done state and selecting DPC is synchronized by introducing another internal register new_cmd_flag. This register holds the value true only for one clock cycle. It is cleared by itself at the next cycle, so that the standard PC generation is enabled again for the normal program execution of the OSIP core for handling the new command. It is important to note that the signal cmd_request generated by the register interface also holds true for only one cycle. Otherwise, the new_cmd_flag register would toggle its value at every clock cycle.

![Figure 3.13: Control of the OSIP state at the PFE-Stage](image-url)
3.2.4 OSIP Code Example

In this section, an exemplary code of programming the OSIP core is shown, explaining in detail how the processor is controlled by the software to handle the commands and especially to synchronize with the register interface, consequently with the PEs of the system.

The code is given in Listing 3.8. It starts with a typical booting process of a processor such as the initialization of the stack and frame pointer and other initializations (line 3–4). By now the processor is at the busy state, because the internal register cmd_done is set to false when the processor is reset.

Then, the instruction get_decoder_address is executed to get the entry point to the handler decoder, in this case the main() function, and to set the processor status to idle. The register DPC now gets the address of the instruction which calls the main() function (line 6). At this point, the processor has been booted, but does not execute further instructions, until a new command request is sent from the register interface.

At the arrival of a command request, the processor is awaken to take the DPC as the next to be executed PC and continues with the program at line 6. Then the command and arguments are read from the register interface (line 16–17). Afterwards, the command is decoded and the corresponding handler of the command is called (line 18–22). After handling the command, results are written back to the argument registers at the interface (line 26), and the program execution jumps directly to the idle section (line 27).

In the idle section (line 8–10), the instruction set_idle (line 10) is executed, which again sets the processor state to idle for receiving the next command. For the next command, the program execution starts from line 6 again.

Note that at the end of the command execution in the handler, it does not necessarily need to directly jump to the idle section. A conventional way can also be followed, in which the program is executed to the end of the main() function, then enters the idle section. However, this can introduce a lot of runtime overhead caused by the calling stacks. In fact, between two consecutive commands no context information (registers and memory content) needs to be maintained, except the static array allocated for the OSIP_DTs. As introduced in Section 3.2.3.2, this array is not reachable by the C-compiler. Therefore, the information inside the array is untouched and safe if skipping the recovery process of the calling stacks, as done in this exemplary code. But there is one step that must be done before entering the idle state, namely resetting the stack pointer and the frame pointer (line 9)\(^2\). This step is important for preventing stack overflow. Otherwise, the stack and frame pointer would continuously increase from one command to the next.

\(^2\)Theoretically, the same thing should also be done for the heap. However, the use of malloc() is avoided and also not needed in the OSIP application.
3.3 Results

In this section, hardware results based on the gate-level synthesis for the OSIP core are reported and compared with those of LT-OSIP, and a preliminary performance analysis is performed, considering the execution time of the most frequently used and critical OSIP commands. Especially, the area and energy efficiency of OSIP and LT-OSIP are compared, using two widely used cost metrics – Area-Time product (AT) and Area-Time-Energy product (ATE).

3.3.1 Area, Timing and Area-Time Product (AT)

The OSIP core is synthesized with Synopsys Design Compiler [187] using a 65 nm standard cell library under typical operation conditions (supply voltage 1.0 V, temperature 25 °C). The area and maximum achievable clock frequency are given in Table 3.1. As a comparison, the synthesis results of LT-OSIP are also given in the table. The
area of memories (program and data memory) is excluded from the reported area in the table, due to missing memory libraries at this technology during this work.

### Table 3.1: Synthesis results of OSIP and LT-OSIP

<table>
<thead>
<tr>
<th></th>
<th>OSIP</th>
<th>LT-OSIP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total area (kGE)</td>
<td>35.7</td>
<td>23.1</td>
</tr>
<tr>
<td>Combinational (kGE)</td>
<td>25.2</td>
<td>14.8</td>
</tr>
<tr>
<td>Sequential (kGE)</td>
<td>10.5</td>
<td>8.3</td>
</tr>
<tr>
<td>Max. clock frequency (MHz)</td>
<td>690</td>
<td>690</td>
</tr>
</tbody>
</table>

As shown in the table, OSIP and LT-OSIP achieve the same maximum clock frequency. In both processors, the critical path is caused by the read access to the data memory in the MEM-stage. This is because the output data delay from the data memory is considerably long, if a large static array is allocated for the OSIP_DTs.

In contrast to the same maximum clock frequency, OSIP has much larger area than LT-OSIP. An area increment of 54% is reported, caused by the special hardware features in the OSIP core. However, even with an area of 35.7 kGE, the OSIP core is still rather small if considering a multi-processor system.

The special hardware features in OSIP lead to significant performance improvement. For an initial performance analysis, the execution time of the commands in OSIP is compared with that in LT-OSIP. In the current systems, there are dozens of commands supporting the OSIP APIs. However, not all commands are critical for the system performance, e.g., the commands for configuring systems. These commands are executed only once at the beginning of an application, therefore, irrelevant for the entire system performance. The commands for getting and setting the system information, which normally occur occasionally and are simple, are typically also not critical for the system performance.

The most critical commands are task-related and frequently issued, and/or often involve task scheduling and mapping or synchronization. Through the profiling of the applications, five “hot-spot” commands are identified, which are for creating a ready task (CRT), creating a dependent task (CDT), synchronizing tasks (ST), requesting a task (RT) and fetching a task (FT). Their occurrence frequencies ($f_{cmd}$) are given in Figure 3.14, the sum of which is 99.9%. The figure also shows the average execution cycles for these commands when running an H.264 decoder application. As OSIP and LT-OSIP have the same clock frequency, the ratios of the execution cycles are also the speed-up factors for these commands by OSIP. Certainly, the execution time of the commands varies from one application to another, and also varies within an application at different phases. It largely depends on the size of task lists and the scheduling and mapping algorithms. The numbers presented in the figure result from a complex scheduling and mapping algorithm for a moderate-sized system with 7 ARM processors, and the system and task configuration is quite generic. In practice, the configurations and the scheduling and mapping algorithms can be specifically
simplified or optimized for different target applications. However, this figure is meant to provide a first impression how much the special instructions in OSIP can improve the command execution time. For these “hot-spot” commands, up to a speed-up of $9.1 \times$ can be achieved. If considering the frequency how often the commands are executed, the average speed-up is $7.7 \times$, following Equation 3.1. The relative low speed-up for the commands CDT and FT is due to the fact that they do not cause scheduling or mapping during the execution.

$$\tau_{\text{LT-OSIP}} / \tau_{\text{OSIP}} = \frac{\sum_{\text{cmd}} (\text{Cycles}_{\text{cmd,LT-OSIP}} \cdot f_{\text{cmd}})}{\sum_{\text{cmd}} (\text{Cycles}_{\text{cmd,OSIP}} \cdot f_{\text{cmd}})} = 7.7 \quad (3.1)$$

If taking the area overhead of OSIP into consideration, the Area-Time-Product (AT), i.e. the area efficiency, is improved by a factor of $5 \times$ by OSIP, as calculated in Equation 3.2.

$$\frac{A_{\text{LT-OSIP}}}{A_{\text{OSIP}}} \cdot \frac{\tau_{\text{LT-OSIP}}}{\tau_{\text{OSIP}}} = 5.0 \quad (3.2)$$

The performance analysis in this section is rather preliminary, in which the OSIP efficiency is only analyzed in an isolated way. However, the performance of a sys-
tem does not only depend on the efficiency of the task manager, even though it is certainly an important factor, but also on many others, such as task sizes, the communication architecture, etc. The bottleneck component of the system finally determines the system performance. The bottleneck could be the manager, or the communication architecture as well as the PEs. A systematic system-level analysis needs to be made for a more comprehensive evaluation of the OSIP performance, which will be shown in the next chapter.

### 3.3.2 Power, Energy and Area-Time-Energy Product (ATE)

The power consumption of the OSIP core is estimated using post-synthesis gate-level power simulation with Synopsys PrimeTime [188], running a H.264 video decoding at a clock frequency of 690 MHz with a supply voltage of 1.0 V. The power of the memories is not considered.

As described in Section 3.2.3.3, the OSIP core has two states: busy and idle. In Table 3.2, the average power consumption of OSIP at both states is listed. At the idle state, the OSIP core consumes about 7.2% of the power at the busy state. While the static power stays almost unchanged, since it is only influenced by the area, the dynamic power is reduced by a factor of $17.3 \times$ from the busy state to the idle state.

<table>
<thead>
<tr>
<th>State</th>
<th>OSIP (mW)</th>
<th>LT-OSIP (mW)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Busy</td>
<td>17.20</td>
<td>8.54</td>
</tr>
<tr>
<td>Dynamic Power</td>
<td>16.92</td>
<td>8.38</td>
</tr>
<tr>
<td>Static Power</td>
<td>0.28</td>
<td>0.16</td>
</tr>
<tr>
<td>Idle</td>
<td>1.25</td>
<td>0.95</td>
</tr>
<tr>
<td>Dynamic Power</td>
<td>0.98</td>
<td>0.80</td>
</tr>
<tr>
<td>Static Power</td>
<td>0.27</td>
<td>0.15</td>
</tr>
</tbody>
</table>

A detailed analysis of the power consumption of the OSIP core is depicted in Figure 3.15. For both OSIP states, the main contributor to the power consumption is the clock tree, including the clock gating elements and the clock pins driving the registers. In the busy state, the contribution by the registers and the combinational logic is also considerably large in comparison to the idle state. For the latter, the contribution by the registers and the combination logic is only due to the static power, as the complete pipeline is deactivated, i.e., no data switching in the pipeline.

Table 3.2 also presents the average power consumption of LT-OSIP. Compared to OSIP, LT-OSIP consumes less power than OSIP, both in the busy and idle state, which

---

3 In a gate-level synthesis, the clock tree buffers are typically not generated. Therefore, Figure 3.15 does not present the power consumed by the clock tree buffers.
is natural. In the busy state, OSIP has to finish the same amount of work as LT-OSIP, but within a shorter time. For the idle state, OSIP has a higher power consumption mainly due to the larger area. The higher dynamic power of OSIP in this state is caused by a larger clock tree which is also resulted from more registers.

It is, however, more important to compare the energy efficiency, in this case, the average energy consumption per task scheduling and mapping. The ratio of the energy efficiency between OSIP and LT-OSIP can be calculated by Equation 3.3, in which the number of tasks (#Tasks) is the same for a given application, independent of which task manager is used.

$$\frac{E_{\text{task},\text{LT-OSIP}}}{E_{\text{task},\text{OSIP}}} = \frac{(P_{\text{busy,LT-OSIP}} \cdot T_{\text{LT-OSIP}})/\#\text{Tasks}}{(P_{\text{busy,OSIP}} \cdot T_{\text{OSIP}})/\#\text{Tasks}} = 3.8 \quad (3.3)$$

Together with Equation 3.1, it is shown that OSIP only consumes 26.3% of the energy of LT-OSIP to handle a task, while improving the task management performance by a factor of $7.7 \times$. Considering further the OSIP area overhead, the Area-Time-Energy Product (ATE) of both task managers per task is compared in Equation 3.4.

$$\frac{ATE_{\text{task,LT-OSIP}}}{ATE_{\text{task,OSIP}}} = \frac{A_{\text{LT-OSIP}} \cdot T_{\text{LT-OSIP}} \cdot T_{\text{LT-OSIP}}}{A_{\text{OSIP}} \cdot T_{\text{OSIP}} \cdot T_{\text{OSIP}}} = 18.9 \quad (3.4)$$

The energy analysis above assumes that OSIP is completely busy during the application execution. In reality, this is not the case. For a more accurate analysis, the energy consumption at both busy and idle state as well as the application execution time need to be considered, which will be shown in Section 4.3 of the next chapter.
3.4 Summary

In this chapter, an overview of OSIP-based systems is given, and the major advantages of such systems are highlighted from the efficiency and flexibility perspective.

As the key component of the system, the architecture of OSIP – an application-specific processor for OS, is described in detail. A preliminary performance analysis already shows the efficiency of OSIP in the task management by comparing it with a generic RISC processor. The control-centric OSIP architecture development is challenging. But it is effectively overcome with the special hardware features for handling list-based operations, fast memory accesses and comparing list nodes as well as compact branch instructions. The hardware results including the area, timing and the power are presented, and the area and energy efficiency of OSIP is highlighted.

3.5 Discussion

The special hardware features in OSIP improve the OSIP efficiency, but at the cost of the reduced flexibility. Among these hardware features, the OSIP-AGU and the node comparator are the most prominent ones, which are discussed in this section.

3.5.1 OSIP Address Generation Unit (OSIP-AGU)

The OSIP-AGU, which introduces little hardware overhead, greatly speeds up the memory accesses to the OSIP_DTs. This is thanks to the regular structure of the OSIP_DTs with eight words and their arrangement in a static array, for which a simple address generation using a shift operation by a constant and basic logic operations instead of arithmetic operations becomes possible. However, this can be a limitation in the programming, if over eight words are needed for a node.

Normally, eight words are already large enough for storing the necessary information in a node, whether for a task node, a scheduling/mapping node or for a PE node. In fact, there are still fields in the nodes, which are not used, but reserved for future extensions. In case that more words are really needed for storing the information, a workaround has to be done. For example, the node information can be distributed in two consecutive OSIP_DTs, so that the OSIP-AGU is still applicable. Of course, necessary information conversions between the node and the OSIP_DT structure for the index and word offset must be made in the OSIP software. If the number of the needed words for the node is not a multiple of eight, some words get wasted.

On the other hand, if the designer wants to reduce the memory usage by reducing the node size, in case that less information is needed for a node, it is not possible with the OSIP-AGU. This also means wasting memory words.

A more generic way of implementing the OSIP-AGU would be using a hardware multiplier, which multiplies the node index by the node size (i.e., the number of words per node) to calculate the base address of a node. Then, the word offset is added with the base address to obtain the final word address. In comparison with the current OSIP-AGU, this implementation would result in a larger area in the pipeline
and possibly worsen the timing. However, for certain applications, it could enable better memory utilization and also has higher flexibility. Hence, a trade-off can be considered.

### 3.5.2 Node Comparator

The node comparator is another hardware feature, which can have flexibility limitations when using the `cmp_node` and `cmp_node_e` instructions. Naturally, it is impossible for a hardware node comparator to cover all possible comparison rules and different combinations, and the current comparator already supports a quite wide range of rules. However, if new rules should be applied, software implementation is needed in the OSIP scheduling algorithms to support them. To still be able to use these two special instructions for node comparison, an additional flag can be introduced in the software to distinguish the currently supported rules and the new ones.

Another approach of implementing a flexible comparator, which at the same time can cover an even larger range of rules in hardware, is using a Coarse-Grained Reconfigurable Architecture (CGRA) [39, 76, 129, 172], in which different rules can be configured statically or dynamically.
Chapter 4

System-Level Analysis of OSIP Efficiency

As a central task manager, the efficiency of OSIP undoubtedly has a big impact on the system performance. The previous chapter has made a preliminary analysis of the OSIP efficiency by analyzing the execution time of the most critical OSIP commands for scheduling and mapping tasks. In comparison to a RISC-based task manager, OSIP is able to execute these commands within much less time. However, from the system perspective, this analysis is only isolated and rather one-sided, as it does not show how the OSIP efficiency influences the overall system performance.

For a complete system, its performance depends on many factors, among which the performance of PEs, the task sizes and the communication architecture are especially important in addition to the task manager. These factors need to be jointly investigated in order to analyze the OSIP efficiency in a system context. In this chapter, a thorough characterization of the performance of OSIP is provided from the system point of view. A special focus is laid on the joint impact of the communication architecture and the OSIP efficiency, as the communication architecture has become one of the dominant factors for the performance of modern MPSoCs.

This chapter is organized as follows. First, the system setup for the analysis and the benchmarking applications — a synthetic application and a real-life H.264 video decoding application — are introduced. Then, the OSIP-efficiency is analyzed in systems without considering the communication overhead by idealizing the communication architecture. Afterwards, the impact of the communication architecture on the OSIP-based system performance is highlighted. Following this, optimized realistic communication architectures are presented. Based on the resulted different communication architectures, the impact of the OSIP efficiency and the communication architecture on the system performance is jointly investigated. Finally, a summary is made for the OSIP efficiency from the system perspective.

4.1 System Setup

For evaluating the OSIP efficiency at the system-level, a virtual platform is built using Synopsys Platform Architect. The platform consists of several instruction-accurate ARM926EJ-S processor models, an OSIP model, a shared memory and some periph-

erals such as input stream, virtual LCD and UART. Without loss of generality, the clock frequency of the system is set to 100 MHz during the analysis.

In the platform, all components are SystemC-based. Three OSIP models – UT-OSIP, LT-OSIP and OSIP, which are introduced in Section 3.2.1, are employed in the system alternatively for comparison. A SystemC wrapper is created for the OSIP models, modeling the behavior of the register interface and interrupt interface.

The communication architecture of the system is AHB-based, to which all system components are connected. As the arbitration scheme in the bus, round-robin is selected. As will be shown later, the communication architecture is stepwise extended and optimized. A simplified overview of the system is given in Figure 4.1. The different transactors for the communication protocol translation between the system components, as well as the clock and reset generators are not shown in the figure for clarity. In this system, all slave components (OSIP, shared memory and peripherals) of the bus can be accessed by all ARM processors through the bus.

4.2 Benchmark Applications

Two applications are selected to benchmark the OSIP performance in the task management. The first one is a generic synthetic application, intended for covering a wide range of analysis cases, helping to investigate the limits of OSIP by considering the number of tasks, task sizes, the number of PEs and the amount of data traffic. The second application is a popular real-life application from the multimedia domain – H.264 video decoding. In the following, both applications will be briefly introduced, and the mapping of the applications onto the system will be described.
4.2. Benchmark Applications

4.2.1 Synthetic Application

The synthetic application consists of three major task types: data producing (Task\textsubscript{p}), task generation (Task\textsubscript{gen}) and data consuming (Task\textsubscript{c}). The dependencies between them are shown in the upper part of Figure 4.2. First, Task\textsubscript{p} produces a set of data into the shared memory and issues the execution of Task\textsubscript{gen}, after the program is started. Then Task\textsubscript{gen} generates a set of Tasks\textsubscript{c}, which consume the data by summing them up and send the result to the I/O.

The lower part of Figure 4.2 shows how the system is configured to execute the application. The ARM processors are divided into two PE classes. The first class contains only one processor (ARM\textsubscript{0}), onto which both Task\textsubscript{p} and Task\textsubscript{gen} are mapped. This processor is named Producer PE (PPE). The second class contains the rest ARM processors, executing Tasks\textsubscript{c}. These processors are named Consumer PEs (CPEs). All CPEs have the same priority to execute a task, if no tasks are running on them. OSIP is responsible for deciding which task should be executed as the next and mapping it to an available CPE. In order to push OSIP to its performance limit, a priority-based scheduling algorithm is chosen for deciding the next to be executed task. A system with \( n \) consumer PEs is further referred to as an \( n \)-CPE-system.

Following parameters are configurable in the synthetic application to create different workloads for OSIP:

- \( \text{N}_{-}\text{CPE} \in \{1, 3, 5, 7, 9, 11\} \): This parameter configures the number of CPEs. Increasing \( \text{N}_{-}\text{CPE} \) potentially creates requests to OSIP for task execution more frequently, hence increases the OSIP workload.
• \( \text{N\_TASK} \in \{11, 88, 165\} \): This parameter configures the number of the generated \( \text{Tasks}_c \). It is chosen as a factor of 11, such that the tasks could be distributed to CPEs in a balanced way in the largest target system within this analysis, namely the 11-CPE-system. As mentioned above, the task scheduling is priority-based. To find the best candidate task for execution, OSIP has to loop the complete task list. This means that the workload of OSIP increases linearly with the list size. When increasing \( \text{N\_TASK} \), the size of the task list potentially increases, which implies higher workload for OSIP.

• \( \text{N\_ACCESS} \in \{1, 6, 11\} \): This parameter configures the frequency of accessing the same data from the shared memory by a \( \text{Task}_c \). The size of the data set produced by \( \text{Task}_p \) is fixed to 500 32-bit words, which are stored in the memory as an array. Without considering the communication overhead, the task size with an \( \text{N\_ACCESS} \) of 1, 6 and 11 corresponds to 2.5 kcycle, 15 kcycle and 27.5 kcycle, respectively. The tasks of the different sizes are further referred to as small tasks, medium tasks and large tasks.

These configuration parameters also have a big impact on the system communication, which will be discussed later in detail.

### 4.2.1.1 OSIP Working Scenarios

Based on the configuration parameters above, three scenarios are defined from the perspective of the workload of OSIP, representing different types of applications:

• **Best case scenario – Low workload:** In this scenario, low workload is generated to OSIP by configuring the size of \( \text{Task}_c \) to the maximum (\( \text{N\_ACCESS} = 11 \)) and the number of \( \text{Tasks}_c \) to the minimum (\( \text{N\_TASK} = 11 \)). In this type of applications, it takes a PE quite a long time to finish a task before the PE requests a new task from OSIP. This makes the frequency of accessing OSIP low. Furthermore, it takes OSIP less time to handle a request (in this case, to find the best candidate task from the list), because the task list is short. So, from the OSIP perspective, this scenario is the best case for it, because it is only little stressed.

• **Worst case scenario – High workload:** This scenario is exactly the opposite case to the scenario above. Here the task size is set to the minimum (\( \text{N\_ACCESS} = 1 \)) and the number of \( \text{Tasks}_c \) is set to the maximum (\( \text{N\_TASK} = 165 \)). In this scenario, the PEs are able to finish the task within a short time, hence request new tasks from OSIP very frequently. In addition, the scheduling effort of OSIP becomes much higher due to a much larger task list. Therefore, for OSIP this appears to be the worst case.

• **Average case scenario – Medium workload:** In this scenario, both the task size and the number of tasks are set to medium (\( \text{N\_ACCESS} = 6, \text{N\_TASK} = 88 \)). This configuration averages the workload of OSIP between the best case and worst case scenarios.
4.2. Benchmark Applications

OSIP Configuration

![Task graph and OSIP configuration of H.264](image)

**Figure 4.3:** Task graph and OSIP configuration of H.264

In addition, for all three scenarios, the configuration set of N_CPE is iterated during the performance analysis.

### 4.2.2 H.264 Video Decoding

The software implementation of the H.264 video decoding follows a 2-D wave concept, which is similar to the one introduced in [127]. In this implementation, the video frames are built up with the so-called Macroblocks (MBs), which are the basic data elements that the decoding algorithm operates on. Using the 2-D wave concept, the decoding of each MB has dependency on its three possible neighboring MBs, which lie on its left, top, and top left side. A simplified task graph and the OSIP configuration for the application are given in Figure 4.3.

In the task graph, the main functional blocks of the H.264 decoding are presented. First, the entropy information of compressed video frame data is decoded, out of which the MB data structures are built. The data of each MB are then re-scaled by an inverse quantization (IQT) and transformed by an inverse discrete cosine transform (IDCT). Afterwards, an intra-frame prediction block is conducted on the data out of IDCT, which utilizes the spatial correlation to the previous decoded neighboring MBs and predicts the current MB. Finally, a deblocking filter is applied to remove the spikes on the edges of the MB.

Among the main functional blocks given in the figure, IQT, IDCT and intra-frame prediction are the task types, that are computationally intensive and at the same time can be highly parallelized. In the OSIP configuration, the PEs are divided into
two classes. The first class contains a single PE (ARM), on which the task type entropy decoding is mapped. After the entropy decoding is finished, parallel operations on the MBs are possible. The second PE class then includes all PEs, and all task types except the entropy decoding are mapped onto this class. In the actual software implementation, IQT and IDCT are merged.

For the system performance analysis purpose, the number of PEs is set to be configurable in the software implementation. The different PE number results in different frame rates, which are given in Frame per Second (fps). This frame rate is used as the criterion for evaluating the OSIP efficiency.

4.3 Efficiency Analysis with an Ideal Communication Architecture

The performance of a system depends on many factors, among which the communication architecture has a large impact. In this section, to isolate the effect of the scheduling and mapping efficiency of OSIP, the communication architecture is first bypassed in the simulation platform. With this, an idealized communication architecture is obtained, i.e., the data communication through the bus introduces no latency in the system. In other words, the communication architecture of the system has an infinitive bandwidth.

4.3.1 Synthetic Application

In the synthetic application, the execution times of the application using the three different OSIP implementations, namely OSIP, LT-OSIP and UT-OSIP, are compared. In the upper part of Figure 4.4, an overview of the application execution time in different scenarios is presented. The following observations can be made.

First, the OSIP-based systems and the UT-OSIP-based systems have a close system performance. In general, with more workload generated onto OSIP, the difference of the execution times between these two systems increases. However, even in the worst case scenario, the difference is rather small. In the best case scenario, the difference is only marginal. In contrast, the system performance based on LT-OSIP is much worse than that of the other two for all scenarios.

Second, the system behaviors of the OSIP-based and UT-OSIP-based systems are very similar. With the increasing number of the PEs, the execution time decreases. In the average case and worst case scenarios, the performance becomes saturated in large systems in terms of a large number of PEs. In the former case, the saturation starts with a 7-CPE-system and in the latter case it already starts with a 3-CPE-system. In these systems, the system performance is limited by the degree of the task parallelism, which is obviously not able to match the computational power provided by the given PEs.

In the LT-OSIP-based systems, the curves of the execution time present a very different performance behavior. In all three scenarios, the execution time first decreases
4.3. Efficiency Analysis with an Ideal Communication Architecture

Figure 4.4: Synthetic application: OSIP efficiency analysis in systems with an idealized communication architecture

with the increasing number of the PEs. Then, beyond a certain amount of PEs, the execution time increases again. Adding more PEs into the system certainly introduces more computational power. But at the same time it also generates requests to the task manager more frequently, hence increases its workload. If the workload exceeds the scheduling and mapping ability of the manager, the manager becomes the bottleneck of the system. Apparently, LT-OSIP is the bottleneck in these large systems. In this case, adding more PEs worsens the system performance. Furthermore, it can be observed that the more workloads LT-OSIP receives, the earlier it becomes the bottleneck and the faster the execution time increases afterwards.

The busy time of OSIP ($t_{OSIP-busy} (%)$), shown in the lower part of Figure 4.4, illustrates the task management efficiency of the three OSIP implementations from another perspective. Formally, $t_{OSIP-busy} (%)$ is calculated by Equation 4.1, in which $t_{cmd}$ is the time that OSIP spends on executing the commands and $t_{app}$ is the total...
execution time of an application. The higher $t_{\text{OSIP-busy}}(\%)$ is, the more stressed OSIP becomes.

$$t_{\text{OSIP-busy}}(\%) = \frac{t_{\text{cmd}}}{t_{\text{app}}} \cdot 100\% \quad (4.1)$$

Naturally, as a hypothetical extremely fast task manager, which can execute any command within zero time, UT-OSIP always has a busy time of zero. At the other extreme, LT-OSIP has a very high busy time percentage in most system configurations. In the average case and worst case, its busy time is close to 100% in the large systems. In comparison to LT-OSIP, OSIP has a much lower busy time. Most of the time, it is below 50%. This indicates that OSIP still has high potential to perform much more scheduling work.

Figure 4.5 compares the energy consumption of OSIP and LT-OSIP in the synthetic application. In contrast to the energy comparison by Equation 3.3 in Section 3.3.2, in which OSIP is analyzed in an isolated way and assumed to be always busy, here the comparison takes both the OSIP busy time and the application execution time into consideration. The calculation of the energy consumption ratio between LT-OSIP and OSIP is given in Equation 4.2, 4.3 and 4.4.

$$\frac{E_{\text{LT-OSIP}}}{E_{\text{OSIP}}} = \frac{P_{\text{LT-OSIP}} \cdot t_{\text{app,LT-OSIP}}}{P_{\text{OSIP}} \cdot t_{\text{app,OSIP}}} \quad (4.2)$$

with

$$P_{\text{OSIP}} = P_{\text{busy,OSIP}} \cdot t_{\text{OSIP-busy}}(\%) + P_{\text{idle,OSIP}} \cdot (1 - t_{\text{OSIP-busy}}(\%)) \quad (4.3)$$

$$P_{\text{LT-OSIP}} = P_{\text{busy,LT-OSIP}} \cdot t_{\text{LT-OSIP-busy}}(\%) + P_{\text{idle,LT-OSIP}} \cdot (1 - t_{\text{LT-OSIP-busy}}(\%)) \quad (4.4)$$

The curves of the energy ratio in Figure 4.5 show an increasing trend of the energy improvement by OSIP with the increasing system size. A maximum energy improvement factor of $4.6 \times$ can be observed in the figure.

### 4.3.2 H.264

The analysis of the OSIP efficiency in the H.264 application is based on the comparison of the frame rates of the video decoding, which are given in the upper part of Figure 4.6. The lower part of the figure presents the OSIP busy time.

As shown in the figure, in this real application OSIP also performs very closely to UT-OSIP, similarly as in the synthetic application. With more PEs integrated in the system, the frame rate increases steadily. Starting with the 10-PE-system, the frame rate tends to become saturated. The biggest frame rate difference between the OSIP-based and UT-OSIP-based systems exists at the configuration of 8 PEs, which is 3 fps, corresponding to a performance degradation of 9%, while for the small systems such as the 1-PE- or 2-PE-system, the difference is actually negligible.
4.3. Efficiency Analysis with an Ideal Communication Architecture

Figure 4.5: Synthetic application: Energy consumption ratio between LT-OSIP and OSIP

Figure 4.6: H.264: OSIP efficiency analysis in systems with an idealized communication architecture
As a comparison, the frame rates in the LT-OSIP-based systems are much lower. The largest difference compared to the UT-OSIP-based systems exists in the configuration of 10 PEs, which is around 18.5 fps, corresponding to a performance degradation of 52.8%. For small systems, the frame rate differences are also quite obvious.

It can also be seen from the figure that the highest frame rate achieved in the LT-OSIP-based systems is in the 5-PE-system. Afterwards, the frame rate drops slightly due to the more workload to LT-OSIP created by the more PEs. It means that for this application, LT-OSIP can effectively support up to 5 PEs. In comparison, OSIP can effectively support much more PEs.

If comparing the busy time of OSIP and LT-OSIP, the former one increases much slower than the latter one with the increasing number of PEs. For LT-OSIP, almost every additional PE increases its busy time significantly. In contrast, OSIP has a very low busy time even in the large systems, which clearly shows its efficiency.

The energy consumptions of LT-OSIP and OSIP in H.264 are compared in Figure 4.7. Again, OSIP also gains more advantages for energy consumption in larger systems. In the largest system, the energy consumption of OSIP is only 23.3% of that of LT-OSIP.

The two case studies in this section compare the efficiency of three different OSIP implementations for task management. Based on the ASIP concept, OSIP is able to improve the system performance significantly in comparison to the RISC-based LT-OSIP in multi-processor systems with much lower energy consumption. Especially in large systems, in which LT-OSIP in fact fails in task management, OSIP still provides high scheduling and mapping ability. This indicates that for such systems an ASIP-based task manager is more applicable than a RISC.

Overall, for the synthetic application, the performance of the OSIP-based systems is at maximum 4.3 times as much as that of the LT-OSIP-based systems, with OSIP
4.4. Impact of Communication Architecture

Certainly, an ideal communication architecture with zero communication overhead does not exist in reality. In this section, an investigation is made to analyze what kind of impact a realistic communication architecture can have on OSIP-based systems.

Generally, both the communication architecture and OSIP are shared resources in the system. Thus, they have a commonality if considering the relation between their load and the system size. When a large number of PEs are used in the system, high workload is generated to OSIP. At the same time, heavy traffics are generated to the communication architecture as well. Therefore, it is of extreme importance to consider the impact of the communication architecture and the OSIP efficiency on the system performance jointly.

In order to obtain a first impression how a realistic communication architecture can influence the performance of OSIP-based systems, an initial analysis of the communication overhead is made. The analysis is based on the AHB bus, which has been bypassed in the simulation platform in the previous section and is now really used in the system. In this initial analysis, only OSIP and the H.264 video decoding are chosen as the task manager and the target application, respectively. The analysis results are presented in Figure 4.8.

In the figure, the frame rates in the systems using an ideal bus and using a real AHB bus are compared. A large performance gap between both systems can be consuming 22.1% of the energy of LT-OSIP. For H.264, the maximum performance improvement factor is $2.0 \times$, with OSIP consuming 25.0% of the energy of LT-OSIP.

In the remaining part of the thesis, only the system performance is further considered when analyzing the OSIP efficiency.

4.4 Impact of Communication Architecture

Figure 4.8: H.264: Impact of the communication architecture in OSIP-based systems
observed for all system configurations. The average drop of the frame rate using the real AHB bus lies at a factor of $2.4 \times$, and the maximum factor lies at $2.8 \times$ in the system with 11 PEs.

This large performance gap is not only due to the latency of the real bus, but also due to bus contention which additionally introduces high communication overhead. The effect of the bus contention can be seen from the trend of the frame rate. In the systems with an ideal bus, the frame rate has an increasing trend until the performance becomes saturated. In comparison, the frame rate based on the real AHB increases at the beginning, then starts to decrease slowly after integrating more than 6 PEs into the system. In these systems, the intention of improving the system performance by adding more PEs is not fulfilled. The benefit gained from the parallel execution of the tasks on more PEs is completely reduced by the additional overhead caused by the bus contention.

### 4.4.1 Detailed Analysis of Communication Overhead

To gain insight into the communication overhead in the systems with the real AHB bus, a detailed analysis is made, the result of which is presented in Figure 4.9. From the perspective of a PE, the execution time of each PE is contributed by three parts:

- **Active time ($t_A$):** $t_A$ is the time that a PE spends on executing the instructions. In case that the execution of an instruction needs to access a shared component (e.g., the shared memory) through the bus, the time spent on the bus is excluded from $t_A$.

- **Idle time ($t_I$):** $t_I$ is the time, during which a PE does not get tasks assigned and stays in a low-power state. In this state, the PE does not execute instructions, is therefore idle. It recovers from the idle state and becomes active again when it receives an interrupt from OSIP at the arrival of a new task.

- **Communication time ($t_C$):** $t_C$ is the time that is explicitly spent on the communication initiated by a PE to the other system components that are connected to the bus.

In the figure, for each system configuration, the average of each part of the execution time described above is calculated among the total PEs and presented in percentage. It is easy to identify that for all systems, the communication time is the main contributor to the total execution time. In all system configurations, more than 50% of the total execution time is spent on the communication, which is doubtlessly the system bottleneck.

In comparison to the communication time, the active time consistently decreases with the increasing number of PEs, because each PE obtains less tasks. In the end, it becomes only a small portion of the total execution time in the large systems. The fact that less tasks are executed in each PE in large systems makes them more frequently enter the idle state, which is illustrated by the trend of the increasing idle time.
The impact of the communication architecture on the system performance is also reflected by the OSIP state. For the same implementation of an application, the number of tasks is fixed. Due to this fact, the total number of requests generated from the PEs to OSIP, in form of OSIP commands, is approximately constant independent of the communication architecture. So, for the same system size, the total scheduling and mapping time of OSIP does not differ much for different communication architectures. However, a slow communication architecture increases the total execution time of the application, hence reduces the busy time of OSIP in percentage. As shown in
Figure 4.10, the busy time of OSIP in the systems with a real AHB is much lower than that with an ideal bus. In the former, OSIP mostly stays at an idle state ($t_{OSIP-busy}(%)$ below 8%).

On the other hand, with the same communication architecture, logically requests would be generated to OSIP more frequently, if there are more PEs integrated in the system. So in a large system OSIP would be kept in the busy state more often. However, in Figure 4.10, this statement only holds true for the systems with the ideal bus, in which the busy time of OSIP increases steadily with more PEs. For the systems with the real AHB, the busy time of OSIP first increases with the increasing number of PEs, then becomes nearly unchanged. The traffic contention in these large systems prolongs the actual task execution time, hence slows down the request generations to OSIP.

Based on the analysis above, it can be concluded that an unoptimized communication architecture can have a disastrous effect on the system performance. It easily makes OSIP largely underutilized, delaying the task scheduling and mapping. Therefore, in order to improve the utilization of OSIP, the communication architecture has to be optimized. This is addressed in the next section.

### 4.5 Optimized Communication Architecture

Three optimization steps for the communication architecture of the OSIP-based systems studied in this chapter are proposed, which include employing a multi-layer AHB, a cache system with coherence control and write buffers. These optimizations are quite common in general, but the goal is to evaluate the impact of different communication architectures in OSIP-based systems.

#### 4.5.1 Multi-layer AHB

The advantages of a multi-layer AHB [10] against an AHB bus is that parallel accesses from bus masters to bus slaves are possible, as long as the masters do not address the same slave at the same time. This type of parallel communication fits well into the communication characteristics in the OSIP-based systems, in which three types of independent data communications can be differentiated:

- **Communication between PEs and OSIP**: This communication is specific for OSIP-based systems. Based on it, the information is exchanged between the PEs and OSIP, which is needed for the system-wide scheduling, mapping and synchronization. Therefore, this communication type is very control-centric. Typically, the number of the tasks and PEs influences this communication.

- **Communication between PEs and shared memory**: In this communication, data are exchanged between different tasks, which may run on the same PE or on different PEs. The data payload of the tasks determines the required communication bandwidth of the system. So, this communication is data-centric.
• **Communication between PEs and peripherals**: This communication serves for reading the input data stream and sending the results to the peripherals. It however only contributes to a minor part of the total communication.

By applying the multi-layer AHB, the communication overhead caused by bus contention in the AHB bus can be effectively reduced, especially if the control-centric communication and the data-centric communication are well balanced. In the current simulation platform, all PEs are treated equally. Therefore, each processor is connected to a dedicated bus layer, which means that the multi-layer AHB is fully connected. However, the area of a full multi-layer AHB for a large number of processors can be very big, which should be carefully considered during the system design. Tradeoffs need to be made by e.g., sharing a bus layer between several PEs.

### 4.5.2 Cache System

Compared to a multi-layer AHB, which mainly focuses on improving bus contention, a cache focuses on reducing bus accesses by exploiting data locality. The low latency of accessing data locally can result in a significant reduction of the total communication time, if the cache works properly. Moreover, the reduced accesses to the bus potentially also reduce the communication overhead caused by bus contention.

However, in multi-processor systems cache coherence is of vital importance. Data inconsistency occurs if a PE is updating the data in the shared memory, while another PE is still reading the old data from its local cache. This leads to system malfunction. So, the cache system must ensure that the data fetched from the PE must be the latest updated data, and the local copy of the data in the cache must be consistent with the data in the shared memory. In the literature [58, 116, 196, 197], this problem has been thoroughly studied. In this work, a cache coherence system based on the write-broadcast approach has been implemented at the system-level, which is illustrated in Figure 4.11.

The cache system contains two basic functional modules: a local cache module for each individual PE and a global Cache Coherence Management Unit (CCMU), which manages the data consistency crossing the local cache modules.

Each local cache module consists of a local cache controller and a cache memory, following the principle of a four-way set associative cache. The data replacement policy in case of cache miss is **Least Recently Used** (LRU). The cache controller has two functions. On the one hand, it manages the data in the cache memory, such as accessing the local or shared memory and updating the cache data. On the other hand, it acts as a bridge to the CCMU using an additional interface. Through this interface, information is exchanged from one cache over the CCMU to the other caches.

When reading data, this multi-cache system behaves exactly in the same way as a normal single-cache system. The local cache controller checks whether there is a valid data copy in the cache. If so, the controller reads the data from the cache and sends the value to the PE. Otherwise, the controller fetches the data from the shared memory and updates the cache correspondingly.
The difference from a single-cache system exists when a local controller writes the data to the shared memory. In this case, the CCMU is involved. The write mechanism works as the following: Whenever a cache controller receives a write request from the PE, it sends the request to the bus to store the data into the shared memory. In addition, it forwards the request, including both the address and the data, to the CCMU, which then broadcasts the request to all other cache modules. Upon receiving the broadcasting information from the CCMU, each cache module checks whether there is a data entry at the required write address in its cache memory. If an entry is found at that address, it has to be updated with the data from the CCMU. Otherwise, no further actions need to be taken in the cache. Afterwards, each cache controller sends a confirmation to the CCMU, which in turn generates a response over the initial cache module back to the PE to complete the write process. In this way, the multi-cache system ensures the consistency of the data stored in the caches and the shared memory.

Note that this cache system is only applied onto the data accesses to the shared memory. It does not help in improving reading data from OSIP. Reading OSIP mostly happens in two situations: a) reading the return values of a command and b) reading certain status information, such as the OSIP status or the status of a spinlock. For the former situation, the return values are always freshly generated by OSIP. Therefore, there is no use of caching these return values. For the latter situation, the PEs poll the information, until the desired information is set either by OSIP or by the other PEs. This polling time cannot be reduced by using the cache. Even if the cache is hit, but if it does not contain the desired information, the polling still continues.

Figure 4.11: Cache system
4.5.3 Write Buffer

Write buffers are typically used to improve writing data to bus slaves. Instead of reducing the number of bus accesses and bus contention, they enable parallelization of the communication and computation. The PEs offload data transfers to the write buffers and can continue with further instructions of the task execution, without having to wait for the write responses from the bus slaves.

In this work, the implementation of write buffers follows a similar principle as introduced in [179], in which a hardware FIFO queue is inserted between each cache module and the bus. It confirms the write request from a PE before the request actually reaches the bus. When a write request arrives, it first appends the request to the FIFO queue if the FIFO is not full, and generates immediately a response to the PE. If the FIFO is full, the write request stays pending until one entry in the FIFO becomes free.

The behavior of a write buffer becomes somewhat complicated, if there is a cache miss when reading an address. In this case, the word at the address has to be fetched from the shared memory. However, if the word currently happens to be in the write buffer, reading from the shared memory returns a wrong word. To avoid this, the read access is suspended until all requests in the write buffer are processed. Only then the word is read from the shared memory. In other words, a read cache miss empties the write buffer.

Similar to multiple caches, multiple write buffers also have a consistency problem. Suppose that PE\textsubscript{A} needs to write a word to the shared memory, which shall be used by PE\textsubscript{B}. It can happen that the word from PE\textsubscript{A} is still located in its write buffer at the time PE\textsubscript{B} reads the shared memory. So, PE\textsubscript{B} fetches an outdated value. However, in OSIP-based systems this kind of race condition is prevented by the spinlock registers in the OSIP register interface, if following the programming model of OSIP-based systems. The spinlock registers are generally used to prevent shared resources from being accessed by multiple PEs concurrently. Briefly, if multiple PEs try to access a shared resource, e.g., an shared memory element, they must first acquire a spinlock register before they access it. The spinlock is only owned by one PE at a time. After accessing the shared resource, the owner PE releases the spinlock and the ownership is passed to the next PE. Before the spinlock is released, the other PEs keep on polling it. The detailed spinlock mechanism in OSIP-based systems is described in Chapter 5.

The example in Figure 4.12 illustrates how the race condition is prevented in OSIP-based systems with spinlocks. Since PE\textsubscript{B} reads data generated by PE\textsubscript{A}, there is a data dependency between the tasks on PE\textsubscript{A} and PE\textsubscript{B}. Let Task\textsubscript{A} be the task running on PE\textsubscript{A} that writes word \textit{A} to the address \textit{Addr}\textsubscript{A} in the shared memory, and let Task\textsubscript{B} be on PE\textsubscript{B} that reads \textit{A}. Task\textsubscript{B} depends on Task\textsubscript{A}. To write the address, PE\textsubscript{A} first acquires a spinlock for \textit{Addr}\textsubscript{A}. Then, it writes \textit{A} to the memory. Afterwards, it releases the spinlock, which is a special type of write to the OSIP register interface. Both write requests, i.e., “Write A” and “Release spinlock” are added into the FIFO queue of the write buffer of PE\textsubscript{A}. Till now, Task\textsubscript{B} has still not yet been activated, because Task\textsubscript{A} has not been finished. After releasing the spinlock, Task\textsubscript{A} activates Task\textsubscript{B}, and Task\textsubscript{B}...
Figure 4.12: An example of preventing data race conditions for write buffers by using spinlocks

starts to poll the spinlock to get the access to $A$. At this time, $A$ could still be in the write buffer of $\text{PE}_A$ because of bus contention. On the other hand, as long as $A$ is in the write buffer, the spinlock cannot be released because of the FIFO principle. This means, $\text{Task}_B$ is never able to read from the memory, if $A$ is still in the write buffer. In other words, if $\text{Task}_B$ acquires the spinlock, $A$ has already been written into the memory. Therefore, the race condition never occurs.

4.5.4 Results

In this section, the three optimization steps for the communication architecture are evaluated based on the H.264 and the synthetic application with detailed analysis presented only for H.264. The following notations are made for the different communication architectures resulted from the different optimization steps:

- **CA0**: AHB bus, which is the baseline communication architecture for the analysis and considered as unoptimized.
- **CA1**: Multi-layer AHB bus.
- **CA2**: Multi-layer AHB bus with the cache system.
- **CA3**: Multi-layer AHB bus with the cache system and write buffers.
- **CA4**: Ideal communication architecture, in which the AHB bus is bypassed and there is no communication overhead.

In addition, the local cache memory size and write buffer size are set to 4 kB and four words respectively in the following discussions.
4.5. Optimized Communication Architecture

4.5.4.1 H.264

Figure 4.13 gives an overview of the effect of the different optimizations on the system performance by comparing the frame rates of H.264. The performance improvement by the optimizations is significant: In the largest system with 11 PEs, from CA0 to CA3 the frame rate is increased by 11.7 fps, corresponding to a factor of $1.9 \times$. In the smallest system with a single PE, the frame rate is increased by 2.5 fps, corresponding to a factor of $1.7 \times$.

The contribution of the optimizations to the system performance improvement is different for different system configurations. For this application, up to 6 PEs, the improvement is mainly introduced by the cache system and the write buffers. For example, in the 6-PE-system, 91.9% of the improvement is achieved with these two optimization steps. In contrast, the contribution of the multi-layer bus is only 8.1%, because in these configurations the communication is more data-centric than control-centric.

However, in larger systems, in which more and more communication takes place between the PEs and OSIP, the control-centric and data-centric communications becomes more balanced. In these systems, the multi-layer bus plays a more important role in the performance optimization than in small systems. For example, in the 11-PE-system, 28.1% of the improvement is contributed by the multi-layer bus. This shows that in OSIP-based systems, the optimization for the communication architecture should not only focus on the data communication, but also on the control-centric communication between the PEs and OSIP, which also has a big impact on the system performance.

Naturally, the performance gap between the systems with an ideal communication architecture (CA4) and the systems with a realistic communication architecture (CA0 – CA3) gains advantage from the increasing number of PEs, as depicted in the figure.
In latter systems, more bus contentions happen when using more PEs, which largely lowers the benefits obtained from the task parallelism. This is however not a limitation for the systems with an ideal bus.

A detailed analysis of the communication overhead at different optimization levels is given in Figure 4.14, based on the 11-PE-system. It is easy to observe that the communication time ($t_C$) is significantly reduced by a factor of $2.8 \times$ from CA0 to CA3. In the meantime, the idle time of the PEs ($t_I$) is decreased by a factor of $1.6 \times$, which is an important side effect of the optimization for the communication architecture. By optimizing the communication, the execution time of the tasks is reduced, which consequently reduces the time of resolving dependencies between the tasks and preparing new tasks. Therefore, the PEs do not need to wait for that long anymore before they receive a new task.

In comparison to $t_C$ and $t_I$, the active time ($t_A$) of the PEs is only slightly reduced, since each PE still gets the same amount of tasks assigned, independent of the optimization for the communication architecture. An exception exists in the system with CA4, which has a much higher $t_A$. The cause are the significantly increased software API calls. Due to the zero-latency in the bus, each polling of the interface information of OSIP needs less time, for which the number of the polling operations increases. In this sense, the communication overhead is partially shifted to the execution of the software APIs.

### 4.5.4.2 Synthetic Application

The performance of the synthetic application also significantly benefits from the optimizations for the communication architecture. Figure 4.15 summarizes the execution time of the synthetic application in different scenarios with different communication
4.6 Joint Consideration of OSIP Efficiency and Communication Architecture

In the previous sections, the impact of the OSIP efficiency and the communication architecture on the system performance has been analyzed in one dimension. Section 4.3 fixes the communication architecture and varies the OSIP implementation, while in Section 4.4 and 4.5, the OSIP implementation is fixed and the communication architecture varies. The analysis results show that both OSIP and the communication architecture are essential for performance. However, it is also important to analyze the OSIP efficiency in a communication context, i.e., how is the impact of the OSIP efficiency changed with different communication architectures? To answer this ques-

 architectures. From CA0 to CA4, large reduction of the execution time can be observed for all scenarios.

Similar to the observations made for H.264, the effect of the different optimization steps largely depends on the application characteristics. For example, the cache system leads to the highest performance improvement in the best case scenario, but relatively low improvement in the worst case scenario. In contrast, the multi-layer bus has only little effect in the best case scenario, but increases the execution time in the average case scenario significantly in the large systems. The same phenomenon can also be found for the write buffers. Therefore, it is extremely important to take the target applications into consideration when optimizing the system communication architecture. Especially for OSIP-based systems, both the control-centric and data-centric communications must be considered with care.

![Figure 4.15: Synthetic application: Execution time at different optimization levels](image)

4.6 Joint Consideration of OSIP Efficiency and Communication Architecture
4.6.1 H.264

The joint effect of the OSIP efficiency and the communication architecture for H.264 is illustrated in Figure 4.16. Naturally, for all analyzed communication architectures the frame rate of the system with LT-OSIP is lower than that with UT-OSIP or OSIP, as LT-OSIP is not an efficient task manager. However, the performance gap between the systems with a fast manager (UT-OSIP/OSIP) and a slow manager (LT-OSIP) is gradually narrowed from CA4 over CA3 to CA0. Taking the 11-PE-system as an example. At CA4, using UT-OSIP/OSIP instead of LT-OSIP increases the frame rate by a factor of $2.06 \times / 1.94 \times$, while at CA3 this improvement factor is reduced to $1.89 \times / 1.83 \times$, and at CA0 the factor is only $1.58 \times / 1.55 \times$. This shows that without the support of a well designed communication architecture, the efficiency of a fast task manager is wasted to a large extent.

On the other hand, once the communication architecture is highly optimized, a fast task manager gets more actively involved in the task management. In contrast, a slow manager could more easily become the system bottleneck with an optimized communication architecture, especially in large systems. As shown in the figure, from CA3 to CA4 the performance of the LT-OSIP-based systems increases relatively slowly. Even an ideal communication architecture is not able to help in improving the performance much. However, in the OSIP-based systems, the optimization from CA3 to CA4 still results in significant performance improvement with the support of OSIP. This indicates that OSIP is more suitable for high-performance systems than LT-OSIP, which can also be confirmed by the low busy time of OSIP in the figure.

Furthermore, while a slight performance difference still exists between the OSIP- and UT-OSIP-based systems at CA4, this difference becomes only marginal in the systems with a real communication architecture (CA0 and CA3). This observation clearly shows the OSIP efficiency from the view of the practical use of OSIP in a realistic system.

4.6.2 Synthetic Application

The synthetic application provides a wider view of the joint impact of the OSIP efficiency and the communication architecture. In the following, detailed discussions are made using the three OSIP workload scenarios defined in Section 4.2.1.1.
4.6. Joint Consideration of OSIP Efficiency and Communication Architecture

Figure 4.16: H.264: Joint impact of OSIP and the communication architecture

4.6.2.1 Best Case Scenario

The upper part of Figure 4.17 compares the execution time of the synthetic application in the best case scenario, in which low workloads are generated to OSIP. Overall, the UT-OSIP- and OSIP-based systems have a very similar performance profile, while the LT-OSIP-based systems have a reduced performance characteristic.

However, when using an unoptimized communication architecture, the performance difference with the different OSIPs is small. This is due to the fact that the communication dominates the execution time of the application, which is the bottleneck of the system.

By optimizing the communication architecture, the system bottleneck starts to move from the communication architecture to other system parts. As can be seen from the figure, in small systems with an optimized communication architecture, the impact of the different task managers on the system performance is still relatively small. In these systems, the dominating factor is the task execution time. In comparison to the task execution time, the scheduling and mapping time is short. With
the continuously increased number of CPEs, LT-OSIP begins to reach its performance limit with respect to the scheduling and mapping. Starting from the 7-CPE-system, the performance gap between the LT-OSIP-based systems and the UT-OSIP-/OSIP-based systems becomes very large. In this case, a slow task manager is not able to efficiently handle frequent requests originated from the large number of CPEs anymore. The busy time of LT-OSIP given in the lower part of figure shows that LT-OSIP in large systems with a fast communication architecture is highly stressed. Mostly it is busy with task management for more than 80% of the time.

In general, a more optimized communication architecture makes the task managers (except UT-OSIP) more frequently enter the busy state, because the task execution time is shortened and the CPEs send more frequently requests to the manager correspondingly. So, the busy time of a task manager increases with the optimization of the communication architecture. However, in comparison to LT-OSIP, OSIP is much less stressed for all communication optimization levels.
4.6.2.2 Worst Case Scenario

The execution time of the application in Figure 4.18 demonstrates a more peculiar system behavior in the worst case scenario than in the best case scenario. In this scenario, high load is created to the task manager, for which using LT-OSIP as the task manager should not be considered. While it is still able to reduce the execution time from the 1-CPE-system to the 3-CPE-system, the system performance becomes disastrous if more CPEs are used. Which communication architecture to use in the system only plays a minor role for the performance. In this case, not only the task management is inefficient, the time spent on the scheduling and mapping algorithm itself contributes to the major part of the total execution time of the application.

In contrast, in both OSIP-based and UT-OSIP-based systems, the optimization for the communication architecture plays a key role in improving the performance. Thanks to its scheduling and mapping efficiency, OSIP has high potential of cooperating with highly optimized communication architectures without becoming the system bottleneck. As a comparison, LT-OSIP does not have this potential. Its busy time is already very close to 100% at CA3 and CA4 in most of the system configurations.

It can also be observed that applying more than 3 CPEs cannot help in further improving the system performance, even with an ideal task manager (UT-OSIP) and an ideal communication architecture (CA4). The main reason lies in the task parallelism. In the application, the PPE creates $\text{Tasks}_c$ based on the APIs. Although the time for executing the APIs is short, memory allocations need to be made for creating the tasks. The memory allocations are protected by a spinlock, which is at the same time also required by the CPEs. So competitions for the same spinlock exists between the PPE and the CPEs, which becomes especially critical when the task size is small. In this scenario, the CPEs execute each task very fast and the possibility that multiple CPEs compete with the PPE is high. It leads to the fact that the PPE is not able to prepare the tasks fast enough for the CPEs. In other words, the task parallelism cannot match the number of the CPEs. This is basically a synchronization problem, which will be discussed in more detail in the next chapter.

In the wide sense, synchronization can also be regarded as one special type of communication, which appears to be the bottleneck of the system in the worst case scenario. This also explains why in this case, the OSIP-based systems still perform almost as well as the UT-OSIP-based systems. In fact, mostly OSIP stays at an idle state in the system with a real communication architecture, as shown in the figure. And even in systems with the ideal bus, OSIP stays at the busy state at most for 44% of the time.

4.6.2.3 Average Case Scenario

The analysis results in the average scenario (see Figure 4.19) demonstrates high similarity as those in the best scenario. Only in the systems with an unoptimized communication architecture, the LT-OSIP-based systems have a comparable performance as the other two. In other cases, the UT-OSIP- and OSIP-based systems, which are similar to each other in the performance, are much better.
To again highlight the fact that an unoptimized communication architecture can become the system bottleneck, adding more than 5 CPEs at CA0 even worsens the system performance for all configurations. With this communication architecture, high bus contention takes place, which in fact impairs parallel task execution.

The effect of the task synchronization can also be observed in this scenario, e.g., in the OSIP-based systems using CA4. The busy time of OSIP first increases with the increasing number of CPEs. Then starting from the 9-CPE-system, it decreases, in which task synchronization becomes the most critical factor influencing the performance. In comparison to the worst case scenario, the critical effect of the synchronization occurs only in large systems in this scenario. This explains why here the busy time of OSIP in the 5-CPE- and 7-CPE-system at CA4 is even higher than in the worst scenario. In the latter one, the inefficient synchronization blocks a smooth task generation, which in the end results in a shorter task list in OSIP than in the average case and hence reduces the OSIP load.

Figure 4.18: Synthetic application: Joint impact of OSIP and the communication architecture in worst case scenario
4.7 Summary

In this chapter, the efficiency of OSIP is analyzed from the system level, based on a synthetic application and a real-life application. Especially the joint impact of the OSIP efficiency and the communication architecture is thoroughly investigated by comparing the systems with an unoptimized AHB bus, an ideal bus and a realistic but highly optimized bus. The analysis results show that from the system point of view, OSIP has very high efficiency in task scheduling and mapping, which is comparable to a hypothetical extremely fast task manager.

In addition, the communication architecture plays an important role in the OSIP-based systems. In order to fully utilize the OSIP efficiency, an optimized communication architecture is required. Otherwise, the OSIP efficiency would be wasted to a large extent. More generally, designing MPSoCs based on a central task manager
must take special care of the balance between the efficiency of the task manager and the communication architecture.

As shown in the synthetic application, the synchronization is another important factor, which can limit the utilization of OSIP. In the next chapter, how to effectively handle the synchronization problem is addressed by using the flexibility of OSIP.
Chapter 5

OSIP Support for Efficient Spinlock Control

In MPSoCs, applications are partitioned into tasks for parallel execution on different PEs. To protect shared resources like memories or I/O peripherals and to prevent data corruption, mutual exclusion needs to be guaranteed. Spinlocks are widely used to ensure that a shared resource can only be accessed by one task at a time. This introduces the synchronization effect between the tasks, which can have a large impact on system performance.

Generally, without a priori application knowledge, the control of spinlocks is often highly random, which can delay the resolving of the synchronization and consequently degrade system performance. A simple example is depicted in Figure 5.1, in which two PEs compete for the same spinlock. In the example, two different execution sequences of tasks are possible by acquiring the lock in different orders. The execution time shows that assigning the spinlock to PE2 first (Figure 5.1(b)) is a better choice, which achieves higher performance.

Despite its simplicity, this example shows that a smart control of spinlocks can increase system performance. To guide the spinlock control in a smart way, application knowledge such as task sizes and task dependencies needs to be considered.

In this chapter, an advanced spinlock control mechanism is introduced, using OSIP-based systems as the experimentation platforms. In this mechanism, the application knowledge is included into the spinlock control flow. As application knowledge varies from one application to another, it is important to have user-defined

---

**Figure 5.1:** Impact of the spinlock acquisition order
spinlock control algorithms, which are able to be adapted to applications. This calls for a programmable processor, either a RISC or an ASIP, to execute the control algorithm. Instead of developing yet another controller, which would introduce high area cost, in OSIP-based systems, the existing task manager – OSIP, is re-used for this purpose in addition to its original design purpose for task scheduling and mapping.

This chapter is organized as follows. First, some related work on the spinlock control is discussed. Then, the basic spinlock control flow in OSIP-based systems is introduced. This basic flow is then extended to an application-aware control flow, which starts from an high-level integration of application knowledge down to a low-level realization. The two case studies in the previous chapter — the synthetic application and the H.264 video decoding — are further analyzed and discussed based on the extended spinlock control mechanism. Finally a summary is given and some further discussions are made.1

5.1 Research on Spinlocks

Spinlocks are a very commonly used technique to address mutual exclusion in multiprocessor systems. Many implementations have been proposed for spinlocks in the literature.

The simplest approach is the test-and-set lock, which repeatedly tries to replace the flag of a lock with true in order to acquire the lock. Although the implementation is rather simple, it introduces heavy traffic load due to continuous updates of the flag, especially in cache coherent systems. One improvement is suggested by using the test and test-and-set lock [156], in which updates are only made when the lock is assumed to be available. Another important improvement is made in [7] by adding a certain delay (backoff) between two unsuccessful trials.

Having the test-and-set lock, the grants to the requests can be very unevenly distributed. In contrast, a queuing lock provides better fairness, in which the PEs acquire a lock in turns (FIFO fairness). In [7], an array-based queuing lock is introduced, in which the requests of PEs are maintained using an array per lock in the shared memory. When a PE releases a lock, the control of the lock is passed to the next requesting PE in the array. To support array-based locks, large memory space is required, given as O(pn) for p processes and n locks. This space is reduced to 2p+n words by the Mellor-Crummey and Scott (MCS) queuing lock in [130]. It employs a linked list for the requests, constructed by pointing each request to its successor. A similar approach to the MCS lock is the Craig, Landin and Hagersten (CLH) lock presented in [48,118], in which each process points to its predecessor in the linked list.

Recent research also applies hardware support for queuing locks to improve the efficiency. A dedicated hardware controller is presented in [212], which is provided to each PE and uses a set of register pairs to locally maintain the queue information

and grant the locks. Each pair contains a TOTAL register and an AHEAD register, showing how many PEs are requesting a lock in total and how many PEs have requested the lock before its own PE, respectively. The controller updates the registers by observing the lock requests on the bus, and can furthermore decide whether to switch the PE to a low-power state, when waiting for a lock. In the work of [38], the synchronization problems in NoCs are addressed. A hardware module called Synchronization Handler (SH) is used at each node of the network for the queue management. Different from these two distributed approaches, a central hardware module called Synchronization-Operation Buffer (SB) is introduced in [134]. It is located in the memory controller to manage the lock queue and notifies a PE automatically if it decides that the PE should be the next one to acquire the lock. By this, the traffic overhead due to the polling can be significantly reduced. Another central approach, not directly related to the queuing lock, but to reduce the memory contention caused by polling is presented in [94]. They suggest a hardware lock unit separated from the memory to support spinlocks.

However, in all above-mentioned approaches, the acquisition of locks features high randomness, since the grant decision solely depends on the time stamp when the locks are requested, regardless of the fairness. As mentioned at the beginning of this chapter, a proper control of spinlocks would most likely have a positive impact on system performance. Priority-based spinlocks are a way of implementing this, in which locks are granted by comparing the priorities of the requests in the queues. In the work presented in [48], which process receives the grant to a lock as the next, is determined by looping over the queue at the time the lock is released. In contrast, the work in [98] suggests sorting the queue by the priority during the time that the lock is being held by others. However, while the former approach potentially increases the workload of the PEs, the latter one needs to extend the process to support the insertion of spinlock requests into the queue at proper positions. And in general, priority-based approaches can suffer from the starvation problem. In [185], a central hardware memory controller is proposed to arbitrate the competitions for protected memory regions based on the dynamic priority scheme [204]. However, no architectural detail has been given by the authors, and the presented results based on a dual-core system are not representative for typical multi-processor systems.

The spinlock control mechanism presented in the chapter is based on a centralized way, which has some similarities to the approach introduced in [94]. However, the main distinctions of this approach are made by:

- Enabling the programmer to introduce application-specific spinlock control information into the control flow to guide spinlock decisions. The control information can be, but is not limited to priorities.

- Utilizing the flexibility of an existing task manager, the OSIP, to execute user-defined, application-aware spinlock control algorithms. If needed, runtime system information can be included into the algorithms to make the most suitable decisions. Furthermore, with the user-defined control algorithms, the starvation problem can be efficiently handled.
5.2 Basic Spinlock Control in OSIP-based Systems

The analyzed OSIP-based systems presented in the previous chapter use a basic compare-and-swap mechanism in controlling spinlocks. A set of spinlock registers are implemented in the OSIP register interface to maintain the spinlock information. These spinlock registers including their control logic do not interact with the OSIP core, therefore can also be implemented as an external module outside OSIP (see Section 3.1.3). Depending on the system and target application requirements, this spinlock control unit needs to be customized, for example, for configuring the number of spinlock registers. However, in the following, the spinlock registers and their control logic are still considered as a part of OSIP register interface for convenient discussions.

The spinlock registers are accessed from the PEs (or the tasks running on the PEs) through a pair of APIs, assisted by the control logic. The control flow is illustrated in Figure 5.2. Before entering the critical section of a task, in which a shared resource must be protected from being accessed by other tasks, the task first requires a spinlock. It sends a locking request to the control logic for acquiring the corresponding spinlock register at the interface. This is done by executing the API function SpinlockAcquire(LockID) (step 1a). LockID specifies the spinlock register. Upon receiving the request, the control logic checks the spinlock status (step 1b). If the spinlock is available, it grants the spinlock to the requester and marks the lock further as unavailable. Otherwise, it returns a failed signal, and the task has to poll the lock register repeatedly until it acquires the lock. After the task obtains the lock and finishes the critical section, it releases the spinlock (step 2a) by calling another API function SpinlockRelease(LockID). The control logic sets the spinlock to available again (step 2b). This control mechanism is rather simple. However, its high randomness could lead to suboptimal performance in spinlock-intensive applications.

![Figure 5.2: Basic spinlock control flow](image-url)
5.3 Application-Aware Spinlock Control

This section introduces an enhancement to the basic spinlock control mechanism used in the OSIP-based systems described above. In the enhanced mechanism, the application information is taken into consideration when making spinlock decisions.

5.3.1 Key Concept

The key idea is to allow the programmer to include application knowledge into the spinlock control flow, and to use OSIP to do an application-aware spinlock control based on that knowledge.

In this context, the term *application knowledge* specially refers to spinlock-related information. It typically considers the task execution time and the time, during which the tasks are blocked by spinlocks. In addition, task dependencies also need to be considered. This is because different spinlock acquisition orders can influence resolving task dependencies, and consequently the task parallelism.

Having the application knowledge, the programmer needs to abstract it, e.g., as priorities indicating how urgent the spinlocks are needed by the requesting tasks, and adds it into the application. This abstracted information is the spinlock control information, which is considered by OSIP at runtime. It is important to note that the programmability of OSIP allows the programmer to define different semantics of the spinlock control information for different algorithms, such as priority-based, round-robin or weighted fair queuing. The programmer can even specify that some PEs always have a higher priority than the others when requesting a certain lock. The algorithm running on the OSIP core can be adapted to interpret the semantics.

Running additional software for the spinlock control naturally introduces more workload to the OSIP core. However, the fast task scheduling and mapping of the OSIP core leaves sufficient time for it to afford additional workload, which has already been shown in the previous chapter.

To make good decisions on the spinlocks, OSIP should consider spinlock control information from different PEs together. To collect the information from different PEs, a spinlock reservation mechanism is proposed in this work. In the following, this mechanism will be explained in detail. Furthermore, the complete extended spinlock control flow will be introduced (see Figure 5.3). The extensions to the original flow, both from the API side and from the OSIP side, are shown in **bold** in Figure 5.3.

5.3.2 APIs for Spinlock Reservation

The spinlock reservation mechanism is used to inform OSIP that a PE wants to obtain a spinlock, and to request OSIP to reserve the spinlock for the PE. Along with the reservation request, the spinlock control information is also provided to OSIP. During the time, in which the required spinlock is acquired by a PE, OSIP can collect several reservation requests from other PEs. Based on the collected information, OSIP judges which one(s) out of these PEs should get the spinlock as the next and makes reser-
Once the spinlock is reserved, the PE(s) that has/have got the reservation has/have advantages over the other PEs when requesting the spinlock. In this way, the randomness of granting spinlock requests is largely reduced.

To help the programmer add the control information into the application in a simple way, an API called SpinlockReserve() is introduced. It contains two parameters: LockID and LockInfo, specifying the required spinlock register and the control information, respectively. For minimizing the communication overhead, both parameters are combined into a 32-bit word before they are transferred to OSIP. Note that the reservation request does not try to acquire the spinlock immediately, but makes a preparation for the spinlock acquisition. Therefore, it must be sent to OSIP before the actual acquisition request starts. Complementarily, another API SpinlockClearReservation() is defined to clear the reservation after the spinlock is released.

These two APIs should be always used in pair to enclose spinlock acquisition and release. A simple example is given in function Task() in Figure 5.3. In practice, this API pair can also be used to enclose multiple spinlock acquisition/release pairs, if desired, as long as they require the same lock. And it is not always necessary to do the reservation for all spinlocks. Instead, identifying the most critical ones and making reservations of these ones with proper reservation information is essential for better system performance.

### 5.3.3 Enhanced Spinlock Control

The APIs for reserving spinlocks require an extension to the original spinlock control logic to store the reservation information and to interact with the OSIP core for the enhanced control of spinlocks. Four stages are defined for the extended spinlock control in OSIP: spinlock information collection stage, decision stage, release stage and reservation stage. The first three stages are purely controlled by the hardware in the interface, while the last stage needs a collaboration between the hardware interface and the software control of the OSIP core.
5.3. Application-Aware Spinlock Control

5.3.3.1 Information Collection Stage

Upon receiving a spinlock reservation request from a PE, the reservation information is stored in an internal register at the interface, which is named Spinlock Reservation Register (SRR) (Figure 5.3, step 1). Each PE has a dedicated SRR with the following four fields:

- **LockID (8 bits):** This is the spinlock ID, indicating which spinlock should be considered for reservation. The current system has 256 spinlock registers in the interface.
- **LockInfo (4 bits):** It is the information needed by the OSIP core to make reservation decisions on the corresponding spinlock.
- **ReservationFlag (1 bit):** This flag indicates whether the corresponding PE of the SRR wants to reserve the spinlock.
- **ReservedFlag (1 bit):** This flag shows whether the spinlock has been reserved for the PE.

The information bits for the spinlock ID and spinlock control information are adjusted to the current system, which can be easily extended.

To avoid confusion, this stage only collects control information, but does not make reservations. The spinlock reservation for a requesting PE is only triggered by other PEs, which is explained in detail later in the reservation stage (Section 5.3.3.4).

5.3.3.2 Decision Stage

When the interface receives a spinlock acquisition request (step 2a), the decision stage starts. In this stage, the interface decides whether the requesting PE acquires the spinlock. The control flow is depicted in Figure 5.4.

First, the interface checks the status of the spinlock (step 2b). If the spinlock is currently already locked, the PE fails to acquire the spinlock and has to poll it again, which is the same as in the original approach. The difference from the original approach exists when the spinlock is currently available. In this state, the interface needs to further check the reservation status of the spinlock (step 2c). If the spinlock has not been reserved for the requesting PE, but reserved for other PEs, the current request still fails. In other words, the request can only become successful, when the requesting PE has got the reservation (no matter whether other PEs also have got the reservation of the same spinlock), or the spinlock currently has not been reserved for any PEs.

From this control mechanism, it can be seen that the PEs that have got the spinlock reserved have a better opportunity to acquire it than the other PEs. By introducing additional control information to the spinlocks, the programmer is now able to influence the reservation decisions made by the OSIP core, consequently the spinlock acquisition order.
5.3.3.3 Release Stage

In this stage, the spinlocks are released (step 3a and 3b), which is the same as in the previous basic spinlock control approach.

5.3.3.4 Reservation Stage

As the key step of the proposed mechanism, the spinlock reservation is made in this stage. It plays an essential role in making proper spinlock decisions. Therefore, the programmer needs to define a proper reservation algorithm for a given application. Since the reservation algorithm runs on the OSIP core, it can also include additional system information besides the spinlock control information, if necessary. This is possible, because the OSIP core has an overview of the system status.

In addition, it is also important to decide when the OSIP core should be triggered to make reservations for the PEs. If the core is triggered too frequently (e.g., it is always triggered whenever there is a reservation request), it would often become unnecessarily busy. This could impair the primary purpose of OSIP, namely the normal task scheduling and mapping. On the other hand, if the OSIP core is only rarely triggered for the reservation, the effectiveness of the reservation mechanism would be reduced.

With these considerations, in the current implementation the OSIP core is triggered only when the following conditions are met:

- The interface receives a clearing signal to the reservation of a spinlock from a PE, after it releases the spinlock. At this point, both flags in the corresponding SRR of the PE are cleared (step 4a).
5.3. Application-Aware Spinlock Control

- The released spinlock is currently not reserved for any other PEs (step 4b). It means that the ReservedFlag of all other SRRs with the same LockID as the spinlock is unset. This is for reducing the workload to the OSIP core.

- There is at least one reservation request for the spinlock from other PEs (step 4b). This means that at least one other SRR with the same LockID as the spinlock has the ReservationFlag set.

- The OSIP core is currently at the idle state (step 4c).

The block diagram of triggering the OSIP core for spinlock reservation is given in Figure 5.5. It is important to mention that the PE clearing the reservation triggers the spinlock reservations for other PEs. However, for clarity, the logic for excluding the PE that clears the reservation is omitted in Figure 5.5.

When the OSIP core is triggered for the reservation, only the SRRs that contain the spinlock ID, the reservation of which has just been cleared by the previous owning PE, are considered by the core. The spinlock control information of these SRRs is transferred to the core through the argument registers in the interface. A command for spinlock reservation is generated from the spinlock control logic to the OSIP core through the command register (step 4c). In the current implementation, the bits of the first argument register are used to mark the SRR IDs that should be considered for reservation. The control information of the SRRs is stored in the remaining argument registers in a halfbyte-aligned way, ordered by the SRR IDs. However, the arrangement of the control information in the argument registers can be easily adapted, if a different number of the information bits is used.
After the OSIP core is triggered, the spinlock reservation is executed as a normal OSIP command (step 4d). The OSIP status is set to busy to prevent further commands from being issued to the OSIP core. The reservation algorithm is user-defined and decides which PE(s) should get the reservation, based on the provided spinlock control information. As mentioned before, the algorithm can also take internal system status into consideration when making the decision. Afterwards, the spinlock control logic gets informed of the decision result by the OSIP core and update the ReservedFlag of the corresponding SRR(s). Note that the reservation of a spinlock is not necessarily limited to one PE. In fact, making a reservation for multiple PEs at the same time helps reducing the workload of the OSIP core and accelerating the response to the spinlock request.

5.3.4 Hardware Overhead

As shown above, the previous basic spinlock control unit is enhanced with additional hardware to support the spinlock reservation mechanism. A number of 14-bit SRRs are needed, as many as PEs. Besides, the spinlock control logic is extended. However, since the computationally most intensive part for making reservation decisions is performed in software, the additional control logic (mainly for storing/extracting information into/from the SRRs and equivalence comparison of spinlock IDs) only results in low hardware overhead, both from the area and timing perspective. For a 12-PE-system, the largest system considered in this thesis, the additional hardware has an area of 3.3 kGE, achieving a maximum clock frequency of 2.0 GHz (synthesized with Synopsys Design Compiler for a 65 nm standard cell library, supply voltage 1.0 V, temperature 25°C). As a comparison, the OSIP core has an area of 35.7 kGE and achieves 690 MHz under the same synthesis conditions (see Section 3.3.1).

The hardware logic for the extended spinlock control in the interface can theoretically be completely implemented in software running on the OSIP core. However, this would on the one hand introduce too much workload to the core. On the other hand, this control logic only requires very low flexibility for a given system, which naturally calls for a native hardware implementation.

5.4 Case Studies

In this section, the two applications introduced in the previous chapter — the synthetic application and H.264 — are further used as the case study applications for analyzing the proposed enhanced spinlock control mechanism. The communication architecture of the systems in the case studies is based on a multi-layer AHB bus with the cache coherence system and write buffers (CA3 defined in Section 4.5.4).

5.4.1 Synthetic Application

As shown in Section 4.6.2.2, in the worst case scenarios, the execution time of the application in UT-OSIP-/OSIP-based systems cannot be improved by applying more
than three CPEs, even if the system uses an ideal bus. A major cause is inefficient task synchronization. If looking deeper into the application, it can be found that Task$_c$ and Task$_{gen}$ have to compete for the same spinlock for protecting the used standard C library functions. Task$_{gen}$ needs a malloc() process for creating tasks while Task$_c$ calls printf() to send the results to the I/O. Since the GNU C compiler for the ARM processor, which is currently used in the simulation platforms, internally issues malloc() when calling printf(), the same spinlock has to be used in Task$_{gen}$ and Task$_c$ to avoid executing multiple malloc functions at the same time. Note that it is not always the case if using other compilers. However, this does not influence the generality of the analysis performed below. For example, it can be imagined that Task$_{gen}$ could also print information to the I/O or Task$_c$ could do memory allocations as well.

In general, the execution time of creating a task in Task$_{gen}$ is much shorter than that of Task$_c$. However, since Task$_c$ are executed by multiple processors, it can happen that the task generation speed of Task$_{gen}$ cannot catch up with the speed of executing Task$_c$ to consume data. As a consequence, some CPEs in the system might often or even always stay at an idle state, waiting for Task$_c$ and wasting cycles. This becomes especially critical if Task$_{gen}$ is frequently blocked by Task$_c$ because of the competition for the spinlock. On the other hand, if currently enough Task$_c$ have been generated in the system, waiting for being executed, Task$_{gen}$ should not block the execution of Task$_c$ when competing for the spinlock. This can allow Task$_c$ to be more smoothly completed. Therefore, a balance between Task$_{gen}$ and Task$_c$ is desired when they are executed.

The traditional way of assigning different priorities to the tasks does not help much in this situation. Task priorities only determine the task execution order, but not the spinlock acquisition order. If the spinlock control is done randomly, a high-priority task can still be blocked by a low-priority task, simply because the high-priority task loses the competition for the spinlock. Therefore, the spinlock reservation approach is applied here to handle the spinlock competition between Task$_{gen}$ and Task$_c$ in a better way.

As explained above, in this application a balance between Task$_{gen}$ and Task$_c$ is desired to maximize the system performance. To achieve this, the execution of the reservation algorithm should be dynamically adapted to the system status. In the implementation, spinlock reservations are used in both Task$_{gen}$ and Task$_c$, but with different spinlock control information. Assume without loss of generality that two values $A$ and $B$ are assigned to the control information of Task$_{gen}$ and Task$_c$, respectively, and $A$ is unequal to $B$. In addition, a threshold is set in the reservation algorithm executed on the OSIP core, which is meant to help judging whether enough Task$_c$ have been prepared in the system. A prepared task is a Task$_c$ which has been generated but not yet been executed. Since OSIP is aware of the system status, it compares the number of the prepared tasks with the threshold value, when it executes the reservation algorithm. If the number is lower than the threshold, more tasks need to be generated in the system to meet the requirement of data consumption. In this situation, the reservation algorithm reserves the spinlock for the task with the control value $A$, namely for Task$_{gen}$. If the system has a higher number of prepared tasks than
the threshold, the task generation speed is presently not the limiting factor of the system performance. So the reservation is made for the tasks with the control value $B$, namely for $Tasks_c$, in order to boost their execution speed. In this case study, the threshold value is double of the number of CPEs. The pseudo code of this algorithm is given in Algorithm 5.1.

Algorithm 5.1 Spinlock Reservation Algorithm for Synthetic Application

1: procedure SpinlockReservation(ReservationFlags, ReservationInfo, ListSize)
2:    declare ReservedFlags ← 0
3:    for PE_ID ← 0, N_CPE do
4:      if (ReservationFlags & (1 ≪ PE_ID)) ≠ 0 then
5:        if ListSize < ThresholdValue then
6:          if ReservationInfo(PE_ID) = Value_A then
7:            ReservedFlags ← ReservedFlags | (1 ≪ PE_ID)
8:        end if
9:      else
10:        if ReservationInfo(PE_ID) = Value_B then
11:            ReservedFlags ← ReservedFlags | (1 ≪ PE_ID)
12:        end if
13:      end if
14:    end for
15: return ReservedFlags
16: end procedure

Since in the spinlock reservation mechanism, OSIP collects the requests from multiple PEs in order to make the best decision for them, the mechanism exhibits a statistical behavior. Therefore, the categorization of the three scenarios (best case scenario, average case scenario and worst case scenario) in Section 4.2.1.1 is not suitable for the case study here. Instead, the task list size is fixed to maximum, which is 165, and analysis is made for small tasks ($N_{ACCESS} = 1$), medium tasks ($N_{ACCESS} = 6$) and large tasks ($N_{ACCESS} = 15$).

The reduction of the execution time ($\Delta t_{exec}$) of the application with the spinlock reservation mechanism is given in percentage in Figure 5.6. It compares the original execution time ($t_{exec,orig}$) with the optimized execution time by the reservation ($t_{exec, opt}$) in different system configurations in terms of the number of CPEs and Task$_c$ sizes.

Two trends can be observed by comparing the execution time in the upper part of the figure. First, the applications with small tasks benefit more from the spinlock reservation mechanism than those with large tasks. Second, large systems benefit more than small systems. In both cases, either with small tasks or in large systems, the competition for the spinlock happens frequently. This increases the opportunity for the spinlock reservation mechanism to take actions to improve the spinlock control. In contrast, in systems with a small number of processors and relatively large tasks, the performance improvement is rather limited. There are also cases, in which the execution time becomes longer, e.g., in the 3-CPE-system with a large task size. In
5.4. Case Studies

![Graph showing performance improvement based on spinlock reservation.](image)

Figure 5.6: Synthetic application: Performance improvement based on spinlock reservation. $\Delta t_{\text{exec}}(\%)$ is the reduction of the execution time, which is calculated with $\Delta t_{\text{exec}}(\%) = \frac{t_{\text{exec,orig}} - t_{\text{exec,opt}}}{t_{\text{exec,orig}}}$. $\Delta t_{\text{spin}}(\text{Task}_{\text{gen}})(\%)$ is the reduction of the spin time of Task$_{\text{gen}}$ in percentage, which is calculated with $\Delta t_{\text{spin}}(\text{Task}_{\text{gen}})(\%) = \frac{t_{\text{spin,orig}(\text{Task}_{\text{gen}})}}{t_{\text{exec,orig}(\text{Task}_{\text{gen}})}} - \frac{t_{\text{spin,opt}(\text{Task}_{\text{gen}})}}{t_{\text{exec,opt}(\text{Task}_{\text{gen}})}}$.

In this system configuration, the matching of the execution patterns of Task$_{\text{gen}}$ and Task$_{c}$ in the original system is already close to optimum. Spinlock competitions between them take place very seldom, and the competition durations are also only very short, if they take place. Applying the spinlock reservation mechanism into this system introduces the overhead caused by the additional APIs and the OSIP workload, but does not have benefits in reducing the competitions. In fact, in this case the additional overhead results in a slight change of the task execution patterns, which introduces even more spinlock competitions into the system.

The lower part of Figure 5.6 shows the reduction of the spin time of Task$_{\text{gen}}$ in percentage ($\Delta t_{\text{spin}}(\text{Task}_{\text{gen}})(\%)$) when the spinlock reservation mechanism is applied. For systems with a large number of consumer processors or small tasks, in which Task$_{\text{gen}}$ more often needs the spinlock for the task generation, the spinlock reservation...
mechanism can greatly reduce the spin time of Task\textsubscript{gen}. In contrast, for small systems with relatively large tasks, in which Task\textsubscript{gen} does not need the spinlock frequently, the adaptive reservation algorithm tends to increase the spin time of Task\textsubscript{gen} for the benefit of executing Tasks\textsubscript{c}.

5.4.2 H.264

The spinlock reservation mechanism also applies to H.264. As introduced in Section 4.3.2, the IQT, IDCT and intra-frame prediction tasks can be highly parallelized in this application. The IQT and IDCT can be processed for each MB independently, while the prediction for an MB has dependency both on the IQT/IDCT for the same MB and on the predictions for the neighboring MBs on the left, top and top left of the current one. The task synchronization between the IQT/IDCT and prediction for an MB, and especially between the predictions for the neighboring MBs, has a big impact on the frame rate of the video decoding. If the task synchronization is not efficiently resolved, the frame rate could be largely reduced.

However, in the original implementation the task synchronization is often delayed by spinlock competitions. Similar to the synthetic application, the memory allocation for creating new IQT/IDCT and prediction tasks is often blocked by the I/O accesses for outputting MBs. To change this situation, spinlock reservations are made for the creation of these tasks. Furthermore, the reservation for creating a prediction task is assigned with a higher priority than for creating an IQT/IDCT task, because resolving the synchronization between the predictions is more critical for improving the system performance.

The reservation algorithm is given in Algorithm 5.2. First, the number of the PEs which require a spinlock reservation is counted and the IDs of these PEs are recorded (line 4 – 9). Then, a search is made among these PEs to find the ones with the highest reservation priorities (line 11 – 20), which have been given as the spinlock control information when reservations are requested by the PEs. Finally, the reservation result is returned.

The performance improvement with the spinlock reservation mechanism is highlighted in the upper part of Figure 5.7 by comparing the video frame rates. A significant increment of the frame rate can be observed starting from a 6-processor-system, which clearly shows the efficiency of the proposed approach. For large systems, a speedup of up to $1.2 \times$ is achieved. The low improvement for small systems with less processors results from the low competition for the spinlock, in which the improvement opportunities are rather limited.

Similarly as done for the synthetic application, the spin time for the lock is analyzed in order to have a closer look at the reason for the speedup. Shown in the lower part of Figure 5.7, the average spin time of trying to acquire the lock in the application is largely reduced by-reserving the spinlock, which effectively improves the utilization of the processors. As a side effect, the communication traffic is also reduced.
Algorithm 5.2 Spinlock Reservation Algorithm for H.264

1: procedure SpinlockReservation(ReservationFlags, ReservationInfo)
2:     declare ReservedFlags, CandIdx[N_PE], TempWinner
3:     declare Count $\leftarrow 0$
4:     for $PE_{ID} \leftarrow 0, N_{PE} - 1$ do
5:         if (ReservationFlags & (1 $\ll$ PE_{ID})) $\neq 0$ then
6:             CandIdx[Count] $\leftarrow$ PE_{ID}
7:             Count $\leftarrow$ Count + 1
8:         end if
9:     end for
10:     ReservedFlags $\leftarrow$ CandIdx[0]
11:     TempWinner $\leftarrow$ CandIdx[0]
12:     for $i \leftarrow 1, Count - 1$ do
13:         if ReservationInfo[TempWinner] < ReservationInfo[CandIdx[i]] then
14:             ReservedFlags $\leftarrow$ (1 $\ll$ CandIdx[i])
15:             TempWinner $\leftarrow$ CandIdx[i]
16:         else if ReservationInfo[TempWinner] = ReservationInfo[CandIdx[i]] then
17:             ReservedFlags $\leftarrow$ ReservedFlags | (1 $\ll$ CandIdx[i])
18:         end if
19:     end for
20:     return ReservedFlags
21: end procedure

5.5 Joint Impact of OSIP Efficiency and Flexibility

The case studies above show the efficiency of the proposed spinlock control mechanism in resolving spinlock competitions. The programmability of OSIP is the key to support this mechanism, which enables a smart spinlock control based on application knowledge.

However, the efficiency of OSIP in task scheduling and mapping also indirectly has an impact on the efficiency of this mechanism. As introduced in Section 5.3.3, the OSIP core can only be triggered when it is idle. So the control mechanism actually utilizes the idle time of the OSIP core to execute the reservation algorithm. The idle time is the time interval between OSIP command executions. Too short idle time can have two negative impacts. The first one is that the OSIP core might probably only be seldom triggered for executing spinlock reservation, which reduces the effectiveness of the proposed spinlock control mechanism. The other one is that, when the reservation algorithm becomes active during the idle time, its execution can often prevent normal commands from entering the OSIP core since it sets the OSIP status to busy. So this postpones the normal task management operations, thereby reducing the system performance. Therefore, it is important to analyze the joint impact of the efficiency and flexibility of OSIP on the proposed mechanism. For this purpose, a comparison is carried out between OSIP-based, UT-OSIP-based and LT-OSIP-based systems.
5.5.1 Synthetic Application

An overview of the performance analysis of different system configurations for the synthetic application is given in Figure 5.8, considering the three different OSIP implementations, task sizes and whether or not using the spinlock reservation mechanism. In addition, the workload of the different OSIPs is presented in the figure by the OSIP busy time ($t_{\text{OSIP-busy}}$).

Note that due to the inflexibility of UT-OSIP, which is a hypothetical ASIC, it is impossible for it to adapt the user-defined control information and the reservation algorithm for different applications. Therefore, it is assumed that the proposed spinlock control cannot be applied to the UT-OSIP-based systems.

- **OSIP vs. UT-OSIP:** Naturally, the extremely high scheduling and mapping efficiency of UT-OSIP defines an upper bound of the system performance. This is shown by the shortest execution time of the application in the UT-OSIP-based systems for all system configurations without spinlock reservation. Compared with the UT-OSIP-based systems, the original OSIP-based systems have slightly
worse performance. However, as discussed in Section 4.6.2, even UT-OSIP can not improve the system performance as much as desired by simply adding more processors into the system. The figure shows that for all task sizes, the reduction of the execution time is very limited beyond a certain number of processors in the UT-OSIP-based systems. As explained in Section 5.4.1, in these large systems, the system performance strongly depends on the task generation speed, and the spinlock competition is one major factor of limiting this speed. The same effect can also be observed in the OSIP-based systems.

By applying the spinlock reservation mechanism in OSIP, a large number of undesired spinlock competitions are effectively reduced. This largely makes up for the slightly lower efficiency of OSIP (compared to UT-OSIP). In fact, for the
configurations with intensive spinlock competitions, the improved OSIP-based systems perform slightly better than the UT-OSIP-based systems, as shown in the figure. The obvious increment of the busy time of OSIP with the reservation mechanism in these configurations shows exactly that OSIP is actively involved in handling spinlock competitions. To highlight the efficiency of OSIP, even with the additional spinlock control overhead, OSIP is still not much loaded.

- **OSIP vs. LT-OSIP:** Compared to the high performance improvement in the OSIP-based systems for most system configurations, the improvement caused by the spinlock reservation mechanism can only be observed in a few configurations in the LT-OSIP-based systems. One obvious improvement can be found in the 3-CPE-system with the medium task size. Here LT-OSIP is still able to manage the task scheduling and mapping, and has enough idle time for the spinlock control, shown by the increased busy time of LT-OSIP. However, in most of the other configurations, the system performance has very low improvement, often even becomes slightly worse. The high percentage of the busy time of LT-OSIP, given in Figure 5.8, largely prevents the spinlock reservation algorithm from being executed by LT-OSIP. Therefore, in these configurations the proposed mechanism can not be actively in operation because of the low scheduling and mapping ability of LT-OSIP. On the contrary, it wastes the execution time caused by the additional APIs.

### 5.5.2 H.264

In this application, similar observations can be made in Figure 5.9, as for the synthetic application, considering the impact of the different OSIP implementations on the proposed spinlock reservation mechanism.

The spinlock reservation can improve the frame rate in the OSIP-based systems with a large number of PEs greatly, because the synchronization is better solved. It is important to note now the OSIP-based systems perform clearly better than the UT-OSIP-based systems, which highlights the advantage of using an ASIP over an ASIC, if flexibility is needed.

In contrast, with LT-OSIP, the same mechanism works for small systems, but fails in large systems. As shown in the figure, till the 6-PE-system the frame rate is effectively improved by using LT-OSIP to resolve the spinlock competition for better synchronization. From the 6-PE-system on, this mechanism starts to lose its effectiveness. Even worse, starting from the 8-PE-system, the frame rate becomes even lower by applying it. Again, the low scheduling and mapping ability of LT-OSIP is the cause. Compared to the other OSIP-based systems, the generally low frame rate in the LT-OSIP-based systems already implies that the task synchronizations in H.264 are not resolved efficiently here. However, in the small systems, the impact of the inefficient synchronizations on the system performance is higher than that of LT-OSIP. Therefore, LT-OSIP can still helps in improving the performance by running the spinlock reservation mechanism. In the large systems, LT-OSIP is the bottleneck. Adding
more workload onto it makes the synchronization time even longer, which worsens the system performance further.

If comparing the frame rate improvement in small systems with OSIP and LT-OSIP, one question could be raised: Why can LT-OSIP make improvement in small systems while OSIP cannot? The answer is that different task managers have different scheduling and mapping time, which influences the system behavior in a different way and in this case results in different task synchronization patterns. While in the OSIP-based systems synchronizations are still not critical in small systems, in the LT-OSIP-systems synchronizations already cause performance issues due to the inefficiency of the task manager. Therefore, when designing the software (e.g., task partitioning) for a system with a central manager, the scheduling and mapping time should also be taken into consideration to avoid large deviation between the desired task execution pattern during the design phase and the real task execution pattern.
Based on the comparison between the systems using different OSIP implementations, it can be concluded that the flexibility of a system can help increasing the system performance significantly by using application knowledge. In this work, the application knowledge is applied to spinlock control. Large improvements can be observed in both OSIP-based and LT-OSIP-based systems. Especially the OSIP-based systems, supported by the application knowledge, perform even better than the systems using an extremely fast ASIC task manager in many cases.

On the other hand, considering additional application knowledge in the system introduces additional workload, which can also lead to worse system performance. This can be confirmed by the reduced performance of the LT-OSIP-based systems in most of the analyzed cases. Compared to LT-OSIP-based systems, OSIP-based systems, however, can achieve much more stable performance improvement. This shows that the efficiency of OSIP enables more advanced system control techniques. More in general, the efficiency is essential for guaranteeing an effective utilization of the system flexibility.

5.6 Summary

In this chapter, a centralized spinlock control mechanism is introduced, in which spinlocks can be reserved for the tasks to prevent random spinlock acquisitions. The reservation information is user-defined, based on application knowledge. A simple spinlock reservation API pair enables easy integration of the application knowledge into the spinlock control flow. The OSIP processor, originally designed for task scheduling and mapping, is used additionally to support this mechanism for making application-aware spinlock decisions. The performance improvement in the presented case studies shows the efficiency of this mechanism in spinlock-intensive applications. Furthermore, an extensive analysis of this mechanism considering different OSIP implementations is made. The analysis results highlight the efficiency and the flexibility of OSIP, thanks to the ASIP concept, which enables successful employment of this mechanism in practice.

Although the experiments in this chapter are done in the OSIP-based systems, the basic concept of spinlock reservation can be generalized to systems using a central programmable controller. Certainly, the efficiency of the controller should be considered when applying this mechanism.

5.7 Discussion

The spinlock reservation mechanism shows its efficiency in improving the spinlock control in general. However, there are still several potential improvement possibilities, which are discussed in the following.
5.7. Discussion

5.7.1 Tool Support for Spinlock Reservation

The reservation information, also including whether and where to make reservations, plays a key role in this approach. In the two case studies given in this chapter, the information is derived by analyzing the applications manually. This requires much tedious work, even though both applications have relatively regular task execution and spinlock competition patterns. And in these two case studies, both spinlock control algorithms are simple, but still demonstrate high efficiency in improving the system performance. This is possible because they focus on solving the most critical spinlock competitions that have a large impact on the task parallelism degree. However, there are certainly applications, in which more complicated and optimized spinlock acquisition orders are needed to maximize task parallelism. This requires complex control algorithms. However, too complex algorithms could introduce high additional workload to OSIP, which can potentially impair the normal task scheduling and mapping operations of OSIP. Therefore, a balance needs to be found between the possible gain in task parallelism achieved by the complex algorithms and the additional workload caused by them. All these naturally call for tool support such as MAPS compiler [30], which on the one hand can provide a more systematic analysis, on the other hand can considerately accelerate the complete design flow of the proposed approach.

5.7.2 Scalability Problem

As a central mechanism, the proposed spinlock control approach can inherently be faced with the scalability problem. If the system size becomes very large, the central controller would likely become the system bottleneck. This is, however, not the case in the presented case studies if OSIP is used as the controller, which is indicated by its low busy time shown in Figure 5.8 and Figure 5.9. In fact, in systems based on a central bus, the scalability problem at the central controller is not necessarily critical. If an efficient enough central controller like OSIP is employed, the system bottleneck would most likely be shifted to the bus communication due to heavy traffic in large systems, as shown in Chapter 4, in which a detailed joint analysis of the communication architecture and the OSIP efficiency is made.

However, in systems with high communication bandwidths, e.g., using NoCs, the scalability problem of the proposed central mechanism can become serious. In these systems, a combination of this mechanism and other distributed mechanisms like the ones presented in [212] and [38] can be considered, which are complementary to each other. OSIP, being the system task manager, can be used to perform global task-level spinlock control, focusing on maximizing task parallelism. In contrast, the control jobs at more fine-grained granularity levels, such as word-level read/write locks, can be done in a distributed way, thereby reducing traffic and the OSIP workload. The simple system integration of OSIP makes it possible to combine both mechanisms, and the user only needs to define separate address-mappings for the spinlocks to distinguish the requests to the central controller and local control units.
5.7.3 Nested Spinlocks

In the current experiments, nested spinlock reservations are not supported. This means, a later reservation of a PE always overwrites its previous reservation. To support nested reservations, buffers need to be introduced in the spinlock control unit to allocate multiple reservation requests from the same PE.
Chapter 6

OSIP Integration in NoC-based Systems

In Chapter 4 and Chapter 5, the advantages of using OSIP for handling task scheduling and mapping have been presented from the efficiency and flexibility perspective. In these two chapters, the systems are built on a shared bus with a shared memory. As this kind of system architectures is centralized, a central task manager such as OSIP fits well into such systems. Its integration into the system can be easily done as a standard memory-mapped peripheral.

However, in systems with a large number of PEs towards many-core systems, the shared bus based communication architectures are not suitable anymore. As introduced in Section 2.4, they inherently suffer from the scalability problem caused by long bus wires in large systems. For these systems, NoCs are a preferred communication architecture paradigm, in which the long wires are segmented by using packet-based communications through routers. In comparison to bus-based systems, NoC-based systems have a distributed architecture. As OSIP is a central manager, some general questions would be: Is it still possible to integrate the central OSIP into distributed NoC-based systems? How to integrate OSIP, if it is possible? Is OSIP still efficient in such systems? In this chapter, answers to these questions are given.

This chapter is organized as follows. First, two main problems of integrating OSIP into NoC-based systems are presented. Following this, the basic idea of addressing the integration problems is illustrated. Afterwards, a complex case study is provided. In this case study, both the NoC architecture used in this work and the target application are introduced. The application is taken from the wireless communication domain, which is a $2 \times 2$ Multiple-Input Multiple-Output (MIMO) Orthogonal Frequency-Division Multiplexing (OFDM) digital receiver. Then, the system implementation at a high abstraction level using VPUs is introduced and the concrete integration of OSIP into the system with the NoC is described in detail. Also, preliminary results about the impact of the NoC on the system performance are reported. Based on the preliminary results, optimizations are proposed for the communication and in-depth analyses are performed. Finally, a summary is made and further possible extensions for OSIP integration into NoC-based systems are discussed.\(^1\)

\(^1\)Portions of this chapter have been published by the author in [215] in Advancing Embedded Systems and Real-Time Communications with Emerging Technologies, edited by Seppo Virtanen. Copyright 2014, IGI Global, www.igi-global.com. Posted by permission of the publisher.
6.1 OSIP Integration Problems in NoC-based Systems

As introduced in Section 3.2.3, OSIP is featured with two interfaces: a register interface (REG_IF) and an interrupt interface (IRQ_IF) for the hardware integration. These two interfaces serve as the basic physical communication channels to enable the communication between OSIP and PEs: the register interface for exchanging the task management information and special system status and the interrupt interface for triggering PEs to execute tasks through direct interrupt lines. In bus-based systems, they enable easy integration of OSIP as an off-the-shelf IP component. However, in large NoC-based systems, even though it is theoretically possible to integrate OSIP in the same way, the integration has two major problems at both interfaces in practice:

- **Polling register interface:** In OSIP-based systems, the communication between PEs and OSIP heavily relies on the polling of the information in the register interface. Every time after a PE issues a command to OSIP, it polls the OSIP status from the interface for reading the results. Also, PEs poll for acquiring OSIP spinlocks in order to obtain the access to shared resources or OSIP itself. However, polling the register interface through the network introduces high communication overhead, as each polling has to go through several routers, thereby reduces system efficiency.

- **Interrupt lines:** Direct interrupt lines from OSIP to PEs in large systems indicate very long wires crossing the whole chip, which are typically undesired in modern chips. They introduce signal integrity problems and potentially reduce the system clock frequency. In general, using long wires has conflict with the NoC concept.

Considering these two problems, adaptations need to be made for OSIP before integrating it into NoC-based systems, such that the communication with both hardware interfaces of OSIP does not raise performance and integration issues.

6.2 Basic Concept of OSIP Integration in NoC-based Systems

In this section, the basic idea of tackling the above-mentioned OSIP integration problems in NoC-based systems is presented.

Since in large NoC-based systems direct remote communications between PEs and OSIP through both OSIP interfaces is not realistic, local communications with OSIP need to be created. To achieve this, small proxies can be employed for the PEs. These proxies are physically closely located to OSIP and act as a bridge between the PEs and OSIP. Concretely, on the OSIP side these proxies implement the original OSIP communication primitives, e.g., polling OSIP’s register interface and catching interrupt signals. On the NoC side, the proxies communicate with the PEs using messages based on the NoC protocol. The messages contain the information needed for the
6.2. Basic Concept of OSIP Integration in NoC-based Systems

Implementation of the OSIP communication primitives, such as requests to OSIP or responses from OSIP. In this way, the proxies separate the direct communications between the PEs and OSIP into: a) remote communications between the PEs and the proxies over the NoC; and b) local communications between the proxies and OSIP through a local communication architecture such as buses and interrupt lines.

![Figure 6.1: OSIP subsystem for NoC integration](image)

The proxies and OSIP together with a local communication architecture and other peripherals build up a so-called OSIP subsystem. Instead of only using OSIP in bus-based systems, the complete OSIP subsystem is regarded as the task manager in NoC-based systems. Figure 6.1 shows an exemplary OSIP subsystem. Without being shown in the figure, an adapter is typically needed in the OSIP subsystem to access

![Figure 6.2: An exemplary NoC-based system with integrated OSIP](image)
the NoC and to distribute the information from NoC to different proxies as a kind of address decoder. An exemplary NoC-based system containing OSIP is illustrated in Figure 6.2, which consists of several PE subsystems, memory subsystems and the OSIP subsystem.

In the OSIP subsystem of Figure 6.1, each PE has a dedicated proxy. Suppose that a PE wants to request a new task from OSIP. It sends a message to its proxy. When receiving the message, the proxy generates an OSIP command and tries to access OSIP by polling its register interface. Assuming that currently a new task is available for the PE, OSIP generates an interrupt to the proxy after handling the command. Upon receiving the interrupt, the proxy fetches the task information from OSIP and sends a response message to the PE. So, polling and receiving interrupts only take place locally in the OSIP subsystem and the interrupt information to the PE is implicitly contained in the response message. Furthermore, using a proxy for polling also enables the PE to perform other independent tasks at the same time, hence improves the PE utilization.

Different implementations can be made for the proxies. They can be ASIC implementations using finite state machines (FSMs), or programmable processors like RISCs or even ASIPs. Which implementations to choose largely depends on the required flexibility of the communication with OSIP. In other words, it depends on the regularity of the communication patterns with OSIP.

It is also important to note that it is not necessary to always create a dedicated proxy for each PE. Message-based communications between PEs and proxies enable sharing a proxy among multiple PEs, which has the advantage of less area overhead, if the proxy is able to provide sufficient support for the PEs.

6.3 Case Study

In order to validate the concept introduced in the previous section, a case study is designed. For the case study, a $2 \times 2$ MIMO-OFDM digital receiver from the wireless communication domain is selected. Applications in this domain can typically be presented as DFGs, which are suitable for distributed memory architectures in NoC-based systems. For the NoC implementation, the mesh-like NoC used in [167] and [168] is selected. In the following, both the NoC structure and the application are introduced.

6.3.1 NoC Structure

The NoC structure used in this case study has a 2D mesh-like topology, shown in Figure 6.3. It has a dimension of $5 \times 4$, excluding the nodes at the four corners. The identifier of the nodes consists of the X and Y coordinates and is denoted as $(x, y)$ in the figure. Each node includes a router and a network interface. Virtual channels implemented as FIFO buffers are employed in the routers. For the routing, the wormhole routing scheme is applied.
6.3.1.1 NoC Packets

In this NoC, two types of NoC packets are defined: multi-flit packets and single-flit packets. In a multi-flit packet, it always starts with a header flit (HD) and ends with a tail flit (TL). Between the HD flit and the TL flit, there can be several data flits (DT) or none. In contrast, a single-flit packet contains only one flit, which is both the header flit and the data and tail flit (HDT). The structures of both packets types and the flit formats are given in Figure 6.4.

![Figure 6.3: 2D mesh-like NoC topology](image)

![Figure 6.4: Structure of multi-flit packets and single-flit packets](image)
Each flit consists of three fields: flit type (HD/DT/TL/HDT), virtual channel ID (VCID) and flit body. VCID specifies which virtual channel of the next router is required to buffer the flit. The flit body contains the payload data of the packet. In the HD/HDT flit, it also contains the routing header, which provides the necessary information needed for routing. The information includes:

- **src**: Coordinates of the source node, from which the packet is sent.
- **dst**: Coordinates of the destination node, to which the packet is sent.
- **prio**: Priority of the packet, which is used for the arbitration when competing with other packets for virtual channels and router output ports.
- **size**: Payload size of the packet, which is given in bytes.

### 6.3.1.2 Router

The router architecture of this NoC is scalable regarding the following parameters: the number of router input and output ports, the number of virtual channels, the size of virtual channels and the data link width between routers. The block diagram of the router architecture is shown in Figure 6.5, in which the following five main functional blocks can be identified:

- **Receiving (RCV)**: In this functional block, the flits from the neighboring routers are received. The VCID information of the flits is extracted and used for selecting the corresponding virtual channels for the flits. There are as many RCV blocks as the router ports, and within one clock cycle, one flit can be received by one port at most.

- **Routing (RUT)**: In this block, the routing direction of the flits to the next router is determined. A routing table is defined in each router, following a modified XY routing algorithm, because the normal XY routing algorithm cannot be directly applied to this NoC due to the missing nodes at the four corners. In this modified algorithm, the routers first forward the packet in X-direction, until the X-coordinate of the destination node is reached or the packet cannot be forwarded any further if the boundary of the NoC in X-direction is reached. Then the packet is forwarded in Y-direction until the Y-coordinate of the node is reached. If the node is exactly the destination node, the routing is finished, which is the same as in the normal XY routing. Otherwise, the routing continues in X-direction again until the destination node is reached. One example for the latter case is the routing from node \((1, 3)\) to node \((4, 2)\), which has the routing path \((1, 3) \rightarrow (2, 3) \rightarrow (3, 3) \rightarrow (3, 2) \rightarrow (4, 2)\). Note this modified XY routing is also deadlock-free as the normal XY routing, because no cycles can be created during the routing.

The routing table is hard-wired and configurable at the design time. When the RUT block receives a flit from the RCV block, it first examines the flit type. In
case of an HD/HDT flit, the coordinates of the destination nodes are extracted from the \textit{dst} field of the flit and used as the input to the routing table. The outcome is the output port of the router, to which the flit should be sent. If the flit is a DT/TL flit, it simply takes the previous routing result for the HD flit of the same packet.

- **Virtual Channel Allocation (VCA):** After the routing direction (i.e., the output port of the router) for a flit is determined, a virtual channel at the corresponding input port of the next router needs to be allocated for the flit. A virtual channel at the current router and at the next router are further referred to as an input and output virtual channel, respectively. As introduced in Section 2.4.2, virtual channels are common means to improve the NoC throughput.

For the allocation of an output virtual channel, competitions can happen between the flits in the current router, which are not only from different input ports, but also from different input virtual channels at each port. To reduce the hardware complexity, the arbitrations are performed in a hierarchical way consisting of two levels.

At the first level, local arbitrations are performed to find an available output virtual channel for each input virtual channel individually. The availability of an output virtual channel is indicated by a so-called \textit{credit} signal, showing whether the buffer of the virtual channel is full. This level of arbitrations is static. The output virtual channels with smaller IDs, if they are available, have higher priorities than the ones with higher IDs.

At the second level, a global arbitration across all input virtual channels is performed. This arbitration is dynamic, based on the \textit{Fair-Among-Equal} scheme. Here the priorities in the \textit{prio} field of the HD/HDT flits from different input virtual channels are compared if they compete for the same output virtual channel. The input virtual channel with the highest flit priority wins. In case that multiple input virtual channels have the same highest priority, a round-robin arbitration is further performed for fairness.

To support the wormhole routing scheme, all flits of a packet always use the same virtual channel of a router, and the flits of other packets are not allowed to be inserted in between. This means that if an output virtual channel is currently allocated for a packet, it is in general not available for the flits of other packets, until the complete packet has been received by the virtual channel.

As the outcome of this functional block, at most one output virtual channel per output port can be allocated. However, it is possible that multiple input virtual channels at one input port can allocate an output channel each, while the input channels of some other input ports allocate none.

- **Switch Allocation (SWA):** With an allocated output virtual channel, a flit is ready for being transferred to the next router. As there are more virtual channels than physical channels between neighboring routers, competitions can also
happen among the flits from different input virtual channels for allocating the physical channels. For fairness of transferring packets from different input ports, at most one flit from each input port can obtain the access to a physical channel within one clock cycle. Therefore, another two-level arbitration is applied, using the same dynamic Fair-Among-Equal scheme described above for both levels. At the first level, arbitrations are performed among the input virtual channels at each input port to find a winner input channel (i.e., a winner flit) of the port. Then, at the second level, the winner flits from all input ports are compared to determine which ones can finally obtain the access to the physical channels.

Note that different from the allocation of a virtual channel, which must be continuously allocated for a complete packet, the allocation of a physical channel does not have this limitation. The flits of different packets can be transferred to the next router in an interleaved way. This effectively improves the utilization of the physical channels and prevents the situation that a packet with a higher priority is blocked by a packet with a lower priority for a long time.

- **Crossbar Transfer (XT):** This is the last functional block of the router. The flits which have obtained the access to the physical channels are finally transferred to the next routers through a crossbar.

In the current implementation, pipeline registers are inserted between the RUT and VCA block to improve the maximum clock frequency. The pipeline can be stalled for each input virtual channel independently. This means that, if a flit in the pipeline register cannot be forwarded to the next router, only the output of the corresponding input virtual channel is stalled while the other input virtual channels are not affected. Certainly, more pipeline registers can be inserted between other functional blocks, but at the cost of more area overhead and higher latency.

### 6.3.1.3 Network Interface

The Network Interface (NI) provides an interface for the system IP components such as processors and memories to access the network. At the source side of a packet communication, it creates the flits of the packet. Depending on the payload size specified by the source node, NI might need to do zero-padding to complete the tail flit of the packet. At the destination side, it extracts the data payload out of the flits to the node.

The block diagram of the NI is shown in Figure 6.6. It has a register interface connected to the IP component and a router interface connected to the network. The register interface is accessed by the IP component in the memory-mapped way. Data are transferred to/from the NI_IN/NIL_OUT register, which are bound to an input and output FIFO, respectively. And the NI can be configured through the NI_CFG register, specifying how to generate interrupt and suspend signals to the IP component. For example, an interrupt can be generated to the IP for receiving new data, if a new packet arrives from the network. Similarly, a suspend signal can be generated to suspend the data transfer from the IP, if the input FIFO is full. In addition, the NI
Figure 6.5: Router block diagram
status such as the FIFO status can be read from a status register \( (\text{NI\_ST}) \). A simple control block is designed in the NI to control its functional behavior.

![NI block diagram](image)

**Figure 6.6:** NI block diagram

### 6.3.1.4 Communication Protocol

The network itself is only responsible for transferring data from one system node to another, using the routing information in the header flit of a packet. To enable the destination node to understand the purpose of a received packet, semantics needs to be defined for the packet, which is a part of its data payload. With the semantics, different packet types can be distinguished.

In this NoC, five types are defined, which are *read request packet* (PCK\_R), *read response packet* (PCK\_RR), *write request packet* (PCK\_W), *write response packet* (PCK\_WR) and *synchronization packet* (PCK\_S). The first byte of the payload data after the routing header field is used for encoding the packet types. After the identification of the packet type, more semantics information can be interpreted correspondingly. For example, a PCK\_R packet includes further read address and read data size at the destination node in its semantics.

The first four packet types are needed for a master/slave communication, typically between a processor and a memory, while the last one is for a master/master communication. For the master/slave communication, a master node sends a PCK\_R or a PCK\_W to a slave node, and the slave handles the received packet and sends back a PCK\_RR or a PCK\_WR as the response, respectively. To avoid out-of-order data transmission, a master node is not allowed to send another packet to the same
slave, before it receives the response packet to the previous packet of the same type. For the master/master communication, a master node is allowed to issue multiple requests without waiting for the responses.

More detailed information about the packet type definition and the communication protocol can be found in Appendix A.

6.3.1.5 Hardware Results: Area, Clock Frequency and Power Consumption

This 2D mesh-like NoC architecture is scalable with respect to the link data width and the number of the virtual channels as well as their depth. In this section, hardware results for a NoC configuration with 128-bit data width and four virtual channels with a depth of eight at each router port are presented. This NoC configuration is used in [167] and [168] as the base model for developing a high-level power estimation methodology for communication architectures.

The NoC is synthesized with Synopsys Design Compiler for a 65 nm standard cell library under typical conditions (supply voltage 1.0 V, temperature 25°C), then further placed and routed using Cadence SoC Encounter. During the Place & Route, black boxes are used as the “place holders” for the system IP components including eight processors and eight memories. In total, an area of 2.56 MGE and a maximum clock frequency of 340 MHz are reported for the NoC.

For estimating the power consumption of the NoC, three scenarios to mimic the traffics on the network are used:

- **Scenario A**: Processors access two memories in the center of the network heavily, while accessing the other six memories at the edges/corners lightly.

- **Scenario B**: Each processor communicates with a dedicated slave at the opposite side of the network.

- **Scenario C**: The traffic pattern of this scenario is extracted from a real wireless communication application.

The power estimation is performed using Synopsys PrimeTime after Place & Route, and the clock frequency is set to 316 MHz. The power consumption results are listed in Table 6.1.

<table>
<thead>
<tr>
<th>Scenario</th>
<th>A (mW)</th>
<th>B (mW)</th>
<th>C (mW)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Power (mW)</td>
<td>108</td>
<td>220</td>
<td>73</td>
</tr>
</tbody>
</table>

In Figure 6.7, a detailed power profile is shown for Scenario C, in which the distribution of three main power consumption groups (registers, combinational logic and clock network) of the NoC is given. It is well noticeable, that the major part of the power consumption is contributed by the clock network due to a large number of registers used in the virtual channels.
6.3.2 Application – MIMO-OFDM Iterative Receiver

NoCs have big advantages over traditional buses in throughput and scalability. To make a reasonable study on the integration of OSIP into a NoC-based system, suitable applications are needed, which should be able to exploit the high bandwidth and the distributed architecture of the NoC. Therefore, the parallel implementation of the H.264 video decoding, which heavily relies on the communication over a shared memory, makes it ill-suited for NoC-based systems. Instead, another application from the wireless communication domain, namely a $2 \times 2$ MIMO-OFDM digital receiver is selected for this purpose. Applications in this domain are typically presented as DFGs, which fit well into distributed memory architectures, and normally come along with a large amount of communication data.

In this case study, a doubly iterative design with a similar specification as that for the 4G LTE communication standard [2] is applied to the receiver. In the following, the receiver structure is briefly described.

6.3.2.1 Structure of MIMO-OFDM Iterative Receiver

A typical block diagram of a $2 \times 2$ MIMO-OFDM doubly iterative receiver is given in Figure 6.8, which consists of six main building blocks: Radio Frequency (RF) frontend, Fast Fourier Transformation (FFT), channel estimation, MIMO preprocessing, MIMO demapping and channel decoding.

- **RF frontend**: At the receiver side, the RF frontend processes the received analog signals at antennas by using filters, Low-Noise Amplifiers (LNAs) and Analog-
to-Digital Converters (ADCs) to generate digital signals for further baseband processing.

- **FFT**: For an OFDM system, signals at different frequency spectra are sent in an orthogonal way to the receiver. Therefore, after the conversion into digital signals at the RF frontend, FFT is needed to transform these signals again from the time domain back to the frequency domain.

- **Channel estimation**: When the transmitter transfers data to the receiver, the data can be distorted by fading channels and co-channel interference. The channel estimation is used to estimate the channel characteristic to help restore the distorted transmitted signals at the receiver. It can be training-based or blind. In the training-based channel estimation, reference data such as preambles or pilots, which are known to the receiver, are sent by the transmitter in addition to the payload data to aid the estimation. In the blind channel estimation, no such training data are sent, and the estimation is only based on the signal statistics.

- **MIMO preprocessing**: MIMO processing is often divided into MIMO preprocessing and MIMO demapping. In the preprocessing, the estimated channel matrix is typically decomposed, considering the algorithmic performance and numeric stability. The decomposed matrices, partly multiplied by the output vectors from the FFT, are then further used in the MIMO demapping. Popular decomposition algorithms are QR Decomposition (QRD), Sorted QRD (SQRD) and Givens rotations.

- **MIMO demapping**: During the MIMO demapping, the constellation points of the signals are determined and demapped into binary bits. There are both algorithms based on linear approaches such as Zero Forcing (ZF), Minimum Mean Square Error (MMSE) and algorithms based on non-linear approaches such as Sphere-Decoding (SD).

- **Channel decoding**: The data sent from the transmitter contain redundant information for making error correction possible at the receiver side. The channel
decoding removes the redundant information from the demapped bits to extract the original information bits. Often, iterative decoding such as Turbo Decoding (TD) or Low-Density Parity-Check (LDPC) is applied to improve the Bit-Error-Rate (BER). In a doubly iterative receiver, outer iterations are further performed, which feed the extrinsic information from the decoder back to the demapper in order to achieve more algorithmic performance gain.

In this case study, only the digital part of the receiver is considered, which is marked by the dashed frame in Figure 6.8. The application that runs on the receiver is at the downlink side. It is similar to the Frequency Division Duplex (FDD) mode of the LTE standard, with respect to the radio frame structure, pilot locations and the way of data processing.

A LTE radio frame in the FDD mode has a duration of 10 ms, which contains 10 subframes, shown in Figure 6.9. Each subframe, corresponding to a Transmission Time Interval (TTI), is further divided into two slots. Each slot contains 7 or 6 OFDM symbols, depending on whether a normal Cyclic Prefix (CP) or an extended CP is used in the OFDM symbol. A normal CP is considered in this work, meaning 7 OFDM symbols per slot.

The data processing in LTE is based on so-called Resource Blocks (RBs), which consist of both time and frequency resource elements. When using a normal CP, a RB contains 12 subcarriers of the seven OFDM symbols in a slot. In addition to the actual data payload, RBs also contain pilots to aid the channel estimation. In different antenna systems, the pilot patterns are defined differently. Figure 6.10 gives the pilot patterns for a $2 \times 2$ MIMO-OFDM system, in which each RB has four pilots, located at the first and fifth OFDM symbol, respectively.

In addition, guard bands by means of null subcarriers are employed in the OFDM symbols to prevent significant leakages between adjacent frequency bands. This means that only a subset of the OFDM subcarriers are actually used for transmitting the data.

### 6.3.2.2 Definition of Timing Constraints

For the analysis, the timing constraints of the digital receiver are defined. The system targets the peak downlink data rate for a $2 \times 2$ MIMO system, which is $150 \text{ Mbit/s}$, as defined in the LTE standard. This is the throughput constraint. The most relevant system parameters for achieving this data rate are listed in Table 6.2.

In comparison to the throughput constraint that is defined in the LTE standard, the latency constraint of the digital receiver at the PHYsical layer (PHY) is not fixed. It depends on the signal processing time at the RF frontend and the Medium Access Control (MAC) layer, and the transmission time at PHY as well. Therefore, some assumptions are made to define the latency constraint.

The round trip duration of a Hybrid Automatic Repeat reQuest (H-ARQ) is used as the base for making the assumption. According to the LTE standard, the round trip must be finished within four TTIs, which correspond to 4 ms. It means that the
One radio frame, $T_1 = 10$ ms

Figure 6.9: Radio frame structure of LTE (FDD mode, normal CP)

Figure 6.10: Pilot pattern of $2 \times 2$ MIMO system
receiver has 4 ms to generate an acknowledgment (ACK/NACK) to the transmitter after the transmitted signal arrives. This time period at the receiver can be divided into five parts: RF receiving time, digital PHY receiving time, MAC processing time, digital PHY transmission time and RF transmission time. As the RF processing time is very short, which is about tens of nanoseconds, they can in general be ignored. Furthermore, as the algorithm complexity at the MAC layer and that for the digital PHY transmission are much lower than that for the digital PHY receiving, less processing time is required for the MAC layer and PHY transmission. Combining these considerations, it is assumed that 60% of the total 4 ms-latency are consumed by the digital receiver, 40% by the rest system components. So, the latency constraint for the digital receiver is set to 2.5 ms.

To achieve good algorithmic performance gain, four outer iterations between the MIMO demapper and the channel decoder are implemented in the system. Beyond four outer iterations, the performance gain improvement is only very limited, which is shown in [186].

### 6.4 System Implementation

As the main purpose of the case study is to analyze the OSIP integration into NoC-based systems, the most relevant information required for the analysis are the task execution time, task dependencies and the data communication. In this sense, a real implementation of the algorithmic kernels of the MIMO-OFDM receiver is not necessarily needed. Therefore, VPU’s are employed to model the implementation of these kernels with a dummy behavior for enabling fast system exploration. However, accurate timing annotations are made for executing the dummy behavior in the VPU’s. The data communication between the VPU’s is also modeled accurately with respect to the data amount and the data flow. The whole system is driven by a clock frequency of 300 MHz.
6.4. System Implementation

6.4.1 Data Flow Graph

The execution time information of the algorithmic kernels is obtained from the ASIC implementations in the literature for the VPU timing annotation. Instead of detailing the implementation of these kernels, the references are listed in Table 6.3 together with the execution time.

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Implemented Algorithm</th>
<th>Latency $^a$ (ns)</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>FFT</td>
<td>2048-point FFT</td>
<td>51690/OFDM symbol</td>
<td>[27] $^b$</td>
</tr>
<tr>
<td>Channel estimation</td>
<td>Matching Pursuit (MP)</td>
<td>133467/Slot/SISO channel</td>
<td>[117] $^c$</td>
</tr>
<tr>
<td>Preprocessing</td>
<td>Sorted QRD (SQRD)</td>
<td>27/Channel matrix</td>
<td>[88]</td>
</tr>
<tr>
<td></td>
<td>$Q^H Y$</td>
<td>13/Symbol vector</td>
<td>[133] $^d$</td>
</tr>
<tr>
<td>Demapping</td>
<td>Sphere Decoding (SD)</td>
<td>333/Symbol vector</td>
<td>[202] $^e$</td>
</tr>
<tr>
<td>Channel decoding</td>
<td>Turbo Decoding (TD)</td>
<td>37000/Codeword</td>
<td>[126] $^f$</td>
</tr>
</tbody>
</table>

$^a$ In this case study, the system clock frequency is 300 MHz, which is the maximal achievable clock frequency by the selected channel decoder used in [126] to meet the throughput constraint of 150 Mbit/s. The implementation in [126] is performed using a 65 nm low power CMOS technology. However, the implementation of other algorithms does not use the same technology. Therefore, a linear clock frequency scaling is applied for these implementations to estimate the latency at 300 MHz.

$^b$ This large size of FFT is needed to support the transmission bandwidth of 20 MHz.

$^c$ This latency is based on the implementation using 8 parallel multiplication and storage units (MSUs) and running 50 MP iterations.

$^d$ The latency is estimated by assuming that four clock cycles are needed to perform a matrix-vector multiplication in the architecture.

$^e$ Simulation results show that in average 10 visited nodes can lead to good algorithmic performance for detecting a symbol vector in a $2 \times 2$ MIMO system. In this architecture, at every cycle one node can be visited. In addition, 3 clock cycles are needed for the initialization, which result in total 13 clock cycles as a reasonable time for detecting one symbol vector.

$^f$ The latency is based on the execution of 6 iterations inside the Turbo Decoder.

After the implementation of the kernels is selected, the data flow of the digital part of the receiver can be extracted, which is shown as a Cyclo-static Synchronous Data Flow (CSDF) graph in Figure 6.11. Note that only the data in the 1200 occupied subcarriers at the output of the 2048-point FFT are further processed in the algorithmic kernels after the FFT.

6.4.2 VPU Assignment

Considering the throughput constraint together with the data flow, the minimum number of the VPUs required for the kernels can be derived. The definition of the VPUs as well as the required numbers are summarized in Table 6.4.
Figure 6.11: CSDF of $2 \times 2$ digital MIMO-OFDM receiver

Table 6.4: Definition of VPUs

<table>
<thead>
<tr>
<th>VPU Name</th>
<th>VPU\textsubscript{FFT}</th>
<th>VPU\textsubscript{ChEst}</th>
<th>VPU\textsubscript{PreProc}</th>
<th>VPU\textsubscript{SD_Cluster}</th>
<th>VPU\textsubscript{TD}</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>4</td>
</tr>
</tbody>
</table>

For the demapping, three sphere decoders are needed to meet the throughput constraint. However, as in a doubly iterative receiver, they operate on the same data set generated from the preprocessing unit and the channel decoder, they are typically tightly coupled. A simple hardware arbiter can be applied to distribute the data to different sphere decoders. Therefore, they can be considered together as an SD cluster and mapped onto one VPU. In this sense, the processing time of demapping a symbol vector can be approximately calculated as one third of the sphere decoding latency given in Table 6.3.

In VPU\textsubscript{FFT}, a traffic generator mimicking the RF frontend (represented as the actor \textit{Src} in Figure 6.11) is also included, in addition to the FFT task itself. This traffic generator creates an OFDM symbol to the FFT task every 71.4 $\mu$s through a FIFO channel, as defined in the LTE standard.

6.4.3 Node Mapping

After the VPUs are defined, they are mapped to the nodes of the NoC. An overview of the node mapping is given in Figure 6.12. There are three types of subsystems (S/Sys): VPU subsystems, memory subsystems and the OSIP subsystem. The OSIP subsystem is placed at the center of the NoC to reduce the communication latency with the VPUs, since all VPUs need to communicate with it. The memory nodes are placed closely to the VPU nodes, to which they have intensive data communications. For
example, Log-Likelihood Ratio (LLR) data and extrinsic information are exchanged frequently between demapping and decoding. Therefore, the memories for storing these data are placed in the middle of the SD_Cluster and TD nodes. For the same reason, the memory for storing the symbol vectors and channel information is placed in the middle of the other VPU nodes.

![Diagram](image)

**Figure 6.12:** Node assignment in the NoC-based system

### 6.4.4 VPU Subsystem

As each VPU is based on an ASIC implementation, it emulates the behavior of an ASIC. This means that the VPUs are unlikely to be able to communicate with the other nodes of the system directly without the implementation for the NoC protocol. Therefore, each VPU subsystem includes an additional local controller for this purpose. Figure 6.13 shows the basic structure of a VPU subsystem.

In the VPU subsystem, three components (a controller, the VPU and a local memory) are connected using a local shared bus. The controller is the master of the subsystem. It has the following functions:

- It implements the NoC communication protocol, sending and receiving the packets between the remote memory and the local memory.

- It communicates with the OSIP subsystem for requesting tasks, receiving task information and also sending a synchronization message to OSIP, after a task is finished. Since the VPUs are ASICs, the tasks themselves are already determined in VPUs. So, the information needed for the task execution mainly contains the source and destination address of the to be processed data at a remote memory as well as the data size.

- It activates the VPU to process the data.
Certainly, if the VPU is a programming processor, a local controller is not necessary, which can be replaced by the VPU. For the task execution, additional task information would be needed such as function pointers to the tasks.

The general execution pattern of the VPU subsystem can be simplified into the following five steps:

1. Requesting a task from OSIP to get input data information, including the remote data address and the data size;
2. Loading the remote data into the local memory;
3. Executing the task to process the data;
4. Sending the output data of the task from the local memory to remote;
5. Sending synchronization information to OSIP, acknowledging the end of the task execution and requesting OSIP to solve possible task dependencies.

The communication between the VPU and OSIP subsystem follows a request-response scheme. It means, whenever the VPU controller sends a new task request or a synchronization request, it always waits for the response message from the OSIP subsystem, before sending the next one.

6.4.5 OSIP Subsystem

The OSIP subsystem is the core component of the system. In Section 6.2, the basic concept of integrating OSIP into a NoC is introduced. Following the concept, different implementations of the OSIP subsystem can be made depending on the applications and the implementations of other system components. In this section, one possible implementation of such an OSIP subsystem is shown, targeting the current MIMO-OFDM application.
An overview of the OSIP subsystem is given in Figure 6.14. Structurally, it is almost the same as the classical bus-based OSIP system, with the exception of the use of an adapter. It contains several PEs (an ARM processor and several proxies), an AHB bus and a shared memory. The ARM processor is used to program the complete system, such as configuring the system, creating tasks and task dependencies for the VPU subsystems, etc. The task-related information is stored in the shared memory.

In order to enable the OSIP-system to control the VPU subsystems, the proxies must be able to cooperate with the corresponding VPU subsystems properly, supported by the adapter.

- **OSIP adapter**

  The adapter is responsible for interfacing with the network. On the one hand, upon receiving a packet from a VPU subsystem, it interprets the message contained in the packet and forwards the information to a corresponding proxy, based on the coordinates of the packet source node. In this OSIP subsystem, a dedicated proxy is defined for each VPU subsystem. The message contains the instructions from the VPU subsystem that the proxy should follow. As introduced in Section 6.4.4, there are two types of instructions sent from the VPU subsystem to OSIP: requesting a task (in more exact words, requesting the data information needed for a task) and synchronizing tasks. On the other hand, the
adapter prepares packets containing the response information from the proxy, after it finishes the instructions from the VPU. The information exchange between the adapter and the proxy is through a register interface.

- **Proxy**

  The key idea of using a proxy is to reduce frequent communications between PEs and OSIP over the NoC. A PE only needs to send a basic instruction remotely to the proxy, and the further detailed execution of the instruction is offloaded locally to the proxy, which implements the OSIP APIs or a subset of them, depending on the applications. In the following, it is shown how the two requests from the VPU subsystems are handled in this case study.

  - **Requesting a task:** When the proxy receives the instruction for requesting a task, it performs the following steps:
    1. Sending a command to OSIP to request a new task;
    2. Waiting for an interrupt from OSIP;
    3. Sending a second command to OSIP to fetch the task;
    4. Obtaining the concrete task information from the shared memory;
    5. Sending the task information with a unique task ID back to the VPU subsystem through the adapter.

    Certainly, for sending commands to OSIP, the proxy needs to lock OSIP and later unlocks it. It also needs to poll the OSIP status to check whether the commands are finished by OSIP.

  - **Task synchronization:** After the VPU finishes the task, it acknowledges OSIP of this by sending a message containing the task ID. The proxy checks whether there are tasks, which have a dependency on this task. If not, it simply sends a confirmation back to the VPU subsystem without involving OSIP. Otherwise, the proxy generates a task synchronization command to OSIP before sending the confirmation.

    There are situations, that a task has dependencies on multiple tasks. For example, the MIMO preprocessing can only be started after the channel estimations at both antennas have been made. In this case, the task synchronization cannot be started until both VPU\textsubscript{ChEst} subsystems send back the acknowledgment. As each proxy is only dedicated for one VPU subsystem, a shared memory address, which is created by the ARM processor when creating the tasks, is used to maintain the update of the synchronization information from both VPU\textsubscript{ChEst} subsystems. This shared address needs to be protected using a spinlock to avoid accessing it by both proxies at the same time. So the polling operation to the spinlocks is also implemented in the proxy.

There are also some other basic functions implemented in the proxies, such as interrupt control and system booting check.
6.4. System Implementation

In this work, both the OSIP adapter and the proxies are modeled using SystemC. The implementation is more ASIC-like as a state machine, because the VPU-subsystems are ASIC-based and require only low flexibility. As introduced in Section 6.2, software implementations using RISCs or ASIPs are also possible, depending on the requirements of the application.

6.4.6 Preliminary System Analysis

A preliminary analysis for the OSIP integration in the NoC-based system is made with respect to the throughput and latency requirements defined in Section 6.3.2.2. For the analysis, an ideal and a real NoC are selected. The ideal NoC has an unlimited communication bandwidth with a link data width (DW) towards infinity, and the real NoC has an initial width of 32 bits as the baseline NoC. The 32-bit link data width is the minimal supported width in this NoC. The achieved throughputs and latencies are depicted in Figure 6.15.

![Figure 6.15: Throughputs and latencies of MIMO-OFDM receiver with different NoC configurations](image)

Two conclusions can be drawn from the figure. First, the OSIP integration into the NoC works properly. With enough communication bandwidth provided, both the throughput and latency requirement can be well met. Second, similar to the bus-based systems, the NoC communication architecture also has a big impact on the OSIP-based systems. If using the baseline NoC instead of the ideal NoC, the latency of the receiver increases by a factor of 4.4×, while the throughput decreases from 150 Mbit/s to a moderate 50 Mbit/s. Note that 150 Mbit/s is the maximum achievable throughput due to the maximum input data rate received at the antennas.

However, a NoC with an unlimited bandwidth does not exist in reality. In the following section, realistic enhancements are made on the NoC-based communication
architecture (not limited to the NoC) to make better evaluations on the OSIP efficiency in NoC-based systems.

6.5 Enhancements for NoC-based Communication Architecture

To the baseline NoC-based communication, three performance enhancements are presented to improve the communication throughput and latency of the NoC and, more importantly, to parallelize the OSIP execution and the VPU computation in a pipelined fashion.

6.5.1 Increasing link data width

As indicated in Figure 6.15, increasing the link data width of the NoC certainly helps increase the system performance. There are also other parameters, such as the communication protocol, buffering scheme, number of virtual channels and packet size, which also have a high impact on the system performance. However, to optimize all these different parameters is out of the scope of this work. Therefore, most of the parameters are fixed to typical values (4 virtual channels, 8-flit buffers and 4 kB packet size), and only the link width is adjusted. This parameter allows to easily adjust the average interconnect throughput and latency.

6.5.2 DMA Support

It is common practice to have DMA support for communication on distributed memory architectures. Good examples can be found in the Cell processor [103] and the KeyStone II processor [20]. A DMA offloads the processors from transferring the data to other nodes of the system so that communication and computation take place in parallel. The enhancement to the VPU subsystem with a DMA is shown in Figure 6.16. The DMA contains a separate read and write channel, corresponding to the bi-directional port of the network interface. Using the data addresses and sizes provided by the controller, the DMA performs data transfer between the local memory and the remote memory over the NoC.

6.5.3 Pipelined Execution

DMAs allow to pre-fetch data so that once a PE is ready, the computation can start immediately. This hides the data communication overhead. Different from normal distributed systems, in OSIP-based systems, another type of communication overhead is introduced by accessing the central task manager – OSIP. This communication overhead is not only caused by the traffic in the network, but more due to the fact, that there is a latency when OSIP processes the requests from PEs. Therefore, it can help in improving the system performance if the OSIP latency can also be hidden, e.g.,
by means of pre-fetching the next tasks during the execution of PEs. In fact, in this system, pre-fetching tasks is obligatory for the pre-fetch of data. Without a task, the data information including the addresses and the data size is unknown to the DMA.

To hide the communication overhead of both data transfer and the accessing to OSIP, the pipelining concept is applied. Referring to the execution steps listed in Section 6.4.4, three pipeline stages are built: step 1) and 2) are grouped into the first pipeline stage, step 3) is in the second stage, and step 4) and 5) are grouped into the third stage. The pipelining is only applied to the demapping (SD_cluster) and the channel decoding (TD) subsystems due to frequent data and task information exchange between them, which is caused by the outer iterations. Deeper pipelining would be also possible by assigning each step to a separate stage. Note that in the non-pipelined execution of the system, in which DMAs are not used, the controller of the VPU subsystems has to transfer the data in step 2) and 4) between the local and remote memory.

An exemplary execution pattern of the non-pipelined and pipelined control flow of the master subsystem is illustrated in Figure 6.17. As shown in the non-pipelined pattern, the executions of Task_A, Task_B and Task_C (including requesting task, reading input data and sending output data) take place in a sequential order in the VPU subsystem. In contrast, in the pipelined pattern, their execution are interleaved. As shown in the figure, while executing Task_B, Task_C is pre-fetched from OSIP and the input data are prepared. In parallel, the output data of Task_A are sent and a task synchronization is issued to OSIP.

In the pipelined execution flow, there is a situation that deadlocks would happen between task executions, if no additional actions are taken on the OSIP side. Suppose that Task_C has a dependency on Task_B. If the controller is currently sending a request to OSIP for Task_C, it is not able to receive the task, because Task_B is not finished yet. On the other hand, when Task_B is finished, the controller is not able to send the
Figure 6.17: Two execution flows of VPU subsystems. Following annotations are made: R2L(i) for loading input data of Task_i from remote memory to local memory; L2R(i) for sending output data of Task_i from local memory to remote memory; req(i) for requesting Task_i from OSIP; exec(i) for executing Task_i; sync(i) for sending request to OSIP to synchronize possible tasks that depend on Task_i.

synchronization request, because the pipeline is blocked by requesting Task_C. So, a deadlock is resulted.

To solve this problem, the proxy first sends dummy task information to the VPU subsystem, if it cannot receive a new task from OSIP immediately. By identifying a dummy task, the VPU controller then performs the next pending operation in the pipeline, in this case sending synchronization information of Task_B. When the dependency between Task_B and Task_C is resolved by OSIP, the proxy generates the actual Task_C to the VPU subsystem. Till the VPU controller receives Task_C, the pipeline is stalled and no further new task request is sent to OSIP. In this way, all open operations in the pipeline are finished, and the pipeline execution can be continued.
6.5.4 Performance Analysis with Enhanced Communication Architecture

The performance improvement of the NoC-based system with the above-mentioned communication enhancements is illustrated in Figure 6.18.

![Figure 6.18: Improvement of system performance with enhancements in the NoC-based communication architecture](image)

By varying the link data width, a better application performance can be obtained. For the MIMO receiver, both the latency and the throughput are significantly improved. Considering the non-pipelined communication, the latency is reduced by a factor of $3 \times$ by increasing the data width from 32 bits to 1024 bits, and the throughput is increased by a factor of $2.8 \times$. However, in the non-pipelined communication, even at the largest link data width, the achieved throughput is still below the required throughput, and the latency just matches the requirement. Moreover, it can be observed that the performance first increases very fast from 32-bit to 128-bit data width. Then, beyond the 128-bit width the performance increment is slowed down. This results from the fact that in the small configurations, the communication bandwidth is the major factor which influences the system performance. In contrast, in the large
NoC configurations, the efficiency of OSIP has an increasing impact, because the communication becomes less critical than before. However, the latency of task scheduling and mapping by OSIP is almost constant for a given application. Therefore, the total improvement in percentage is reduced.

Of course, higher performance can be achieved by further increasing the link data width. This is however impractical due to the high area overhead and also limits the achievable clock frequency. To be mentioned, the 128-bit-based NoC achieves only slightly higher clock frequency than the system clock frequency of 300 MHz. Therefore, instead of having further large NoC configurations, pipelining communication and computation is a better choice. Shown by the dotted lines in the figure, by hiding the latency of the data transfer of DMAs and the requests to OSIP during the task execution based on the pipelining concept, more performance can be gained than simply increasing the link data width. Already at the data width of 128 bits, both the latency and the throughput requirement are fulfilled. And the performance at the 64-bit configuration with pipelining is comparable with the one at the 1024-bit configuration without pipelining. The performance saturation beyond 128-bit is caused by the maximum input data rate at the antennas.

6.6 Analysis of OSIP Efficiency in NoC-based Systems

In this section, the OSIP efficiency in NoC-based systems is analyzed. For this purpose, the different OSIP implementations (UT-OSIP, LT-OSIP, OSIP) and optimizations for the NoC-based communication are jointly investigated, similarly as done in the previous chapters.

Figure 6.19 shows the latency and throughput results with different OSIPs and communication optimizations. Naturally the best performance results are obtained with UT-OSIP. Especially in the non-pipelined communication configurations, the UT-OSIP has a clear advantage against the other two OSIP implementations. In these configurations, the latency of OSIP and LT-OSIP is simply added to the communication latency, resulting in additional communication overhead. However, in the NoCs with a low communication bandwidth (from the 32-bit to 128-bit link data width), in which the data communication is the main contributor to the overall communication overhead, the advantage of using UT-OSIP is not very large. In these cases, the system performance with OSIP is very close to that based on UT-OSIP. For the very low communication bandwidth based on the 32-bit width, even LT-OSIP results in a comparable performance as the other two OSIP implementations. By increasing the communication bandwidth, a fast OSIP implementation gains more advantage. Starting from the 256-bit link data width, UT-OSIP performs much better than OSIP. If comparing LT-OSIP and UT-OSIP, big performance difference already occurs at the 64-bit width. This shows that in this application, without pipelining communication and computation, OSIP can efficiently support a link data width up to 128 bits, but LT-OSIP is only suitable for the NoC with a link data width of 32 bits.
After pipelining the communication and computation, the advantage of UT-OSIP against OSIP is greatly reduced. Shown in the figure, beyond the 128-bit configuration, both UT-OSIP and OSIP result in the same system performance. In these configurations, the latency of OSIP is completely hidden by the task execution, and the data processing speed of the application processors is the major factor influencing the system performance. The biggest performance difference based on these two OSIP implementations can be found at the 64-bit configuration. In this configuration, the data transfer can be hidden by the task execution, but adding the additional OSIP latency to the data transfer exceeds the task execution time. However, the performance loss of using OSIP instead of UT-OSIP is relatively small. If choosing an even smaller configuration of 32-bit, for both UT-OSIP- and OSIP-based systems, the communication and the computation cannot totally overlap with each other and the data transfer plays a more important role than the scheduling and mapping efficiency. Therefore, their performance difference becomes very small again.
Pipelining the LT-OSIP-based systems also increases the system performance. However, in comparison to the other two fast OSIP implementations, the high latency of LT-OSIP largely prevents the total communication overhead from being hidden by the task execution. This results in much lower system performance improvement even with the pipelining support, especially in the systems with a high communication bandwidth for the data transfer. As shown in the figure, starting from the link data width of 128 bits, the performance increment becomes very small. In these systems, the overhead caused by the data transfer is not critical, but the latency of LT-OSIP dominates the communication time, which cannot catch up with the task execution speed.

A more direct indication of the OSIP efficiency – the OSIP busy state is compared in Figure 6.20. Compared to LT-OSIP, OSIP has much less busy time, which clearly highlights its efficiency in task management. The pipelined execution in the system involves OSIP more frequently into operation, but in total its busy time is not more than 50%, showing that OSIP still has enough capability for handling more tasks. The almost unchanged OSIP busy time starting from the 128-bit configuration with the pipelined execution confirms that the latency of OSIP is completely hidden by the task execution.

<table>
<thead>
<tr>
<th>Data Width</th>
<th>OSIP, non-pipelined</th>
<th>OSIP, pipelined</th>
<th>UT-OSIP, non-pipelined</th>
<th>UT-OSIP, pipelined</th>
<th>LT-OSIP, non-pipelined</th>
<th>LT-OSIP, pipelined</th>
</tr>
</thead>
<tbody>
<tr>
<td>32-bit</td>
<td>20%</td>
<td>20%</td>
<td>20%</td>
<td>20%</td>
<td>20%</td>
<td>20%</td>
</tr>
<tr>
<td>64-bit</td>
<td>40%</td>
<td>40%</td>
<td>40%</td>
<td>40%</td>
<td>40%</td>
<td>40%</td>
</tr>
<tr>
<td>128-bit</td>
<td>60%</td>
<td>60%</td>
<td>60%</td>
<td>60%</td>
<td>60%</td>
<td>60%</td>
</tr>
<tr>
<td>256-bit</td>
<td>80%</td>
<td>80%</td>
<td>80%</td>
<td>80%</td>
<td>80%</td>
<td>80%</td>
</tr>
<tr>
<td>512-bit</td>
<td>70%</td>
<td>70%</td>
<td>70%</td>
<td>70%</td>
<td>70%</td>
<td>70%</td>
</tr>
<tr>
<td>1024-bit</td>
<td>90%</td>
<td>90%</td>
<td>90%</td>
<td>90%</td>
<td>90%</td>
<td>90%</td>
</tr>
</tbody>
</table>

**Figure 6.20:** OSIP busy state at different NoC-based communication architectures

### 6.7 Summary

In this chapter, the potential problems of integrating OSIP into NoC-based systems are presented, which are caused by frequent polling to OSIP and long interrupt lines. A proxy-based approach is proposed, which acts as a bridge between the remote PEs and OSIP. It handles the polling and the interrupts and implements the OSIP APIs
locally to reduce the communication overhead between the PEs and OSIP. The case study based on a digital MIMO-OFDM receiver shows the feasibility of this approach in general. With the OSIP efficiency, both the throughput and latency constraints are met.

As for bus-based systems, an optimized communication architecture in NoC-based systems is also needed to effectively utilize the OSIP scheduling and mapping efficiency. OSIP itself can also be considered as a part of the communication, which has certain similarities as a shared memory. Requesting tasks from OSIP is somewhat like reading data from the memory. Therefore, hiding the latency of accessing OSIP should also be considered during the communication optimization, which is similar to hiding data accesses using a DMA. Pipelined execution in PEs by pre-fetching tasks from OSIP is a good way of achieving this.

In the case study presented in this chapter, the PEs are ASIC-based and require low flexibility, and each PE has a dedicated proxy. Therefore, simple state machine based OSIP proxies are used. It is also possible to share a common proxy for multiple PEs. If high flexibility is required, a flexible implementation using programmable processors can also be considered for the proxies.

6.8 Discussion

This chapter shows a general way of integrating OSIP into NoC-based systems by means of using proxies. There is still much research work to do for this area. In the following, discussions are made considering the proxy complexity and the possibilities of using OSIP in very large-scale NoCs.

6.8.1 Proxy Complexity

Using proxies certainly introduces additional area overhead into the system. Therefore, it is important to decide which OSIP APIs should be implemented in the proxies, depending on the applications.

In the simplest case, a proxy can only provide very primitive functions, mainly for addressing the polling and interrupt problems of OSIP-based systems in NoCs. All OSIP APIs are implemented at PEs. In this case, the proxy behaves more like a memory adapter and communicates with OSIP in a very passive way by simply forwarding the instructions from the PE to OSIP. For an interrupt from OSIP, the proxy only needs to create a packet to the PE, informing of the arrival of the interrupt signal without performing further operations. The kind of communication between the PEs and the proxies is fine-grained. Figure 6.21 shows an example how a command would be issued to OSIP in this case. The operations performed at the PE side and at the proxy side are basically one-to-one mapping except the polling. With only a few functions implemented, the proxy has very low complexity. But it is easy to see from the figure that the communication overhead between the PE and the proxy is high.
This can become critical if the communication channels of the system are already congested.

For reducing the communication overhead, more APIs can be implemented in a proxy. Considering the example in Figure 6.21, the PE only needs to send a command packet and then waits for the results packet, if the proxy can recognize that it is a command packet and automatically executes the necessary APIs required for issuing an OSIP command (locking/unlocking OSIP, checking the OSIP status, writing the command and reading results). Even further, it is possible that the PE sends a compact instruction to the proxy and the latter translates it into multiple OSIP commands, as done in the case study of this chapter. For example, when the PE requests a new task, the proxy first sends the command requesting a task to OSIP. It then autonomously sends a second OSIP command fetching a task to obtain the task, after it receives an interrupt signal from OSIP. It can be imagined that even more functions are implemented at the proxy for the sake of communication, which however increases the proxy complexity.

In principle, it is a multi-dimensional tradeoff that needs to be made for the decision on the implementation of a proxy:

- **Proxy complexity vs. Communication overhead**: The remote communications between a PE and OSIP can be costly in terms of the performance if these are too many and the communication bandwidth is limited. Adding more functions into the proxy locally can certainly reduces the remote communication overhead, but it is at the cost of the proxy complexity.
• **Proxy complexity vs. NoC complexity:** It is also possible to improve the remote communications between a PE and OSIP by e.g., increasing the communication bandwidth of the NoC or even creating dedicated communication channels for these communications. By this, without having a complex proxy, the remote communication overhead can still be reduced with respect to performance. So, tradeoffs can be made by considering whether adding more complexity into the NoC or into the proxy.

• **Proxy performance vs. Proxy flexibility:** Another tradeoff exists between the proxy complexity and flexibility. With fined-grained communications between a PE and its proxy, such as the example illustrated in Figure 6.21, the PE is the one that manages the communication with OSIP. The PE itself determines when and what to send to OSIP or read from OSIP, and the proxy is only a kind of memory adapter for accessing OSIP. In this sense, the proxy provides full flexibility to support any instruction from the PE with low complexity, but the communication overhead is high. In contrast, to support compact instructions from a PE to reduce the communication overhead, several OSIP commands are issued by the proxy itself to OSIP in sequence. However, this is subject to the condition that the execution pattern of the commands are always fixed, which limits the flexibility of the proxy in supporting the PE.

• **Proxy complexity vs. Proxy performance:** To support a large number of APIs in a proxy, which should at best also be able to cover different possible combinations of the instructions sent from a PE, a highly complex state machine would be expected if following the ASIC implementation. Using a programming processor can handle this with much less hardware complex, but the performance would be reduced. Here tradeoffs are also needed and an ASIP implementation seems to be a good choice.

Considering these tradeoffs, it is always important to analyze the application and system requirements for making the final decisions on the proxy implementation.

### 6.8.2 Multi-OSIP-System

The scale of the NoC used for the case study of this chapter is relatively small. In future NoCs for supporting many-core systems with hundreds or even thousands of PEs, using a single OSIP processor for task management is impractical. It is not necessarily the OSIP performance that would first become the system bottleneck, but the communication from PEs to OSIP could be more critical. The latency of sending commands to OSIP and receiving responses from OSIP could be very large caused by long paths through a large number of routers and heavy competitions in the NoC region near OSIP.

To address this problem, multiple OSIP instances can be considered in the system. They can be organized hierarchically or in a distributed way, or using a mixture of them, as illustrated in Figure 6.22. The whole system can be divided into several small
MPSoCs and each one is controlled by one OSIP subsystem. If these MPSoCs are only loosely coupled with each other in executing tasks, a distributed organization is well suitable. On the contrary, if the task management needs to cross multiple MPSoCs, a hierarchical organization is preferred. In this case, an OSIP subsystem at a lower hierarchy level becomes a PE subsystem of an upper-level OSIP subsystem. However, despite of the different structural organizations, they are all possible thanks to the flexibility of OSIP.
Chapter 7

Conclusion and Outlook

The development of today’s submicron silicon technology brings both benefits and challenges. On the one hand, much more functionalities can be integrated onto the chip of the same size than before, which has driven the evolution of embedded systems since the past decade. On the other hand, power and energy issues are becoming extremely critical, and the gap between chip performance and capacity is becoming larger and larger. To address these challenges, the MPSoC technology provides a convincing solution. The key is its task-level parallelism.

To efficiently exploit the parallelism of MPSoCs, task management plays an essential role. As introduced in Chapter 2, task management in MPSoCs can be static, semi-static or dynamic, among which dynamic task management is especially important for coping with high dynamism of today’s embedded systems. However, the increasing system complexity, from both the hardware architecture and the application software perspective, makes the design of good dynamic task management highly challenging. It should be fast and have high quality, and at the same time be flexible due to its dynamic nature.

In the literature, most task manager implementations are either based on RISC processors or dedicated hardwired solutions. Both have their limitations, the former lacking efficiency while the latter the flexibility. This work argues that ASIPs are a more promising way for implementing task managers, which combine the advantages of the other two implementation types.

In this work, an ASIP task manager called OSIP is proposed and analyzed in depth with respect to its efficiency and flexibility. The design of the OSIP architecture focuses on accelerating the typical and the most critical kernel operations in task management, such as list operations, various task comparison metrics, fast extraction of task information and frequent branches. For these operations, a set of customized instructions are developed, which largely speed up the major components of task management – scheduling, mapping and synchronization. Meanwhile, OSIP contains a normal RISC instruction-set, which provides enough flexibility for its possible functional extensions.

The efficiency of OSIP is assessed by comparing it with two other task manager implementations – a hypothetical extremely fast ASIC annotated as UT-OSIP and a RISC processor annotated as LT-OSIP. It is important to mention that a meaningful comparison must be performed within a system context, considering e.g., the system size in terms of the number of PEs and task sizes, etc. They influence the load to the task manager. It is also important to include the impact of the communication architecture into the assessment. Both being a shared resource, the task manager and
the communication architecture in fact compete with each other during the system execution: which one is finally the system bottleneck? The analysis results in Chapter 4 show that the high efficiency of OSIP in task management makes it unlikely become the system bottleneck. The system performance is mostly limited by the communication bandwidth or the task parallelism. In this case, adding more efficiency to the task manager, e.g., by using UT-OSIP, is rather a waste of the efficiency. On the other hand, the RISC-based LT-OSIP can easily become the system bottleneck due to its low efficiency, especially for large systems.

The evaluation for the flexibility of OSIP is performed when it is applied for improving the spinlock control mechanism in OSIP-based systems by using a-priori application knowledge, which is presented in Chapter 5. The programmability of OSIP enables the designer to extend the OSIP functionality by software to support more advanced spinlock control mechanism based on the so-called spinlock reservation, which results in significant performance improvement. By this, the OSIP-based systems have an even better performance than the UT-OSIP-based systems in many cases in the presented case study. For the latter, the extension is not applicable due to the flexibility limitation of ASIC. The successful application of the OSIP flexibility onto the spinlock improvement is supported by its efficiency, which supplies OSIP with enough performance margin for much more additional workload. In contrast, LT-OSIP also has flexibility, but it often fails in improving the spinlock control with the extended mechanism. Its low efficiency does not allow it to afford the additional work. In this sense, its flexibility is largely wasted.

The efficiency and flexibility are usually considered to be contradictory when designing a system. However, in this thesis it is shown that they actually enhance each other in dynamic task management, which requires both. With efficiency, the utilization of the flexibility can largely increase. And with flexibility, more efficiency can be achieved. Being able to make good trade-offs between the efficiency and flexibility, an ASIP provides a very promising solution to implement a dynamic task manager in complex MPSoCs. It can be more generally concluded that ASIPs are not only suitable for intensive data processing, but also able to be well applied in supporting control-centric applications such as system control.

The task management in OSIP-based systems is organized in a centralized way, i.e., OSIP manages tasks for several PEs and each of these PEs communicates with OSIP. This centralized organization would most likely be the preferred organization, if an ASIP is used as the task manager. A fully distributed task management using an ASIP manager at each PE would be an overdesign, which would cost too much hardware. However, in the systems with a NoC, which has a distributive characteristic, the centralized management might introduce too much communication overhead between the task manager and the PEs, which can impair the management efficiency of the ASIP. In OSIP-based systems, this problem is solved by introducing proxies for the PEs. An OSIP subsystem is created based on the proxies and OSIP, which is considered as the task manager as a whole towards the outside. The PEs do not communicate with OSIP directly, but with the OSIP subsystem by only sending a small number of instructions in form of packets. Inside the OSIP subsystem, the proxies in-
interpret the instructions and perform detailed communications with OSIP, complying with the OSIP programming model. In this way, the remote communication overhead through the NoC is effectively reduced to a large extent. Although the solution is rather specific for OSIP-based systems, the idea of converting remote communications into local communications can be generalized for the centralized task management.

Using an ASIP as the task manager, or more generally as the system controller in MPSoCs is still not popular in the research in spite of its advantages over RISC- or ASIC-based approaches. Among others, a major reason could lie in its design complexity. In comparison to the ASIC design, a complete software development tool chain needs to be developed for an ASIP, which largely increases the design complexity. In comparison to the RISC design, identifying efficient customized instructions for an ASIP is also not a trivial task. An ADL-based ASIP design environment, such as the Processor Designer used in this work, eases the ASIP design flow. But there are still many challenges to overcome. For example, developing a highly optimized compiler for an ASIP, which can effectively exploit the customized instructions, is still highly challenging. In this work, most of the customized instructions of OSIP are used based on the Compiler-Known Functions (CKFs), which certainly decreases its usability.

Concretely for OSIP-based systems, there are also several things which can be further optimized:

- **Current systems always use a RISC processor as the master of the complete system.** It boots up the system and creates an initial configuration of scheduling and mapping hierarchy. As OSIP is programmable, these tasks can in principle also be performed directly by OSIP. As a consequence, the RISC master processor can be removed from the system, which not only reduces the area cost, but also creates a clearer system control flow structure.

- **OSIP takes actions upon the requests from PEs through its register interface.** The information, on which OSIP makes task management decisions, is also completely provided by PEs through the register interface using commands. This makes the behavior of OSIP passive. As the task manager, it should be able to take the responsibility in controlling systems more actively, which is however limited by its current two interfaces. OSIP does have a good overview of the system tasks and load. But if it should take more active decisions, it sometimes needs to know the context of tasks. However, a direct access from OSIP to the task context is currently not available. A workaround through the PEs over the register interface has to be made, if needed. This is certainly not an efficient way. Therefore, a bus interface, e.g., compliant to the AHB protocol, is desired, which enables a direct access of OSIP to the shared memory over the bus for fetching required information.

- **The task management of OSIP-based systems follows the centralized approach,** which has to be faced with the scalability problem. At the end of Chapter 6, several possibilities have been shown to support very large-scale systems by
introducing multiple OSIPs. However, concrete case studies are still needed to evaluate the different multi-OSIP structures and many researches can further be carried on in this area.

Dynamic task management in MPSoCs is highly complex and challenging. This thesis has proposed an ASIP-based approach to overcome the challenges. The results show significant performance gain by applying this approach. Although there are still many things to work on in ASIP-based task management, using ASIPs turns out to be the best way to meet both the efficiency and the flexibility requirements in dynamic task management of MPSoCs.
Appendix A

Packet Types and Communication Protocol of NoC

In this appendix, the packet types and the communication protocol of the NoC presented in Section 6.3.1 are introduced in details.

A.1 Packet Types

There are four packet types for reading and writing shared memory and one packet type for the synchronization between master subsystems. In addition, application-specific packet types can be defined. The first byte of the packet payload is used to encode the packet type, which is called Operation Code (OP Code). The meaning of the remaining payload data depends on the packet type. Table A.1 gives a summary of the packet types. The M_S/Sys and S_S/Sys in the table are abbreviations of master subsystem and slave subsystem, respectively.

<table>
<thead>
<tr>
<th>Name</th>
<th>Direction</th>
<th>OP Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PCK_R</td>
<td>M_S/Sys → S_S/Sys</td>
<td>0x00</td>
<td>read request packet</td>
</tr>
<tr>
<td>PCK_RR</td>
<td>S_S/Sys → M_S/Sys</td>
<td>0x01</td>
<td>read response packet</td>
</tr>
<tr>
<td>PCK_W</td>
<td>M_S/Sys → S_S/Sys</td>
<td>0x02</td>
<td>write request packet</td>
</tr>
<tr>
<td>PCK_WR</td>
<td>S_S/Sys → M_S/Sys</td>
<td>0x03</td>
<td>write response packet</td>
</tr>
<tr>
<td>PCK_S</td>
<td>M_S/Sys → M_S/Sys</td>
<td>0x04</td>
<td>synchronization packet</td>
</tr>
<tr>
<td>PCK_APP</td>
<td>M_S/Sys → M_S/Sys</td>
<td>0x80 – 0xFF</td>
<td>application-specific packet</td>
</tr>
</tbody>
</table>

The structure of the payload contained in the different packet types is given by Table A.2. The column Byte range specifies the byte positions of the field in the packet (byte 0 – 3 are occupied by the routing header).

A.2 Communication between IP components

A complete communication initiated by a M_S/Sys to a S_S/Sys is always supported by a packet pair. For reading data from a S_S/Sys, the communication starts with a read request packet from a M_S/Sys, ends with a read response packet from the
Table A.2: Payload structure of packet types

<table>
<thead>
<tr>
<th>Type</th>
<th>Byte range</th>
<th>Semantics</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PCK_R</td>
<td>4</td>
<td>OP Code</td>
<td>0x00 for PCK_R</td>
</tr>
<tr>
<td></td>
<td>5 – 8</td>
<td>address</td>
<td>Start address in the target S_S/Sys for reading data.</td>
</tr>
<tr>
<td></td>
<td>9 – 12</td>
<td>size</td>
<td>Read data size (in bytes) from the target S_S/Sys (maximal data size: $2^{12} - 1 - 1 - 4 = 4090$ bytes).</td>
</tr>
<tr>
<td>PCK_RR</td>
<td>4</td>
<td>OP Code</td>
<td>0x01 for PCK_RR</td>
</tr>
<tr>
<td></td>
<td>5 – last</td>
<td>data</td>
<td>Read data returned to the target M_S/Sys.</td>
</tr>
<tr>
<td>PCK_W</td>
<td>4</td>
<td>OP Code</td>
<td>0x02 for PCK_W</td>
</tr>
<tr>
<td></td>
<td>5 – 8</td>
<td>address</td>
<td>Start address in the target S_S/Sys for storing data.</td>
</tr>
<tr>
<td></td>
<td>9 – last</td>
<td>data</td>
<td>Data to be stored. The data size should be at least 1 byte.</td>
</tr>
<tr>
<td>PCK_WR</td>
<td>4</td>
<td>OP Code</td>
<td>0x03 for PCK_WR</td>
</tr>
<tr>
<td>PCK_S</td>
<td>4</td>
<td>OP Code</td>
<td>0x04 for PCK_S</td>
</tr>
<tr>
<td></td>
<td>5 – 8</td>
<td>sync val</td>
<td>Synchronization value known to both the initiator M_S/Sys and the target M_S/Sys.</td>
</tr>
</tbody>
</table>

S_S/Sys. This also applies for writing data, which starts with a write request packet and ends with a write response packet.

Synchronizations between two M_S/Sys using synchronization packets are slightly different. A typical use case of a synchronization is to resolve data dependency between two M_S/Sys. In the case, the producer M_S/Sys first writes data into a remote S_S/Sys such as a shared memory. Then it sends a synchronization packet to the consumer M_S/Sys, informing it of the data availability. After receiving the packet, the consumer M_S/Sys reads the remote data and sends another synchronization packet to the producer as the response. Before receiving the synchronization packet from the consumer, the producer should make sure that no changes are made to the data to avoid data corruption.

In the following, the communication steps between the subsystems are shown in details. Due to different semantics in the packet payload, the interactions between the subsystems and NIs vary for different packets.

A.2.1 Transmission of Read Request Packet (PCK_R)

Table A.3 shows the communication between a M_S/Sys and a S_S/Sys to complete the transmission of a PCK_R.
A.2.2 Transmission of Read Response Packet (PCK_RR)

Table A.4 shows the communication between a M_S/Sys and a S_S/Sys to complete the transmission of a PCK_RR.

A.2.3 Transmission of Write Request Packet (PCK_W)

Table A.5 shows the communication between a M_S/Sys and a S_S/Sys to complete the transmission of a PCK_W.

A.2.4 Transmission of Write Response Packet (PCK_WR)

Table A.6 shows the communication between a M_S/Sys and a S_S/Sys to complete the transmission of a PCK_WR.

A.2.5 Transmission of Synchronization Packet (PCK_S)

Table A.7 shows the communication between two M_S/Sys to complete the transmission of a PCK_S.

A.2.6 Constraint

In the current NoC, out-of-order communication is not supported in communication between a M_S/Sys and a S_S/Sys. To avoid this, the system designer has to make sure that a M_S/Sys is not allowed to send another request packet to the same S_S/Sys, if the corresponding response packet to a previous request packet of the same type has not been received yet. However, this constraint does not apply to the communication between two M_S/Sys if both are able to handle out-of-order communication.
### Table A.3: Transmission of a PCK_R

<table>
<thead>
<tr>
<th>Step</th>
<th>Direction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>M_S/Sys → NI</td>
<td>M_S/Sys writes packet routing header, operation code, remote read address and data size to register NI_IN. NI creates a PCK_R and sends it.</td>
</tr>
<tr>
<td></td>
<td>NoC</td>
<td>Packet is transferred over NoC and reaches destination node.</td>
</tr>
<tr>
<td>2</td>
<td>NI → S_S/Sys</td>
<td>NI notifies S_S/Sys via interrupt.</td>
</tr>
<tr>
<td>3</td>
<td>S_S/Sys ← NI</td>
<td>S_S/Sys reads the first flit(s) for getting packet routing header, operation code, read address and data size from NI_OUT. The following information is extracted from the packet header: <em>payload size</em>, <em>priority</em>, <em>src/dst node</em>.</td>
</tr>
<tr>
<td>4</td>
<td>S_S/Sys</td>
<td>S_S/Sys stores <em>src/dst, payload size, priority</em> information for creating PCK_RR later.</td>
</tr>
</tbody>
</table>

### Table A.4: Transmission of a PCK_RR

<table>
<thead>
<tr>
<th>Step</th>
<th>Direction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>S_S/Sys → NI</td>
<td>S_S/Sys writes packet routing header, operation code and the first data bytes to register NI_IN. The header information is obtained from the corresponding PCK_R. NI obtains the knowledge of data payload size.</td>
</tr>
<tr>
<td></td>
<td>NoC</td>
<td>Packet is transferred over NoC and reaches destination node.</td>
</tr>
<tr>
<td>2</td>
<td>S_S/Sys → NI</td>
<td>S_S/Sys writes data repeatedly to register NI_IN. In parallel, NI creates and sends the packet.</td>
</tr>
<tr>
<td>3</td>
<td>NI → M_S/Sys</td>
<td>NI notifies M_S/Sys via interrupt.</td>
</tr>
</tbody>
</table>
| 4    | M_S/Sys ← NI | M_S/Sys reads the first flit(s) for getting packet routing header, operation code and the first data bytes from NI_OUT. The following information is extracted from the packet header: *payload size*, *src/dst node*.

| 5    | M_S/Sys ← NI | M_S/Sys reads data payload from NI_OUT.                                                                                                                                                                   |
### Table A.5: Transmission of a PCK_W

<table>
<thead>
<tr>
<th>Step</th>
<th>Direction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>M_S/Sys → NI</td>
<td>M_S/Sys writes packet routing header, operation code, remote write address and the first data bytes to register NI_IN. NI obtains the knowledge of data payload size.</td>
</tr>
<tr>
<td>2</td>
<td>M_S/Sys → NI</td>
<td>M_S/Sys writes the remaining payload data to NI_IN. In parallel, NI creates and sends the packet.</td>
</tr>
<tr>
<td>-</td>
<td>NoC</td>
<td>Packet is transferred over NoC and reaches destination node.</td>
</tr>
<tr>
<td>3</td>
<td>NI → S_S/Sys</td>
<td>NI notifies S_S/Sys via interrupt.</td>
</tr>
<tr>
<td>4</td>
<td>S_S/Sys ← NI</td>
<td>S_S/Sys reads the first flit(s) for getting packet routing header, operation code, write address and the first data bytes from NI_OUT. The following information is extracted from the packet header: payload size, priority, src/dst node.</td>
</tr>
<tr>
<td>5</td>
<td>S_S/Sys ← NI</td>
<td>S_S/Sys reads the remaining payload data from NI_OUT and writes the data to target address.</td>
</tr>
<tr>
<td>6</td>
<td>S_S/Sys</td>
<td>S_S/Sys stores src/dst, priority information for creating PCK_WR later.</td>
</tr>
</tbody>
</table>

### Table A.6: Transmission of a PCK_WR

<table>
<thead>
<tr>
<th>Step</th>
<th>Direction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>S_S/Sys → NI</td>
<td>S_S/Sys writes packet routing header and the operation code to register NI_IN. The header information is obtained from the corresponding PCK_W. NI creates a PCK_WR and sends it.</td>
</tr>
<tr>
<td>-</td>
<td>NoC</td>
<td>Packet is transferred over NoC and reaches destination node.</td>
</tr>
<tr>
<td>2</td>
<td>NI → M_S/Sys</td>
<td>NI notifies M_S/Sys via interrupt.</td>
</tr>
<tr>
<td>3</td>
<td>M_S/Sys ← NI</td>
<td>M_S/Sys reads flit(s) for getting packet routing header and operation code from NI_OUT. The following information is extracted from the packet header: src/dst node.</td>
</tr>
</tbody>
</table>
### Table A.7: Transmission of a PCK_S

<table>
<thead>
<tr>
<th>Step</th>
<th>Direction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>M_S/Sys1 → NI</td>
<td>M_S/Sys1 writes packet routing header, operation code and synchronization value to register NI_IN.</td>
</tr>
<tr>
<td>-</td>
<td>NoC</td>
<td>Packet is transferred over NoC and reaches destination node.</td>
</tr>
<tr>
<td>2</td>
<td>NI → M_S/Sys2</td>
<td>NI notifies M_S/Sys2 via interrupt.</td>
</tr>
<tr>
<td>3</td>
<td>M_S/Sys2 ← NI</td>
<td>M_S/Sys2 reads flit(s) for getting packet routing header, operation code and synchronization value from NI_OUT. The following information is extracted from the packet header: payload size, src/dst node.</td>
</tr>
<tr>
<td>4</td>
<td>M_S/Sys2</td>
<td>M_S/Sys2 might store src/dst, synchronization value information for creating another PCK_S as response later.</td>
</tr>
</tbody>
</table>
## Glossary

### Acronyms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACE</td>
<td>AXI Coherency Extension</td>
</tr>
<tr>
<td>ACO</td>
<td>Ant Colony Optimization</td>
</tr>
<tr>
<td>ADL</td>
<td>Architecture Description Language</td>
</tr>
<tr>
<td>AHB</td>
<td>Advanced High-Performance Bus</td>
</tr>
<tr>
<td>AMBA</td>
<td>Advanced Microcontroller Bus Architecture</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>ASIC</td>
<td>Application-Specific Integrated Circuit</td>
</tr>
<tr>
<td>ASIP</td>
<td>Application-Specific Instruction-set Processor</td>
</tr>
<tr>
<td>AT</td>
<td>Area-Time product</td>
</tr>
<tr>
<td>ATE</td>
<td>Area-Time-Energy product</td>
</tr>
<tr>
<td>AXI</td>
<td>Advanced Extensible Interface</td>
</tr>
<tr>
<td>BER</td>
<td>Bit Error Rate</td>
</tr>
<tr>
<td>CCMU</td>
<td>Cache Coherence Management Unit</td>
</tr>
<tr>
<td>CGRA</td>
<td>Coarse-Grained Reconfigurable Architecture</td>
</tr>
<tr>
<td>CKF</td>
<td>Compiler-Known Function</td>
</tr>
<tr>
<td>CP</td>
<td>Constraints Programming</td>
</tr>
<tr>
<td>CP</td>
<td>Cyclic Prefix</td>
</tr>
<tr>
<td>CSDF</td>
<td>Cyclo-static Synchronous Data Flow</td>
</tr>
<tr>
<td>DFG</td>
<td>Data Flow Graph</td>
</tr>
<tr>
<td>DMA</td>
<td>Direct Memory Access</td>
</tr>
<tr>
<td>DSP</td>
<td>Digital Signal Processor</td>
</tr>
<tr>
<td>DT</td>
<td>Data flit of a NoC packet</td>
</tr>
<tr>
<td>DVFS</td>
<td>Dynamic Voltage-Frequency Scaling</td>
</tr>
<tr>
<td>DW</td>
<td>Data Width</td>
</tr>
<tr>
<td>FCFS</td>
<td>First-Come First-Served</td>
</tr>
<tr>
<td>FDD</td>
<td>Frequency Division Duplex</td>
</tr>
<tr>
<td>FFT</td>
<td>Fast Fourier Transformation</td>
</tr>
<tr>
<td>FIFO</td>
<td>First In, First Out</td>
</tr>
<tr>
<td>flit</td>
<td>flow control unit</td>
</tr>
<tr>
<td>Abbreviation</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>FPGA</td>
<td>Field-Programmable Gate Array</td>
</tr>
<tr>
<td>fps</td>
<td>Frame per Second</td>
</tr>
<tr>
<td>FSM</td>
<td>Finite State Machine</td>
</tr>
<tr>
<td>GA</td>
<td>Genetic Algorithm</td>
</tr>
<tr>
<td>GE</td>
<td>equivalent gate count in units of two-input drive-one NAND gate</td>
</tr>
<tr>
<td>GPP</td>
<td>General Purpose Processor</td>
</tr>
<tr>
<td>GPU</td>
<td>Graphical Processing Unit</td>
</tr>
<tr>
<td>H-ARQ</td>
<td>Hybrid Automatic Repeat reQuest</td>
</tr>
<tr>
<td>HD</td>
<td>Header flit of a NoC packet</td>
</tr>
<tr>
<td>HDT</td>
<td>Header-Data-Tail flit of a NoC packet</td>
</tr>
<tr>
<td>HOSK</td>
<td>Hardware Operating System Kernel</td>
</tr>
<tr>
<td>HW-RTOS</td>
<td>HardWare Real Time Operating System</td>
</tr>
<tr>
<td>IDCT</td>
<td>inverse discrete cosine transform</td>
</tr>
<tr>
<td>ILP</td>
<td>Integer Linear Programming</td>
</tr>
<tr>
<td>IQT</td>
<td>Inverse Quantization</td>
</tr>
<tr>
<td>ISS</td>
<td>Institute for Integrated Signal Processing Systems at the RWTH Aachen University</td>
</tr>
<tr>
<td>ISS</td>
<td>Instruction-Set Simulator</td>
</tr>
<tr>
<td>ITRS</td>
<td>International Technology Roadmap for Semiconductors</td>
</tr>
<tr>
<td>LDPC</td>
<td>Low-Density Parity-Check</td>
</tr>
<tr>
<td>LISA</td>
<td>Language for Instruction Set Architecture</td>
</tr>
<tr>
<td>LLR</td>
<td>Log-Likelihood Ratio</td>
</tr>
<tr>
<td>LNA</td>
<td>Low-Noise Amplifier</td>
</tr>
<tr>
<td>LRU</td>
<td>Least Recently Used</td>
</tr>
<tr>
<td>LTE</td>
<td>Long Term Evolution</td>
</tr>
<tr>
<td>LTRISC</td>
<td>a template RISC processor provided by Synopsys Processor Designer</td>
</tr>
<tr>
<td>MAC</td>
<td>Medium Access Control</td>
</tr>
<tr>
<td>MIMO</td>
<td>Multiple-Input Multiple-Output</td>
</tr>
<tr>
<td>MMSE</td>
<td>Minimum Mean Square Error</td>
</tr>
<tr>
<td>MoC</td>
<td>Model of Computation</td>
</tr>
<tr>
<td>MP</td>
<td>Matching Pursuit</td>
</tr>
<tr>
<td>MPSoC</td>
<td>Multi-Processor System-on-Chip</td>
</tr>
<tr>
<td>M_S/Sys</td>
<td>Master subsystem</td>
</tr>
<tr>
<td>NI</td>
<td>Network Interface</td>
</tr>
<tr>
<td>NoC</td>
<td>Network-on-Chip</td>
</tr>
<tr>
<td>OFDM</td>
<td>Orthogonal Frequency-Division Multiplexing</td>
</tr>
<tr>
<td>OMAP</td>
<td>Open Multimedia Application Platform</td>
</tr>
<tr>
<td>OSIP</td>
<td>Operating System application-specific Instruction-set Processor</td>
</tr>
</tbody>
</table>
OS  Operating System
PC  Personal Computer
PC  Program Counter
PE  Processing Element
PHY  PHYsical layer
PSP  Processor-Support Package
QAM  Quadrature Amplitude Modulation
QoS  Quality of Service
QRD  QR Decomposition
RB  Resource Block
RF  Radio Frequency
RISC  Reduced Instruction Set Computer
RR  Round-Robin
RTL  Register-Transfer Level
S/Sys  subsystem
SA  Simulated Annealing
SDFG  Synchronous Data Flow Graph
SDF  Synchronous Data Flow
SD  Sphere-Decoding
SIMD  Single Instruction Multiple Data
SMP  Symmetric Multiprocessing
SPE  Synergistic Processing Element
SQRD  Sorted QR Decomposition
S_S/Sys  Slave subsystem
SSRAM  Synchronous Static Random Access Memory
TD  Turbo Decoding
TI  Texas Instruments
TL  Tail flit of a NoC packet
TL  Transaction-Level
TS  Tabu Search
TTI  Transmission Time Interval
VC  Virtual Channel
VLIW  Very Large Instruction Word
VPU  Virtual Processing Unit
ZF  Zero Forcing
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>ITRS 2012: Predictions for number of transistors per chip at production and</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>clock frequency.</td>
<td></td>
</tr>
<tr>
<td>1.2</td>
<td>Qualitative comparison between flexibility, performance and power consumption</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>of different PE types (adapted from [23]).</td>
<td></td>
</tr>
<tr>
<td>2.1</td>
<td>An exemplary system connected by a bus.</td>
<td>22</td>
</tr>
<tr>
<td>2.2</td>
<td>An exemplary system connected by a NoC, with 8 PE nodes and 8 memory nodes</td>
<td>23</td>
</tr>
<tr>
<td>3.1</td>
<td>Basic concept of OSIP-based systems.</td>
<td>32</td>
</tr>
<tr>
<td>3.2</td>
<td>OSIP software layer.</td>
<td>33</td>
</tr>
<tr>
<td>3.3</td>
<td>Hardware integration of OSIP-based systems.</td>
<td>35</td>
</tr>
<tr>
<td>3.4</td>
<td>Design flow for OSIP development.</td>
<td>39</td>
</tr>
<tr>
<td>3.5</td>
<td>An exemplary scheduling hierarchy.</td>
<td>41</td>
</tr>
<tr>
<td>3.6</td>
<td>Instruction-level profiling of C-implementation.</td>
<td>42</td>
</tr>
<tr>
<td>3.7</td>
<td>Pipeline structure of the OSIP Core.</td>
<td>45</td>
</tr>
<tr>
<td>3.8</td>
<td>Structure of AGU for OSIP_DT.</td>
<td>47</td>
</tr>
<tr>
<td>3.9</td>
<td>Load/Store of OSIP_DT and normal data.</td>
<td>47</td>
</tr>
<tr>
<td>3.10</td>
<td>Update OSIP_DT.</td>
<td>49</td>
</tr>
<tr>
<td>3.11</td>
<td>Hardware-supported node comparison.</td>
<td>50</td>
</tr>
<tr>
<td>3.12</td>
<td>Compare nodes and Compare nodes &amp; Continue.</td>
<td>52</td>
</tr>
<tr>
<td>3.13</td>
<td>Control of the OSIP state at the PFE-Stage.</td>
<td>54</td>
</tr>
<tr>
<td>3.14</td>
<td>Execution cycles of “hot-spot” commands and their occurrence frequency in</td>
<td>58</td>
</tr>
<tr>
<td></td>
<td>percentage.</td>
<td></td>
</tr>
<tr>
<td>3.15</td>
<td>Power profile of OSIP.</td>
<td>60</td>
</tr>
<tr>
<td>4.1</td>
<td>Setup of baseline system.</td>
<td>64</td>
</tr>
<tr>
<td>4.2</td>
<td>Task graph and OSIP configuration of the synthetic application.</td>
<td>65</td>
</tr>
<tr>
<td>4.3</td>
<td>Task graph and OSIP configuration of H.264.</td>
<td>67</td>
</tr>
<tr>
<td>4.4</td>
<td>Synthetic application: OSIP efficiency analysis in systems with an ideali-</td>
<td>69</td>
</tr>
<tr>
<td></td>
<td>zed communication architecture.</td>
<td></td>
</tr>
<tr>
<td>4.5</td>
<td>Synthetic application: Energy consumption ratio between LT-OSIP and OSIP.</td>
<td>71</td>
</tr>
<tr>
<td>4.6</td>
<td>H.264: OSIP efficiency analysis in systems with an idealized communication</td>
<td>71</td>
</tr>
<tr>
<td></td>
<td>architecture.</td>
<td></td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
<td></td>
</tr>
<tr>
<td>4.7</td>
<td>H.264: Energy consumption ratio between LT-OSIP and OSIP</td>
<td></td>
</tr>
<tr>
<td>4.8</td>
<td>H.264: Impact of the communication architecture in OSIP-based systems</td>
<td></td>
</tr>
<tr>
<td>4.9</td>
<td>H.264: Composition of the execution time from PE’s view</td>
<td></td>
</tr>
<tr>
<td>4.10</td>
<td>H.264: Impact of the communication architecture on the OSIP state</td>
<td></td>
</tr>
<tr>
<td>4.11</td>
<td>Cache system</td>
<td></td>
</tr>
<tr>
<td>4.12</td>
<td>An example of preventing data race conditions for write buffers by using spinlocks</td>
<td></td>
</tr>
<tr>
<td>4.13</td>
<td>H.264: Frame rate at different optimization levels</td>
<td></td>
</tr>
<tr>
<td>4.14</td>
<td>H.264: Composition of the execution time at different optimization levels of the communication architecture in the 11-PE-system</td>
<td></td>
</tr>
<tr>
<td>4.15</td>
<td>Synthetic application: Execution time at different optimization levels</td>
<td></td>
</tr>
<tr>
<td>4.16</td>
<td>H.264: Joint impact of OSIP and the communication architecture</td>
<td></td>
</tr>
<tr>
<td>4.17</td>
<td>Synthetic application: Joint impact of OSIP and the communication architecture in best case scenario</td>
<td></td>
</tr>
<tr>
<td>4.18</td>
<td>Synthetic application: Joint impact of OSIP and the communication architecture in worst case scenario</td>
<td></td>
</tr>
<tr>
<td>4.19</td>
<td>Synthetic application: Joint impact of OSIP and the communication architecture in average case scenario</td>
<td></td>
</tr>
<tr>
<td>5.1</td>
<td>Impact of the spinlock acquisition order</td>
<td></td>
</tr>
<tr>
<td>5.2</td>
<td>Basic spinlock control flow</td>
<td></td>
</tr>
<tr>
<td>5.3</td>
<td>Enhanced spinlock control flow</td>
<td></td>
</tr>
<tr>
<td>5.4</td>
<td>Flow of granting a spinlock request</td>
<td></td>
</tr>
<tr>
<td>5.5</td>
<td>Block diagram of triggering the OSIP core for spinlock reservation</td>
<td></td>
</tr>
<tr>
<td>5.6</td>
<td>Synthetic application: Performance improvement based on spinlock reservation</td>
<td></td>
</tr>
<tr>
<td>5.7</td>
<td>H.264: Performance improvement based on spinlock reservation</td>
<td></td>
</tr>
<tr>
<td>5.8</td>
<td>Synthetic application: Comparison of performance improvement using spinlock reservation between OSIP-, LT-OSIP- and UT-OSIP-based systems</td>
<td></td>
</tr>
<tr>
<td>5.9</td>
<td>H.264: Comparison of performance improvement using spinlock reservation between OSIP-, LT-OSIP- and UT-OSIP-based systems</td>
<td></td>
</tr>
<tr>
<td>6.1</td>
<td>OSIP subsystem for NoC integration</td>
<td></td>
</tr>
<tr>
<td>6.2</td>
<td>An exemplary NoC-based system with integrated OSIP</td>
<td></td>
</tr>
<tr>
<td>6.3</td>
<td>2D mesh-like NoC topology</td>
<td></td>
</tr>
<tr>
<td>6.4</td>
<td>Structure of multi-flit packets and single-flit packets</td>
<td></td>
</tr>
<tr>
<td>6.5</td>
<td>Router block diagram</td>
<td></td>
</tr>
<tr>
<td>6.6</td>
<td>NI block diagram</td>
<td></td>
</tr>
<tr>
<td>6.7</td>
<td>Power profile: Contribution of different power groups</td>
<td></td>
</tr>
<tr>
<td>6.8</td>
<td>Block diagram of a MIMO-OFDM doubly iterative receiver</td>
<td></td>
</tr>
<tr>
<td>6.9</td>
<td>Radio frame structure of LTE (FDD mode, normal CP)</td>
<td></td>
</tr>
<tr>
<td>6.10</td>
<td>Pilot pattern of 2×2 MIMO system</td>
<td></td>
</tr>
<tr>
<td>6.11</td>
<td>CSDF of 2×2 digital MIMO-OFDM receiver</td>
<td></td>
</tr>
<tr>
<td>6.12</td>
<td>Node assignment in the NoC-based system</td>
<td></td>
</tr>
</tbody>
</table>
6.13 Structure of VPU subsystem ........................................... 132
6.14 Block diagram of OSIP subsystem ............................... 133
6.15 Throughputs and latencies of MIMO-OFDM receiver with different NoC configurations ............................................ 135
6.16 Structure of VPU subsystem with DMA ....................... 137
6.17 Two execution flows of VPU subsystems ......................... 138
6.18 Improvement of system performance with enhancements in the NoC-based communication architecture ......................... 139
6.19 Joint impact of OSIP and NoC-based communication architecture on system performance ......................................... 141
6.20 OSIP busy state at different NoC-based communication architectures .......................... 142
6.21 An example of issuing an command to OSIP based on fine-grained communications between PE and proxy .......................... 144
6.22 Examples of multi-OSIP-systems in different organizations .......................... 146
List of Tables

2.1 List of task management references in the literature ................. 11
3.1 Synthesis results of OSIP and LT-OSIP .............................. 57
3.2 Power consumption of OSIP and LT-OSIP ............................ 59
6.1 Power consumption of NoC ........................................... 123
6.2 System parameters for achieving a throughput of 150 Mbit/s in a $2 \times 2$ MIMO system ................................................. 128
6.3 Selected implementations of algorithmic kernels ..................... 129
6.4 Definition of VPUs ..................................................... 130
A.1 Packet Types ......................................................... 151
A.2 Payload structure of packet types .................................... 152
A.3 Transmission of a PCK_R ............................................. 154
A.4 Transmission of a PCK_RR .......................................... 154
A.5 Transmission of a PCK_W ............................................ 155
A.6 Transmission of a PCK_WR .......................................... 155
A.7 Transmission of a PCK_S ............................................. 156
Bibliography


[90] IBM, “Power8 Processor.” Online: http://www-03.ibm.com/systems/power


Curriculum Vitae

Name  Diandian Zhang
Date of birth  May 17th, 1977, Jiangsu, China

since Jun. 2017  Continental Teves AG & Co. oHG, Frankfurt am Main, Germany
Apr. 2015 – May. 2017  Cypress Semiconductor Corp., Langen, Germany
Apr. 2014 – Mar. 2015  Spansion Inc., Langen, Germany
Feb. 2006  Graduation as Dipl.-Ing.

Diploma thesis at the Institute for Integrated Signal Processing Systems, RWTH Aachen University, Germany: “Analysis and Automatic Determination of Power-efficient Instruction Encoding for Application-Specific Instruction-Set Processors (ASIPs)”

Oct. 2000 – Feb. 2006  Study of Electrical Engineering and Information Technology with focus on Information and Communication Engineering, RWTH Aachen University, Germany
Sep. 1998 – Jul. 2000  German teacher at Deutschkolleg of Tongji University, Shanghai, China
Sep. 1994 – Jul. 1998  Bachelor study in German studies

Bachelor thesis at Tongji University, Shanghai, China: “A Thematic Comparison between Chinese and German Pop Music”

Bad Homburg, June 2017  Diandian Zhang