Memory-Aware Mapping Strategies for Heterogeneous MPSoC Systems

Dissertation
zur Erlangung des Grades eines

DOKTORS DER INGENIEURWISSENSCHAFTEN

der Technischen Universität Dortmund
an der Fakultät für Informatik
von

Olivera Holzkamp (geb. Jovanovic)

Dortmund
2017

Tag der mündlichen Prüfung:	16.03.2017
Dekan / Dekanin:	Prof. Dr. Gernot A. Fink
Gutachter / Gutachterinnen:	Prof. Dr. Peter Marwedel
	Prof. Dr. Jens Teubner

Acknowledgments

First and foremost, I would like to express my sincere gratitude to my advisor Prof. Dr. Marwedel for the continuous support and guidance of my Ph.D study and for his patience and immense knowledge. I would also like to thank him for giving me the chance to work in his group and to realize this thesis. Next, I would like to thank Prof. Dr. Teubner for his support and commitment to review this thesis. Furthermore, many thanks to my committee members, Prof. Dr. Heinrich Müller and Prof. Dr. Peter Buchholz.

A lot of colleagues supported me throughout my thesis. Many thanks to Robert Pyka, Markus Buschhoff, Helena Kotthaus, Michael Engel, Andreas Heinig and Florian Schmoll. Robert, Helena and Markus helped me a lot through my thesis with fruitful discussions, proof-reading and motivation. Robert Pyka deserves special thanks for helping me through hard times and always keeping me motivated. In fact, we accompanied each other through all highs and lows during our work. Andreas Heinig, Florian Schmoll, Daniel Cordes and Robert Pyka helped me with technical discussions and technical setup within the MNEMEE framework. I also wish to acknowledge the contribution and cooperation that was provided by the student Nils Kneuper.

Many people outside the department deserve my special thanks. Prof. Dr. Christiane Floyd provided advice and motivation which has been a great help during my work. I would also like to extend my thanks to Iuliana Bacivarov and Sander Stujik for their support and constructive feedback during the planning and development of the tools in this work. Next to Prof. Dr. Marwedel, Prof. Dr. Petru Eles, Dr. Alexandru Andrei and Prof. Dr. Jens Wagner contributed in awakening my interest in research and PhD.

A part of this work was kindly supported by the SFB 876 research project of the Deutsche Forschungsgemeinschaft (DFG). However, a great part was developed within the MNEMEE project, i.e. EC Seventh Framework Program FP7 / IST-216224. The members of these projects deserve special thanks for their kind cooperation, support and the fruitful discussions. Furthermore, the mentoring^3 program initiated by Universitätsallianz Metropole Ruhr (UAMR) supported me with great input, a great mentor and great PhD students. Many thanks to all these institutions for their resources and their guidance.

Last but not least, I wish to thank my family and friends for their support and encouragement throughout my study. Words cannot express how grateful I am to my parents Elica and Stojan, who always believed in me and encouraged me to keep going. My special thanks are extended to my beloved husband Stephan for his support, motivation and all his patience during these years. My mother-in-law Ingrid also deserves my very great appreciation for her support and motivation. I thank all of them for keeping me free of other tasks, so that I was able to finish this thesis.

Abstract

Embedded systems, such as mobile phones, integrate more and more features, e.g. multiple cameras, GPS sensors and many other sensors and actuators. These kind of embedded systems are dealing with increasing complexity due to demands on performance and constraints in energy consumption. The performance on such systems can be increased by executing application tasks in parallel. To achieve this, multiprocessor systems-on-chip (MPSoC) devices were introduced. On the other side, the energy consumption of these systems has to be decreased, especially for battery-driven embedded systems. A reduction in energy consumption can be achieved by efficiently utilizing the hardware resources on these devices. MPSoC devices can be either homogeneous or heterogeneous. Homogeneous MPSoC devices usually contain the same type of processors with the same speed, i.e. clock frequency, and the same type and size of memories for each processor. In heterogeneous MPSoC devices, the processor types and/or clock frequencies and memory types and/or sizes may vary.

During the last decade, research has dealt with optimizations for the efficient utilization of hardware resources on MPSoCs. Central issues are the extraction of parallelism from sequential code and the efficient mapping of the parallelized application tasks onto the processors of the system. A few frameworks have been developed which distribute parallelized application tasks to available processors while optimizing for one or more objectives such as performance and energy consumption. They usually integrate all required, foregoing steps such as the extraction of parallelized tasks from sequential code and the extraction of a task graph as input for the mapping optimization. These steps are performed either manually or in an automated way. These kind of frameworks help the embedded system designer to significantly reduce design time. Unfortunately, the influence of memories or memory hierarchies is neglected in mapping optimizations, even though it is a well-known fact that memories have a drastic impact on the runtime and energy consumption of the system.

This dissertation investigates the effect of memory hierarchies in MPSoC mapping. Since a thread based application model is used, a thread graph extraction tool is introduced. Furthermore, two approaches for memory-aware mapping optimization for homogeneous and heterogeneous embedded MPSoC devices are presented. The thread graph extraction tool extracts a flat thread graph with important annotations for software requirements, hardware performance and energy consumption. This thread graph represents all required input information for the subsequent memory-aware mapping optimizations. Dependent on the complexity of the application, the designer can choose between a fine-grained and a coarse-grained thread graph and thus influence the overall design time.

The first presented memory-aware mapping approach handles single objective optimizations, which reduce either the runtime or the energy consumption of the

system. The second presented memory-aware mapping approach handles a multi-objective optimization, which reduces both, runtime and energy consumption. All approaches additionally reduce the work of the embedded system designer and thus the design time. They work in a fully automated way and are integrated within the MACCv2/MNEMEE tool flow. The MNEMEE tool flow also provides all required foregoing steps such as the parallelization of sequential application code. The presented evaluations show that considering memory mapping during MPSoC mapping optimization significantly reduces the application runtime and energy consumption. The single objective optimizations are able to achieve an average reduction in runtime by about 21% and an average reduction in energy consumption by about 28%. The multiobjective memory-aware mapping optimization achieves an average reduction in runtime by about 21% and an average reduction in energy consumption by about 26%. Both presented optimization approaches were validated for homogeneous and heterogeneous MPSoC devices. The results clearly show that neglecting the memory subsystem can lead to wasted optimization potential.

Contents

Chapter 1

Introduction

Contents

1.1 Introduction

The process of miniaturization of electronic circuits began with the invention (1947) and distribution of transistors, which replaced large vacuum tubes. The first integrated circuit (IC) was realized 1958 at Texas Instruments where transistors, capacitors and resistors formed an electronic circuit on an area of only few square millimeters. The next important step towards miniaturization and computing power was the introduction of the first, commercial microprocessor *Intel 4004* with 2,300 transistors in 1971 [1]. In the following years, the computation power increased while the size of the components and the manufacturing costs shrunk continuously. In 2012, Intel introduced the multicore Xeon Phi Coprocessor with about five billions transistors and up to 62 cores [2].

This increase was consistent with Moore's law, which states that the number of transistors on integrated circuits doubles every two years [3]. In 2010, the International Technology Roadmap for Semiconductors (ITRS) confirmed this trend up to the end of the year 2013 [4]. In 2014, ITRS decided that Moore's law would no longer be followed. Instead, application requirements are the basis of the new "application guided technology" roadmap. Seven focus teams are going to analyze applications and identify new technology requirements. They include heterogeneous components and heterogeneous integration as well as "continued shrinking of horizontal and vertical physical feature sizes to reduce cost and improve performance" [5].

However, the low manufacturing costs of electronic devices and the miniaturization led to a proliferation of special-purpose systems. Contrary to general-purpose systems as personal computers (PCs), special-purpose systems fulfil a specific, customized task. Some examples are smart phones, digital cameras or tablets. These special-purpose systems are defined as embedded systems or as cyber-physical systems.

Definition 1.1 (Embedded Systems) *Embedded systems (ES) are information processing systems embedded into a larger product - Peter Marwedel [6]*

Definition 1.2 (Cyber-Physical Systems) *Cyber-Physical Systems (CPS) are integrations of computation with physical processes - Edward Lee [7]*

Cyber-physical system can be also defined as embedded systems in a physical environment [6]. These systems are characterized by their interaction with the environment through information processing with the help of sensors and actuators. Since they are usually also embedded in larger products, e.g. automotive, their presence is less apparent. The main goal of these systems is to make our lives easier or more comfortable. They are already an inherent part of our daily life. We are surrounded by these systems everywhere:

- *consumer electronics*: television, cameras, tablets, mp3 player, media player, video game consoles, etc.
- *telecommunication*: smart phone, telephone switches for network, modem / router, USB Internet sticks, etc.
- *transportation*: automotive (e.g. ABS, airbag, navigation), aircraft (e.g. collision detection), railway, etc.
- *household appliance*: refrigerator, vacuum robots, washing machine, microwave oven, etc.
- *home automation*: security, control lights, climate, surveillance, etc.
- robotics, medical equipment, etc.

1.2 Embedded Systems Architecture

The progress in hardware development is making embedded systems more efficient, powerful and faster. The hardware progress and the cumulative demands of applications of these systems are a reason for their increasing complexity. For example, the first functions of a mobile phone were short message service (SMS) and telephone service. Nowadays, a smart phone has an integrated camera, internet access, touch screen, GPS, different applications, etc. With the increasing demands, the complexity in the design of these systems has also increased. Furthermore, in the beginning

of embedded systems, the hardware was plain, e.g. consisting of micro controller including memory and input/output functions. Afterwards, systems-on-a-chip (SoC) was introduced where all functions or components of an electronic system are integrated into one circuit, including processors that are more powerful. Nowadays, different hardware architectures are available for embedded systems, from SoCS to homogeneous and heterogeneous multiprocessor system on chips (MPSoCs). MPSoCs integrate multiple processors on a chip, where the processors can be identical (homogenous) or of different types (heterogeneous). Depending on the application area of the embedded system, the hardware has to be chosen properly by the embedded system designer. Some designers work already on a given, fixed architecture, other designers have to design an appropriate architecture.

Different terms occur considering concurrent execution: multiprocessor, multicore processor and MPSoCs. A multiprocessor is a hardware architecture containing two or more processing units, which share (main memory and) peripherals. A multicore processor is a processor containing two or more cores or central processing units (CPU), respectively. Both are processing blocks while MPSoCs integrate a complete system solution, e.g. which can contain video and graphics solution [8]. Figure 2.1 on page 13 illustrates a MPSoC, i.e. a functional diagram of the OMAP 5912 from Texas Instruments.

To sum up: In embedded systems, concurrent execution is usually realized through homogeneous or heterogeneous MPSoC systems, which are required in order to satisfy the demands of high-performance computing applications. The development of MPSoCs and state-of-the-art systems are described in more detail in Section 2.1.

1.3 Design of embedded systems

For the design of embedded systems, different design steps have to be performed including optimizations, e.g. for the reduction of runtime, energy consumption, code size, bus traffic, etc. Next to these different optimizations, the designer has to face various other challenges, e.g. managing concurrency and also meeting the main demands such as security, robustness, safety, timing aspects, dependability, reliability, availability, maintainability, decrease hardware and software costs of design, etc. In a first step, the designer has to decide which application model, specification language and hardware or hardware description are suitable for the characteristics of the system that has to be designed. Afterwards, optimizations can be performed for hardware (e.g. dynamic voltage scaling, utilize memories and buses, etc.) and software (e.g. code optimizations). All these design steps confront the designer with huge and complex tasks, which have to be accomplished as fast as possible due to time-to-market constraints. Time-to-market constraints define the time of a product from design until its availability for sale. With an earlier release of the new system, the industry gains advantages in competition and market share. In the last decade, MPSoCs and the resulting requirement to manage more and more concurrency as

well as the therefore resulting increasing performance and functionality demands on these systems increased the complexity for embedded system design even more.

Different optimization tools are introduced in research for the different design steps. However, these tools frequently cannot be connected to each other for many reasons. For example, the tools work on different internal models for application and architecture specification due to the manifold models that are existing for embedded systems. Even if the tools work on the same models, it is not guaranteed that the optimization tools can interconnect. Due to different interfaces, the output does not fit to the input of other design optimization tools or the internal data structures of one tool does not fit to other tools, respectively. Here, the designer has on the one side help in form of an optimization tool, which speeds up the design time and decreases the complexity. However, on the other side, now the designer has either to take care of the interconnection of different tools or design an optimization tool on his/her own, which can be connected to another optimization tool.

In both cases, this means again a huge delay in design time. Another problem, which occurs during design time, is the proper analysis and validation of the optimizations on the system. For this, cycle-accurate simulators are required in order to obtain valid results. However, the setup of these simulators, including the setup of the operating system and the interconnection with the underlying synchronization and communication library (e.g. OpenMP, MPI) are a very time-consuming job, which requires a lot of knowledge in these fields. Moreover, an energy and runtime model has to be available for the chosen architecture. It is very difficult and time-consuming to obtain all these energy and runtime values. These issues confront the designer and the researchers with huge timing and management problems. For this reason, research is also concentrating on the automated integration for the different design optimization steps. Some frameworks, which integrate the most important design steps, were introduced. An overview of these frameworks is given in Section 2.4.4. The development and updating of these frameworks is taking years, and their scope can fill several PhD theses.

1.4 Mapping problem

In this section, two important design optimization steps are introduced. Both optimizations solve an allocation problem where software (application threads or memory objects) has to be efficiently mapped onto the hardware (processing elements or memories). These optimizations optimize for a single objective goal or for several objective goals as the reduction of energy consumption or runtime.

1.4.1 Mapping of memory objects to memories

The mapping of memory objects to memories is an important optimization that is often neglected during design. Memories or the memory subsystem have a drastic influence on the system's runtime and energy consumption due to the still existing *memory wall problem* [9]. This problem describes the huge gap between the speed

of processors and the speed of memories. The speed of processors grows much faster than the speed of memories. Thus, the access time to memories limits the performance and memories consume a lot of energy. In a SoC, the access to the main memory can take up to 100 cycles. This problem is also valid for MPSoC systems where the access time to off-chip DRAM also consumes a great amount of time.

Memory hierarchies were introduced in order to cope with this significant problem by placing smaller, faster and more energy-efficient memories (i.e. on-chip memories) close to the processor, building a memory hierarchy with one or more levels. The idea is to place frequently used instruction or data memory objects onto on-chip memories and thereby to reduce runtime and energy consumption. For example, an on-chip memory on level-1 is located next to the processor and has ideally only one cycle access time. On-chip memory on level-2 are also common. There is a larger distance between the level-2 memory and the processor. Dependent on this distance, the access time increases as well as the energy consumption compared to the access and energy consumption to the level-1 memory.

On-chip memory hierarchies either consist of cache or scratchpad memories. Especially in the design of real-time embedded systems, scratchpad memories are extensively used [10]. The content of these memories is known in advance. They are predictable with respect to runtime and energy consumption. These advantages are achieved because they are explicitly allocated by the application designer or an optimization software, respectively. Furthermore, they consume less energy and die area than caches since no additional hardware in the form of control logic is required for the management of their content. For these reasons, we explicitly consider only scratchpad memories instead of caches.

Due to the drastic influence of the memories on runtime and energy, intelligent algorithms or optimizations are required in order to map efficiently the most accessed memory objects onto the memories. Important achievements in this research field is described in Section 2.4.1. Next, some trends in the memory hierarchy of MPSoCs are introduced, which will lead the research in a new direction with new challenges in the memory mapping optimization.

The trends in MPSoC systems show that level-1 and level-2 caches are common. One example is the Exynos Octa 5410 architecture, which contains two level-1 and one level-2 cache for each processor. Another trend is the heterogeneity in the memory hierarchy. For example, the OMAP architecture contains usually a GPU and an ARM processor. In the OMAP 5912, the ARM926EJ processor has access to a 16 KB instruction and to a 8 KB data cache. The TMS320C55x DSP core has access to a 64 KB on-chip dual-access RAM, a 96 KB on-chip single-access RAM and a 24 KB instruction cache [11].

The OMAP 5430 includes a dual-core with two ARM Cortex-A15, a DSP subsystem with a TMS320DM64 DSP processor and an image-processing unit (IPU) subsystem. Each Cortex-A15 core contains a separate instruction and data cache of 32 KB and both cores have access to a level-2 cache of 2 MB. The DSP has access to a level-1 32 KB cache and a 128 KB level-2 cache. The dual-core Cortex-M4 in

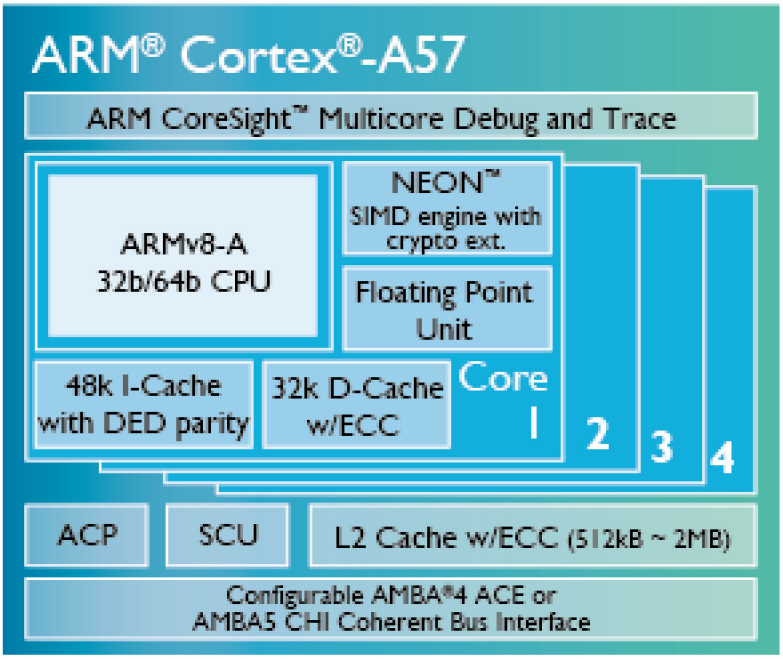

Figure 1.1: ARM Cortex-A57[1]

the IPU subsystem has a shared access to level-1 32 KB cache and a shared level-2 64 KB cache and a 16 KB ROM. Also, a multicore GPU architecture is integrated with two SGX544 cores containing a shared system-level cache of 128 KB [11].

The ARM Cortex-A57 processor can have a more heterogeneous memory hierarchy containing a 48 KB instruction cache and a 32 KB data cache, as illustrated in Figure 1.4.1. The level-2 caches can have a size from 512 KB to2 MB.

These real-world examples show the upcoming trend of heterogeneity in the memory subsystems in MPSoC systems. A more detailed overview of MPSoCs is given in Section 2.1. A classification of MPSoC systems concerning their memory organization is described in Section 2.3.1.

1.4.2 Mapping of application tasks to processors

The design step of mapping of application tasks onto the processing elements of the architecture has evolved from the development of parallel execution in hardware architectures. The goal is to efficiently map parallel application tasks in order to achieve an increase in performance or in order to decrease the energy consumption.

The preceding step is a parallelization, which is performed either manually by the designer or automatically by special parallelization tools. After this step, the former sequential application or parts of this application are split into several tasks, which can be executed in parallel. These tasks can have the same workload, different

[1]Image reproduced with permission of the rights holder, ARM Ltd.

workloads, or a mixture of both. This depends on the application, its types of operations, access patterns, complexity, etc. Moreover, the type of parallelization performed on the application influences the workload of each task (pipeline, loop-level, etc.). After the parallelization, the mapping optimization has to distribute these tasks efficiently. For this, the characteristics of the underlying architecture have to be considered, as the number of processors and the characteristics of each individual processor (e.g. clock rate, energy consumption). The complexity and solution/design space for this optimization increases if two or more contradicting objective goals are considered. In research, this design step was also immensely explored. Related work of this topic is described in Section 2.4.4. As already stated in the previous Section 1.2, the trends in MPSoC design go towards heterogeneity and towards heterogeneous processors and thus to the mapping of tasks to these processors.

1.5 Contributions

Research tries to alleviate the burden of the designer by introducing different optimization techniques for the diverse, complex design steps. This thesis focuses on memory optimizations in homogeneous and heterogeneous MPSoC systems. It combines two separately considered optimizations into one optimization step since state-of-the-art MPSoC architectures require this combined view in order to utilize the full optimization potential. Usually, two optimization steps are performed. One optimization step is the mapping of concurrent application threads among the processors of a homogeneous/heterogeneous MPSoC. The other optimization step is the mapping of memory objects to a local memory or to different memories in the memory hierarchy. Both are common optimization steps in the design of embedded systems, which try to efficiently utilize the resources of the system.

This thesis gives a more detailed view on the complexity of the combined mapping of the application threads together with their memory requirement onto the architecture's processors and their underlying memory hierarchy in 2.3.3. Since the trend of MPSoC systems goes towards heterogeneous systems with heterogeneous processors and a heterogeneous memory subsystem, this thesis focuses on this kind of architecture. Heterogeneous MPSoC systems increase the complexity of this optimization.

In application to processor mapping, the focus lies only on the characteristics of the processors (e.g. speed, energy consumption, type). Due to the drastic influence of the memory wall problem on runtime and energy, it is crucial to consider the underlying memory hierarchy. The architecture resources have to be matched to the application's requirements, or vice versa. This thesis focuses on this matching. Focus is laid on both, the architecture and application characteristics with the goal to efficiently utilize the system resources in order to achieve a high optimization potential. The optimization goals are the reduction of energy consumption and/or runtime.

The thesis concentrates on the extraction of a detailed analysis model for all introduced optimization. On the architecture side, the processor's energy consumption for active and idle mode is included as well as the performance capability. Since processors have access to their underlying memory hierarchy with different memories on different levels, these are captured in detail as well. All memories have various characteristics as size, type (instruction, data or unified). Furthermore, access speed and energy consumption differ depending on the access type, i.e. read or write access and access width (Byte, half-word, word). The characteristics of the underlying buses, that are accessed, are also considered. On the application side, the characteristics of the application threads have to be considered in detail, i.e. workload of a thread and thus all requirements of the memory objects as size, type of memory object (data or instruction), number of read and write accesses, etc. Furthermore, communication between threads is considered as well, taking into account the data send over a communication channel (i.e. data size, number of data send, etc.). All this information is taken into account for the optimization and analysis model. It is verified against cycle-accurate simulation.

For the separate consideration of either the reduction of runtime or energy consumption, an integer linear programming (ILP) optimization is introduced. Furthermore, a multiobjective optimization for the reduction of runtime and energy consumption at the same time is also introduced. All optimizations are available as separate tools, which can be used by designers, dependent on the designer's system requirements. Both optimizations use the detailed analysis model. An evolutionary algorithm is used for the multiobjective memory-aware mapping optimization. It generates a set of mapping solutions by crossover and mutation. These solutions are evaluated and generated in a design space exploration loop. At the end, a set of mapping solutions is provided as solution.

Another important design step is the extraction of an extended thread graph from parallelized source code. This thread graph extraction gained also attention in research and is handled by this dissertation. This step is performed before the application-to-architecture mapping and is required in order to perform the mapping optimization properly. The challenge is an accurate extraction of a detailed representation of the parallel threads, including dependences as control-flow and data-flow. Furthermore, the mapping tools also require information on threads, which include architectural characteristics, e.g. as the runtime of a thread on different processors or the memory sizes of memory objects, which are dependent on the architecture. All these manifold and additional information have to be extracted and annotated to the thread graph. This extraction is handled by this thesis.

Except for the memory-aware mapping ILP optimization, these optimizations were developed within the EU project MNEMEE with the goal to integrate memory-awareness in the embedded system design and to alleviate the work of the designer by providing an automated tool flow for the complex design optimization steps.

To the best of our knowledge, we are the first, which integrate a homogeneous and a heterogeneous memory subsystem in the optimization step of mapping concurrent threads onto processors.

1.6 Outline

The remainder of this book is organized as follows:

- **Chapter 2** describes the common architecture and application models as well as all underlying models of this work and a more detailed problem description. Furthermore, an overview over the related work is given.
- **Chapter 3** presents the EU project MNEMEE including the goals and the description of the fully automated design framework.
- **Chapter 4** presents the *Thread Model Extraction Tool* which extracts an annotaded thread graph from parallelized C-Code.
- **Chapter 5** describes the *Memory-Aware Mapping Optimization Tool* based on integer linear programming (ILP) for the reduction of energy consumption or for the reduction of runtime, respectively.
- **Chapter 6** describes the *Memory-Aware Mapping Optimization Tool* based on an evolutionary algorithm for the multiobjective optimization for energy consumption and runtime.
- **Chapter 7** presents the summary of this work and gives an overview over future work.

1.7 Authors Contribution to this dissertation

In §10(2) of the "Promotionsordung der Fakultät für Informatik der Technischen Universität Dortmund vom 29. August 2011", a dissertation has to provide a separate list which presents the author's contribution to research and results in cooperation with other researchers.

Therefore, the following list provides an overview over the contribution of the author on the presented results for each chapter:

- Chapter 2: This chapter describes related work and gives an overview over application and architecture models. Thus, in this chapter the author of this thesis presents research results by other authors.
- Chapter 3: The MNEMEE toolflow [12],[13] was created by almost all MNEMEE partners. However, the main part of the MACCv2 framework [14], which is the basis of the toolflow, was developed by ICD in cooperation with TU Dortmund. The author has also worked on this framework, especially by integrating the tools/optimizatons that are introduced in the next chapters. A great part of the framework is used for analysis, optimization and evaluation of this thesis. All tools within the MNEMEE toolflow are described in publications [12],[13].

- Chapter 4: The *Thread Model Extraction Tool* was developed within the MNEMEE project [12], [13]. The pre-processing steps of the thread graph extraction were developed in collaboration with IMEC, Sander Stuijk(TUE Eindhoven) and the author. The thread graph extraction itself was entirely developed by the author and is described in detail in Chapter 4.6. The author also integrated the thread graph extraction and 70% of the pre-processing step into the MACCv2 framework.

- Chapter 5: The memory-aware ILP optimizations for the reduction of runtime and energy were developed by the author in cooperation with her master's student Nils Kneuper, who realized the majority of the implementation. About 60% of the ILP formulations were contributed by the author. The resulting publication [15] was written by the author. Guidance and inspiration was provided by Prof. Peter Marwedel. The employed cycle-accurate CoMET simulator was donated by Synopsys Inc. [16]. The implementation of the heterogeneous platform within CoMET and the MACCv2 framework was entirely performed by author.

- Chapter 6: The memory-aware multiobjective optimization was entirely developed by the author and published in [17]. Guidance, inspiration and motivation were provided by Prof. Peter Marwedel and Prof. Lothar Thiele. The initial problem definition was provided by Prof. Marwedel. Iuliana Bacivarov helped with guidance and a lot of fine-tuning in the optimization and implementation. The system-level framework called distributed operation layer (DOL) [18] was used as a basis for the implementation of the optimization. The optimization goals of DOL were exchanged by the author's optimization goals. Furthermore, the author implemented several extension and changes. First, the process network model was exchanged by a thread-based model, which includes parallel sections. The application model was extended to include memory objects and all required characteristics/information of memory objects (e.g. size, number of reads/write, instruction/data/shared, etc.). The architecture model was extended to include more memory characteristics (memory types, energy consumption/runtime for different type of accesses, etc.). The author also performed the integration and interfacing of this tool into the MNEMEE tool flow or MACCv2 framework, respectively. This optimization is also described in [12] and [13] as part of the MNEMEE tool flow.

Chapter 2

Models and Tools

Contents

The design of embedded systems is a challenging and quite complex task. Consumer demand and improvements in hardware have even increased this complexity. State-of-the-art embedded systems get more and more complex and have to fulfill many aspects such as efficiency (energy, runtime, code, costs), timing aspects (hard/soft deadlines, real-time aspects, etc.), dependability (i.e. reliability, safety, security), etc. [6]. All these aspects must be considered and have to be an integral part in the design flow in order to guarantee the full functionality of the desired system. It is impossible to develop a standard design flow for the abundance of different embedded systems and all their resulting manifold characteristics (i.e. functionality and requirements). Furthermore, the time and effort spent in the design of the system also depends on the characteristics of the desired system. For example, the design of a flight control system in an airplane is much more complex than an electric-driven, classic DVD player. Some of these different systems need the same optimization steps in the design flow. However, since these different systems have

different underlying software and hardware, it can happen that the same optimization problem has to be implemented in a different way with a different approach for different design flows.

Section 2.1 gives an overview over the development of MPSoCs and state-of-the-art MPSoCs. Next, Section 2.2 introduces the common application and architecture models in embedded system design. Afterwards, Section 2.3 gives an overview over the considered models for the mapping optimization problem in this work. Related work is presented in Section 2.4.

2.1 MPSoCs

MPSoCs arose from the requirement to perform parallel execution instead of sequential execution in order to achieve an increase in performance. The first area of application was the fast solving of large and complex problems such as weather modeling, simulation of the evolution of galaxies, data mining, etc. This includes all scientific and engineering calculations. The first supercomputer CDC 6600 was invented 1964 in order to solve large scientific problems and to use time sharing for smaller problems. The supercomputer included 10 peripheral processors, each of them containing a small memory for program and buffer area (4096 memory words with 12 bit length). A central processor was also included. All processors had access to a central memory. The first multiprocessor occurred in the 1970s. The Illiac IV processor included four control units, which controlled 64 arithmetic logic units (ALUs). With this multiprocessor, vector and array operations could be performed in parallel. The C.mmp multiprocessor contained 16 processors which were connected to a memory through a crossbar [19]. Also, the occurrence of *superscalar*, *very long instruction word (VLIW)* and *explicitly parallel instruction computing (EPIC)* processors have the common goal to improve performance by the parallel work of a certain number of execution units.

MPSoCs have emerged in the past decade. The architecture design mainly depends on the underlying embedded application. In this section, some representative MPSoCs are introduced. One of the first MPSoC, *Lucent Daytona* was introduced in the year 2000. It contained four CPUs, which were attached to a high-speed bus. Each CPU has a 8 KB 16 banks local memory. Each bank can be configured as instruction/data cache or scratchpad memory. This MPSoC was designed for wireless base stations where identical signal processing was performed on a number of data channels [19].

Another well-known MPSoC for multimedia processing is the *Philips Viper Nexperia* which was designed for advanced TVs, set-tops and home media servers. It consists of two CPUs: a MIPS and a Trimedia VLIW processor. Buses are integrated for each CPU as well as for the external memory interface. The MIPS processor is the master running the operating system, while the Trimedia is integrated as the slave which executes the commands from MIPS. Hardware accelerators performed computations such as color space conversion [19].

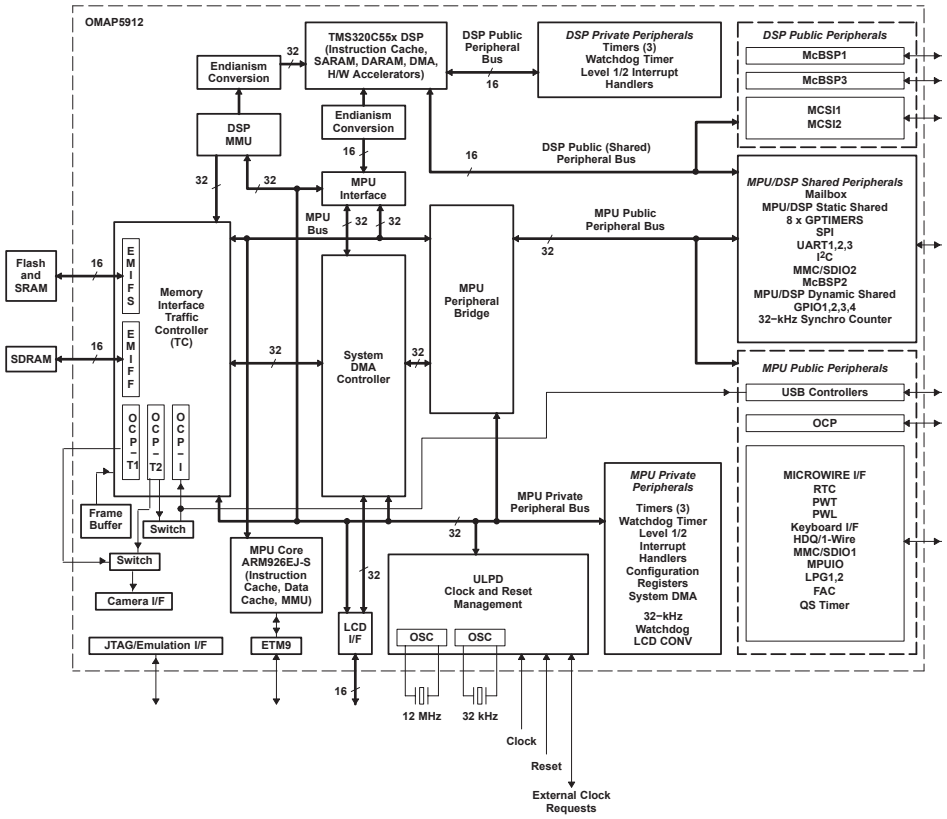

Figure 2.1: TI Omap 5912 - Functional Diagram [11]

The Texas Instruments (TI) OMAP architecture is a well-known MPSoC which was widely used for mobile phones. Many implementations exist for this architecture. For example, the OMAP 5912 implementation has an ARM9 and a DSP processor (TMS320C55x). A functional diagram of the OMAP 5912 is illustrated in Figure 2.1.

Almost all implementations of OMAP contain a (dual-core) ARM processor together with a DSP or PowerVR graphics processing unit (GPU). Another famous MPSoC is the CELL architecture. It contains one 64-bit power processor element (PPE) and 8 specialized synergistic processor elements (SPEs) together with a high-bandwidth bus and a high-speed memory controller. The SPEs consist of a synergistic processor unit (SPU) which includes a 256 KB local memory. All processing elements communicate over a high-speed bus. A direct transfer between the local memories of different SPUs is possible through a DMA controller. The CELL architecture was implemented in the Playstation 3 video-game console from Sony and for some HDTVs from Toshiba. Other application areas such as visualization, image and signal processing, and various scientific and technical workloads are also suitable [19, 11].

The ARM11MPCore is a multiprocessor consisting of one to four (identical)

processors that are known for their low dynamic power consumption and which are often used within MPSoC systems for smartphones. Each processor has a separate instruction and data cache. The sizes for each cache can vary from 16 KB to 64 KB. The Arm Cortex-A processors are the follow-on products of the Arm11MPCore. They also contain separate instruction and data caches on level 1 with different memory sizes from 8 to 64 kB and a shared L2 Cache with sizes from 128 kB to 2 MB. The ARM big.little architecture is actually one of the state-of-the-art MPSoCs. The first-generation architecture consists of one to four ARM Cortex-A15 processors together with one to four ARM Cortex-A7 processors. The second generation of this architecture consists of ARM Cortex-A57 or Cortex-A72 processors together with ARM Cortex-A53 or Cortex-A35 processors. The *big* Cortex-A15 can be used for heavy workloads while the *little* energy-efficient Cortex-A7 is used for smaller workloads that have to be accomplished all the time, e.g. operating system activities, user interface, etc. Different variations in the clock rate and the number of coprocessors are available as a quad core with two copies of each processor or also a system with four *big* and two *little* processors [20], [21]. This architecture can be used for many computation intensive applications as well as for a mixture of network traffic and computation (e.g. STB, WLAN), for OS GUI environments in netbooks, smartphones etc. Actually (in 2016/2017), this big.little architecture is implemented in state-of-the-art smartphones such as *Samsung Galaxy S7* or *Samsung Galaxy Note* devices within different available Samsung Exynos processor chips.

The Exynos 5 Octa chip (5410/5420) contains four Cortex-A15 cores and four Cortex-A7 cores. Compared to earlier Exynos processors, Samsung claims that it reduces power consumption (by up to 70%) while maximizing performance. The 5410 chip contains a PowerVR SGX544 MP3 533 MHz GPU for console-like 3D games and a 13 Mp 30 fps ISP (Image Signal Processor). The (local) memory sizes are not published yet by Samsung [22], [23].

An existing Exynos Octa 5420 development board is based on Exynos 5 Octa and integrates four Cortex-A15 processors with 32 KB of instruction and 32 KB of data level-1 Cache and a 2 MB level-2 Cache. The four Cortex-A7 processors also include a separate 32 KB instruction and 32 KB data level-1 cache and a 512 KB level-2 cache. A 64 KB ROM for secure booting and 336 KB internal RAM for security functions is included. Other features are a 3D graphic accelerator with multi-core GPU, and a separate 2D graphic accelerator [24].

The Exynos 7 Octa chip (5433/7420) introduced the new ARM Cortex-57 and 53 cores and integrated four Cortex-A57 and four Cortex-A53 cores. In 2016, Samsung maintained the big.little architecture and introduced the Exynos 8 Octa chip containing a custom core CPU by Samsung (based on Armv8) and Cortex-A53 cores [25].

Further processor chips are based on the big.little architecture such as Qualcomm Snapdragon 808 MSM8992 and 810 MSM8994 (two to four ARM Cortex-57, four Cortex A-53 cores in LG, HTC and Microsoft Lumia smartphones) [26], [27] and Nvidia Tegra X1 (four ARM Cortex A57, four Cortex A-53, Nvidia Android TV) [28].

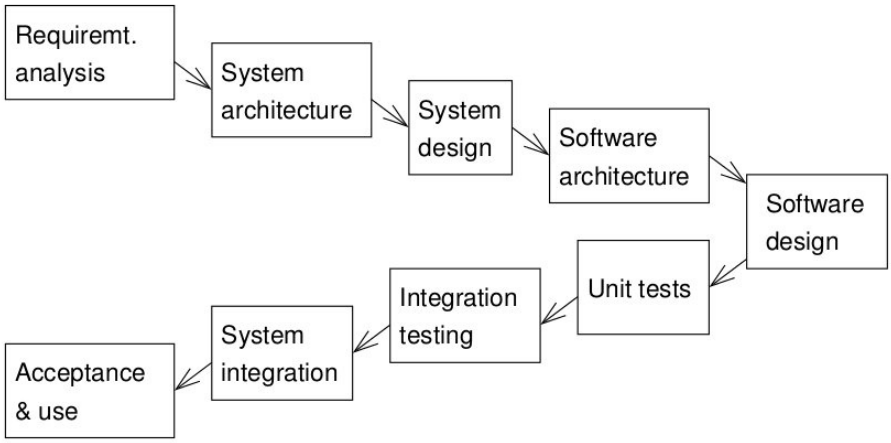

Figure 2.2: V-model (rotated standard view) [6]

To sum up: Different MPSoC alternatives exist which integrate homogeneous or heterogeneous components such as processors and memory subsystems. The modeling of these different systems is described in the next section.

2.2 Application and architecture models

Various specification languages, models and design flows exist. The designer is confronted with a huge number of possibilities and has to decide which design flow, specification language or model specifies the behavior and functionality of the desired system properly. Unfortunately, no specification or design flow exists which covers all requirements completely.

The embedded system design includes several steps, which can start from an idea up to the prototype of a system. Typically, the design flow starts with a specification of the system behavior, hardware and software with the aid of models and specification languages. Here, the specification depends on the preliminary infrastructure, i.e.: Is the hardware and/or software (partly) already fixed or is a new hardware or software design required? Based on the infrastructure, the designer has to decide which design flow to choose.

Many design flows exist. If the designer cannot find a proper design flow, he/she can choose an individual design flow which reflects the desirable design steps in the most suitable way. Here, three instances of design flows are introduced, which represent the common steps in the design of embedded systems.

The V-model (version 97) with all required steps is shown in Figure 2.2. The first axis represents the design and implementation phase while the other axis represents the validation and test phase. It is widely used in the German government sector. A

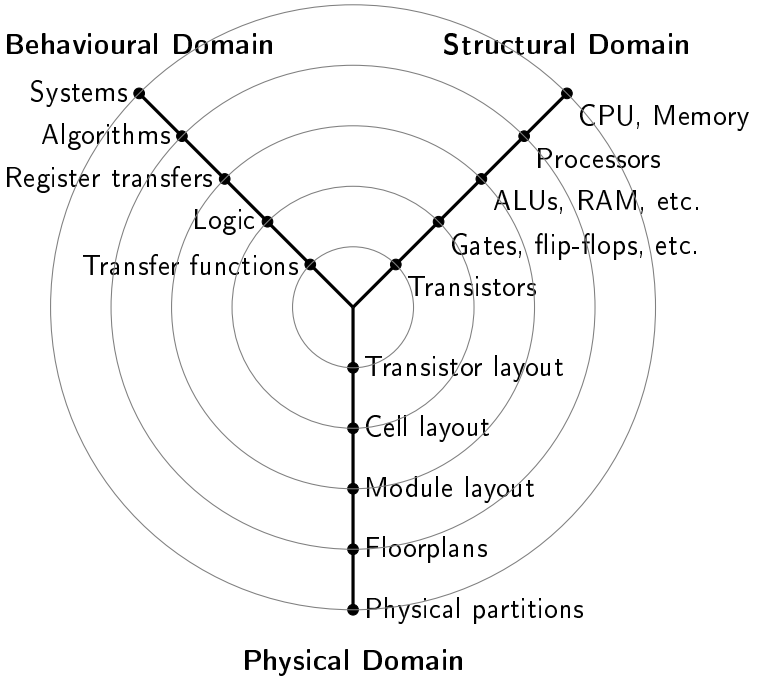

Figure 2.3: Gajski-Kuhn Y-chart

more recent model with better scalability is available with extended and modifiable design steps [29].

Gajski's Y-chart is another well-known flow for hardware design, which is illustrated in Figure 2.3 [30]. This design flow has three dimensions: the behavioral, structural and geometrical dimension. The geometrical layout contains information about chips and the structural layout contains information about hardware components. The abstraction increases from the inner to the outer circle. Each circle and axis defines a model. The high-level model describes the overall behavior while the models on the lower-level describe the behavior of components. Different design paths can be chosen. They are usually performed step by step and typically take their path from a coarse behavioral level to a fine-grained geometrical level.

Another design flow, which includes essential design steps, is shown in Figure 2.4 and is taken from [6]. Boxes with rounded corners represent stored information in a design repository. The rectangles represent transformations on data. After the specification of the system behavior, the hardware components and the system software, further design steps and optimizations are required. First, the application tasks have to be mapped onto the execution platform. Different optimizations can be performed during the mapping and afterwards, e.g. high-level transformations, runtime and energy minimization strategies, etc., can be applied. At the end, an evaluation step should be performed in order to evaluate the effect of the different optimizations on the system (i.e. performance, energy consumption, etc.) and to adjust some optimization if necessary. Furthermore, a validation step should check

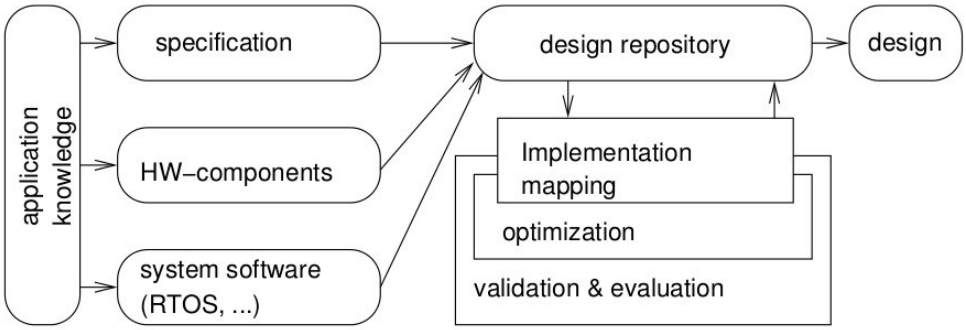

Figure 2.4: Design flow from Marwedel [6]

the correctness of the design.

However, next to the decision which design flow to choose, also a decision has to be made concerning the hardware and concerning the software model or model of computation, respectively. The model of computation is an abstraction for certain functionality and represents design specifications. No model of computation can meet all specification requirements. Linked to the decision of hardware and software specification, the decision about the parallel programming model is also important.

The decision about the hardware platform depends on the characteristics of the system that has to be designed. Often, the architecture is already fixed since it is reused or only slightly changed from previous designs. In this case, the focus lies more on software design. Even if the hardware is fixed, it is often desirable to model the hardware at another abstraction level because at the validation step, it is less time-consuming to simulate the system and perform optimizations.

Next, a short overview is given over the common hardware (memory / communication models) and models of computation for multiprocessor systems.

2.2.1 Memory Architecture Model

First, different memory architectures for multiprocessor systems exist:

- shared memory architecture (UMA and NUMA)
- distributed memory architecture
- hybrid distributed shared memory architecture

In the shared memory architecture, the processors can access all memory in the global address space [31]. This means, that the memory and all changes made to it are visible to all processors and all processors can change the memory content. The shared memory architecture can be divided into two classes, which differ in the memory access time: UMA (uniform memory access) and NUMA (non-uniform memory access). Figure 2.5 illustrates the UMA shared memory architecture and

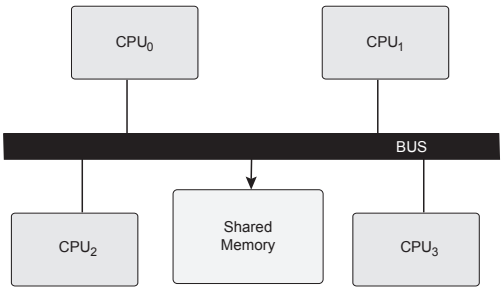

Figure 2.5: Shared Memory Architecture (UMA)

Figure 2.6 shows a NUMA shared memory architecture. The UMA shared memory architecture usually contains identical processors which have equal access time to the memory. The NUMA shared memory architecture often links two UMA architectures together as illustrated in Figure 2.6. The memories are accessible by both sides through the link. Since the access across the link is slower, the memory access times can vary. Contrary to the shared memory architecture, in the distributed memory architecture, each processor has exclusive access to its own memory as shown in Figure 2.7.

A communication network connects the memories of the different processors. If data has to be shared among different processors, the programmer has to define the communication and synchronization. Both memory architectures have their advantages and disadvantages. In the shared memory architecture, there is usually one bus for several processors. Several conflicts can occur on the bus when two or more processors want to access the bus at the same time. This problem increases when the number of processors increase. In the distributed memory architecture, the

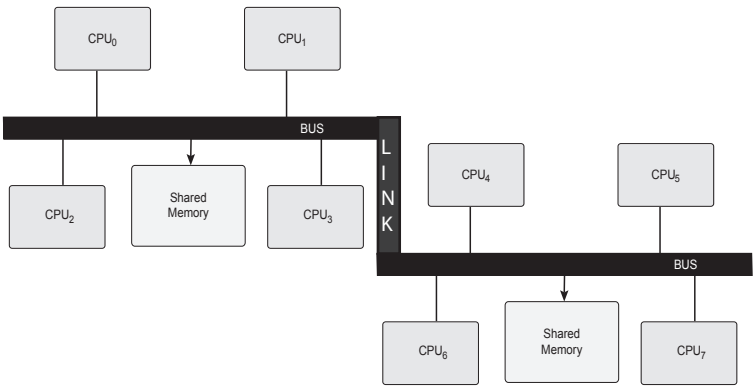

Figure 2.6: Shared Memory Architecture (NUMA)

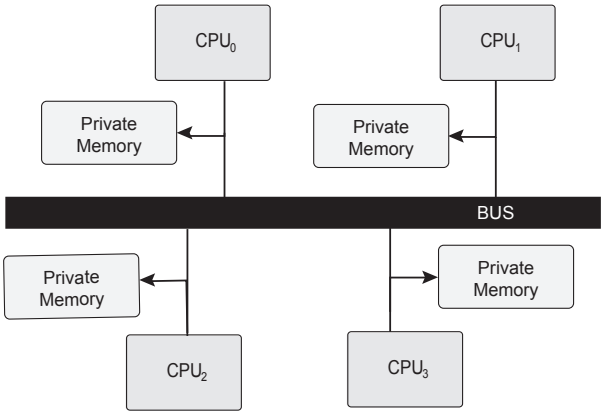

Figure 2.7: Distributed Memory Architecture

processors can access their memory rapidly without any conflicts. The disadvantage here is that the programmer has to take care for the data communication between the different processors. Also, mapping an existing data structure from a shared memory architecture to a distributed memory architecture is extremely difficult. The shared memory architecture is more user-friendly since the data sharing is fast and in the most architectures uniform. However, the hybrid distributed shared memory architecture combines both memory structures and also the advantages of both. The processors can work on their own memory without any bus conflicts. The access to a shared memory is only performed for access to shared data [31]. An example is illustrated in Figure 2.8.

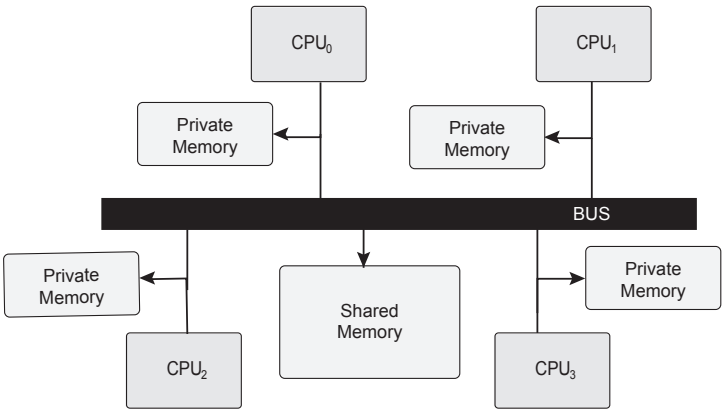

Figure 2.8: Hybrid Distributed Shared Memory Architecture

2.2.2 Model of Communication

The selection of the hardware influences the model of communication and the selection of the proper model of computation and vice versa. The model of communication can be either shared memory or message passing. Shared memory communication is performed through a common memory which is accessible by all processors. Message passing is performed by sending and receiving messages. It can be also implemented when no common memory is available, but it is slower than shared memory communication. Three different strategies can be implemented for message passing: asynchronous, synchronous or remote invocation. For the asynchronous message passing, the sender can send its messages and a channel buffer stores these messages. The recipient does not need to confirm the received message. In the synchronous message passing, it is contrary to the asynchronous process. Here, the sender and the recipient have to be ready for the communication exchange. In the remote invocation strategy, the sender can only send its message after it has received a confirmation from the recipient [6]. The designer selects the model of computation based on the underlying architecture and communication model.

2.2.3 Model of Computation

The requirements for any design-flow are adequately defined abstraction levels and models. The hardware and software/specification models on any level trade off accuracy for efficiency but alleviate the complexity of the system design specification. On the other hand, models have to be defined with the right amount of detail that will allow rapid and meaningful (design space) exploration, synthesis and validation. These abstraction levels are also valid for the models of computation, which are required for the description of system behavior. The system behavior is specified in a first step. The models of computation impact the design of specification languages.

There are several requirements for specification models: timing behavior, concurrency, reliability, modularity, synchronization, communication, security, etc. Since no model can meet all specification requirements, it is extremely important to choose the appropriate model for a successful design of the system. Different types of models exist: state based, thread based, actor based or data flow models. State based models are usually finite state machines (automata), state charts and timed automata. They describe state-oriented behavior. Timed automata include timing information. The state charts model includes several features such as the modeling of hierarchy (super and sub-states), concurrency (and/or states), history mechanism, etc. Another state-based model is the specification and description language (SDL). The basic elements are processes, which are modeled as finite state machines. Processes also perform operations on data (declarations, assignments, decisions) and are interacting with other processes. For this interaction, asynchronous message passing is performed through FIFO queues where each process has one queue for signals. SDL was standardized by the International Telecommunication Union (ITU). SDL is suitable for distributed applications, and it also has been used for specifying

ISDN. However, it is not deterministic. For an implementation, the upper bounds for the length of FIFOs have to be determined. It is not a suitable model for hard deadlines [6].

Furthermore, task graphs or process networks can be used to represent dependences between computations, i.e. control- and data-flow between different tasks or processes. Even more detailed task graphs can be selected, which can represent e.g. the amount of data consumed at each edge. Also, an organization as a hierarchical task graph is possible if required by the designer. Task graphs can be used at different steps in the design, e.g. for mapping of tasks to processing units or for aggregation or generation of tasks in the parallelization step.

Further models can be actor-based, such as communicating finite state machines (cfsm) and data flow models such as synchronous data flow graphs (SDF) or Kahn process networks (KPN). Kahn introduced the Kahn process networks (KPN) for parallel/distributed execution [32]. The actors can be implemented as processes in the programming language C. The communication is asynchronous and performed via unbounded FIFO channels. The channels are point-to-point queues with one producer and one consumer per channel. The synchronization is performed via blocking read and non-blocking write. An advantage is that KPNs are deterministic.

A disadvantage is that KPNs are difficult to implement because the size of infinite FIFOs has to be implemented on limited physical memory [33]. Furthermore, too small buffers can lead to an artificial deadlock which has to be resolved at runtime [34].

The synchronous data flow graph by Lee and Messerschmitt is also an actor-based model where the actors are executed concurrently [35]. Each actor produces and also consumes a fixed number of tokens per firing. Let us assume an actor A which produces tokens is connected to an actor B which consumes the tokens of actor A. In a SDF, a balance equation exists for each channel where the number of firings f_A of actor A multiplied by the number of tokens N produced by actor A is equivalent to the number of firings f_B of actor B multiplied by the number of tokens M consumed by this actor: $f_A N = f_B M$. The communication between the actors is buffered. The data flow is synchronous since all tokens are consumed at the same time. The message passing is asynchronous, i.e. the actors do not have to wait until an output is accepted. SDF can be scheduled statically. The schedule can be determined at compile-time. Furthermore, the buffer memory requirements and deadlocks are decidable problems.

Carl Adam Petri introduced Petri nets in 1962 [36] They model casual dependencies and are suited for message passing. The key elements are conditions, events and flow relations. Conditions can be met or not, and events take place when certain conditions are met. Flow relations relate conditions and events. Petri nets model resources, mutual exclusion and synchronization. Three different kinds of Petri nets exist: condition/event nets, predicate/transition nets and place/transition nets. Condition/event nets are simple nets with only one token per condition. They are a special case of bipartite graphs. With place/transition nets, more than one token per condition is possible. Here, conditions are defined as places, and transitions

represent events. Predicate/transition nets are useful for large condition/events or place/transition nets since they are able to reduce the size of these nets. A famous example used for predicate/transition nets is the "dining philosopher's problem".

Thread based execution is defined within the Von-Neumann model. The Von-Neumann model is characterized by sequential execution. It is still vastly used in many designs since languages as C, C++ and Java are widely spread and many applications were already written in these languages. However, many languages do not provide communication and synchronization mechanisms for the execution on multiprocessor systems. As described in [6], the languages CSP and ADA have already built-in communication. In the remaining languages, the communication is provided by selecting different libraries. Java supports concurrency by using threads. The communication type can be selected by choosing different libraries. In Kahn process networks (KPN), the processes are also executed in a sequential manner. However, the emphasis lies in the communication while the details of the execution within the processes has less emphasis. In data flow languages, the movement of data has the most priority while in von-Neumann-languages the control-flow has more priority. The disadvantage of thread-based models is that they are not deterministic and that through the use of mutexes, deadlocks can occur. Here, the programmer has to take care of deadlocks properly.

Discrete event based languages are VHDL, Verilog and Sytem C. They are usually used in shared memory systems. In discrete event modeling, the events are sorted in a timeline queue by the time at which they are processed. The event is processed at its scheduled time, all corresponding actions are performed, and it is then removed from the queue. Sometimes the event enters new events into the timeline queue. A comparison over models of computations and languages is given in [6].

In a design flow, there can often exist a mixture of different languages or models of computation. An overview over the different models of computation and languages and their references to the different communication models is given in Figure 2.9.

Communication libraries

MPI, POSIX threads and OpenMP are among the communication libraries that exist for C/C++. As the name already indicates, the message passing interface (MPI) is used for message-based communication between processors. It allows asynchronous or synchronous message passing. POSIX threads is an application programming interface (API) for threads at operating system level. It provides procedures for thread management (create, join, etc.), synchronization between threads and also for mutexes. POSIX threads are used for shared memory hardware. "OpenMP is a specification for a set of compiler directives, library routines, and environment variables that can be used to specify shared memory parallelism in Fortran and C/C++ programs" [37]. Here, parallelism is expressed with pragmas and it takes the least amount of effort for parallelization for the user. For more information on the libraries, please refer to [6]. For all communication libraries, the programmer has

Communication/ Organization of components	Shared memory	Message passing	
		synchronous	asynchronous
Undefined components		Plain text, use cases	
		(Message) sequence charts	
Communicating finite state machines	StateCharts		SDL
Data flow	(not useful)		Kahn networks SDF
Petri nets		C/E nets, P/T nets, ...	
Discrete event (DE) model	VHDL, Verilog SystemC	(Only experimental systems) Distributed DE in Ptolemy	
Von Neumann-model	C, C++, Java	C, C++, Java, ... with libraries	
		CSP, ADA	

Figure 2.9: Overview over MOCs and languages considered [6]

to define parallel processes or threads and take care for the accurate communication and synchronization.

2.3 Mapping Problem description

This section describes the considered underlying architecture and application models of this work. Furthermore, the complexity of the integration of memory-awareness into the application to architecture mapping optimization is described in more detail. The underlying optimizations that are introduced in this work were mainly developed within the EU-project MNEMEE which is introduced in Section 3. The application and architecture model that is used in this work were specified in cooperation with all MNEMEE partners, based on their requirements and the tools that were already partly provided by these partners.

2.3.1 Architecture Model

The considered architecture model is based on the hybrid shared distributed memory model. Hence, a shared memory is available, which is accessible by all processors. All shared communication is performed on this memory. Each processor has also exclusive access to its own private memory where instructions and data can be allocated. This prevents conflicts on the bus as in the shared memory architecture, and it also prevents slow communication between the tasks as in the distributed mem-

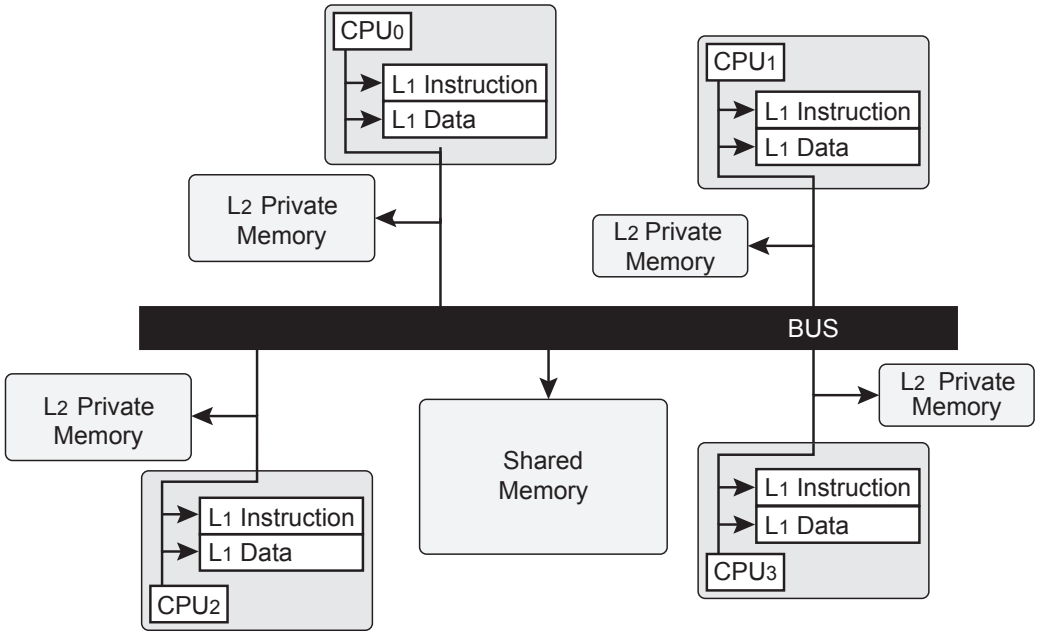

Figure 2.10: Heterogeneous MPSoC architecture with multi-level memory hierarchy

ory architecture. The architecture is already given and fixed and thus it does not have to be designed. The architecture has n processors, which can be homogeneous or heterogeneous, i.e., differ in type and/or clock rate. Each processor can have a distinct memory hierarchy, which can have different hierarchy levels and different sizes on each level. The local memories can differ in memory type (instruction only / data only / unified). Each memory has different energy and runtime values, which depend on type, level and size of the memory. Furthermore, the energy and runtime for bus accesses which lead to the memory have to be considered.

Figure 2.10 illustrates an example architecture. In this architecture, four processors CPU_0-CPU_3 are defined. Each processor has a separate local instruction and data scratchpad memory on level 1 and a private main memory on level 2. The processors have exclusive access to these memories. Furthermore, the architecture has at least one shared memory, which is accessible by all processors and which is used for inter-thread communication and synchronization. The shared memory can be accessed through an on-chip network or bus. Section 2.1 shows that this model is based on state-of-the-art MPSoC architectures.

2.3.2 Application Model

In the application model, it is assumed that an already parallelized application is given. A representation of the application in the form of an acyclic task graph is required for the mapping optimization. A thread-based application model as depicted in Figure 2.11 is used. Here, a main thread accomplishes computation,

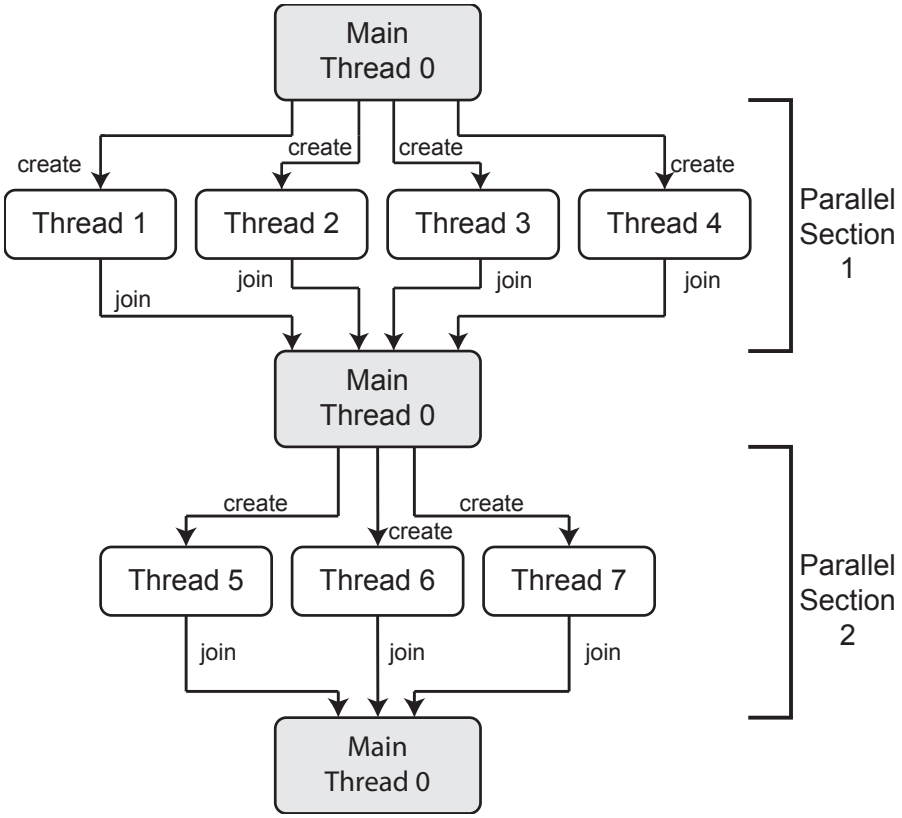

Figure 2.11: Thread-based application model

creates new threads and hereby initiates the parallel execution. The newly created threads can run in parallel and communicate via FIFOs. After they accomplish their computation, the main thread joins them and continues its execution. The section where newly created threads run in parallel is called *parallel section*. An application can have one or more parallel sections. An example is illustrated in Figure 2.11. Here, the application has two parallel sections with four threads in parallel section 1 and three threads in parallel section 2. Each thread, including the main thread, has to be mapped onto a processor of the underlying architecture platform. This application model can be generated from existing sequential C-code applications by using automated tools. More details on these tools are given in section 3.2.1.

Each thread consists of memory objects, which are either instruction code or data. They are mainly characterized by their size and the frequency of read and write accesses. The memory objects define the computation requirements of a thread. Depending on the memories they are allocated to, they essentially influence the performance and energy consumption of the system.

Based on the characteristics of the application, our model can also contain FIFO (First-in First-out) queues for inter-thread communication as depicted in Figure 2.12. In this case, a thread can be composed of several thread nodes. The edges be-

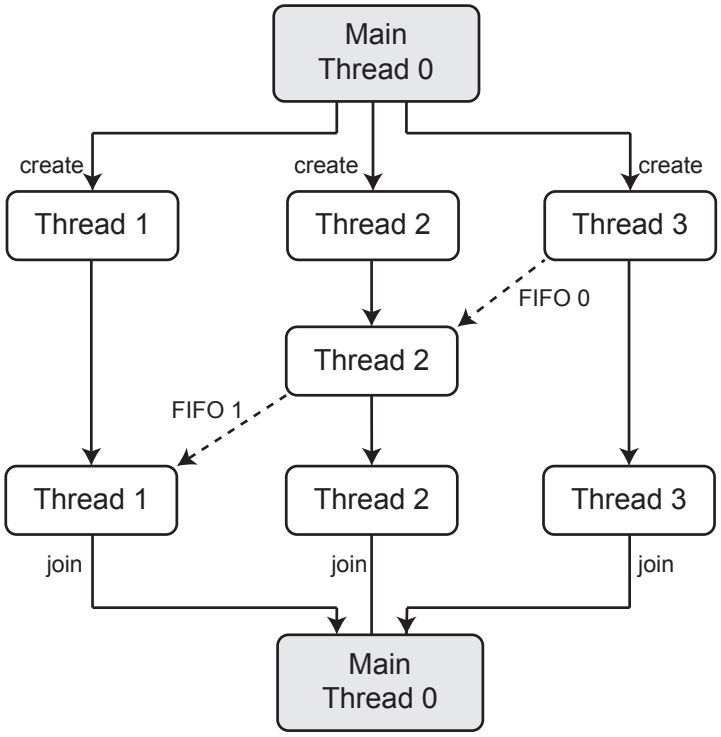

Figure 2.12: Task graph including FIFO communication

tween the different thread nodes illustrate either FIFO (i.e., communication) edges or control flow edges. As shown in Figure 2.12, the second node of thread 2 cannot proceed its execution until thread 3 has written something into the FIFO. This specification ensures a more precise determination of runtime since a thread node cannot be executed until all predecessor nodes have finished their execution or communication, respectively.

2.3.3 Mapping Complexity

State-of-the-art application to MPSoC mapping tools perform the mapping of threads to available processors, while optimizing a single or several objectives. The mapping optimization problem is known to be NP-hard even for homogeneous multiprocessor systems [38]. Even when considering just the threads to processors mapping, the number of options is significant. Considering in addition the memory hierarchy, and therefore explicitly the mapping of memory objects onto memories, increases additionally the options available and the problem complexity. Now, not only the processors but also the memories are crucial for the mapping of a thread, and different combinations are possible. Mapping multiple threads to one processor comes with additional constraints such as checking if enough memory capacity is available for all mapped threads. Furthermore, the memory objects of a thread can drastically contribute to the overall performance and energy consumption, if the

frequently accessed memory objects are mapped onto a fast and energy-efficient memory. Thereby, not only the processor frequency but the combination of processor frequency and the capacity and speed of the processors underlying memory hierarchy has to be suitable for the mapped threads and their individual resource requirements, i.e., memory objects. Obviously, a large number of design decisions has to be taken during memory-aware mapping optimization, such as: on which memory should the memory objects of a thread be placed? Also, if several threads are mapped onto a processor, the limited size of the underlying level-1 and level-2 memories has to be shared among several threads. Here, the question is: which thread gains the most benefit in terms of runtime or energy and should be thereby placed in the fastest memories? The mapping of *all* threads to the underlying resources have to be optimized in order to get the desired results based on the optimization goal. Briefly, which 'processor/memory' pair is most suitable for which combination of possible 'threads/memory objects' mapping while optimizing the overall energy consumption or system performance (or both)?

2.4 Related Work

The mapping of application tasks/threads onto available processors is a well-known optimization step for multiprocessor systems in embedded system design. The integration of memory awareness into this step was not really well investigated up to now. However, the step of mapping memory objects onto memories is also a very well-known optimization with several approaches. The mapping of tasks/threads to processors and the mapping of memory objects to memories are optimizations that are usually performed separately. This is due to the fact that single processor systems were state-of-the-art for a long time and these systems did not require a mapping of concurrent tasks onto different processors. Nowadays, multiprocessor systems are state-of-the-art, resulting in more complex embedded system design, especially when considering heterogeneous multiprocessor systems. Usually, the different design and optimization steps for embedded system design (i.e. parallelization, mapping, etc.) were considered as standalone steps. Recently, (semi-) automated frameworks were introduced, which provide all important design-flow steps in order to help the designer to cope with the huge and time-consuming design complexity. These frameworks usually also integrate the optimization step of mapping tasks/threads onto processors. However, not all frameworks consider memory optimizations.

An overview of approaches which perform the mapping of memory objects to memories is given in section 2.4.1. The optimization step of mapping concurrent tasks onto processors and frameworks, which also integrate this step are introduced in section 2.4.4. Finally, section 2.4.6 describes previous approaches, which introduce a combined mapping of task/threads onto processors and memory objects to memories.

2.4.1 Mapping of memory objects to memories

The mapping of memory objects to memories is a research topic that was vastly explored in the last years. In the beginning, only instructions were mapped to small scratchpad memories. Afterwards, more complex problems were solved including data and stack memory objects, dynamic allocations, scratchpad sharing strategies and also cache optimizations.

The first part of this section gives an overview over the research for single core systems with an emphasis on scratchpad allocation strategies. The second part presents the research for multicore systems.

2.4.2 Single Core Systems

For single core systems, several optimizations for different memory objects and different optimization goals exist, e.g. performance or energy minimization. Memory objects can be categorized into instruction and data where data can be either global (shared) or local data. Further memory objects which are important and more challenging are stack and heap data, as well as large arrays. Scratchpad allocation for stack and heap data are challenging because they grow during the execution of the application and the determination of their size is a very difficult task, especially for scratchpad or cache memories where the memory size is constrained. The constrained size is also a problem for the allocation of frequently accessed large arrays.

The first memory mapping optimizations were static, i.e. the mapping is performed before the execution of the application and is not changed during execution. One of the first paper which introduced scratchpad memories was [39]. A knapsack algorithm was used for the assignment of code and data blocks to scratchpad memory. The authors compared scratchpad memories against cache memories and showed that the scratchpad memory occupied less die area and consumed less energy than cache memories.

In [40], the authors performed an energy optimization for both instruction basic blocks and data (global scalar and non-scalar variables). Here, the energy per instruction basic block and data was formulated as a knapsack problem where the allocation of these memories objects to the scratchpad memory results in a gain of energy consumption. With integer linear programming (ILP) an optimal solution was obtained where on average 22% of the energy consumption was minimized compared to a system with a cache memory.

In [41], the work of [40] is extended by integrating the allocation of arrays next to instruction and data memory objects. Here, large arrays are partitioned into smaller segments whenever it is beneficial so that this part could fit into the scratchpad memory. All arrays are considered step by step. The algorithm decides whether an array should be partitioned or not in order to reduce the energy consumption. If so, a splitting point is chosen, which leads to a maximum reduction of energy consumption. The decisions if an array should be split and the choice of the splitting point

of the array is solved by ILP. A reduction of 5.7 to 17.6 % in energy consumption was achieved.

An architecture including separate instruction and data caches along with a scratchpad memory is given in [42] and [43]. Here, the scratchpad memory stores instruction memory objects which are chosen by an allocation algorithm. The cache behavior is represented as a conflict graph. A conflict edge is present when two memory objects are mapped to the same cache line. The goal is to minimize the number of conflict edges and thus the energy consumption. A combined effect of scratchpad memories together with caches on the systems energy consumption is considered in this work. For this problem, an ILP formulation [42] as well as a greedy heuristic [43] are presented. In the evaluation, the results are compared to loop-caches as counterparts of scratchpad memories. The presented approach outperforms the loop-caches. The average reduction of energy consumption is given by 20.7%.

Up to now, only allocation strategies were presented considering a system with one single process application. In [44], the authors consider multiprocess application where the scratchpad memory can be shared among several processes. Three allocation strategies are proposed: non-saving, saving and hybrid. In the non-saving strategy, the scratchpad memory size is divided among all processes, i.e the memory consists of disjoints region and it is allocated only at the beginning of the program execution, i.e static allocation. The saving strategy considers overlapping, where the scratchpad memory content is changed for each process. The hybrid strategy is a combination of the saving and non-saving strategy, where a part of the scratchpad memory is used as an overlapping region and the other part is used with disjoint region. The hybrid approach gives the maximum reductions in energy by 27% to 45% compared to a single process approach. On average a reduction of 9-20% is achieved for all proposed strategies. The authors showed that the saving strategy is better for small scratchpad memories while the non-saving strategy is better for larger scratchpad memories.

The following approaches perform dynamic allocation.

In [45], only instruction memory objects are considered. The goal is the reduction of the energy consumption for a single process application. This strategy uses ILP. The energy consumption can be reduced by 30%. However, a reduction in runtime by 25.2% on average was also observed. Compared to a static approach an improvement of 38% is given. In [46], the goal is to reduce the energy consumption with the allocation of instruction memory objects onto scratchpad memories. The difference to [45] is that this approach is based on a heuristic which is more appropriate for large-size applications than an ILP. This strategy is compared to a system with an instruction-cache and a 64% improvement in energy consumption was achieved.

An optimization for runtime is introduced in [47] for global data, stack and heap memory objects. This strategy works for memories, which are available on more than two levels in the memory hierarchy. The evaluation is performed for a Motorola MCore which has three levels of memories. For large programs, the stack is split into two separate memory units. Now, two stack pointers exist which have

to be incremented in both memories. On the other side, the heap is allocated for the first *malloc* calls in the scratchpad memory and afterwards it is allocated in DRAM. The threshold is determined by profiling. The heap size is estimated and multiplied by a safety factor of 2. A dynamic allocation of the scratchpad memory is used, i.e. the content of the scratchpad memory is changed during runtime. For this reason, an overhead for the additional copying of memory objects to the scratchpad memory is also considered in this approach. It is useful for large applications which contain several hot spots which cannot all fit in the scratchpad memory. A 0/1 integer linear program is used for this strategy and it is optimal for global and stack data. A heuristic is used for the heap allocation. On average, the runtime is decreased by 39% for this strategy.

Another dynamic allocation approach for instruction memory objects for single process applications was introduced in [48]. This approach introduces an ILP and a heuristic approach for energy minimization. The average deviation between the optimal and the heuristic solutions are less than 6% for processor cycles and energy. For the ILP-based strategy, an average reduction of 23.4% in energy and 7% in execution time was achieved. This strategy was also compared to a system with a preloaded loop cache. Here, the energy was reduced by 29.4% and execution time by 8.7%. Furthermore, an energy reduction of 40% and an on-chip area reduction of 75% was achieved compared to a system with the most energy-efficient instruction-cache configuration.

In [49], the goal is a runtime reduction with the allocation of global data and stack memory objects to a scratchpad memory. Here, an analysis is performed, which indicates which program point is executed at which time. Then, a cost model for the transfers during runtime is set up where a greedy compiler heuristic computes the maximum overall runtime benefit. Compared to a static allocation, a reduction in runtime by 31.2% is achieved. Another heuristic for runtime reduction for global data, heap and stack is introduced in [50]. It is an extension of the strategy in [49] and allocates the heap to scratchpad memory. Here, not all elements of the heap are stored in the scratchpad memory, but only a subset with a fixed size. With this method, the runtime is reduced by 34.6% and power by 39.9% compared to a scenario where only global data and stack is placed to the scratchpad memory and heap to DRAM only.

The authors in [51] integrate a scratchpad manager in the operating system (RTEMS) which dynamically allocates data and instruction (i.e. functions) memory objects to the scratchpad memory. Their approach determines which memory objects are most efficient for the allocation to the scratchpad memory. Several heuristic methods are integrated and compared to each other. Also, an ILP based approach was implemented for evaluation purposes. The set of memory objects can change at each context switch. The methods were implemented for a multiprocess application system with the goal to minimize the energy consumption. Depending on the implemented methods, a 5 to 60% deviation to the optimal ILP solution was observed. Comparing all strategies against a system without a scratchpad memory, the approaches reduce the energy consumption from 40 to 120%. Also, runtime

reductions were achieved by 38 to 115%.

An overview of the presented publications for single-processor systems is illustrated in Table 2.1.

In [52], data assignment and scheduling is performed for multi-layer memory architectures. The application is given as a task graph together with a description of the memory structure with constraint programming. The memory structure is given with two layers: one scratchpad memory on the first level and one dual SDRAM with two banks and two pages on the second level. The approach consists of three steps, which can be repeated iteratively. The first step produces a Pareto diagram for the whole application based on scheduling estimates and incomplete constraints. This is performed with a branch-and-bound heuristic. The second step performs a data assignment to the underlying memories. The goal is to allow parallel access to the memories for data, which has to be accessed at the same time while considering the bandwidth of the memories. In the last step, a scheduling optimization is performed for the tasks. The evaluation was performed for different feedback and iteration strategies and shows the obtained Pareto points.

Next to the software or compiler-based allocation of memory objects to a scratchpad memory, a hardware-based technique is also possible. Here, several memory ranges of the address space are mapped to the scratchpad memory. Additional logic overhead is required for the address decoding. Furthermore, a partitioning of the scratchpad memory into several physical banks is a step which saves power consumption: while one bank is accessed the others can be turned off. But, additional banks cause an overhead in die area, wiring and decoder complexity. Therefore, an optimal partitioning has to be generated. In [53], such a scratchpad memory partitioning is performed. The authors suggest a dynamic programming algorithm for either energy reduction or speed improvement in a homogeneous multiprocessor platform. The execution time is polynomial in the input size. For this, a trace based approach is used, which obtains the number of accesses for each memory range. Code, heap and stack memory objects are considered. As input, the scratchpad memory size, area and an estimation of the hardware overhead for the partitioning of the scratchpad memory is required.

2.4.3 Multiprocessor Systems/MPSoCs

Research was also performed for the mapping of memory objects to the memories of multiprocessor systems. Here, different multiprocessor architectures have been investigated including different memory characteristics. For example, the considered architectures can be shared or distributed memory architectures or Graphics Processing Unit architectures (GPU), network-on-chip (NOC), CELL etc. For all these architectures, the scratchpad allocation problem has to be examined in different ways since the interconnections to the scratchpad memories and therefore the access possibilities are manifold. Even for one considered architecture model, there can be different memory access specifications. For example, in some shared memory architectures a scratchpad memory can only be accessed by one processor.

	[47]	[45]	[40]	[41]	[49]	[43]	[42]	[44]	[46]	[50]	[48]	[51]
Memory objects												
Instruction		■	■	■		■	■	■		■	■	
Data	(global)		■	incl. arrays	(global)			■		(global)	■	■
Stack	■				■					■		
Heap	■									■		■
Static allocation			■	■		■	■	■	■	■		■
Dynamic allocation	■	■			■			■			■	■
Single process	■	■	■	■	■	■	■		■	■	■	
Multiprocess							■				■	
Optimal	global, stack	■	■	■		■		■	■		■	■
Heuristic	heap				■		■			■	■	

Table 2.1: Overview of scratchpad memory allocation publications

In other shared memory architectures the scratchpad memory can be also accessed by several other processors. For multiprocessor systems, different and more complex strategies are required than for single-core systems. Tasks, threads or memory objects have to be distributed efficiently over the processing units and memories in the system. Also, optimizations for heterogeneous multiprocessor systems increase the complexity drastically since the processing units and memories have different characteristics.

All these architecture configurations result in many different scratchpad memory strategies for many different architectures (or memory configurations). Here, a few approaches will be introduced in order to show the manifold possibilities for different architectures.

The authors of [54] suggest an integration of the scratchpad memory allocation into OpenMP. The designer has to mark arrays, which are frequently used and are therefore good candidates for the allocation to a scratchpad memory. The profiling, allocation and implementation are integrated into OpenMP. The approach can also split arrays if required. The authors consider a shared memory architecture where processors have also access to the scratchpad memories of the other processors. [55] is an extension of [54] where data allocation is automatically performed.

In [56], the allocation to scratchpad memories for message passing in distributed shared memory architecture was implemented. Here, the communication between processors is performed on scratchpad memories instead of shared memories, which results in a speed-up of the system. They also provide hardware support for the direct communication of the scratchpad memories between each other. Due to the limited size of scratchpad memories, only small messages can be sent. For this reason, the processor has to communicate more often. However, the authors show that it is still more beneficial than shared memory communication. This hardware/software approach provides different communication modes, from single word access to burst access. If the size of the scratchpad memory is too small for the message, it is copied to private memory, which is still faster than shared memory. Integer semaphores are used for synchronization. They represent the number of free messages in the queue. The semaphores are distributed among the processing elements in order to reduce communication traffic.

The authors of [57] introduce an integer linear programming (ILP) and heuristic approach, which maximizes the throughput of stream programs. For this, the scratchpad is used for code and communication data. Both techniques integrate the overhead for code overlay and communication. The application is given as a synchronous data flow graph (SDF). The considered architecture is the IBM Cell Broadband engine [58] where the considered processing units have a scratchpad memory with a size of 256 kB (16 kB in experimental results). Both techniques are compared against an ILP approach (called SGMS) which does not include the overhead costs for code overlay and communication. The proposed ILP and heuristic technique outperformed SGMS. For smaller scratchpad memory sizes of 16 kB, SGSM could not find solution for four benchmarks because the scratchpad constraints for the communication buffer was violated.

In [59], the architecture model is a homogeneous multiprocessor system with off-chip DRAM and several processors which have access to their own scratchpads as well as to the scratchpads of other processors (remote scratchpad) by fast on-chip communication links. The access to remote scratchpad memories takes more cycles than to local scratchpad memories, but is still less time consuming than an access to off-chip memory. The input is a loop-level parallelized application. The approach performs a compiler-based optimization for interprocess communication. In a first step, the sizes of the data tiles have to be determined. Afterwards, the access pattern matrix of the data tiles are determined for scheduling as well as for the elimination of extra off-chip memory access. The authors achieved an energy reduction of 1.4 to 22.4%.

The authors of [60] introduce virtual scratchpad memories (vSPMs). Here, the scratchpad memory space is virtualized for security reasons. In Android systems this strategy can avoid malware to access sensible data by exploiting software, e.g. with buffer overflow etc. The applications are not known in advance as in other strategies. It is a multi-tasking environment. The virtualization is realized by a hardware IP block which is similar to an arbiter (called HardSPMVisor). The authors provide an API for the utilization of the vSPM. Only data is assigned. The architecture consists of a set of RISC processors and distributed scratchpad memories and an AMBA AHB on-chip bus with secure DMA.

[61] is an extension of [60] which also reduces execution time. Here, also non-volatile memories are considered as on-chip memories because they also reduce leakage power. But their drawback is the high cost for write operations. Therefore, hybrid on-chip memories were proposed [62] which show up to 37 % [63] reduction in leakage power. A minimalistic API is also provided for the programmers for the allocation of instruction and data. They present a compiler-driven allocation strategy where the designer can annotate the placement for data in scratchpad memories and also preferably instructions into non-volatile memories. This allocation of memory objects can also be additionally performed by a static analysis which is performed in the second step. Afterwards, the compiler generates allocation policies. Based on these policies, a dynamic algorithm decides for the best memory utilization. The hardware and architecture are the same as presented in [60], but include also non-volatile memories.

The authors of [64] proposed a heuristic for variable partitioning (allocation of data to scratchpad memory) and scheduling for virtually shared scratchpad architectures. Here, the processors have their private scratchpad memories but are also allowed to access the scratchpad memories of other processors. The goal is to minimize the runtime. Two variable partitioning heuristic strategies are proposed: High Access Frequency First (HAFF) and Global view prediction (GVP). Also a loop pipeline scheduling algorithm for the (already) parallelized tasks is presented. Note, that the allocation of certain data (variables) can reduce the execution time of a task and thus influences the schedule solution drastically. Therefore, a loop optimization is performed where two phases are alternating: scheduling and allocation (remapping) are performed until all variables are assigned or until the number of

unassigned variables is greater than the space left in the scratchpad memories.

The approach presented in [65] generates several different allocation solutions for data including arrays, local variables and stacks. The architecture consists of several homogeneous processors with private and shared scratchpad memories. Shared scratchpad memories are used for data that has to be accessible by two or more processors. The authors assume that the application is already parallelized. An algorithm partitions the problem into an uniprocessor allocation problem and afterwards merges the solution (set of buffers) into a hierarchical set of buffers. Each buffer can be mapped to a scratchpad memory. The solution shows possible synchronization choices. Local variables and stacks are always mapped to level-1. Buffers are calculated by the analysis of loops and accesses to arrays. The smallest buffer is mapped to level-1 if it fits, the rest is mapped to private or main or shared level-2 memory. The results are Pareto optimal points (energy vs. buffer size). For each point, the total memory subsystem energy consumption is calculated.

The authors of [66] suggest a scratchpad allocation for the optimization of the worst-case response time (WCRT). The allocation is dynamic, i.e. different overlays are generated for tasks with disjoint lifetimes. Here, the processors have only access to their own local scratchpad memories and one off-chip shared memory is given in the system. Code is allocated to the scratchpad memories as well as data memory objects, which have a static number of accesses. The architecture can consist of one to four identical processing elements. Each processing element has one scratchpad memory. The sizes of scratchpad memories are equal for all processing elements. The input is an application modeled as message sequence chart (MSC). A preemptive multitasking environment is assumed. Processes can consist of several tasks, which are mapped to a processing element. The authors suggest and investigate four different schemes, which are performed after a worst-case response time analysis. Schema 1 is a profile-based knapsack problem formulation, schema 2 is an interference clustering, schema 3 a graph coloring problem formulation and schema 4 a critical path interference reduction. After the execution of a scheme, a post-allocation worst case response time analysis is performed in order to update task lifetimes since through allocation and optimization the start and endtime of tasks can be different as well as their interferences with other tasks. All schemes were able to achieve gains from 20.6% (schema 1), 24.9% (schema 2), 45.3% (schema 3) to 52.3% (schema 4). The runtime of all schemes varied from 10 seconds to less than a minute.

Finally, the authors of [67] perform a static data allocation for logical buffers. A fixed task-to-processor mapping is already given as a basis. The mapping is performed on a complex multicore platform containing 80 processing elements. About 1000 buffers are mapped onto this platform with a mixed-integer linear programming (MILP) approach.

To sum up, all introduced publications investigated different scratchpad allocation strategies where the focus lies on different architectures or optimizations (e.g. worst case). None of these publications have investigated a combined view of task/thread mapping and multi-level memory mapping.

2.4.4 Mapping of tasks to processors

Mapping tasks onto processors in MPSoCs is a topic vastly explored in recent years, and various approaches for different architectures targeting different objectives were proposed. The mapping of tasks onto homogeneous processors is usually performed through different scheduling strategies. However, when the processors differ in some smaller characteristics, other optimization strategies have to be considered, e.g. for processors with dynamic voltage scaling (DVS). This is especially valid for a mapping onto heterogeneous processors, where the processor's clock rate and type (e.g. DSP) are important factors in the optimization.

Previous work considered homogeneous platforms with different configurations for power management like dynamic voltage scaling (DVS) [68], real-time requirements [69], dynamic scheduling / mapping [70] (i.e. mapping decision during runtime), as well as a combination of those [71]. Heuristics, stochastic methods, or evolutionary algorithms are handling these complex system optimizations. Among others, heterogeneous systems were investigated in [72, 73, 74]. In [72] a multi-heuristic evolutionary algorithm is provided for a dynamic task mapping. The authors in [73] propose a multilevel graph partitioning and mapping approach for the minimization of the systems runtime. Population-based metaheuristics with greedy and random strategies for dynamic task allocation is presented in [74]. All of them are considering distributed heterogeneous systems.

However, none of these approaches considers memory mapping nor integrates the influence of memories on investigated objectives. Generally, it is assumed that either the system has enough memory to cover all application requirements or the influence of memories on system execution time, and energy consumption is usually abstracted in order to reduce the complexity of the optimizations.

2.4.5 Design Frameworks

Recently, many frameworks for hardware and software synthesis were introduced, which integrate all important design steps and thereby help the designer to reduce the complexity. They also include the step of mapping tasks onto processors. In the following, an overview of well-known design frameworks is given.

HOPES is a programming environment for the design of embedded systems with focus on software for MPSoC systems [75]. The software development framework performs four important steps. In step one, the system behavior is specified with model-based programming. Here, three different models of computation are used. A task model defines the execution condition and communication requirement of the given tasks. The tasks are specified in more detail with an extended synchronous data flow graph (SDF), called SPDF, for signal processing and computation tasks. If control tasks exist, they are modeled with a hierarchical, concurrent FSM model. Two task communication methods are supported: shared memory and message queue. A mapping of the task models to processors has to be performed manually by the designer and in HOPES it is assumed to be given. Based on these

mapping results and the initial specification model, a common intermediate code (CIC) is generated. CIC uses generic API functions in order to express I/O operations from tasks. Furthermore, Open-MP specifications are used to express data parallelism in tasks. The goal is to have an independent software platform and communication architecture with CIC. This model can then be ported to different architectures if required. The memories in this model are specified with size and address range. Also, it is specified which process has access to a shared memory. Now, an optimized code generation for each processor is performed. The generic APIs can be translated into OS APIs of the processors or into communication APIs if no OS is specified. The data parallelism expressed by Open-MP is translated into MPI (Message Passing Interface). Afterwards, a task scheduling code generation is performed. These steps are performed in an automated way. In the last step, verification is performed including static C code analysis and a performance and power estimation. The static code analysis is detecting error locations as for example memory access errors (buffer overrun, memory leak, etc.). Furthermore, run-time errors are detected by running the program on a virtual prototyping system.

Daedalus is an automated framework for MPSoC platforms at system-level which is depicted in Figure 2.13 [76, 77]. In this framework, the designer begins with a sequential application and at the end a MPSoC implementation on an FPGA is generated. In the first step, the sequential application is translated into a concurrent Kahn Process Network (KPN) by the tool KPNgen. For this, the application has to be given as a "static affine nested loop program" which contains a set of statements that can be enclosed in loops and/or by conditions (also called polytope model) [78]. KPNgen is able to generate different input-output KPNs, e.g. with a variable amount of parallelization. This parallelization can be exploited by design space exploration (DSE). Furthermore, the platforms are specified with a library of pre-defined IP components, which include programmable processors, hardwired IP cores, memories and interconnections. The platform is produced as synthesizable VHDL. Data is communicated through distributed memories and each memory unit can be organized as one or several FIFOs Daedalus uses a lightweighted multithreading operating system (MTOS) for runtime scheduling of processes that are executed on a processor if a compile-time scheduling is not possible The system-level DSE is performed within the Sesame modeling and simulation environment [79]. Three optimizations are performed within the design space exploration including synthesis: 1. computing the *allocation* of resources for the architecture, 2. mapping processes to these resources, also called *binding*, 3. temporal *scheduling* of processes/communication on the resources. These are the usual steps for frameworks, which include architecture design or synthesis, respectively.

The objectives are: 1. minimize the maximum processing time including the time spent on memories as well as the time spent on communication, 2. minimize the power consumption of the whole system (memory included), 3. minimize the total cost of the architecture model. A quick evaluation for these steps is performed through fast high-level simulations. Next to the design space exploration through high-level simulation, Sesame also supports heuristic search, i.e. genetic algorithms

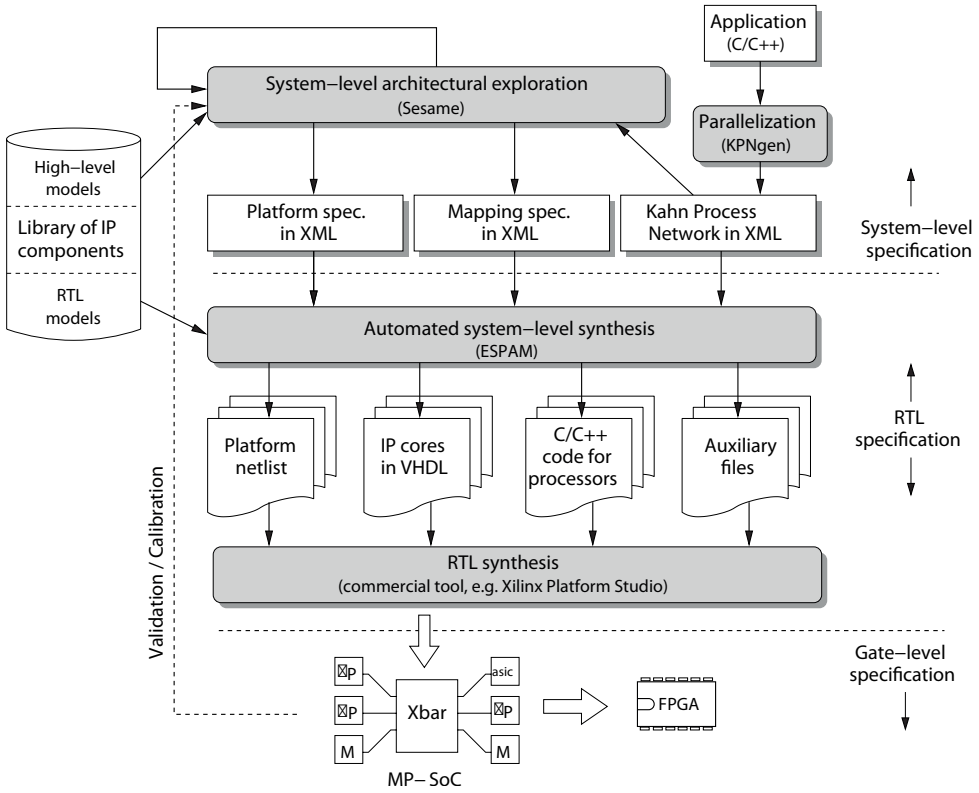

Figure 2.13: The Daedalus Design Framework [80]

for larger design spaces. In an additional step, the design space is trimmed based on analytical models (design space pruning). Design space pruning is performed by collecting knowledge on the platform architectures. This information can be used to guide the designer to select a platform, i.e. select an initial well-known platform or by neglecting platforms, which do not fulfill the requirements. The result of the DSE step are a set of candidate system designs including the process binding that are described in XML. These XML descriptions and the KPN description are required as input for the ESPAM tool. This tool generates synthesizable VHDL which implements a MPSoC platform. Furthermore, C code is generated from the KPN processes which are mapped to the programmable cores. Finally, this implementation can be realized on a FPGA for prototyping.

SystemCoDesigner is another design framework including an automated mapping of System C applications onto a heterogeneous MPSoC platform [76, 81]. The application has to be written in SystemC and represents an actor-oriented application model. Communication is performed through SystemC FIFO channels implemented in one thread. This kind of input can be transformed into a subset of SystemC (called SysteMoc), which is able to represent non-deterministic data flow models (DDF). Here, each actor is defined by a finite state machine and communicates with queues in FIFO semantics. Afterwards, these actors can be transformed

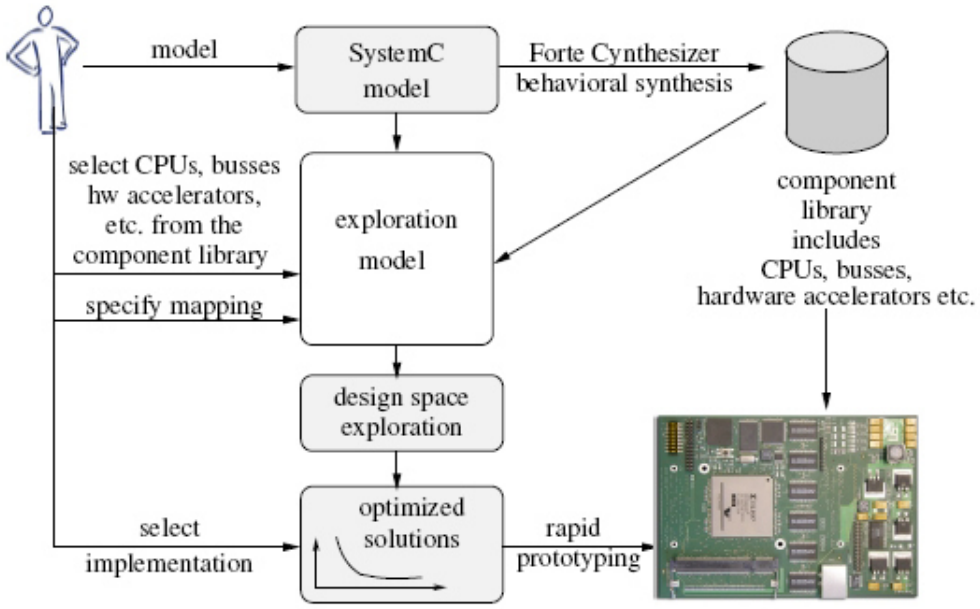

Figure 2.14: SystemCoDesigner Design Flow

into hardware and software modules. The software modules are generated by code transformations while the hardware modules are built with the tool Forte Cynthesizer [82]. The hardware accelerators are automatically generated and stored in a component library. This component library also contains IP cores, i.e. processors, memories, buses, etc. Now, the designer has to specify a heterogeneous MPSoC architecture by connecting cores from the library. Additionally, the designer also has to define mapping constraints for each actor. In the next step, design space exploration is performed. Here, the implementation of the application is optimized while considering several objectives (e.g. throughput, latency, area and power consumption). The simulation is based on the fast "task accurate performance model" as in Daedalus (i.e. high-level simulation). The output of the design space exploration is a set of optimized solutions where the designer can choose a solution. Afterwards, a rapid prototyping of the FPGA implementation can be performed. Here, the designer performs many steps manually. Only some steps, e.g. design space exploration, are performed automatically. The SystemCoDesigner design flow is illustrated in Figure 2.14

The distributed operation layer (DOL) [18] is an automated system-level framework with the goal to efficiently map parallelized application tasks onto heterogeneous MPSoC systems. DOL was mainly developed in two projects. The first development started in the Scalable Hardware/Software Architecture Platform for Embedded Systems (SHAPES) project. Here, this version will be called *basic DOL*. DOL was extended in in the European Reference Tiled Architecture Experiment

(EURETILE) project. This framework provides design space exploration with evolutionary algorithms for multiobjective optimization.

2.4.5.1 DOL (SHAPES)

The input of the DOL framework is an application and an architecture specification. The application specification is specified as an already parallelized application given as a process network model. Each process has to be additionally provided in C/C++. Furthermore, a description for the processes and all FIFOs for the communication (i.e. software-channels) are defined in the application specification which is defined in XML. All processes and software channels have input and output ports for the interconnection. Furthermore, the communication is defined via blocking read/write. The architecture specification is also given in XML. Here, all processors, memories and their interconnections are defined. Performance data is added for the bus throughput and also delays on communication paths as well as processor and bus clock frequencies. A sharing method for the resources (i.e. scheduling) can also be added if required. Furthermore, profiling information has to be added to the application specification, as the number of process invocations, the runtime on the different target processors, etc. For this, a simulation or other ways of profiling can be performed and added to the application specification file.

The design space exploration is performed with a loop with two main phases: the mapping optimization and the performance evaluation. The mapping optimization is performed with multiobjective optimization. The designer has to specify the objectives and integrate them within DOL, e.g. minimizing energy consumption, balance runtime, reduce bus traffic. The mapping optimization creates a population of individuals (chromosomes). These individuals represent a mapping solution and are evaluated based on the objectives. New individuals are generated by crossover and mutation. The optimization is based on the evolutionary algorithm SPEA2 [83] and the PISA interface [84]. The evaluation is performed with the EXPO framework [85]. It evaluates all crucial information as performance and energy. All these values are passed to SPEA2 which selects good individual candidates and communicates this back to EXPO. This is performed in a design space exploration loop until a stop criterion is reached (i.e. maximum number of generations). The output of the optimization is a set of Pareto optimal solutions. These solutions are described in a mapping specification where they define the binding of the processes to the processors and the binding of software channels to the communication paths. A scheduling policy is also defined, but not included in the optimization.

The evaluation of the individuals is usually performed by analytic performance analysis. It is also possible to integrate a cycle-accurate simulation or static or dynamic models as the modular performance analysis (MPA) for real-time analysis. For this, the DOL specifications can be converted into models for modular performance analysis (MPA) The advantage of the analytic or high-level functional simulation is the fast simulation which is usually used in this class of frameworks (e.g. also in Daedalus). A fast evaluation is required because of the huge design

space and because a quick evaluation has to be performed for the numerous mapping solutions which are generated in several loop iterations. Cycle-accurate simulations are very time-consuming and extend the evaluation and design time drastically.

2.4.5.2 DOL (Extended)

DOL was extended by design architectures during the mapping optimization. For this, computation templates are provided which include different processors (DSP, micro controller, etc.) and FPGAs, etc. Furthermore, an automatic selection of the computation templates as well as of communication techniques (different buses, rings) and scheduling (TDMA, EDF, etc.) is provided. The input is given by a set of task graphs and use-cases. Also, executive platforms are specified as architecture graphs. The output also provides the description of the execution platform [6]. The mapping optimization, including evolutionary algorithms, is based on the same strategy as in the previous version of DOL.

To sum up, several frameworks for MPSoC design space exploration with multiple objectives exist, like for instance Daedalus [77], SystemCoDesigner [86], HOPES [75], or DOL [18]. They are based on different applications, architectures, mapping models, different evaluation environments, different strategies to search in the design space, different optimization criteria, design constraints, or abstraction levels. All frameworks have their own characteristics and cannot be compared to each other.

2.4.6 Combined mapping to processors and memories

The integration of memories in the task mapping optimization is a new investigation field which started to gain attention in the last years. Only few research was investigated in this new field.

The authors of [87] perform an allocation of data (also arrays) to scratchpad memories with the polyhedral model Thus, through the polyhedral model also a parallelization is performed which maps computation to processing units. A GPU architecture with multilevel parallel processing units is considered in this work. The scratchpad memories are shared among several processors in the architecture. Multi-level tiling is performed where computation is distributed on multiple levels of parallel units. Data is allocated block-wise to scratchpad memory. In a first step, the approach calculates which data space (array references) is beneficial for allocation. In a second step, the data space is partitioned into disjoint, non-overlapping sets. Here, the bounds are determined and set in a way, which does not exceed the capacity of the local memories. This means, the parallelization is adapted to the sizes of the scratchpad memories in this case. The sizes of the scratchpad memories of each processor are always equal. For the scratchpad memory utilization, a speed-up of 8x for MPEG4 and 10x for Jacobi kernel was achieved compared to a system which only uses GPU DRAM.

The approach in [88] proposes a heuristic which performs task assignment, static

scheduling and allocation of code and data to local memories. Additionally, a final system architecture is generated with the primary goal to minimize the data utilization on the memories. The second goal is to decrease the schedule length by scheduling tasks from the critical path. These are conflicting goals because the memory requirements are usually increased by the tasks which lie on the critical path. In a first step, an estimation on the data memory usage is performed. For this, the task's requirements as the task's start time, task duration and amount of code, data and communication required are considered. Also, the tightest possible schedule which has a higher data memory requirement is integrated in this estimation. In a second step, a cost function is set up which represents the cost of implementing a task t_j on processor P_i where also measurements on the amount of code required, and the time needed to execute a task on processor are integrated. A task is assigned to a processor according to the cost function. As input, an acyclic task graph is required, which is annotated with the estimated execution times of the nodes and the memory requirements of the tasks. The arcs in the task graph represent data transfers. These communication arcs have also to be scheduled and assigned to memory. Preemptions of tasks are not allowed. Data memory has also to be reserved for task communication. Here, more overhead can be caused when tasks are assigned to different processors. In this case, the communication is also scheduled and assigned to communication devices. The architecture can contain ASICS, processors, buses and links. Processors have a separate local data and code memory, and ASICs have one data local memory. The system architecture is modeled by constraint logic programming and is fixed.

In [89], a constructive algorithm is proposed. In a first step, the algorithm chooses a task from the critical path in each iteration. Only tasks can be chosen, where all predecessors were already scheduled. The costs are determined for implementing the task on each processor. This cost mainly depends on the data usage, the communication cost for interprocessor communication and the execution time together with the code memory utilization and the processor's time slots utilization. The task is assigned to the processor with the minimal cost. Also, the communication is scheduled with the minimal communication cost. Furthermore, the data memory is assigned for the data of the task if the memory has enough capacity. If there is not enough capacity in the memory, the mapping decision is reverted and another task is chosen in step one. Their approach is compared to a greedy non-memory-aware scheduling algorithm which also chooses the tasks from the critical path. The memory-aware approach always outperformed the greedy algorithm in terms of runtime and memory utilization. Also, the greedy algorithm was often not able to find schedules when the memory size was decreased. The solution can be used by designers for synthesis. The architecture is identical to the architecture in [88].

The authors in [90] present a partial task assignment (PAT) technique, which is a heuristic. New constraints are added, which limit the possible mapping of tasks. All tasks within one group have to be mapped to the same processor. The goal of this method is to reduce the complexity of the task mapping and scheduling. The sys-

tem architecture, the application and synthesis problem is modeled with constraint logic programming as in [88],[89]. The input is an acyclic task graph with estimates of the task's execution time and code and data requirements. The PAT algorithm assigns the tasks to processor time slots and communication tasks to bus time slots. It tries to utilize the resources as time and memory evenly. The decisions are made based on an estimate of the future usage of resources. The next task that has to be scheduled either increases the critical path length or the data memory usage. The experimental results compare PAT to a simple MATAS algorithm. MATAS has no partial assignment constraints as the PAT algorithm. Therefore, there is no complexity reduction and the full optimization potential cannot be utilized compared to a clustering algorithm as PAT. But, the same metrics as in PAT are used for making clustering decisions. The PAT algorithm always gave better results than MATAS.

[91] use integer linear programming (ILP) to find task mapping, pipelined scheduling, and scratchpad memory partitioning for MPSoC. Here, each processor has its own local scratchpad memory but also has access to other scratchpad memories (remote scratchpads). In this case, partitioning defines the allocation of memory objects to the local and the remote scratchpad memories. During the scratchpad partitioning, memory requirements for each task are calculated. If a task requires more local memory, the scratchpad memories of other processors are partitioned and used. Here, only static data allocation is performed. The multiprocessor system is homogeneous where all memory sizes of the local scratchpad memories are equal. The integration of memory optimization within the task mapping and scheduling was able to improve the performance by 2% up to 80% dependent on the benchmark.

[92] present a memory-aware application mapping for coarse-grained reconfigurable array (CGRA) architectures. CGRAs consist of an array of processing elements (PEs). The goal is the mapping of operations (i.e. loop kernels) to the processing elements and of array data to local memory. A heuristic considers several memory characteristics as number of banks, local memory size and communication bandwidth between local and system memories. Arrays can be shared between processing elements and also be duplicated in different banks of a memory.

Contrary to [88],[89],[90], [92] and [91], this thesis does not perform system synthesis nor scratchpad partitioning. Compared to these approaches, this thesis considers the entire memory hierarchy with several levels and different memory types and sizes. Furthermore, automatic design space exploration and multiobjective memory-aware optimization are also an integral part of our work.

Chapter 3

MNEMEE

Contents

3.1 Introduction

Modern embedded systems have to fulfill more and more requirements. Generally, their focus lies on multimedia and communication applications (e.g. WiFi, HD-Video, coding, etc.). These kind of applications perform a huge amount of computation and data transfers, and they usually also have to meet real time constraints. Nowadays, MPSoCs are the most suitable platforms for these kind of applications because they provide fast computation and a complex memory hierarchy for the data transfers and communication. The designers have to find an efficient mapping of the application onto the limited resources of the system, i.e. processors and memories. For this task, several steps have to be performed by the designer. First, the sequential application has to be parallelized into several tasks that can be mapped to the different processors. Furthermore, the static and dynamic data memory objects have to be efficiently allocated to the memory hierarchy. The huge design space has to be explored in order to obtain a design that fulfils all demands for energy consumption, runtime and memory footprint. Due to the short time-to-market, the designer requires an automated tool flow, which helps the designer to optimize the source code and explore the design space efficiently.

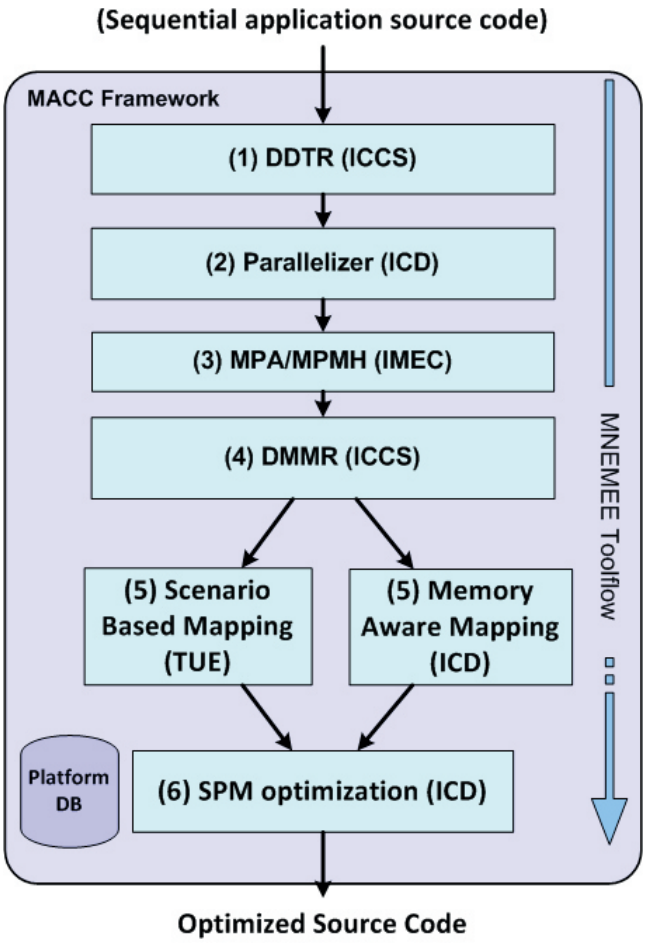

Figure 3.1: The MNEMEE Toolflow

The MNEMEE project [12, 13] addresses all these challenges and introduces an automated tool flow which performs source-to-source optimization with all required steps from parallelization, synchronization, design space exploration up to the mapping of the code and data to the MPSoC platform including state-of-the-art optimization methodologies. The focus lies on the efficient utilization of data memory objects to the available memories in the memory hierarchy since the memory subsystem is usually neglected in the design flow. However, the efficient utilization of the memory hierarchy is directly linked to the performance and energy consumption of the system and has to be optimally exploited. The objective of the MNEMEE project is to achieve a considerable reduction of the energy consumption and the design time of the system. Depending on the designer's underlying system and its requirements, the designer has the possibility to use either the complete MNEMEE tool flow or a part of it.

The MNEMEE project started January 2008 with a duration of 36 months and was funded by the European community The partners included in this project have

all different areas of expertise, which contributed perfectly to the MNEMEE tool flow. Except for the industrial partners, all partners have developed stand-alone tools for individual use, which were extended and integrated into the automated MNEMEE tool flow.

The partners involved in the MNEMEE project are:

- Interuniversity Micro-Electronics Center (IMEC, Leuven, Belgium) - Research Center
- Institute of Communication and Computer Systems (ICCS, Athens, Greece)
- Informatik Centrum Dortmund e.V. (ICD, Dortmund, Germany)
- Universiteit Eindhoven (TUE, Eindhoven, Netherlands)
- Intracom S.A. Telecom Solutions (ICOM, Peania, Greece)
- Thales Communications (TCF, Colombes, France)

3.2 The MNEMEE toolflow

This section describes the complete MNEMEE tool flow and all tools that are integrated in this tool flow. An overview over the complete MNEMEE tool flow is given in Figure 3.1. The underlying architecture and application model is described in Section 2.2. The input of the tool flow is sequential source code in C. The output is parallelized and optimized source code for the target MPSoC architecture. The optimizations are mainly focused on the memory subsystem. All tools automatically generate all required optimizations. The designer has the option to set different settings for the individual tools if required.

3.2.1 The MACCv2 Framework

The MACCv2 framework is a system modeling approach for source-level and memory-aware optimization development for multiprocessor systems [14]. In the MNEMEE tool flow it integrates all tools into one single tool flow. The designer can use a template for optimization and analysis tool development and an interface for the connection to the other tools in the tool flow. Through this interface, it is possible that one tool is providing an optimization decision and another tool is implementing it. Within MACCv2, an ordered execution of the optimization/analysis steps can be specified. All tools can be configured and controlled by tool settings. Furthermore, a common platform model is provided which also models the memory hierarchy in structurally and semantically accessible way. Here, optimization designer are equipped with a database-like interface to the system model. In addition, all required architecture information can be obtained from the system model.

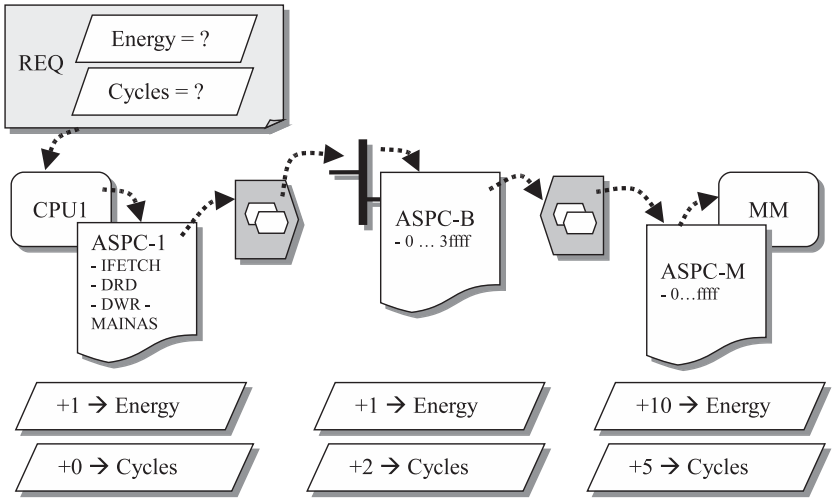

Figure 3.2: The Routing Model within MACCv2 [14]

The architecture description model is a scalable and structural Processor-Memory-Switch-level (PMS) system model. Furthermore, interfaces to backend tools can be provided to the designer (i.e. compilers, linkers, simulators). MACCv2 also has a uniform application representation. The code is represented as an ICD-C based abstract syntax tree. With the ICD-C code representation, the program code can be attached to processing units. In this way, the application code is integrated into the system model. The designer can also use an Eclipse based graphical user interface for system modeling.

The primary goal of MACCv2 is to enable rapid development of source-level architecture independent optimizations. However, the given architectural properties can be still utilized in order to achieve higher optimization gains. Another advantage is the knowledge about the properties of the memory subsystem. Through the query-based interface of the system model, MACCv2 is able to provide immediate results without the requirement of a time-consuming simulation (Figure 3.2). It has been shown that the provided method for accessing the system and the memory hierarchy provides the same quality results as a simulation based approach in modeled MPARM SoC [14].

3.2.2 DDTR Tool (ICCS)

The *Dynamic Data Type Refinement Tool (DDTR)* optimizes dynamic data structures in the source code [93]. Dynamic data structures include dynamic arrays, linked lists and trees. For this, all dynamic data types (DDTs) are profiled in order to obtain their access patterns and their allocation behavior. Based on this access pattern, the implementation of the dynamic data structures are changed with the help of components from the DDTs library. The objective is to increase the perfor-

mance and limit the memory consumption. As an example for the restructuring and its effect on performance, let us have a look at a linked list, which accesses items sequentially. Let us assume that the last items of this list are accessed frequently. A restructuring of this data structure can add an extra pointer to the end of list and thereby increase the performance. The restructuring of the data structures can have effects on the extraction of the parallelization. Thus, this tool is executed before the parallelization tool.

3.2.3 Parallelization Tool (ICD)

The *Parallelization Tool* [94] was developed by ICD within the MNEMEE project and is illustrated in Figure 3.3 The input of this tool is the modified source code that is passed from the DDTR tool. For the automatic extraction of the parallelization, the application is remodeled as a hierarchical task graph. On every level of the task graph, an ILP approach searches for possible parallelism and decides if new tasks are extracted on this level or if earlier extracted tasks on another level in the hierarchy should be maintained. In this way, the whole task graph is processed by a bottom-up search. The goal is to obtain a balanced runtime by executing concurrent tasks on several processors of the system and thereby speed up the execution of the application. The parallelization tool can extract functional parallelism as well as data parallelism or a combination of both. The tool can also limit the number of extracted parallel tasks to the number of available processors in the system. It is also able to extract several parallel sections, i.e. the parallel sections can be executed at different points in the application and in each parallel section, several tasks can be executed concurrently. The results are annotated in the source code, and a parallelization specification is generated. This specification is an interface between the parallelization tool and the subsequent MPMH tool. It specifies all parallel sections, the number of tasks in each parallel section and the type of parallelism for the chosen code segments.

3.2.4 MPMH (IMEC)

The MPMH tool by IMEC performs two different steps in the tool flow [95]. The first step is the implementation of the extracted parallelism by transforming the source code. The second step is the optimization of static array data by generating smaller blocks of these arrays, which can then be mapped to the size constrained memories in the memory hierarchy of the system. This tool only allows C-Code with single-assignment pointers. As input, the parallelization directives, which were specified by the parallelization tool, are required. Otherwise, if the designer wishes to extract the parallelism manually, a parallelization specification has to be provided as well. Based on this specification, MPMH automatically extracts parallel source code by adding synchronization mechanisms and FIFO queues, if required. The tool generates correct-by-construction parallel code. In the next step, the *MPMH Tool* optimizes the access to static arrays. An analysis is performed on array data

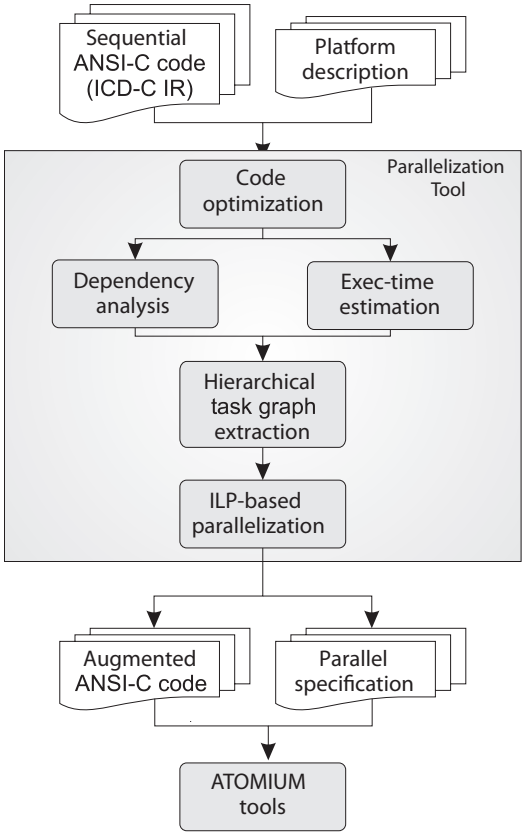

Figure 3.3: The Parallelization Tool [94]

at compile time, and profiling is used. With this information, the tool decides if array data copies should be extracted and mapped to certain levels in the memory hierarchy. The goal is to minimize the energy consumption and runtime. The output of the tool is the parallelized and synchronized source code added with the data copies of the arrays. The source code also contains all required block transfers that need to be performed to the different memories. In the MNEMEE tool flow the data copies are extracted by the MPMH tool, but the mapping to the memories is performed by the *Scratchpad Memory Allocation Tool* which obtains hints from the *MPMH Tool* for the memory mapping. The *Scratchpad Memory Allocation Tool* is described later in this section.

3.2.5 DMMR (ICCS)

After the extraction and implementation of the parallelization, the *DMMR Tool* by ICCS extracts a mapping of dynamic data structures to the different memory layers in the memory hierarchy. For this, a profiling based analysis is performed. The allocation and deallocation timeline for all dynamic objects is extracted. This timeline is analyzed, and the most frequently accessed dynamic objects are identified.

Based on this access information, the tool decides which dynamic objects should be placed at a higher level in the memory hierarchy and which at a lower level. These decisions are passed as hints to the *Scratchpad Memory Allocation Tool*, which finalizes the mapping of data memory objects to the memories.

3.2.6 Thread Model Extraction Tool (TUE / IMEC / ICD & TU Dortmund)

The *Thread Model Extraction Tool* was designed and implemented with the help of several MNEMEE partners, i.e. TUE, IMEC and ICD in cooperation with TU Dortmund. It is an important interface between the parallelization tools and subsequent the mapping tools in the MNEMEE tool flow. Both class of tools work on different internal models and different information. For example, the parallelization tool works on hierarchical thread graphs while the mapping tools require a flat thread graph annotated with certain information for their optimizations. For this reason, the *Thread Model Extraction Tool* automatically extracts a flat thread graph with all dependencies and all required information from the parallelized application source code. This alleviates the work of the designer, since the complex and time-consuming manual extraction is avoided. In a first step, the tool annotates markers to each basic block and extracts a profiling file, which includes the order of executed basic blocks, thread creations and joins, as well as FIFO reads and writes. Based on this information, a flat thread graph is extracted, which includes all dependencies and parallel sections. In a second step, the number of cycles for each basic block as well as all information on memory objects (i.e. number of reads/writes, size, number of executions, etc.) for each thread is extracted and annotated to the thread graph. This thread graph is passed to the mapping tools, which are executed, in the next step of the MNEMEE tool flow. The thread graph is generated in a way which satisfies the input requirements of the Scenario-Aware and Memory-aware mapping tools. In addition, no further profiling has to be performed by the mapping tools, i.e. the requirements for the scenario-aware graph can also be extracted as well as all memory object information for the memory-aware mapping tool. This prevents double profiling work for the same required information. More detailed information on the *Thread Model Extraction Tool* is given in Section 4. Please note that the tool is not illustrated in the MNEMEE tool flow in Figure 3.1. Within the MACCv2 framework, it is configured as a mandatory precedent step, which has to be executed before the mapping tools.

3.2.7 Mapping Tools

The mapping tools are mapping parallelized threads onto the available processors in the MPSoC system. The designer has the choice between two alternative techniques in the MNEMEE tool flow, which are described next. The *Scenario-Aware-Mapping Tool* has a focus on the dynamic behavior of the application and tries to save resources. On the other side, the *Memory-Aware Mapping Tool* has the focus on the

memory subsystem with the goal to find an optimal mapping of threads to processors while matching the memory requirements of the threads to the speed and energy consumption of the available memories in the memory subsystem.

3.2.7.1 Scenario-Aware-Mapping Tool (TUE)

The *Scenario-Aware-Mapping Tool* was developed by TUE [96]. It was extended in the MNEMEE project. This mapping tool considers the dynamic behavior of the application that can occur at different points in the application. These different dynamic points are called scenarios. A scenario is static, i.e. predictable in performance and resource requirements. For the input of the tool, a set of synchronous data flow graphs is required, which can be obtained from the already parallelized source code. In the MNEMEE tool flow, synchronous data flow graphs can be easily generated from the information of the thread graph that was generated by the *Thread Model Extraction Tool* for this purpose. Here, each graph represents a scenario. The goal is to find an efficient resource allocation for each scenario while maintaining timing guarantees when switching between scenarios. The resource allocation includes the allocation of processing (i.e. threads), memories and communication resources. The output of the tool are several trade-offs for the amount of resources for processing, memories and communication. In the MNEMEE tool flow, the mapping, which minimizes the memory usage, is selected and passed to the subsequent tools that implement this mapping. As a standalone tool, it can use a run-time mechanism in order to adapt the mapping of the application during runtime for different use-cases. However, an implementation of a run-time library for this purpose was not implemented since it was not a goal in the MNEMEE project.

3.2.7.2 Memory-Aware Mapping Optimization Tool (ICD / TU Dortmund)

The *Memory-Aware Mapping Optimization Tool* provides a mapping, which minimizes energy consumption and runtime while including the memory subsystem in the optimization process. As input, a flat task graph extracted from parallelized code is required and detailed information on hardware (e.g. memory speed) and application software (e.g. thread execution time and information on memory requirements of threads). All this information is provided by the task graph of the *Thread Model Extraction Tool*. An evolutionary algorithm with the multiobjective goal to minimize the energy consumption and runtime is used for the mapping optimization. An allocation of code and data memory objects to the memories in the memory hierarchy is also performed in this optimization. The tool analyses the task's memory requirements and matches them to the available memory resources in the system while considering the characteristics of the processors (i.e. speed) as well as the characteristics of the memories (i.e. speed and level). This kind of optimization can be very efficient for heterogeneous MPSoC system with different processors and a heterogeneous memory subsystem. However, in the MNEMEE tool flow the

tool passes the mapping of threads to the processors to the remainder of the tool flow. The final implementation of the memory objects to the memories is finalized by the *Scratchpad-Memory Allocation Tool*. As a standalone tool, it is also able to provide the mapping of memory objects to the different memories in the memory hierarchy. More detailed information on the *Memory-Aware-Mapping-Optimization Tool* is given in Section 6.

3.2.8 RTLIB/RTEMS (IMEC/ICD)

A Run-Time Library (RTLIB) API is provided by IMEC in order to specify the platform-dependent part of the application, e.g. thread creation, communication and synchronization. Furthermore, for the execution on a platform an embedded operating system is required. In the MNEMEE/MACCv2 tool flow, RTEMS is implemented as operating system and connected together with RTLIB. The thread mapping decision of the mapping tools is finalized in the source code by this RTLIB/RTEMS step.

3.2.9 Scratchpad Memory Allocation Tool (ICD)

After the mapping of threads to the processors, the *Scratchpad Memory Allocation Tool* decides about the allocation of data to the scratchpad memories of the different processors. The goal is to allocate frequently accessed data objects to the fast scratchpad memories and to thereby reduce the energy consumption and runtime of the system. The optimization is performed with a knapsack-based approach that is implemented as an ILP. As a result, a non-overlaying allocation is presented, which is implemented by source-to-source transformations. Earlier tools have provided hints on the mapping of data structures on memories. This tool decides on the final allocation for data objects.

3.3 Achieved Results

The industrial partners applied the MNEMEE tool flow to their chosen application and implemented their own architecture in the MACCv2 framework database. The target platform for Thales is a OMAP L137 architecture by Texas Instruments which includes an ARM and a DSP processor [97]. The application is based on NATO standard STANAG 4591 implementing enhanced Mixed Excitation Linear Predictive (MELP) algorithm/vocoder at 2400, 1200 and 600 bit/sec, i.e. a speech signal. Intracom Telecom implemented the MSC8144 processor as target platform [98]. It includes high-performance multicore DSP from Freescale for wireline and wireless (infrastructure) applications. Each core has a speed clock of 1GHz. This architecture is optimized for voice, fax, video and data compression processing. An internal QUICC Engine dual-RISC packet processor supports multiple networking protocols. Their application is the Mobile WiMAX IEEE 802.16e, which is a broadband wireless solution. The MNEMEE project showed that the tool flow can also be used in

industrial design tool flows. The industrial partners were able to achieve a reduction in design time by up to 76% and a reduction in energy consumption by up to 52% [12, 13]. The industrial partner Thales (TCF) achieved a reduction in memory footprint by 30% and Intracom (ICOM) was able to achieve a reduction in memory bandwidth by 17%.

Chapter 4

Thread Model Extraction

Contents

4.1 Introduction

The complexity of embedded systems has increased in the past years due to significant improvements in hardware and the corresponding consumer demand. As a consequence, designers are confronted with time-to-market constraints and depend on effective design tools, which take the burden from the designer to perform the complex design steps manually. However, these design tools work at different levels of abstraction. For example, mapping tools, which map application tasks to the different processing elements of the system, work at a high level of abstraction. They require an architecture and application specification as input. At a high level of abstraction, application specifications are often described as task graphs. Task graphs represent the execution of the application. They are generally used for the optimization of complex problems, which can be solved more efficiently at this higher level of abstraction (e.g. mapping and scheduling of parallel tasks). Unfortunately, the application is usually not given in the form of a task graph but in a high programming language, i.e. as C code. On the other side, the majority of optimization tools

expect a task graph as input. Designers usually have to manually transform C code into a task graph, which is a time-consuming, complex and error-prone job. Depending on the complexity of the application code, this job can take hours or even days. This section focuses on automating the translation from C code to a task graph in order to overcome all these obstacles and reduce the amount of developing time.

In the overall tool flow of the MNEMEE project, described in Section 3, several design steps are performed. The overall goal is to provide an automated tool flow, which helps the designer to perform all these single steps automatically. Here, each tool works internally with its own application models. Moreover, the tools require a particular input format, which specifies the information that is needed in order to perform the tools optimization, process properly. However, combining different design tools lead to the problem that the output of the first tool does not usually match the information that is required for the successive tool. Furthermore, the level of abstraction for the various tools differs.

The *Thread Model Extraction Tool* was set up within the MNEMEE project. It is a chain between the parallelization tools and the mapping tools. The output of the parallelization tools is parallelized code, where the parallelization and synchronization are specified through functions and annotations that are defined within the source code. On the other side, the required input of the mapping tools is a flat task graph, which describes the dependencies between threads given by the control flow and FIFO communication (i.e. data dependencies or synchronization). Internally, the parallelization tools work also with information about control flow dependencies and other profiling information. However, it was not planned to export this information as an output for other tools. Moreover, the different tools do not work on the same models, information and level of abstraction. For example, while the parallelization tools work internally on hierarchical task graphs, the mapping tools work with flat task graphs. A translation from a hierarchical task graph to a flat task graph is a very complex and time-consuming step, which would also involve a deeper modification of the parallelization tools. Furthermore, the parallelization tools work on certain information extracted by analysis and profiling. After the extraction and implementation of parallelism, a different set of application information is required for the mapping tools compared to the parallelization tools. For example, the extraction of memory object information is required. Therefore, a separate tool had to be developed, which extracts all required information in an automated way.

4.2 Related Work

First research on the extraction on task graphs was performed by the parallel computing community for massively parallel machines as in [99, 100, 101]. The authors of [102] proposed an approach for embedded system synthesis tools and the authors of [103] an approach for matlab code.

In [99], the authors developed a tool called FAST (Functional Algorithm Simu-

lation Testbed). The goal of the tool is the simulation of large parallel systems with large problem sizes and a focus on computationally hard scientific applications. The applications are implemented on message passing multiprocessors. The input is a sequential, user-annotated parallel program where the user annotates the parallel execution and communication. In the front-end, the tool generates a static task graph including operations and data transfers. The computation is transformed to an intermediate language format with IR-operations and send/receive primitives. The IR-operations are the basic computational blocks while the send/receive primitives represent the data-dependencies between the IR-operations. For each IR-operation, the execution time is obtained by manually counting the corresponding machine instructions. Afterwards, a parser transforms the intermediate format into a weighted task graph in a data-flow manner. Edges represent receive and send data primitives annotated with the size of the communicated data. The back-end maps the task graph onto the parallel architecture. First, the task graph is mapped to an idealized architecture where the number of processors corresponds to the number of tasks. In the next step, a clustering is performed by a heuristic where the communication overhead is minimized without sacrificing the parallelism. In the last step, the clustered parallel task graph is mapped to the architecture with the help of heuristics that use different modifications of priority list scheduling. Furthermore, a load-balancing approach is implemented which sorts the clusters by their load and assigns the cluster with the highest load or processing time to different processors.

In [100], the authors suggest a task graph generator for parameterized task graphs. Parameterized task graphs represent very large task graphs in a compact way and are well suited for coarse-grain parallelism. The parameterized task graph contains a set of symbolic tasks, which are represented by a name and an iteration vector. It also contains a set of communication rules that describe the data that is transferred among tasks. The communication rules model how tasks send or receive data and the dependencies between tasks occurring by this communication. The input is an annotated, sequential program where the designer has annotated task definition directives. The tool extracts source code and computational load of tasks as well as dependencies and communication load between tasks. Dependence analysis is used in order to detect the dependencies between tasks. The authors assume that the dependence analysis is performed by the designer by using an appropriate tool. Afterwards, the communication and synchronization rules are derived from these dependencies. The computation load and communication volume are represented in a problem size independent way. The computational load is computed by the compiler, which sums up the estimated elemental arithmetic costs for each statement. The communication volume for an edge is symbolically computed by the sums of polynomials.

The authors of [101] extract task graphs for large-scale parallel applications on large parallel systems. They are using a combination of analytical, simulation and hybrid model. The considered applications are including MPI directives, which are given from a *High Performance Fortran* program. The goal is to generate a static, symbolic task graph. Additionally, the authors also propose an approach that derives

a dynamic task graph from a static task graph, based on the use of code generation from symbolic integer sets. For the generation of the static task graph, four steps are required where compiler techniques are used. In the first step, the computation and control-flow nodes are generated. A new task node is generated for each *IF/THEN, PROGRAM/FUNCTION/SUBROUTINE,STOP/RETURN*, all other subsequent statements are comprised to one node. In the second step, communication tasks are generated for each logical communication event. The MPI communication directives (i.e. wait-recv, wait-send, isend, irecv) are determined and the task can only continue its execution, after it has received its data. The task synchronization between is also captured in this step. The third step generates symbolic sets and symbolic mappings, where the compiler constructs symbolic scaling functions for each task and communication event. The scaling function of the task is given by the workload of the task, while the scaling function of a communication event is given by the message size. For loops (i.e. DO-nodes), the scaling function describes the number of iterations executed by the processor. The scaling function for functions represents the processor id variables and other symbolic program variables. The last step eliminates excess control flow edges between tasks.

If a less detailed task graph is sufficient for a given model, the compiler can merge adjacent nodes, which are connected by a control flow edge and which do not contain any communication. The scaling functions of the tasks are also considered and treated appropriately by differentiating different cases of these functions. This step generates a more coarse-grain model. The authors also propose a technique for the generation of a dynamic task graph from the static task graph, if required. The static task graph represents a single execution for a particular input. The dynamic task graph has no control flow edges, is acyclic and only regular, non-adaptive code is considered where the task graph does not depend on intermediate computational results. For the generation of the dynamic task graph all Do-nodes are unrolled and the dynamic instances of the branch nodes are resolved.

Contrary to [99, 100], the annotation of the designer for the specification of the parallelism and the communication are not necessary in this work as the parallelization and synchronization are already performed by other tools in the MNEMEE tool flow. Furthermore, this work does not target massive parallel systems and thereby different kind of applications (e.g. streaming applications) is considered. The target of the presented task graph tool is C code for homogeneous or heterogeneous MPSoC systems in the embedded system design field. All steps within this tool are performed in a fully automated way and without any user intervention. Furthermore, we additionally extract detailed information about instruction and data memory objects and a separate architecture specification, which contains information on access times and energy consumption for different memories. Combining both information, the subsequent mapping tools are able to provide a more precise analysis on timing and energy consumption since delays due to memory accesses are integrated in this model. Our tool is also able to extract a compact task/thread graph, which reflects the average timing for large applications, or a detailed task/thread graph, which represents a more precise timing since all loops, and FIFO communications within

loops are unrolled.

In [102], the authors perform a task graph extraction for embedded system synthesis tools. These tools usually require a system specification input that is given in the form of a task graph. In the first step, their task graph extraction tool generates an abstract search tree (AST). The AST is traversed and searched for keywords. These keywords represent variable dependence relationships, program flow and control flow. Program flow keywords track the end of statements and function calls. Control flow keywords mark boundaries of loops or conditional statements. Then, the tool generates an ordered list of events with a combination of keywords and identifiers (functions/variables). In the next step, dependence graphs are determined based on this ordered list of events. Task graphs or dependence graphs are directed acyclic graphs. Therefore, loop and conditionals are turned into a single node where a loop or conditional start and end is captured by the corresponding keywords. Moreover, the tool detects assignment events and variable use and thereby determines dependence relationship between nodes. Furthermore, it generates new nodes for functions, which are not included in loops or conditional statements. For each function, the tool generates a separate dependence graph. The designer can replace the function call nodes with the appropriate task graphs. Some profiling data is required in order to complete the task graph. For this, the source code is annotated, compiled and executed. The annotations capture information about the size of data types and the execution time of each statement and function call. The size of the data types is required to obtain edge weights. Here, pointer variables get user defined size, since it is not possible to get the proper memory size. The execution time is obtained by the system call *gettimeofday()* This system call involves a small amount of error, but the authors claim that this is an insignificant amount of time (25μs). Most designers work on worst-case execution times and the authors suggest adding a safety margin of 20% on the extracted execution time (which they claim should be the common practice for system designers). Finally, combining the dependence graph and the edge weights leads to the final task graph. In the last step, the tool generates a function call graph for interprocedural analysis, which is used to determine dependencies among nodes containing functions.

The authors of [103] propose an automated task graph generation for matlab code of parallel applications. The task graph generation consists of four steps. Step one performs instruction analysis by splitting the instruction into tokens (primary expressions). Afterwards, the execution time of each instruction is calculated with the help of a prepared table, which contains the estimated execution time for matlab code. Step two assigns each instruction to a task. A task is one of the following items:

- a single instruction
- a function body
- one iteration of a single for-loop
- a condition body

For a nested *for-loop* the number of tasks is 2*N*M, where N is the number of iterations for-loop 1 and M is the number of iterations for the nested loop 2. The number of loop iterations has to be given in the code since no profiling is performed. The third step of this task graph generation is the dependency analysis of the tasks where edges are added for control flow and synchronization. In the fourth step, nodes are combined with their predecessor when certain dependencies are given in order to save communication costs between processors. The output of this tool is annotations that specify tasks and a list of tasks with their edges. This strategy is extracting the parallelism on its own.

Comparing [102, 103] to the *Thread Model Extraction Tool* presented in this chapter, the *Thread Model Extraction Tool* is working on sequential C code. Furthermore, it works on a thread-based application including already parallelized and synchronized C code, which is a different starting base. A different type of task/thread graph is extracted, which includes also FIFO communications and more detailed profiling information, e.g. memory objects (size, number of read/write accesses, etc.). The dependence analysis is performed by the ICD-Compiler. Contrary to [103], the *Thread Model Extraction Tool* does not consider single instructions as tasks but threads, which are defined as functions.

Two further interesting publications exist, which target task graph extraction but in different research fields and with different goals. The authors of [104] extract a control and data flow graph for VHDL in VLSI design while the authors of [105] extract a task graph during run time with the help of additional hardware for dynamic task scheduling.

4.3 Problem Description

An example graph of a thread based parallelization is shown in Figure 4.1. Since the underlying application model is a thread based model, the graph will be called thread graph. The challenge is to generate a flat thread graph in an automated way that is based on the given C-Code. Furthermore, various thread information (e.g. execution costs) has to be extracted and annotated to the proper thread nodes. Next to an application specification, the *Thread Model Extraction Tool* has also to extract the proper architecture specification which will be described in section 4.7.

The requirements for the input of the mapping tools for the application specification are given as follows:

- a flat thread graph including control flow edges
- data dependence edges (FIFO) between threads and information about the number of execution of each FIFO and the transferred data per execution
- profiling information about threads (e.g. execution costs and energy consumption for the different processors in the system, number of execution of each thread)

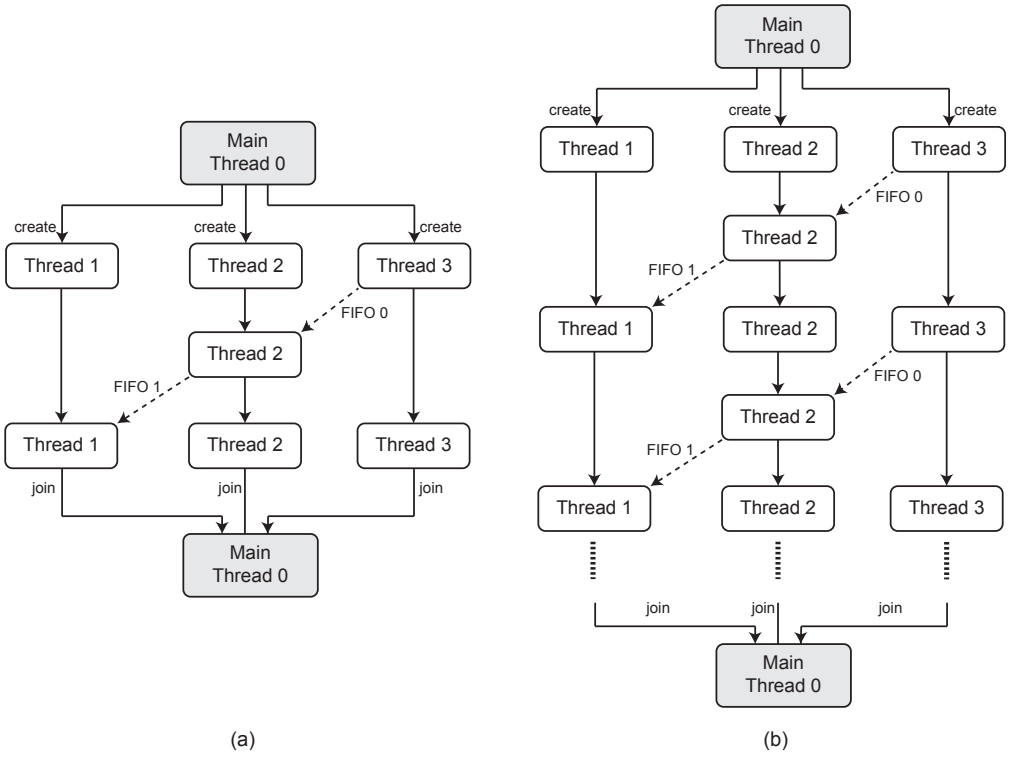

Figure 4.1: Compact and detailed version of a thread model

- extraction of all memory objects including information about the name, number of read and writes accesses, size, etc.

The mapping tools can only perform a proper analysis, optimization and assignment of the threads when all these demands are represented adequately in the thread graph.

The thread graph is static and a directed, acyclic graph. Nodes represent an operation that can be assigned to a processor and edges represent either communication between these operations or the control flow of these operations. Furthermore, the exact timing of FIFO operations must be represented adequately in the thread graph. For example, a thread can execute a part of its code and then wait for data that has to be communicated via a FIFO by another thread. The thread that receives the data can only proceed its execution after the sender thread has produced and send the data. This has to be captured in the thread graph in order to guarantee a proper analysis of the execution and waiting time of a thread, which is crucial for an appropriate mapping optimization.

The starting point for the thread graph extraction is the output of the parallelization tools. The MPMH tool [106] from IMEC implements the parallelization and synchronization with their Run Time Library (RTLIB). RTLIB functions ini-

tiate parallel sections and the creation and join of threads. Furthermore, the synchronization and data communication between threads is provided through FIFO communication.

4.4 Tool Overview

The designer can decide if a compact thread graph or a more detailed version is extracted. A part of the extraction of the thread model is based on profiling. Profiling is an important step in order to obtain essential execution and memory information. The *Thread Model Extraction Tool* consists of three steps. An overview of these steps is given in Figure 4.2.

The first step is a pre-processing step, which extracts timing information for each basic block in the code. It prepares the profiling analysis by adding profiling statements for each basic block. In the second step, the code is executed while all profiling statements are tracked. If FIFO communication is defined within the source code, it is tracked and written out in a separate profiling file. In the last step, a flat thread graph is generated based on the information given in the profiling file and additionally by compiler analysis. Furthermore, the thread graph is annotated with all required information, i.e. number of execution of threads/memory objects, run-time/energy requirement for thread execution, etc. In a separate step, the *Thread Model Extraction Tool* also processes all architecture information given in the MACCv2 database for the underlying platform and generates a proper output file which is also used as input for the mapping tools.

The *Safe-Annotation Tool* was developed internally by TU Eindhoven and performs the (first) pre-processing step. The High-Level time-annotated Simulator (HLSim), which was developed by IMEC, performs the (second) simulation step. HLSim performs a fast simulation for parallel applications at source code level and is also used in the MPMH tool described in section 3.2. It provides information about FIFO communication and thread spawning and joining. For the integration in the *Thread Model Extraction Tool* it was adapted in order to provide information about the execution of basic blocks by tracking the execution of the profiling statements that were included by the *Safe-Annotation Tool*. The last step extracts a thread graph from the profiling information that is provided by the previous steps. It additionally extracts further information on threads and memory objects. A more detailed description of all steps is given in the next subsections.

4.5 Safe-Annotation and Simulation

In a first step, the *Safe-Annotation Tool* annotates each basic block in the C source code with an estimate on the time needed to execute this basic block on the target processor. It does this by analyzing the assembler statements that need to be executed for all C statements inside the basic block. This whole process involves a few steps. First, safe-annotate compiles the application source code to assembler

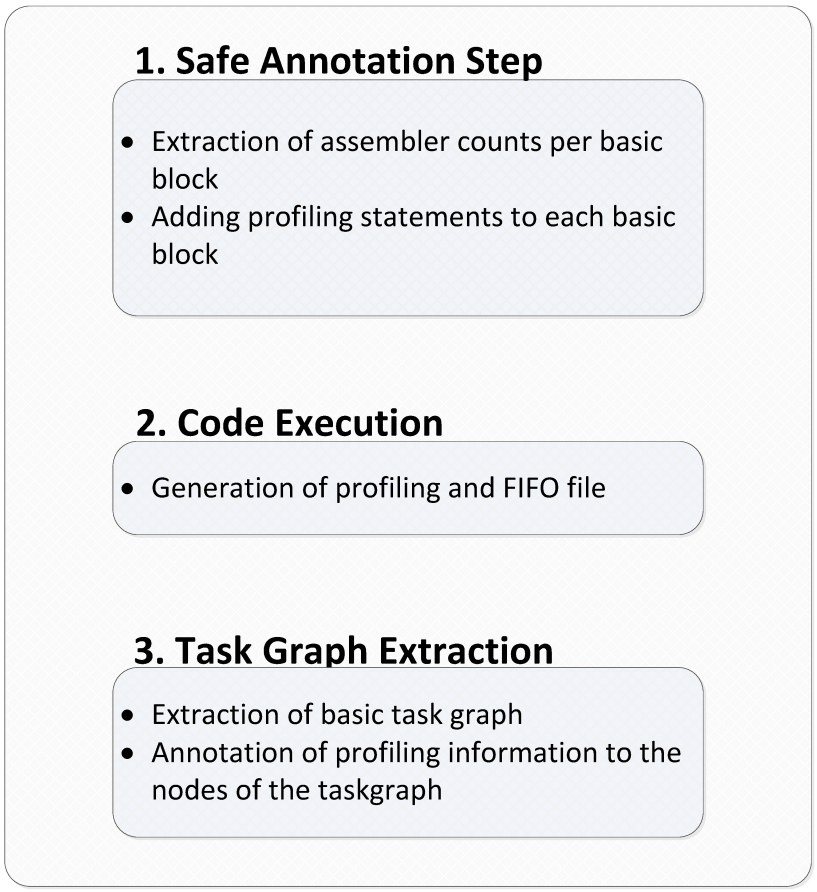

Figure 4.2: Tool flow overview for thread graph extraction

code by using a cross-compiler. Next, the tool analyzes the assembler code and it relates the assembler statements to the C statements by using debug information. In a third step, each assembler statement is analyzed and through a look-up table, its worst-case execution time is determined.

Afterwards, the code is annotated with profiling statements for each basic block including a basic block id for each basic block and its estimated worst-case execution time. Furthermore, the functions for starting and joining of threads as well as every FIFO read and write operations are also marked. With this information, it is possible to track which basic block or which thread/FIFO operation is executed at which time step.

In the second step, the HLSim Simulator by IMEC executes the code and generates two information files. One file contains general information about all existing FIFOs, which contain the FIFO id, its source and destination thread id, the size of one token (element) which is communicated, and the buffer size required for the

communication. The other file contains information about the timing order of each thread creation and join and the FIFO communication between all threads including the communicated data (i.e. number of elements). This profiling file reflects the execution order of the parallelized application code and is the basis for the extraction of the control and data flow for the thread graph.

4.6 Thread Model Extraction

4.6.1 Structure of the Thread Model

Before describing the different steps in the thread graph extraction, the overall thread graph structure has to be defined. The thread model exists of one or more threads and defines one start node and one end node. In our application model these nodes always belong to the main thread with the id *0*. As explained in the last section, a thread graph can consist of several parallel sections. A parallel section starts with the creation of several threads. After the concurrent execution of these threads, they are joined. Afterwards, a new parallel section could be started by the main thread. An example of an application with two parallel sections is shown in Figure 2.11 and Figure 4.5, respectively. A thread has to be mapped to one processor statically and does not migrate to another processor during its execution.

For each thread, we define:

- its id
- the id of the parallel section it belongs to
- a list of nodes that belong to the thread
- the number of executions
- list of memory objects of the thread.

The thread model also defines different types of edges: *CFG*, *CREATE*, *JOIN* and *FIFO* A *CFG* edge represents a control flow edge. A *FIFO* edge is a special edge type which contains additional information as the token size, maximum number of tokens communicated in one iteration and the buffer size. Furthermore, the id of the FIFO is also given. A *CREATE* edge is always set up between the main thread and its newly created child thread. The *JOIN* edge is set up between the last child thread node and the main thread node after a join. Both edges occur at the beginning and end of a parallel section. An overview over the different edges in a thread graph is also illustrated in Figure 4.1. For each edge, the number of accesses and the source and target node is specified.

FIFO edges are special and influence the setup of the thread graph. As already stated before, in a thread graph it is mandatory to illustrate the accurate point of time of FIFO operations. When a thread is reading from a FIFO at a certain time, it can only accomplish a FIFO read after something was written in the FIFO from the

source node. Only then, the thread can proceed with the execution of the subsequent basic blocks. This influences the run-time of a thread and has to be captured very carefully in the thread model. Thus, the thread has to be split into several nodes for an accurate control-flow illustration and timing analysis. Following rules are defined for the proper specification of FIFO operations: If a FIFO communication is performed, a thread consists of several nodes, which define the exact point of time of a FIFO operation as defined in the following. All nodes are executed in a non-blocking way. A FIFO write always occurs at the end of a node and specifies the last operation of this node. A FIFO read can only be specified at the beginning of a node. Only when the FIFO read is executed, the node can proceed with its execution. A FIFO edge has to be set up between the communicating thread nodes. Figure 4.1 illustrates FIFO communication and the proper interconnection between the communicating threads. This interconnection shows that node 2 of thread 2 can only proceed its execution after node 1 of thread 3 has executed the FIFO write operation. If several nodes exist, the nodes within a thread are marked with a node type. The first node of a thread is marked as a start node while the last node of a thread is marked as the join node. Otherwise, the node is marked as a standard node.

For each node, we specify:

- node id
- id of the thread this child node belongs to
- list of memory objects that are accessed within the node
- list of basic blocks that are accessed within the node
- access count
- overall execution costs (assembler counts)
- lists for incoming and outgoing edges
- list of FIFOs and the performed access mode of the FIFOs (read/write)

For each basic block, we define

- a specification of its id
- the numbers of accesses
- the number of assembler counts for one execution
- the number of assembler counts for the overall execution within its node

All memory objects have certain characteristics, which are defined as following:

- name and id
- type (instruction, data or shared)
- size and optionally basic size
- number of read and write calls
- access size (byte, halfword, word)
- mapping

If a memory object is an array, we also need to specify the basic size, which represents the size of one element in the array. This is important in order to determine the proper access time or energy consumption for the access to this memory object. Furthermore, the designer has the possibility to specify if a certain mapping should be performed for a memory object. For example, the designer could specify to map certain interrupt routines to a local scratchpad memory.

Dependent on the type of analysis required or the application complexity, the designer can choose if the thread graph should be extracted as a compact model or as a detailed model. An example of a compact and a detailed thread graph is shown in Figure 4.1. For example, let us assume that a parallel section containing four threads is executed within a loop. Then, for each loop iteration, the four threads are created, executed and joined. The compact thread graph model generates the thread nodes for one iteration and then just counts the number of thread executions. This means, all instances of a thread are represented as *one* thread instance. This thread instance comprises and counts all executed basic blocks containing conditionals (if/else) and loops, which were executed in each iteration of a thread. The advantage of a compact thread graph model is that it can be analyzed more quickly by the mapping tools. It is useful for complex applications. For a detailed thread graph, the thread nodes are unrolled, i.e. for each loop iteration containing a thread creation, a new thread node is created in the thread graph. In addition, if FIFO communications are repeated within a loop in one thread instance, their repetition is detected precisely in the thread graph. As stated before, after each FIFO write a new thread node is created, and each FIFO read is captured at the beginning of a new thread node. This procedure allows a more precise timing and energy analysis, but it also requires more optimization/analysis time within the mapping tools.

4.6.2 Model Extraction

Before extracting the thread model, some pre-processing analysis steps have to be performed by the ICD-Compiler. First, the tool detects the total number of threads existing for the application. Furthermore, it extracts the name of the thread functions and the thread ids. In the next step, a call list is determined for each function, i.e. it can be analyzed which function is called by other functions or by thread functions. This information is important in combination with the analysis of the

profiling file, which was created through simulation. The profiling file reflects the execution order of the parallelized application code. It shows the order of the executed basic block as well as thread creation, thread joins and all FIFO operations.

However, context switches can occur on FIFO reads and writes as well as during the creation and join of a thread. These context switches have the goal to avoid deadlocks. The challenge of the extraction tool is to determine which executed basic block belongs to which thread and thereby to provide the assignment of basic block ids to the proper thread node. With the function call list, it can be determined which functions (i.e. basic blocks) are called by which threads. Additionally, it is detected which functions are never called and which functions are called by more than one thread. In the latter case, shared functions are called by several threads. This means that the executed basic blocks in the profiling file cannot be properly related to a thread and therefore the assumptions about timing and energy consumption of a thread would get imprecise. However, in one of the last steps in the MNEMEE tool flow, the source code is optimized and prepared for the execution of the threads on the different processors. In addition, the proper implementation with the RTLIB of the MPMH tool is finalized. Based on the solution of the mapping tools, this RTLIB step performs a copy of shared functions and assigns these copied functions to each single thread, i.e. each thread owns its own functions and no more shared code exists between threads. This is not valid for shared data. In order to maintain consistency with the optimized source code at the end of the MNEMEE tool flow and for a more precise timing analysis in the thread graph, a copy of the shared functions is also internally generated within this part of the *Thread Model Extraction Tool*. This is conducted with the help of the ICD-C compiler. One further pre-processing step includes the detection of the start and end of loops, i.e. the extraction of basic block ids, which represent the start and end point of loops. This information is required in order to extract a proper *compact* thread model, where loops are not unrolled. Additionally, all memory objects, including all information as size, accesses, etc., are extracted for the threads and all functions that are called by each thread. The memory object information is extracted with the help of the internal MACCv2 tool *Memory-Allocator* which mainly uses ICD-Compiler analysis and architecture information for this extraction.

The thread graph construction begins with the parsing of the profiling file line by line. The profiling file contains keywords that represent start and join of threads, FIFO operations and basic block executions. Each line is processed and interpreted. The start node of the main thread is constructed first. In addition, all basic block executions are indexed and assigned to this thread node. When the main thread creates new child threads, a new thread node for each child thread is created. A *CREATE* edge connects the main thread with its child thread. The tool also performs all connections for the join of threads. According to the rules defined before, FIFO operations are captured and new thread nodes are created after FIFO writes and for FIFO reads. The FIFO edges are also set up between the nodes.

As already mentioned before, context switches occur after a start and join of a thread or after FIFO operations. After each context switch, the tool has to store

the last executed basic block of a thread or the last executed FIFO operation. Afterwards, it has to determine which thread is executed next and continue the construction of the thread graph. Throughout the construction of the thread graph, all information that was extracted by the pre-processing step is used. For example, the function call list is used in order to detect to which function, i.e. thread, an executed basic block belongs to. The tool has to identify which thread continues it execution after a context switch. Furthermore, all memory objects are assigned to the proper threads in the thread model. The tool also detects the number of parallel sections as well as all access counts on edges and the maximum number of elements communicated in FIFO operations.

For the construction of a compact thread graph, the tool has to detect loops. Thus, it has to detect if certain thread nodes were already created, if FIFO operations/edges already exist and are repeated or if they occur for the first time. Based on this information, new nodes have to be created or they are just repeated, i.e. the number of executions is increased. Every execution of single basic blocks, threads and FIFO communication is stored. The total execution costs of one node iteration are dependent on the designers choice. The designer chooses if the average case or worst-case execution costs, i.e. assembler counts, should be stored in the model. The worst-case assembler counts are based on profiling information and not on static analysis. The tool stores the total assembler count for each node iteration. The worst-case execution costs per node iteration are set to the maximum execution costs detected over-all node iterations. The average execution costs are given by determining the average costs over all node iterations. A detailed thread graph contains the proper execution's costs per node, since all loops are unrolled. Thus, it includes the exact execution cost for each node iteration or FIFO edge. For all FIFO edges, the maximum number of elements that were communicated in one iteration is determined. If the designer chose the worst-case execution costs, this value is stored as worst case over all iterations and multiplied with the token size for each iteration. The average case execution costs are extracted by the average elements per iteration multiplied with the token size.

Furthermore, all edges get initial tokens, which is an important requirement for the scenario based mapping tool described in Section 3.2.7.1. The buffer size for the FIFO communication is always set to the worst case, i.e. the maximum number of elements is assumed for each iteration. This number is multiplied with the token size and the access count. This is crucial in order to allocate enough memory space. The buffer size is generated for each FIFO operation and can be stored as a memory object.

Based on the assembler counts and the information from the architecture specification, i.e. processor frequency and energy consumption in active or idle cycle, the run-time and energy consumption of each thread node can be calculated. However, this step will be performed in the mapping tools. It is also possible to generate a graphic illustration of the thread model by using the graphic editor tool YED [107]. YED requires a graphml file for the illustration of nodes and edges. This file is generated in the last step of the *Thread Model Extraction Tool*.

4.6.3 Constraints

The *Thread Model Extraction Tool* extracts a static thread graph, which models execution for specific input. Potential data dependencies that do not occur at run-time are not covered.

Furthermore, this tool is tailored towards the MNEMEE tool flow and can be used only inside this tool flow. A tool of this class always has to be tailored towards some API for parallelization and synchronization (e.g. Open-MP, MPI, MPMH). In this case, it is tailored towards the API of the MPMH tool and its RTLIB library.

The parallelization tool in the MNEMEE tool flow, presented in Section 3.2, is always generating a maximum number of threads, which are smaller or equal to the number of available processors in the architecture. The goal is to have a balanced load on all processors. Therefore, certain scenarios will not occur in the thread graph, i.e. a child thread will not create another child thread. Only the main thread can create child threads and the creation of child threads will not depend on if/else conditions. If this case would not be given, the *Thread Model Extraction Tool* would have to manage also thread creations within if/else clauses and thread creations by child threads. The tool should be able to handle these scenarios, but the timing model would not be that precise anymore for the compact thread graph. For example, for the case where the creation of a new child thread would be executed in the *if* clause. Let us assume that during the program execution one iteration would execute the *if* clause and the next iteration executes the else clause. For the worst-case execution costs, the tool would assume that both clauses are executed in each iteration for the compact model. This could maybe lead to an over-estimation for the timing model and for the resources in the architecture. Nevertheless, it would still represent a model that can be extracted as a thread graph.

For the communication of the FIFO operations, memory buffer is required. This buffer has to be available on a memory for the sender and receiver of the FIFO operation and the buffer size has to be known in advance. In the MNEMEE tool flow, these buffers are mapped to the shared memory of the architecture. The maximum memory requirement is determined by the tool. But, this buffer size does not guarantee that the execution will be deadlock free. For this, further analysis of the application would be required by the designer.

4.7 Architecture Information

Next to an application specification, also an architecture specification is required by the mapping tools in order to match the application's requirements with the architecture capabilities or resources, respectively. In Section 5 and 6, two mapping optimization strategies are described, which perform memory-aware mapping optimization. Both strategies perform a combined optimization for the mapping of threads to processors and memory objects to memories. Therefore, the architecture specification has to contain information about the resources of the architecture, i.e. processors, memories and their contribution to energy consumption and run-time.

The processor has to be specified by an *id* and the *frequency* A detailed memory specification which models a multilevel memory hierarchy and which also contains interconnect characteristics such as the access time or energy consumption for one access with different bandwidth is also required.

Each memory is defined by *id*, *size*, *type*, and *access type*. The *type*, for instance, describes if the memory is a scratchpad memory, shared memory, or private memory. *Access type* defines the type of access, which is possible on the memory, i.e. data only, instruction only or unified access.

The interconnections between processors and memories have to be specified, too. They specify which processor has access to which memory and the cost for an access to a specific memory in terms of run-time and energy. These performance characteristics differ for each processor and memory type. They are important in order to determine the overall run-time and energy costs.

This specification is set up for each processor that has access to a specific memory. Here, first *CPU id* is specified together with access run-times and energy information. A distinction is made between *read* and *write accesses*, and between different access width such as *byte*, *half-word*, and *word*, because run-time and energy can vary for different access bandwidth.

All architecture information is extracted from the architecture database within the MACCv2 [14] framework described in Section 3.2.1. In this framework, the designer can specify and choose among different architectures. The *Thread Model Extraction Tool* acts as an interface to this database, processing all required information and automatically generating the architecture specification file for the mapping tools.

4.8 Evaluation

This section shows the extracted thread graphs for different benchmarks for the detailed and compact thread graph models.

The most benchmarks used in this evaluation are taken from the UTDSP suite [108]. Also real-life benchmarks are used for the evaluations in this thesis: *MPEG4*, *JPEG2000*, *H264* and *Boundary Value Problem*. The UTDSP suite consists of two classes of benchmarks: kernels and applications. Kernels are important calculations in DSP applications as filters, fast Fourier transformation and matrix multiply. Here, *FIR*, *IIR*, *LATNRM* and *MULT* represent kernels. DSP applications are programs, which are usually larger than kernels. The applications from the UTDSP benchmark suite are *ADPCM*, *Compress*, *Edge Detect* and *Spectral*. Table 4.1 illustrates and describes all benchmarks used for the evaluations of this thesis [109].

The code size of the benchmarks ranges from 6.5 kB up to 3 MB with an average code size of 50 kB per benchmark. In a first step, the parallelization tools (described in Section 3.2.1) of the MNEMEE tool flow are applied in order to obtain a parallelized and synchronized application code. Different numbers of threads are extracted for each benchmark, which are shown in Table 4.2. Here, compact and

Benchmark	Description
ADPCM	Adaptive differential pulse-coded modulation speech encoder
Boundary Value	Differential Equations
Compress	Image compression with discrete cosine transforms
Edge Detect	Edge Detection with 2D convolution and sobel operators
FIR	Finite Impulse Response filter
H264 Lblock	H264 decoder
H264 Mblock	H264 decoder with macroblock coding
IIR	Infinite Response Filter
Jpeg2000	JPEG encoder
LATNRM	Normalized Lattice Filter
Mpeg4	MPEG4 encoder
Mult	Matrix Multiplication
Spectral	Spectral analysis with period gram averaging

Table 4.1: Benchmarks and their description

detailed thread graph models are extracted.

Usually, the number of threads for a parallel section is automatically set to the number of processors by the parallelization tools. This can result in homogeneous threads. For a better evaluation of the mapping tools described in the next chapters, the parallelism of some of the benchmarks were manually extended in order to create more threads than available processors in the architecture. By this, an increase in complexity is achieved. Furthermore, it creates more heterogeneity between the individual threads. For the benchmarks *Edge Detect* and *Jpeg2000*, parallel versions containing 6 (i.e. labeled as T6) up to 16 threads (i.e. labeled as T16) per parallel section are added.

In the following subsections, the evaluation results are shown for the compact model and for the detailed model. The number of threads for the benchmarks consists of the main thread and the number of threads in each parallel section.

4.8.1 Compact Model

As already described before, the compact model is suitable for complex benchmarks, which contain many repetitions (i.e. thread creation or FIFOs within loops). The execution costs per node and the FIFO operations are summarized. Based on the choice of the designer, it can represent the average or worst-case costs. These costs do not influence the extraction of the number of threads or parallel sections. The compact model can reduce the complexity of the thread graph and thus the design time, which is mainly spent in the optimization step of the mapping tool.

The results for the number of threads and parallel sections of the extracted thread graph are shown in Table 4.2. Figure 4.4 shows the extracted thread graph for the *Edge Detect T8* benchmark, illustrated with the graphic editor tool YED

Benchmark	Nr. of Threads	Nr. of Parallel Sections
ADPCM	4	1
Boundary Value	5	1
Compress	5	1
Edge Detect	5	1
Edge Detect (T6)	7	1
Edge Detect (T8)	9	1
Edge Detect (T12)	13	1
Edge Detect (T16)	17	1
FIR	5	1
H264 Lblock	8	2
H264 Mblock	9	2
IIR	5	1
Jpeg2000	5	1
Jpeg2000 (T6)	7	1
Jpeg2000 (T8)	9	1
Jpeg2000 (T10)	11	1
LATNRM	3	1
Mpeg4	4	1
Mult	5	1
Spectral	9	2

Table 4.2: Parallelization of Benchmarks for the compact model

[107]. The extracted number of threads and parallel sections on the thread graphs match the number of threads and parallel sections in the source code, which were extracted by the parallelization tools in the MNEMEE tool flow. In addition, all FIFO operations are captured properly at the right point of time. The tool extracted additional thread nodes for the precise representation of the point of time where FIFO write and read operations are performed.

The analysis of the benchmarks show that the *Spectral* benchmark has the most number of threads (25) in this setup with six parallel sections. Although other benchmarks have only one parallel section, most of them still contain complex threads in terms of computation workload, i.e. *Mpeg4*. The complexity of some benchmarks can be seen in the thread graphs of the detailed model. Furthermore, the required time for the thread graph extraction for each benchmark is illustrated in the next Section in Figure 4.3.

4.8.2 Detailed Model

The detailed model is suitable for less complex benchmarks or for a more detailed representation of the workload of the thread graph. Here, all loops are unrolled. This can lead to a repetition of the creation of threads and FIFO operations within

Benchmark	Nr. of Threads	Nr. Parallel Sections	Total Nr. Nodes
ADPCM	4	1	5
Boundary Value	5	1	6
Compress	5	1	6
Edge Detect	5	3	16
Edge Detect (T6)	7	3	22
Edge Detect (T8)	9	3	28
Edge Detect (T12)	13	3	40
Edge Detect (T16)	17	3	52
FIR	5	1	6
H264 Lblock	8	2	15
H264 Mblock	9	2	29
IIR	5	128	641
Jpeg2000	5	1	609
Jpeg2000 (T6)	7	1	613
Jpeg2000 (T8)	9	1	617
Jpeg2000 (T10)	11	1	623
LAT	3	1	4
Mpeg4	4	20	1577
Mult	5	20	401
Spectral	9	2	71

Table 4.3: Parallelization of Benchmarks for the detailed model

threads. The thread graph can grow and get very complex. On the other side, all nodes and FIFO edges contain their individual execution costs. This means that e.g. different execution paths, which can occur due to conditions (if/else-clauses) are captured properly in the execution costs. This can lead to a more precise optimization and resource allocation in the subsequent mapping tools. On the other side, it can increase design time and complexity for more complex benchmarks.

While Table 4.2 showed the total number of threads in the thread graph, Table 4.3 shows additionally the total number of nodes in the thread graph. For example, the benchmark *Edge Detect* has five threads: one main thread and four threads which are executed in parallel. In the detailed mode, the thread creation is enclosed by a loop, which is executed three times. The threads that are created have the same ID and code, but the execution costs can differ for each iteration.

The *JPEG2000* benchmarks have one parallel section, but over 600 nodes. Here, several FIFO operations are performed within loops, which increase the complexity of the thread graphs. The *MPEG4* benchmark contains loops, which repeat the thread creation as well as the FIFO operations. This leads to 1577 nodes. The *Spectral* benchmark contains FIFO operations in loops, which increase the number of nodes. A compact thread graph can reduce the optimization complexity for such

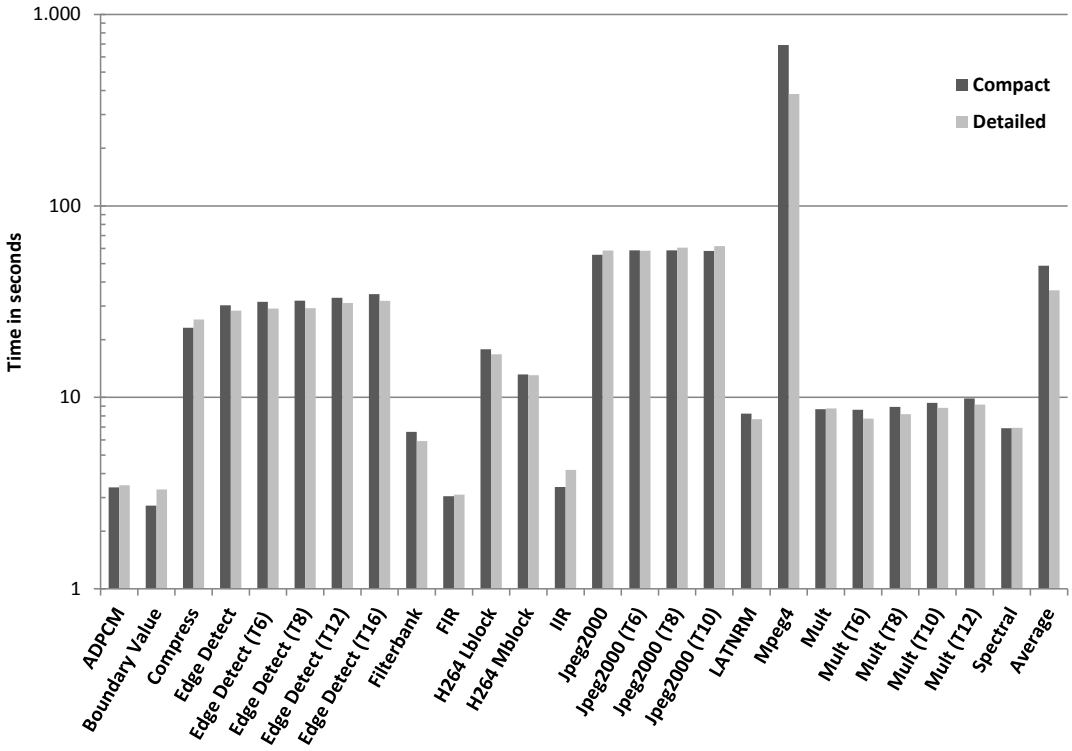

Figure 4.3: Required time for the thread graph extraction

benchmarks.

Some benchmarks have no difference to the compact model. Still, these benchmarks can contain loops in their threads and also complex operations, but no FIFO operations or thread creations in loops. In general, the number of threads and all FIFO operations also match the extracted number of threads and parallel sections of the parallelization tools in the MNEMEE tool flow. By the extraction of the compact and detailed model, the designer can have an overview over the complexity of the thread graphs and choose which model suits better for the demands of the system. Figure 4.5, 4.7 and 4.6 show extracted thread graphs for compact and detailed versions.

Figure 4.3 illustrates the required time for the thread graph extraction for each benchmark for the compact and for the detailed model. The times do not differ very much for both models. The *Thread Model Extraction Tool* traverses the same profiling file for both models. It decides during this traversation if a node has to be created or not dependent if the model is compact or detailed. The time for the traversation of the profiling file seems to be constant for both models.

4.8.3 Extracted Thread Models

This section illustrates extracted thread graphs for different benchmarks.

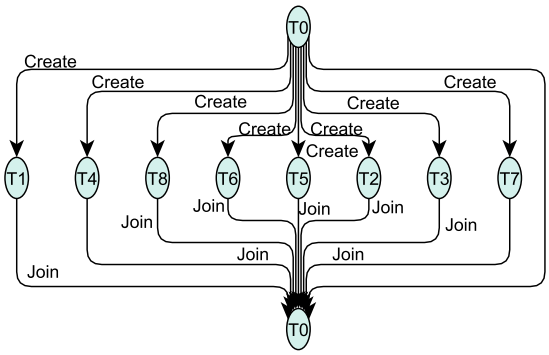

Figure 4.4: Thread Graph for the *Edge Detect T8* benchmark with 8 threads in a parallel section (compact)

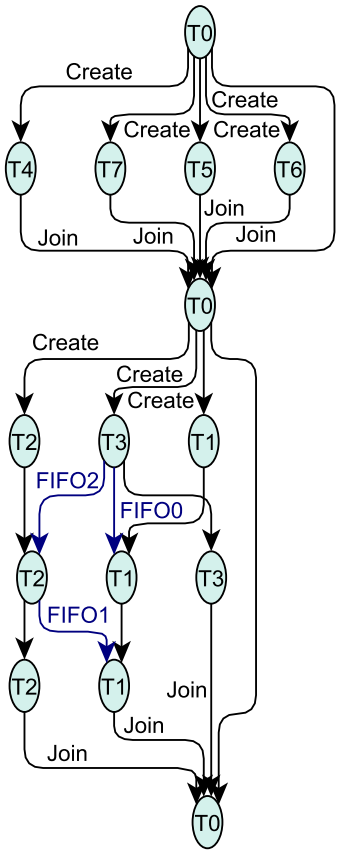

Figure 4.5: Thread Graph for the H264 Lblock benchmark containing FIFOs (compact and detailed version)

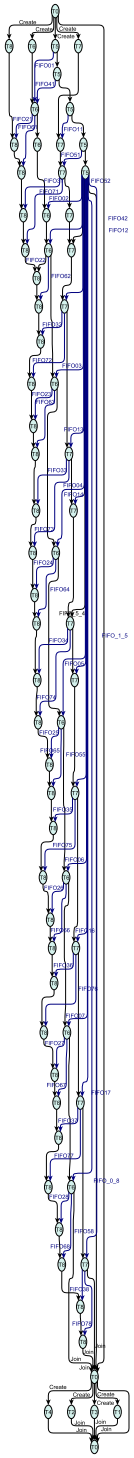

Figure 4.6: Thread Graph for the *Spectral* benchmark containing FIFOs (detailed version)

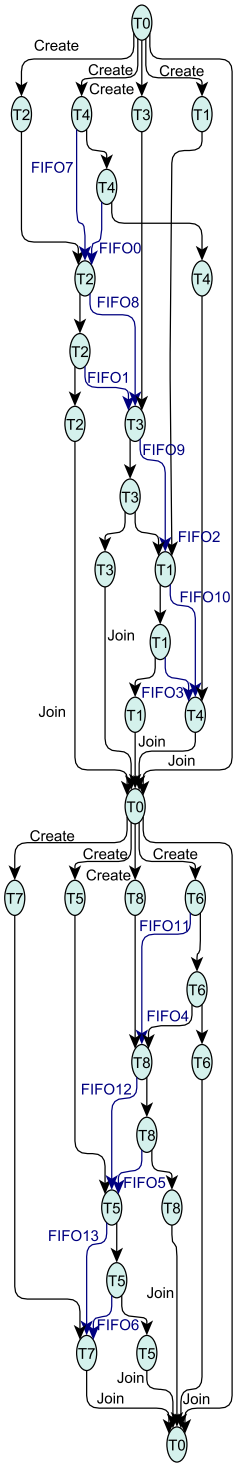

Figure 4.7: Thread Graph for the *H264 Mblock* benchmark containing FIFOs (compact and detailed version)

Chapter 5

Memory-Aware Single Objective Mapping Optimization

Contents

5.1 Introduction

Depending on the domain of the embedded system, a single optimization objective or several optimization objectives can be required for the underlying system. For example, let us assume a mobile and battery-driven embedded system, which contains a sensor and a transmitter that is sending the sensor values at certain time periods to a main station. Such mobile systems are usually optimized to save energy. However, an electricity-powered high-end high-definition (HD) Blu-ray DVD-Player is usually optimized for higher performance and has no need for the optimization of energy consumption. Some embedded systems require only a single optimization objective while others require more than one optimization objective. However, these optimizations have to be performed for state-of-the-art embedded system architectures, which are usually given as homogeneous or heterogeneous multiprocessor systems. All constraint system resources should be optimally utilized in order to obtain the best possible optimization solution for the underlying system.

In the MNEMEE tool flow, two tools are available for memory-aware mapping optimizations. One tool optimizes for a single objective. The second tool performs

multiobjective optimization and is described in Chapter 6. The embedded system designer is able to choose between these two optimization tools dependent on the domain of the underlying embedded system. In this chapter, the focus lies on embedded systems optimizations for a single objective, energy or runtime optimization respectively. The optimizations are based on ILP (Integer Linear Programming) which can provide optimal solutions for the underlying optimization problem. Integer Programming is known to be NP-hard.

The underlying optimization problem is the memory-aware mapping of parallelized threads to the different processors in the system combined with the mapping of memory objects of the threads to the available memories in the memory hierarchy. The reduction of energy consumption or runtime is gained by the proper utilization of the systems resources, i.e. a suitable mapping to the processors and memories. Many parameters have to be considered for this optimization. On the application and architecture side, it is important to manage all requirements as threads and their runtime on the different system processors as well as their energy consumption for idle and active cycles. Furthermore, all memory objects have to be known including characteristic information as their access runtime and energy consumption to the different memories in the system. In addition, it is important to detect if a memory object is a data or instruction object, how many read and write accesses are performed on the memory object and the size of the memory object. This information is required in order to determine the proper runtime and energy consumption for the access to the memory object on the different memories. In order to determine the proper energy consumption and runtime for FIFO operations, all information about the communicated data (size, number of accesses, bus access) have to be known. All these information contribute to a detailed representation of the system architecture and an accurate mapping optimization.

A more detailed description of the complexity was already given in Section 2.3.

Related Work

A detailed overview of all related work is given in Section 2.4.6. To sum up, the approach presented by Suhendra et. al. [91] is most similar to the presented approach in this section. Suhendra suggested an ILP approach for t mapping, pipelined scheduling and scratchpad memory partitioning for homogeneous MPSoC systems. Scratchpad memory partitioning is an approach where scratchpad memories are shared between all processors. Contrary to Suhendra's approach, the architecture considered in this thesis is structured in a different way. Here, homogeneous and heterogeneous architectures can be considered containing memory hierarchies with memories of different sizes, types and with different energy and runtime characteristics at different levels. The processors have exclusive access to their private memories, i.e. the scratchpad memories are not shared among the processors. In addition, pipelined scheduling is not considered in this approach. However, Suhendra's approach considers only data allocation while the presented memory-aware ILP approach considers instruction and data memory objects.

5.2 Tool Overview

In the MNEMEE tool flow, the designer chooses between the scenario-aware mapping and memory-aware mapping. The *ILP Memory-Aware Mapping Tool* is one of the memory-aware mapping tools. The precedent tools are the parallelization and synchronization tools and the *Thread Model Extraction Tool*. As described in Chapter 4, the *Thread Model Extraction Tool* provides an annotated flat thread graph as input for the mapping tools. At the beginning, the flat thread graph and all its annotations are processed. The underlying application and architecture models are described in Section 2.3. The ILP equations are set up based on this information (architecture and application model). The *ILP Memory-Aware Mapping Tool* uses an ILP-solver in order to find an optimal solution for the given ILP equations. If a solution is not possible due to restricted resources, e.g. the capacity of the memories cannot hold all memory objects of the application, the tool stops with an error message. The output of the tool is a mapping of threads to processors and the mapping of memory objects to memories. This information is stored within the MACCv2 framework and provided to the subsequent tools, which implement and prepare this mapping for the underlying system. If the *DMMR Tool* (described in Section 3.2.5) is enabled, the allocation of dynamic memory objects to the scratchpad memories are also provided to the mapping tools. This resulting memory mapping can be considered and added to the mapping solution of the successive mapping tools. The *ILP Memory-Aware Mapping Tool* can be adapted with some further options. It is possible to set a number of threads for the ILP solver when calculating the ILP based mapping. Please note that these threads are not referring to the thread graph, but to the ILP optimization calculation. For example, by setting the value '0', the number of threads is set to the number of available CPUs. This can speed up the calculations of the ILP-solver. Furthermore, the *ILP Memory-Aware Mapping Tool* can be adapted to disable the consideration of all memories, or disable the consideration of scratchpad memories only. It is also possible to consider all memories and memory mapping during the mapping optimization, but not provide the memory mapping solution as output. Only the thread to processor mapping will be provided in this case. These options can be used for different scenarios:

- for the comparison of different optimization possibilities with or without memories
- if certain memories are already reserved by the designer for other purposes (for example, the scratchpad memories)
- if the designer chooses a memory mapping of data only by the *Scratchpad Memory Optimization Tool* in the MNEMEE tool flow

For the last point, the *ILP Memory-Aware Mapping Tool* can consider the memories during its optimizations, but it does not perform a memory mapping, only a thread to processor mapping. However, the threads are mapped to those processors,

which provide the most optimal utilization possibilities for the memory objects of the threads. For example, if a thread has large and data-intensive memory objects, a processor could be chosen, which has access to larger data memories or to those memories which suit the demands of the memory objects of the thread as well as possible.

5.3 ILP Optimization

This section presents the integer linear programming (ILP) equations for the application to architecture mapping with integrated memory-awareness. Depending on the requirements of the embedded system, the designer chooses the proper ILP either for the minimization of runtime or energy consumption.

5.3.1 Optimization for Runtime

This section presents the ILP formulation for the optimization of the overall runtime.

First, the runtime of all threads has to be considered and evaluated. The application starts with the first node of the main thread. The main thread is composed of a start node, optionally one or several intermediate nodes and one exit node (as depicted in Figure 2.11). Between the nodes of the main thread, the child threads are executed in parallel. However, the nodes of the main thread will be mapped to one processor since they belong to one thread. Otherwise, i.e. when mapping the nodes to different processors, additional execution and memory copy cost occur. Depending on the application, some threads also consist of more than one node if communication via FIFOs is specified (as illustrated in Figure 2.12). An application has k threads ($Thread_1$ to $Thread_k$). Each thread consists of j nodes, where j has a value of 1 if no FIFO communication is performed.

Following notations are used for the definition of the equations concerning the threads:

- $Thread_i$ represents a thread with index i, where $i = 1, .., k$,
- $Thread_{i,j}$ defines a node of thread $Thread_i$, where $j = 1, .., n$,
- $StartThrNode_{i,j}$ represents the start time of thread node $Thread_{i,j}$,
- $EndThrNode_{i,j}$ defines the completion time of thread node $Thread_{i,j}$,
- $ExecTimeNode_{i,j}$ specifies the execution time of thread node $Thread_{i,j}$.

The objective function minimizes the completion time of the last thread, which is the exit node of the application:

$$min\,(EndThrNode_{Exit}) \tag{5.1}$$

The completion time $EndThrNode_{i,j}$ of thread i and its node j is given by its start

time $StartThrNode_{i,j}$, added to its overall execution time $ExecTimeNode_{i,j}$:

$$EndThrNode_{i,j} \geq StartThrNode_{i,j} + ExecTime_{i,j} \tag{5.2}$$

The value of $StartThrNode_{i,j}$ is dependent on the predecessors of the thread node $Thread_{i,j}$ and will be explained later in this section.

In the following, additional constraints for the ILP are defined.

5.3.1.1 Overall runtime of a thread node

The runtime of a thread and its thread nodes consist of several parameters. One parameter is the runtime on the processor, another parameter is the runtime for memory access. For the determination of the runtime for memory access, more parameters have to be known: It should be distinguished if a read or write access is performed as well as the access width. Furthermore, the access times for the underlying buses that are accessed are also important for the proper runtime calculation. All parameters are given as input through the *Thread Model Extraction Tool*. The following equations describe the calculation of the runtime for all thread nodes.

Following notations are used for the definition of the equations in this section:

- the set of threads is defined by $T = 1, ..., k$,
- the set of processors is defined by $P = 1, .., p$,
- the set of memories with $M = 1, ..., m$
- MO_i is the set of memory objects for thread i,
- $MO_{i,j}$ is a subset of MO_i and contains all memory objects that are accessed by thread node $Thread_{i,j}$,
- $mObj_{i,j}$ is a memory object of thread node $Thread_{i,j}$ and an element of the set $MO_{i,j}$ and MO_i, respectively,
- $ProcTime_{i,j}$ represents the execution time of thread node $Thread_{i,j}$ on the processor it is mapped to and primarily used as a supporting variable,
- $ExecTimeProc_{i,j,proc}$ specifies the runtime of thread node i, j on a specific processor $proc$,
- $AccTime\_memories_{i,j}$ defines the required access time to the memories which is dependent on the memory mapping of the memory objects of thread node $Thread_{i,j}$; it includes the access time to the underlying buses,
- $X_{i,proc}$ is a binary decision variable and has the value 1 if thread i is mapped to processor $proc$, otherwise it has value 0

- $ProcVarMem_{proc,mem,mObj_{i,j}}$ is a binary decision variable which has the value 1 if the memory object $mObj_{i,j}$ is mapped to memory mem which is accessible by processor $proc$, otherwise its value is 0,

- $NrReadAcc_{mObj_{i,j}}$ defines the number of read accesses to the memory object $mObj_{i,j}$ within thread node $Thread_{i,j}$,

- $NrWriteAcc_{mObj_{i,j}}$ defines the number of write accesses to the memory object $mObj_{i,j}$ within thread node $Thread_{i,j}$,

- $AccTimeRead_{proc,mem,mObj_{i,j}}$ represents the time required for a read access to memory object $mObj_{i,j}$ on a memory mem which is accessible by processor $proc$,

- $AccTimeWrite_{proc,mem,mObj_{i,j}}$ represents the time required for a write access to memory object $mObj_{i,j}$ on memory mem which is accessible by processor $proc$.

The overall runtime of a thread node is composed by the execution time of the thread node on a processor and the time required for memory accesses.

$$ExecTime_{i,j} = ProcTime_{i,j} + AccTime\_memories_{i,j} \tag{5.3}$$

The execution time on a processor $ProcTime_{i,j}$ as well as the access time $AccTime\_memories_{i,j}$ to the memories depend on the mapping decision of this optimization. Their definition will be presented next.

The runtime of thread node (i, j) on a processor in the system is defined by:

$$ProcTime_{i,j} = \sum_{proc=1}^{P} (X_{i,proc} * ExecTimeProc_{i,j,proc}) \tag{5.4}$$

This equation adds the proper execution time of thread i on processor $proc$. The binary decision variable $X_{i,proc}$ indicates if thread i is mapped onto processor $proc$.

It also has to be specified that a thread i and all its nodes can be mapped only to one processor in the architecture. Therefore, following constraint is defined for each thread $Thread_i$:

$$\sum_{proc=1}^{P} X_{i,proc} = 1 \tag{5.5}$$

Next, the overall access time to memories $AccTime\_memories_{i,j}$ caused by a thread node $Thread_{i,j}$ is composed of read and write accesses of the memory object onto

the mapped memory. It is defined by:

$$
\begin{aligned}
&AccTime\_memories_{i,j} = \\
&\sum_{mObj_{i,j}=1}^{MO_{i,j}} (\sum_{mem=1}^{M} ProcVarMem_{proc,mem,mObj_{i,j}} \\
&* (NrReadAcc_{mObj_{i,j}} * AccTimeRead_{proc,mem,mObj_{i,j}} \\
&+ NrWriteAcc_{mObj_{i,j}} * AccTimeWrite_{proc,mem,mObj_{i,j}}))
\end{aligned}
\tag{5.6}
$$

This equation iterates over all memory objects $mObj_{i,j}$ of thread $Thread_i$ and sums up the time that is required for the access to each memory object. The access time depends on the memory to which $mObj_{i,j}$ is mapped, which is indicated by the binary decision variable $ProcVarMem_{proc,mem,mObj_{i,j}}$. The overall access time spent on a memory mem is composed of read and write accesses of memory object $mObj_{i,j}$ to this memory. The number of read accesses $NrReadAcc_{mObj_{i,j}}$ is multiplied with the access time $AccTimeRead_{proc,mem,mObj_{i,j}}$ required for one read access. The same is defined for the write accesses $NrWriteAcc_{mObj_{i,j}}$ to the memory object $mObj_{i,j}$ on memory mem. As mentioned before, read and write accesses can cause different access times, which also depends on size and access width of the memory object. The ILP optimization generates the proper read and write access values for the variables $AccTimeWrite$ and $AccTimeRead$ dependent on the access width and size of the memory objects. Furthermore, only valid combinations of $ProcVarMem_{proc,mem,mObj_{i,j}}$ are generated within the ILP optimization (i.e. up to $M \times O \times MO_i$ variables). This means, only combinations can occur, which represent the valid access possibilities of the underlying architecture: i.e. only memories mem which are accessible by a certain processor $proc$ and which are capable of holding the appropriate memory objects (e.g. a data memory object cannot be mapped to an instruction memory). Please note that this access time also includes the time for the access to the underlying buses. As mentioned in Section 4.7, this information is provided by the architecture database within the MACCv2 framework (described in Section 3.2.1).

5.3.1.2 Memory constraints

The next constraint ensures that the sizes of memories are not exceeded by the memory objects that are mapped to the memories.
Following notations are used:

- $size_{mObj_{i,j}}$ represents the size of a memory object $mObj_{i,j}$,
- $size_{mem}$ specifies the capacity of a memory mem.

The next equation is defined for each memory mem:

$$
size_{mem} \geq \sum_{i=1}^{T} \sum_{mObj_{i,j}=1}^{MO} (ProcVarMem_{proc,mem,mObj_{i,j}} * size_{mObj_{i,j}}) \tag{5.7}
$$

The right hand side of this equation iterates over all threads $Thread_i$ and all their memory objects $mObj_{i,j}$ and sums up the size of all memory objects mapped to memory mem. This equation guarantees that the size of all memory objects does not exceed the size of the memory.

In a memory hierarchy, processors have exclusive access to specific memories (i.e. on-chip and private main memories). Therefore, constraints have to ensure that when a thread $Thread_i$ is mapped to a processor $proc$, the memory objects $mObj_{i,j}$ of thread $Thread_i$ can be only mapped to a memory mem that can be accessed by processor $proc$.

This is defined in the next equation:

$$X_{i,proc} = \sum_{mem=1}^{M} ProcVarMem_{proc,mem,mObj_{i,j}} \tag{5.8}$$

This equation also ensures that a memory object $mObj_{i,j}$ can be mapped only to one memory mem in the system. In detail, if thread $Thread_i$ is mapped to processor $proc$ (i.e. $X_{i,proc}$ has value 1), the memory object $mObj_{i,j}$ can only be mapped to one memory mem that is accessible by processor $proc$ (i.e. through the decision variable $ProcVarMem_{proc,mem,mObj_{i,j}}$).

5.3.1.3 Dependencies in the thread graph

Here, the control flow and FIFO related dependencies between thread nodes are defined. These equations are mandatory in order to guarantee an accurate calculation of the starting and end time of thread nodes.

Following notations are used:

- $pred_{i,j}$ defines a predecessor node of thread node $Thread_{i,j}$,
- $FIFO_{i,h}$ specfies the required time for a FIFO communication between the thread nodes $Thread_{i,j}$ and $Thread_{h,l}$.

Dependencies in the thread graph define at which point of time the execution of a thread node can be started on a processor. A thread node can start its execution when all predecessors have finished their execution. This is defined by following equation:

$$\begin{aligned} &\forall pred_{i,j}: \\ &if(i = h): \\ &StartThrNode_{i,j} \geq EndThrNode_{h,l} + 1 \\ &if(i \neq h): \\ &StartThrNode_{i,j} \geq EndThrNode_{h,l} + 1 + FIFO_{i,h} \end{aligned} \tag{5.9}$$

For all predecessors $pred_{i,j}$ of thread node $Thread_{i,j}$ the following is defined: the start time $StartThrNode_{i,j}$ is greater than the finishing time of the predecessor node $EndThrNode_{h,l}$ of its predecessor node l. Here, the predecessor node belongs to the same thread as the successor node and represents a control flow edge, i.e. for the case if $i = h$. For the case $i \neq h$, i.e. the predecessor belongs to a different thread node, the time required for communication over FIFOs between both threads nodes is added. Here, the number 1 represents one time unit. FIFO communication is optional, i.e. the equations are only set if FIFO communication is defined within the application. Please note, that this equation is not valid for child thread nodes, which are joined to the main thread.

5.3.1.4 Time for FIFO Communication

The time for FIFO communication depends on many parameters, as access and size of a data element that is communicated over FIFO. Furthermore, the bus access time has to be considered as well.

Thus, following notations are used in this subsection:

- $NrElements_{i,h}$ represents the number of data elements communicated within the FIFO $FIFO_{i,h}$,
- $SizeElement_{i,h}$ specifies the size of a data element that is communicated within the FIFO $FIFO_{i,h}$,
- $NrAccesses_{i,h}$ defines the number of accesses to the data element communicated within the FIFO $FIFO_{i,h}$,
- $BusSpeed$ represents the speed of the underlying bus that is used for the FIFO communication of $FIFO_{i,h}$

Please note that one FIFO is defined for the communication of a token (i.e. all variables that needs to be communicated are combined into a token). As already described, if FIFO communication is defined for a certain application thread graph, a thread consists of more than one node. One of these nodes can have an incoming FIFO edge (FIFO read). The predecessor node, belonging to the incoming FIFO edge, must first complete its execution before the target node can proceed with its execution. This was already defined in Equation 5.9.

The following equation defines the time required for a FIFO communication $FIFO_{i,h}$ between the thread nodes $Thread_{i,j}$ and $Thread_{h,l}$:

$$FIFO_{i,h} = \frac{NrElements_{i,h} * SizeElement_{i,h} * NrAccesses_{i,h}}{BusSpeed} \tag{5.10}$$

The term $FIFO_{i,h}$ is specified by the data elements that are communicated via FIFO. Here, the number of data elements $NrElements$ is multiplied by the size $SizeElement_{i,h}$ and the number of accesses $NrAccesses_{i,h}$ to the elements and divided by the speed of the underlying bus.

5.3.1.5 Processor assignment

The next equations ensure that a processor is executing only one thread at a time. First, it has to be determined if two threads are mapped onto the same processor.

Following notations are used in this section:

- $L_{i,h}$ represents a decision variable which has value 0 when thread $Thread_i$ and thread $Thread_h$ are mapped onto the same processor, and else value 1, if they are mapped onto different processors,
- $B_{i,h}$ specifies a decision variable that is set to 1 if thread node $Thread_{i,j}$ and thread node $Thread_{h,l}$ are mapped onto the same processor and thread $Thread_h$ is executed after thread $Thread_i$, else it is set to 0.

The next equation is taken from [91] and was adapted to our thread model. Here, only different threads are considered, i.e. $h \neq i$.

$$\forall proc_p : 1...P, \quad L_{i,h} \leq 2 - X_{i,proc_p} - X_{h,proc_p} \tag{5.11}$$

$$\forall proc_p : 1...P, \forall proc_q : 1...P, proc_p \neq proc_q, \quad L_{i,h} \geq X_{i,proc_p} + X_{h,proc_q} - 1 \tag{5.12}$$

Both equations represent the correct value for $L_{i,h}$ when $Thread_i$ and $Thread_h$ are mapped onto the same or on different processors. The decision variable $L_{i,h}$ is set to 0 or 1 by the values of the decision variables $X_{h,proc_p}$ and $X_{i,proc_p}$ or $X_{i,proc_q}$, respectively. Equation 5.11 constrains $L_{i,h}$ to the value 0 when $Thread_i$ and $Thread_h$ are mapped to the same processor $proc_p$. Additionally, equation 5.12 constrains $L_{i,h}$ to the value 1 when $Thread_i$ and $Thread_h$ are mapped to different processor $proc_p$ and $proc_q$. The variables $L_{i,h}$ are required for all combinations of threads in the parallel section.

While the latter equations work on thread level basis, the next equation adds a constraint at the level of thread nodes. It adds the constraint that two thread nodes are not executed at the same time on one processor. Again, only different threads are considered, i.e. $i \neq h$ which belong to the same parallel section. Thread node $Thread_{i,j}$ belongs to thread $Thread_i$ and thread node $Thread_{h,l}$ belongs to thread $Thread_h$.

$$B_{i,h} + B_{h,i} = 1 \tag{5.13}$$

$$B_{i,h} + B_{h,i} - L_{i,h} = 1 \tag{5.14}$$

$$StartThrNode_{h,l} \geq EndThrNode_{i,j} - \infty * B_{h,i} + 1 \tag{5.15}$$

$$StartThrNode_{i,j} \geq EndThrNode_{h,l} - \infty * B_{i,h} + 1 \tag{5.16}$$

Equation 5.13 allows only one variable $B_{i,h}$ or $B_{h,i}$ to have value 1. If both threads $Thread_i$ and $Thread_h$ are mapped to the same processor, either $Thread_i$ is executed before $Thread_h$ (i.e. $B_{i,h}$ has value 1), or otherwise. Equation 5.14 allows only valid

combinations of $B_{i,h}$ or $B_{h,i}$ with $L_{i,h}$ (in combination with Equation 5.13). For example, variable $B_{i,h}$ is not allowed to have value 1 (i.e. $Thread_i$ and $Thread_h$ are mapped to the same processor) when $L_{i,h}$ has also value 1 (i.e. $Thread_i$ and $Thread_h$ are mapped to different processors). The last two equations 5.15 and 5.16, set the proper start and end time of the thread nodes dependent on the previous value settings of $B_{i,h}$ or $B_{h,i}$. These equations guarantee that precedence constraints are preserved and that threads are not executed at the same time.

Please note, that thread nodes which belong to the same thread are not considered here, since their sequence order is already set in the dependencies constraint in Equation 5.9.

5.3.2 Optimization for Energy

In this section, the ILP formulation for the optimization of the overall energy consumption is presented. All equations that are required for the minimization of runtime are also required in the minimization of the energy consumption, except for the objective function. They are required for the proper calculation of the processor energy where the exact idle and run cycles have to be known. This results in a higher complexity for this ILP optimization.

Following notations are used in this section:

- $CPUEnergy$ represents the energy consumption for all processors of the system,
- $MemEnergy$ defines the energy consumption for all memories in the system,
- $IdleEnergy_{proc}$ specifies the energy consumption in the idle mode of processor $proc$ for one cycle,
- $ActiveEnergy_{proc}$ represents the energy consumption in the active mode of processor $proc$ for one cycle,
- $CPUEnergy_{i,proc}$ is the energy consumption that is specified by the difference between the active mode and idle mode energy for a thread $Thread_i$ that is mapped to processor $proc$,
- $Cycles_{i,proc}$ defines the overall execution cycles of thread $Thread_i$ on processor $proc$,
- $AccEnergyRead_{proc,mem,mObj_{i,j}}$ specifies the energy consumption for a read access to memory object $mObj_{i,j}$ on memory mem and it includes the energy consumption for the accesses to the underlying buses,
- $AccEnergyWrite_{proc,mem,mObj_{i,j}}$ represents the energy consumption for a write access to memory object $mObj_{i,j}$ on memory mem and it includes the energy consumption for the accesses to the underlying buses,

- $FIFOEnergy$ defines the energy consumption for FIFO operations,
- $FIFO_{comm}$ represents the set of all available FIFO operations between thread nodes $Thread_i$ and $Thread_h$,
- $BusEnergyCycle$ specifies the energy of the underlying bus for one access to this bus.

For the minimization of the overall energy consumption of the system, the objective function is defined in the following equation:

$$min\,(CPUEnergy + MemEnergy + FIFOEnergy) \tag{5.17}$$

The goal is to minimize the total energy consumption which consists of the energy consumed by all processors, all memory accesses and optionally for the energy consumption of FIFO operations.

5.3.2.1 Energy consumption of processors

In this model, a processor can be in two different modes. When the processor performs computation, it is in the active mode. Otherwise, it is in the idle mode. The energy consumption for all processors is defined in equation 5.18:

$$\begin{aligned} CPUEnergy = &(\sum_{proc=1}^{P} IdleEnergy_{proc} * EndThrNode_{Exit}) \\ &+ (\sum_{i=1}^{T}\sum_{proc=1}^{P} CPUEnergy_{i,proc} * X_{i,proc}) \end{aligned} \tag{5.18}$$

In the first term, the equation sums up the idle energy $IdleEnergy_{proc}$ for each processor $proc$ for the overall runtime of the application. The overall runtime is defined by the last node of the thread graph $EndThrNode_{Exit}$. The second term represents the additional energy $CPUEnergy_{i,proc}$ that is consumed for each thread $Thread_i$ that is mapped to processor $proc$.

$CPUEnergy_{i,proc}$ is defined by the energy difference between the active mode and idle mode:

$$CPUEnergy_{i,proc} = Cycles_{i,proc} * (ActiveEnergy_{proc} - IdleEnergy_{proc}) \tag{5.19}$$

Only the execution cycles that are required for the computation in active mode of thread $Thread_i$ are multiplied by the difference between the active energy $ActiveEnergy_{proc}$ and idle energy $IdleEnergy_{proc}$.

5.3.2.2 Energy consumption for memory accesses

Here, the energy consumption for all memory accesses is defined:

$$
\begin{aligned}
&MemEnergy = \\
&\sum_{i=1}^{T} \sum_{mem=1}^{M} \sum_{mObj_{i,j}=1}^{MO} ProcVarMem_{proc,mem,mObj_{i,j}} \\
&* (NrReadAcc_{mObj_{i,j}} * AccEnergyRead_{proc,mem,mObj_{i,j}} \\
&+ NrWriteAcc_{mObj_{i,j}} * AccEnergyWrite_{proc,mem,mObj_{i,j}})
\end{aligned} \tag{5.20}
$$

This equation iterates over all threads $Thread_i$, over all memories mem and over all memory objects $mObj_{i,j}$ of $Thread_i$ in the system. If a memory object $mObj_{i,j}$ is mapped to a memory mem (i.e. $ProcVarMem_{proc,mem,mObj_{i,j}}$ is true), the energy for all read and write accesses is added to the overall memory energy consumption $MemEnergy$. In detail, the number of read accesses $NrReadAcc_{mObj_{i,j}}$ for the memory object $mObj_{i,j}$ of thread $Thread_i$ is multiplied by the energy $AccEnergyRead_{proc,mem,mObj_{i,j}}$ that is required for one read access. The energy consumption for the write accesses $AccEnergyWrite_{proc,mem,mObj_{i,j}}$ to $mObj_{i,j}$ is defined in the same way. The optimization generates the proper read and write energy values for $AccEnergyRead$ and $AccEnergyWrite$ based on the access width and size of the underlying memory objects.

5.3.2.3 Energy consumption for FIFO operations

The following equation defines the energy required for a FIFO communication $FIFOEnergy_{i,h}$ between the thread nodes $Thread_{i,j}$ and $Thread_{h,l}$:

$$
\begin{aligned}
FIFOEnergy = &\sum^{FIFO_{comm}} NrElements_{i,h} \\
&* SizeElement_{i,h} * NrAccesses_{i,h} * BusEnergyAccess
\end{aligned} \tag{5.21}
$$

$FIFOEnergy$ is defined by the iteration over all existing FIFO communications $FIFO_{comm}$. For each communication the size $SizeElement$ and the number of data elements $NrElements$ that are communicated via the FIFO are multiplied by the number of accesses $NrAccesses$ to these elements and the energy consumption $BusEnergyAccess$ for an access to the underlying bus.

5.3.3 Restrictions of the ILP model

Here, the restrictions of this ILP model are described. First, the accesses to memories cannot always be modeled as precisely as in real memories. For example, due to the abstraction level of this model, fast accesses that are performed in blocks are not covered. In addition, due to the processor pipeline, access time to memories can be decreased. Furthermore, the bus is not modeled as restricted resource, i.e. bus conflicts and the resulting wait cycles cannot be covered with the proposed model

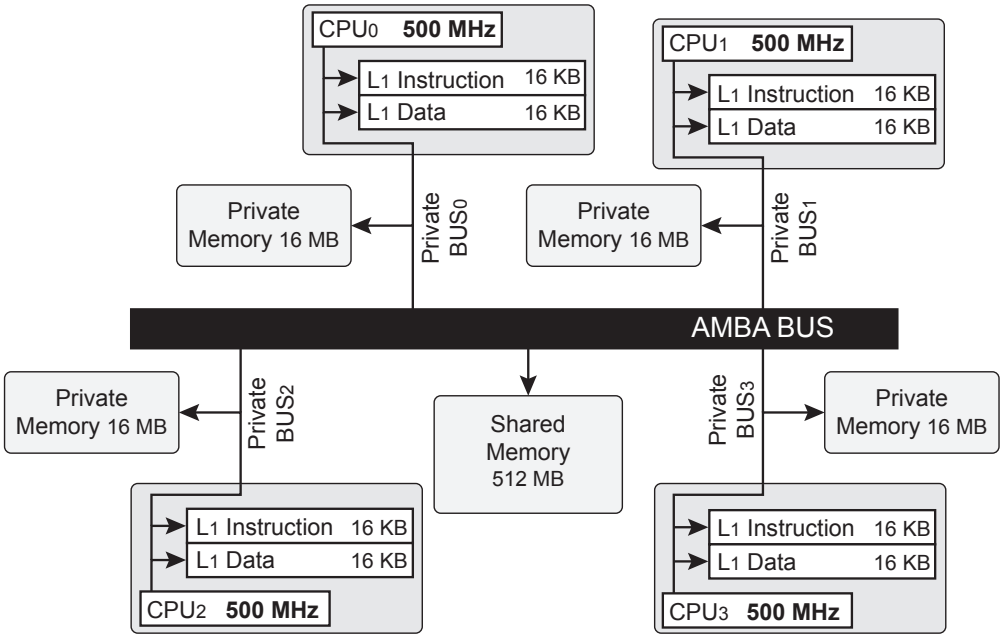

Figure 5.1: Homogeneous MPSoC Architecture for Evaluation

in this work. A precise model for these resources also leads to a more complex ILP formulation or optimization and is not included here.

Furthermore, an ILP solution is not possible if the thread graph includes cycles. With proper analysis as breadth-first search, cycles can be found and deleted. The thread graph should also have one explicit start node and sink. The *Thread Model Extraction Tool*, described in Chapter 4, ensures that the thread graph is following these requirements.

ILP optimizations are known to work fast for smaller inputs, i.e. smaller or compact thread graphs. For more complex thread graphs, the optimization time can increase drastically. The ILP optimization could be stopped after a certain amount of time. In this case, the quality of the solution cannot be clearly validated.

5.4 Evaluation

5.4.1 Simulation Environment

This section describes the simulation environment of the ILP-based memory-aware mapping tool. Section 2.3.1 describes the underlying architecture model including a basic architecture which is illustrated in Figure 2.10.

Evaluation was performed for a homogeneous and a heterogeneous architecture. The homogeneous architecture is illustrated in Figure 5.1. The architecture consists of four ARM11 processors with a clock frequency of 500 MHz. Each processor has exclusive access to its own memories, i.e. one private main memory with 16 MB

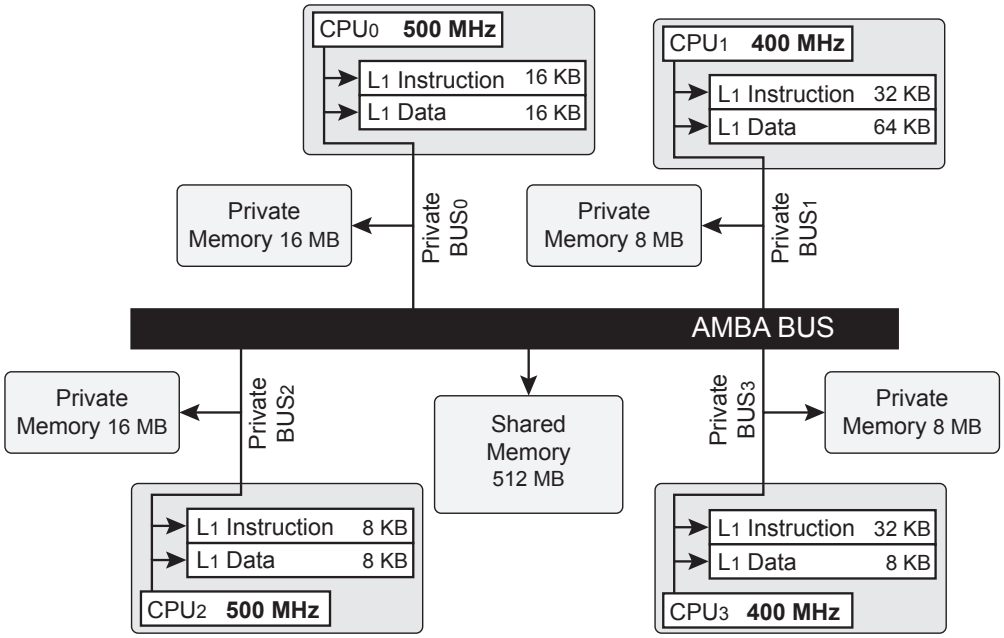

Figure 5.2: Heterogeneous MPSoC Architecture for Evaluation

size and an instruction and data scratchpad memory with 16 KB size. These memories can be reached by private buses. Furthermore, a shared DRAM memory with 512 MB size can be reached by each processor. This memory is accessed through an AMBA bus. This homogeneous architecture was implemented within the MACCv2 framework (described in Section 3.2.1).

The core characteristics of the heterogeneous architecture considered in this evaluation is comparable to the *single-ISA heterogeneous multi-core architecture* suggested in [110, 111]. Here, the authors propose a heterogeneous system with cores from the same architectural family that execute the same instruction set, but differ in power and performance values, cache sizes, raw execution bandwidth, or other characteristics. In [110], the authors compared their architecture against a homogeneous architecture, and the single-ISA heterogeneous architecture outperformed the homogeneous architecture by 63%. Based on this conclusion, our architecture contains processors from the same architectural family, but with different clock frequencies and thus different runtime and energy values that depend on the frequency of the processor. Furthermore, this architecture setup is also comparable to the state-of-the-art ARM big.little architecture ([20], [21]) which is also described in Section 1.2. In this architecture, *big* processors for heavy workloads are combined with *little* energy-efficient processors. They are also available as quad core with two of each processors. The processors used in this evaluation have nearly the same memory hierarchy as an ARM Cortex-A57 processor (illustrated in Figure 1.4.1). This processor is also used in the ARM big.little architecture.

Based on these assumptions, the heterogeneous architecture illustrated in Fig-

ure 5.2 was implemented within the MACCv2 framework (described in Section 3.2.1) The proposed architecture consists of four ARM11 processors with clock frequencies of 400 or 500 MHz. The memory subsystem is fully heterogeneous with different memory sizes on each level and different memory types on level one for the scratchpad memories. Processor CPU_0 has a clock frequency of 500 MHz and contains two level one memories, one instruction and one data scratchpad memory with a size of 16 KB, and one private memory with a size of 16 MB. Processor CPU_1 has a clock frequency of 400 MHZ and a 32 KB instruction and a 64 KB data scratchpad memory. Furthermore, it has a private main memory with 8 MB size. CPU_2 has a clock frequency of 500 MHz, an instruction and data scratchpad memory of 8 KB size as well as a private memory of 16 MB size. CPU_3 has a clock frequency of 400 MHz and access to an instruction scratchpad memory of 32 KB and a data scratchpad memory of 8 KB. The private memory of CPU_3 has a size of 8 MB. The system also contains one shared DRAM memory with 512 MB for shared data and instructions.

Both architectures were implemented for the underlying cycle-accurate COMET simulator (described in next Section 5.4.2), which is used for the validation of the evaluation.

5.4.2 Experimental Setup

In a first step, the parallelization tools [106, 94] are applied in order to obtain a parallelized and synchronized application code. Afterwards, the *Thread Model Extraction Tool* extracts the thread graphs and the architecture specification for the memory-aware mapping tool. The benchmarks, including the number of extracted threads and parallel sections, are described in Section 4.8.

The *Spectral*, all *Edge Detect* parallelization and *Mpeg4* benchmarks are the most complex benchmarks in this setup. The *Spectral* benchmark has 6 parallel sections and 25 threads. The *Mpeg4* benchmark is complex due to the large code size (3 MB) and large number of memory objects that can be potentially mapped to on-chip memories. All *Edge Detect* parallelization, with up to 16 threads per parallel section, represent more complex and heterogeneous threads. However, all benchmarks should represent a good average over mapping complexity and computational workload.

The commercial ILP solver CPLEX [112] is used for the minimization of the objective functions in this ILP formulation. Energy values are computed according to the model defined by Steinke et. al. [40] while the memory models are provided by CACTI [113]. CACTI is a tool for modeling energy consumption and access time for caches and other memories. For the validation of the solutions generated by the ILP-optimization, the cycle-accurate instruction set simulator CoMET [16] is used.

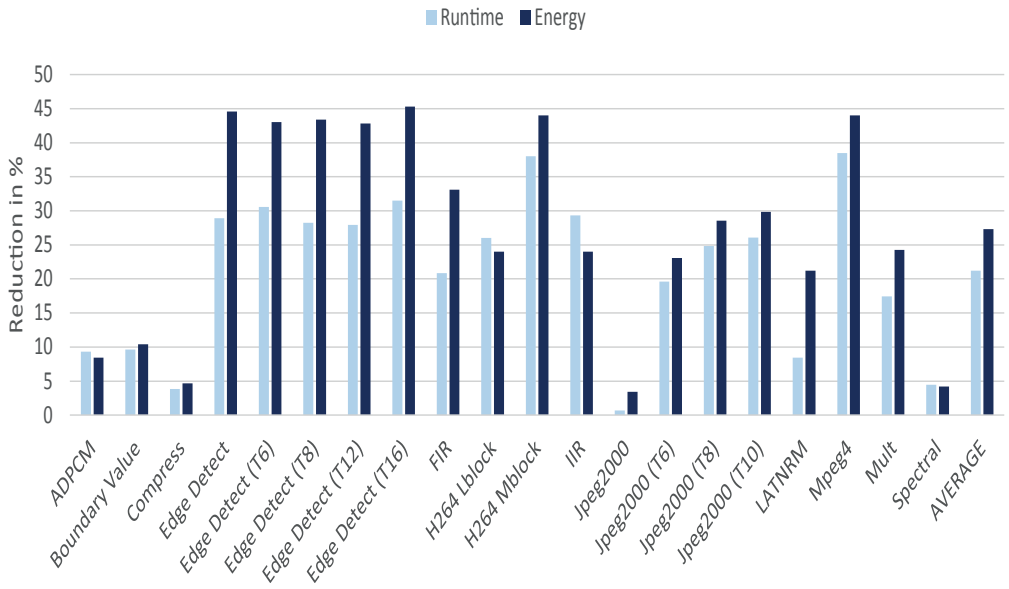

Figure 5.3: Reduction in runtime and energy achieved by Runtime-ILP for a homogeneous platform

5.4.3 Experimental results

5.4.3.1 Comparison against state-of-the-art mapping

In this section, the ILP optimization is validated by comparing the memory-aware mapping optimization against state-of-the-art mapping optimization. An ILP optimization was established, which represents a common state-of-the-art mapping optimization with the same underlying architecture and application model. The ILP optimization includes neither memory-awareness nor mapping of memory objects to on-chip memories. In this way, a comparable reference to the common state-of-the-art mapping optimizations is created. Furthermore, two different state-of-the-art ILP optimizations were established, one with the objective to minimize runtime and one with the objective to minimize the overall energy consumption. For the evaluation, all obtained solutions of both ILP-based mapping optimization tools (for energy and runtime) were simulated on the cycle-accurate CoMET simulator.

In the next subsections, the x-axis describes the different benchmarks, while the y-axis describes the reduction achieved by the memory-aware mapping optimization tool compared to the state-of-the-art mapping optimization tool. Although the goal of the ILP is to minimize the runtime, the resulting system's energy consumption is also illustrated and vice versa.

Homogeneous Architecture

The first set of experiments was performed for a homogeneous architecture which was described in Figure 5.1.

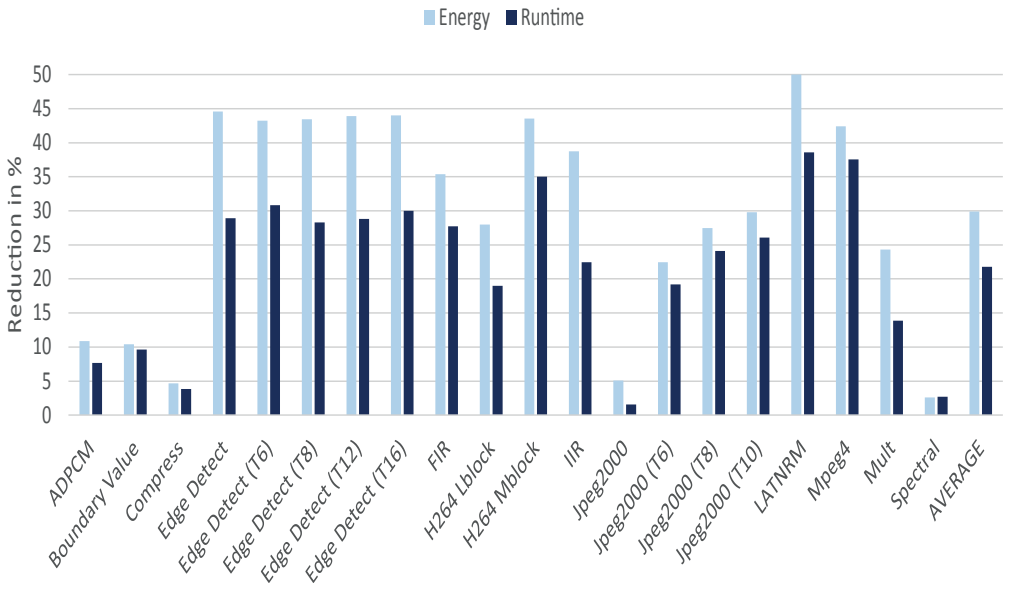

Figure 5.4: Reduction in runtime and energy achieved by Energy-ILP for a homogeneous platform

Figure 5.3 presents the ILP runtime optimization for a homogeneous platform. The best results were obtained for *H264-MBlock*, and *MPEG4* benchmarks with a reduction of runtime by about 38%. This large benefit is due to the fact that almost all memory objects of some benchmarks are mapped to faster and more energy-efficient scratchpad memories, i.e., except of those who are marked as private or shared. The different parallelization of *Edge Detect* and the *IIR* benchmark achieved also remarkable reductions by 28 to 31%. *JPEG2000* and Spectral benchmark reached only a reduction by 1 to 4.5% These benchmarks work primarily with shared data memory objects, which are not allowed to be mapped to a higher memory level. Nevertheless, the instruction code of their threads could be mapped to local scratchpad memories and gain some benefit. The average gain in the reduction of runtime is 21%. Next to the reduction of runtime, a reduction of energy consumption can be observed compared to the energy consumption of a state-of-the-art optimization. The reason for this could be the allocation of memory objects to the local scratchpad memories, which consume less energy per access.

The evaluation results for the energy based ILP memory-aware mapping optimization are shown in Figure 5.4. Please note that the base line here is not the same as in Figure 5.3. The underlying baseline of Figure 5.3 is a state-of-the-art ILP-based *runtime* optimization. In Figure 5.4, the baseline is a state-of-the-art ILP-based *energy* optimization. This means, the resulting values for energy consumption and runtime differ for both baselines. The most reduction in energy consumption was achieved by the *Edge Detect* benchmarks, *H264-MBlock*, *IIR* and *MPEG4* benchmarks. The energy was reduced by 38 to 44.5% compared to state-of-the-art map-

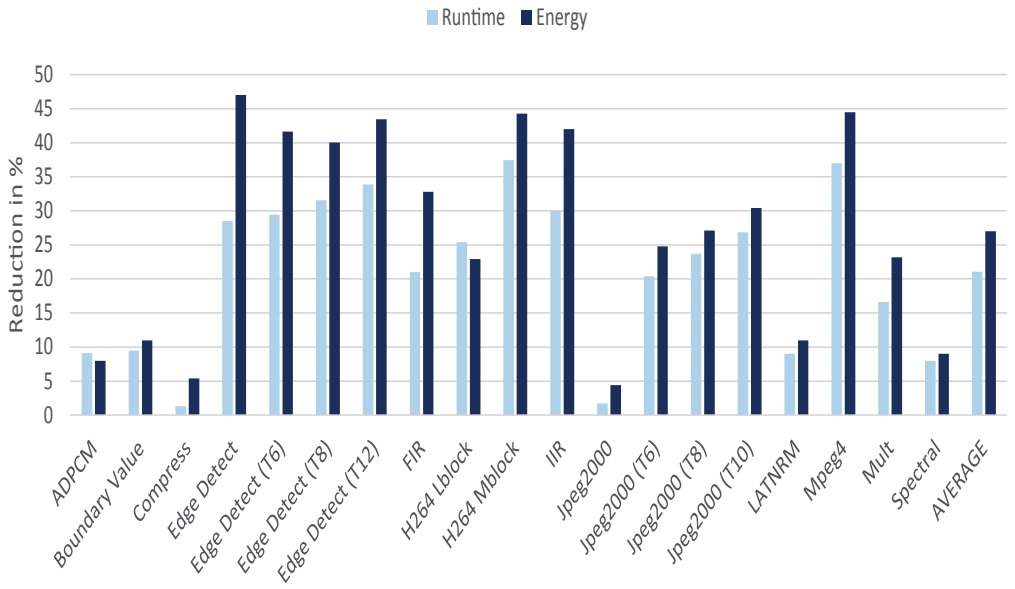

Figure 5.5: Reduction in runtime and energy achieved by Runtime-ILP for a heterogeneous platform

ping. However, *Compress*, *Spectral* and *JPEG2000* achieved a reduction in energy consumption by 2.6 to 5%. On average, the energy consumption was reduced by 28%. The resulting reduction in runtime is also illustrated in this figure. Compared to state-of-the-art mapping, an average reduction in runtime was achieved by 21.8%.

Heterogeneous Architecture

The second set of experiments was performed for a heterogeneous architecture which is described in Figure 5.2.

The results for the ILP-based optimization with the goal to minimize runtime for a heterogeneous platform are shown in Figure 5.5. As for the homogeneous platform, the benchmarks *MPEG4*, *H264-MBlock* and *IIR* showed the most reduction in runtime, from 30% to 37%. In the *MPEG4* benchmark, about 300 kB of code and data memory objects are allowed to be mapped onto the on-chip memories in the system. However, *Compress* and *JPEG2000* reached a reduction of 1.3% and 1.7%. The average reduction for runtime is about 21%. Concerning the resulting overall energy consumption of the system, over all benchmarks the energy was also reduced by a remarkable amount of 27%.

The evaluation results for the energy based ILP memory-aware mapping optimization for a heterogeneous platform are shown in Figure 5.6. As described in the evaluation of the homogenous platform, the base line for energy and runtime differs from the base line of Figure 5.5. In this setup, the benchmarks *MPEG4* and *Edge Detect* reached the most reduction of energy consumption by 55% and 60%, respectively. *Compress* and *Spectral* reached a reduction of about 5.2% and 4.4%.

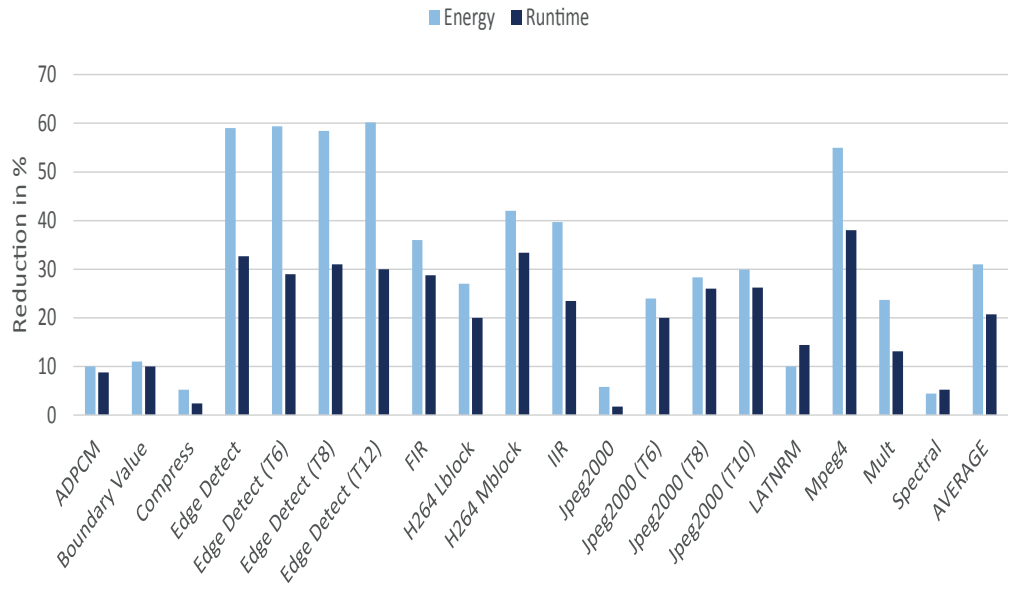

Figure 5.6: Reduction in energy and runtime achieved by Energy-ILP for a heterogeneous platform

The average energy reduction is about 31%. Compared to the overall runtime of the state-of-the-art mapping, a reduction of about 21% on average is achieved.

Comparing the resulting overall runtime of the energy based against the runtime based memory-aware ILP optimization, the overall runtime is increased by 31% by the energy based ILP optimization. On the other side, comparing the overall energy consumption of both optimizations, the energy is increased by 35% by the runtime based ILP optimization. This shows, that there is a trade-off between energy and runtime when optimizing for one of both optimization goals.

5.4.3.2 Runtime of the ILP optimization

This section presents the time that was required for the execution of the memory-aware ILP-based mapping optimization based on compact and detailed thread graphs, which contain either the average case or the worst-case execution time for the benchmarks. The runtime is illustrated in seconds for an AMD Opteron 2.46 GHz on the x-axis. The benchmarks and the average over all benchmarks are described on the y-axis. *WC Detail* and *AC Detail* represent the runtime for a detailed thread graph containing the worst case (WC) or average case execution cycles of the underlying application benchmark. On the other side, *WC Compact* and *AC Compact* represent a compact thread graph which contains the worst and average-case, respectively.

Figure 5.7 illustrates the runtime for the ILP optimization which optimizes the overall runtime for the homogeneous architecture. The average runtime for the compact thread graphs lies between 66 to 71 seconds while the average runtime for

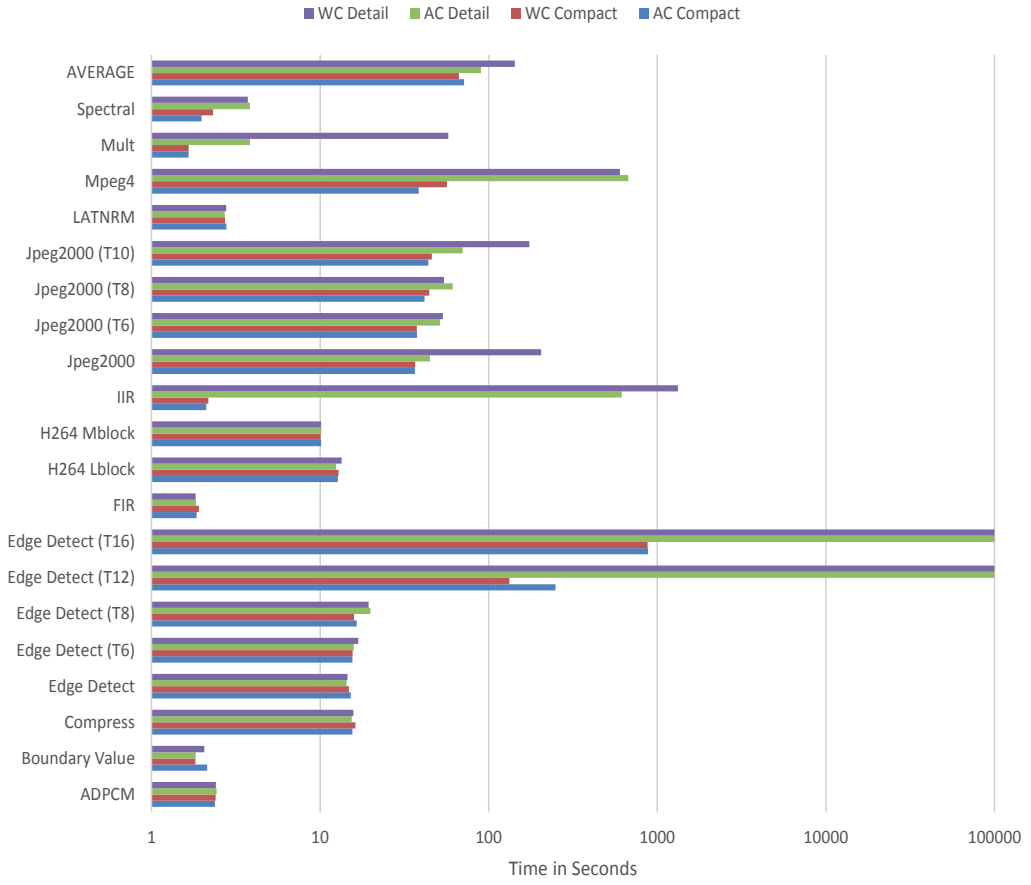

Figure 5.7: Runtime of ILP-optimization for runtime optimization, homogeneous architecture

detailed thread graphs lies between 89 and 142 seconds. The most time consuming benchmarks are *IIR* and *MPEG4* with 600 to 1324 seconds. The optimization for the detailed thread graphs of *Edge Detect* with 12 and 16 threads per parallel section were canceled after three days. Overall, a significant increase in runtime is shown for the more complex benchmarks.

Figure 5.8 illustrates the runtime for the ILP optimization which reduces the overall energy consumption for the homogeneous architecture. Since the optimization for energy is a little bit more complex, an increase in runtime can be observed for each benchmark, especially for the more complex benchmarks *IIR*, *MPEG4* and *Edge Detect (T12)* and *(T16)* The optimization of *Edge Detect (T12)* and *(T16)* was canceled after 3 days. For the compact thread graph of *Edge Detect (T16)*, it was possible to obtain a solution after 9.6 (AC) or 10 (WC) hours, respectively. The average runtime lies between 167 and 265 seconds for detailed thread graphs (not including the runtime for *Edge Detect (T12)* and *(T16)*) For compact thread graphs, the average time lies between 1.764 and 1.868 seconds (including *Edge Detect (T16)*)

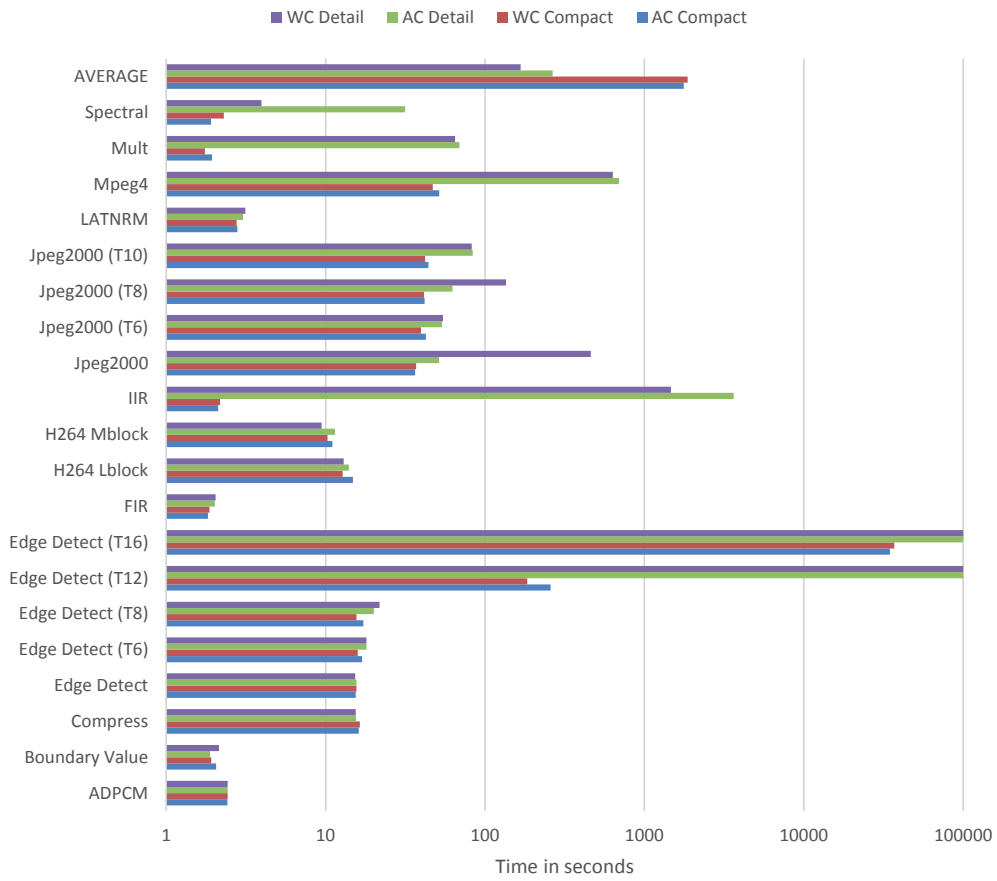

Figure 5.8: Runtime of ILP-optimization for energy optimization, homogeneous architecture

and between 26 and 31 seconds (without *Edge Detect (T16)*). These results shows: It is rather possible to obtain a solution for compact complex benchmarks than for detailed complex benchmarks.

Figure 5.9 illustrates the runtime for the ILP optimization which reduces the overall runtime for the heterogeneous architecture. Compared to the homogeneous architecture, an increase for all *JPEG2000* benchmarks as well as for the *MPEG4* and *Edge Detect (T8)* benchmarks can be observed. Contrary to the homogeneous architecture, it was not possible to obtain a solution for *Edge Detect (T12)* and *(T16)* for the detailed thread graph. However, it was also impossible to obtain a solution for *IIR* benchmark and for *Mult* for detailed thread graphs. The average runtime for compact thread graphs is about 184 seconds while the runtime for detailed thread graphs lies between 243 to 254 seconds.

Figure 5.10 illustrates the runtime for the ILP optimization which reduces the overall energy consumption for the heterogeneous architecture. Here, it is still impossible to obtain a solution for *IIR* and *Mult* benchmark. The runtime for *Edge*

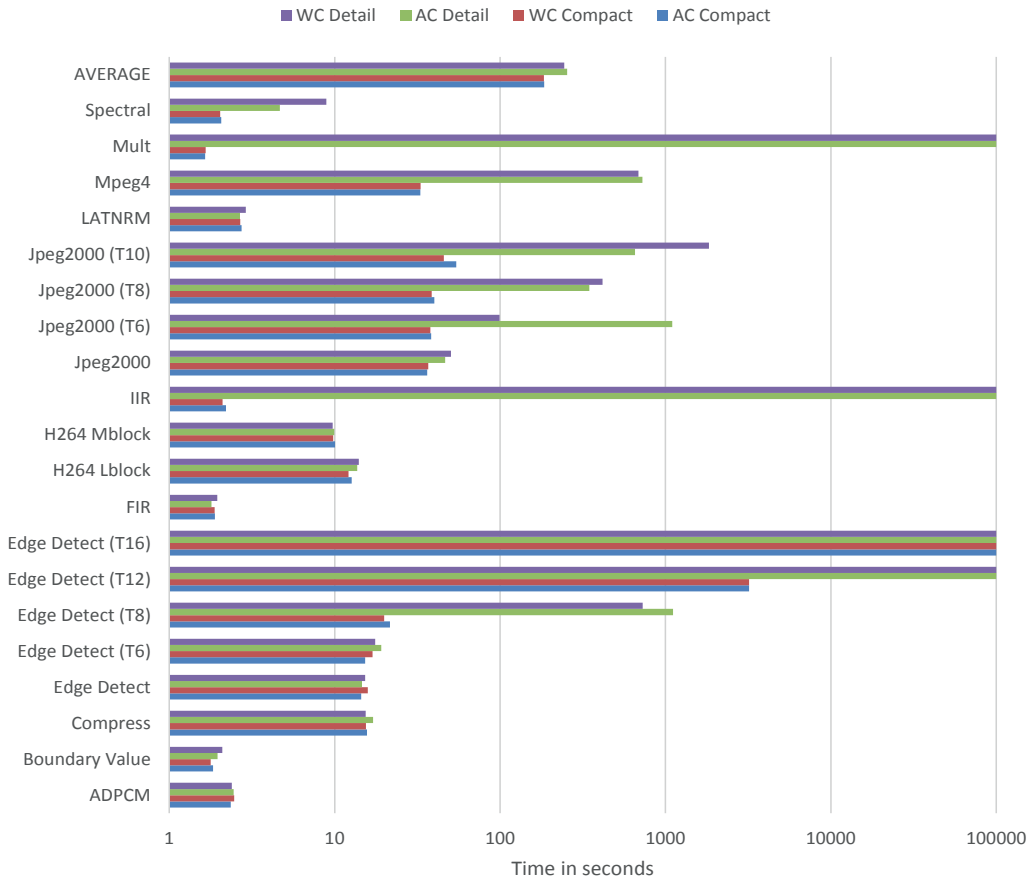

Figure 5.9: Runtime of ILP-optimization for runtime optimization, heterogeneous architecture

Detect (T8) increases to 6 hours or 14 hours, respectively. This shows that a heterogeneous architecture for detailed thread graphs and the optimization for energy is more complex. The average runtime lies about 495 seconds for compact thread graphs and about 3314 seconds (55 min) to 1416 seconds (23 min) for detailed thread graphs.

5.4.4 Conclusions

This section presented an ILP-based memory-aware mapping optimization for homogeneous and heterogeneous MPSoC systems with memory hierarchies. Next to the mapping of application threads to processors, the optimization also allocates frequently used instruction and data memory objects to the different memories in the hierarchy. In this way, the underlying architecture capabilities are exploited and efficiently matched to the application's requirements.

The memory-aware mapping optimization is evaluated by comparing the pro-

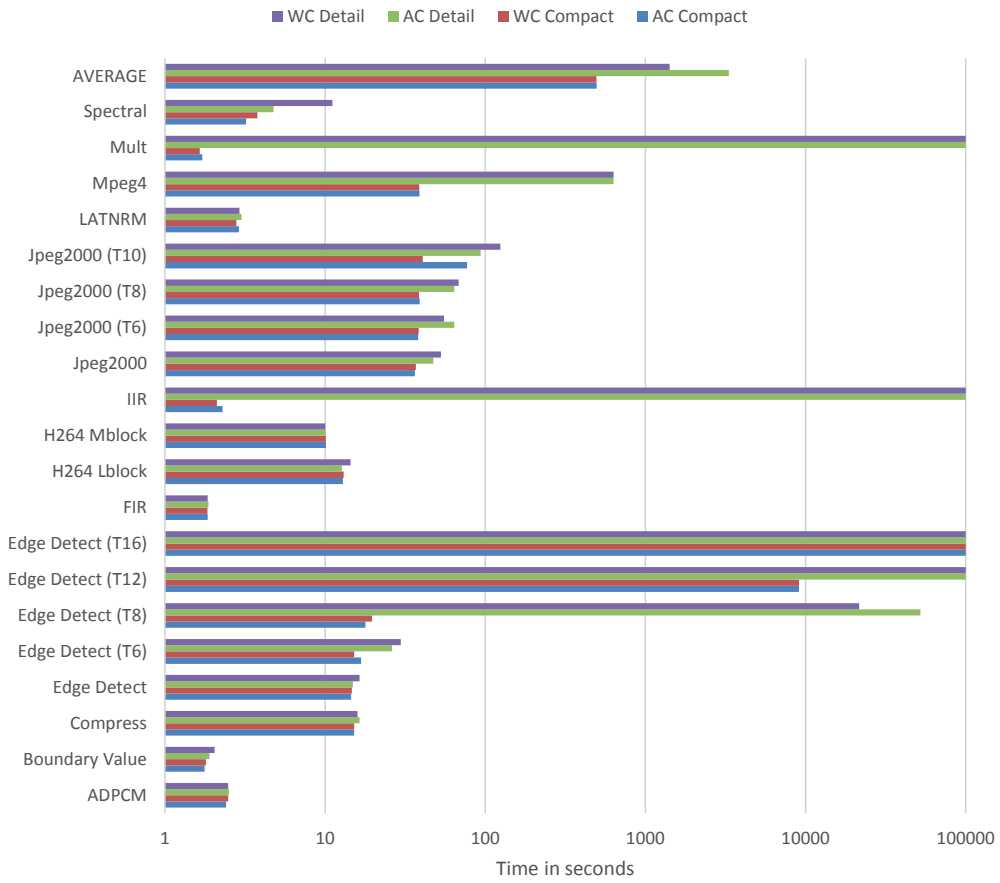

Figure 5.10: Runtime of ILP-optimization for energy optimization, heterogeneous architecture

posed ILP approach to a state-of-the-art mapping optimization, which maps threads to processors, and includes neither the consideration of the memory hierarchy nor the mapping of instruction and data memory objects to memories. For the runtime optimizations of homogeneous and heterogeneous systems, an average reduction of about 21% was reached. For the optimization of energy consumption, a reduction of 28 to 31% was gained.

The most reduction in energy consumption and runtime was observed for benchmarks, which contain many allocatable instructions and data that can be mapped to the faster and more energy-efficient local memory. On the other side, some benchmarks spent the most time accessing shared memory objects. Therefore, only little speed-up or energy reduction can be reached for those benchmarks.

The runtime for an ILP optimization depends on many factors, as the optimization itself (energy or runtime optimization). Furthermore, it depends on the complexity of the benchmarks, the complexity of the thread graph (compact vs. detailed) and on the architecture. While it was possible to obtain a solution for

Edge Detect (T16) in the energy-based ILP optimization for homogeneous architecture for a compact thread graph, it is not possible anymore in the heterogeneous architecture. Furthermore, for some benchmarks it was even not possible to obtain a solution for detailed thread graphs for the heterogeneous architecture. However, it is possible to obtain a solution for compact complex benchmarks rather than for detailed complex benchmarks. Benchmarks that are more complex require more runtime for their optimization.

The ILP-based memory-aware mapping optimization tool is integrated in the automatic MNEMEE tool-flow, where all steps such as application partitioning, parallelization, and memory-aware mapping optimization can be combined in order to help designers in their decision.

Chapter 6

Memory-Aware Multiobjective Mapping Optimization

Contents

6.1 Introduction

Several embedded systems have to fulfill many requirements as providing high performance, low energy consumption, managing concurrency on the system or meeting important deadlines. Some optimization goals that have to be performed for such systems can stand in conflict to each other, e.g. reducing the energy consumption and increasing performance. For example, a high-performance system could increase the processor activity and try to utilize as much performance as possible. Thus, more energy is consumed through higher processor activity. On the other side, a system which has to reduce energy consumption tries to run the processor in an energy mode which reduces power but increases the runtime, e.g. as it is performed in dynamic voltage scaling (DVS) processors. Another strategy switches off some of the processors or processor cores in order to save energy consumption.

Embedded mobile systems as tablets, mobile phones or mobile DVD player usually have to provide high performance and low energy consumption. For such multiobjective optimizations, there are usually many (Pareto optimal) solutions in the huge design and solution space. These systems provide many resources, which influence the optimization goals directly and indirectly. Processors, buses and memories

directly influence performance and energy consumption. The structure of each component can be complex. A good example is the structure and characteristic of a processor or a memory. A processor can have different strategies for reducing energy consumption or boosting the performance: Dynamic voltage or frequency scaling, turning off processor cores or other power management strategies.

However, the memory subsystem also contributes significantly to the energy consumption. In homogeneous and heterogeneous systems, the processors can have access to different memories on different levels, which all have different characteristics and thus influence the energy consumption and performance. More levels of memory results in higher optimization complexity. All this complexity influences the optimization objectives and increases the solution space. Finding an optimal solution is a challenging task in this case. The embedded system designer has to find a solution that satisfies all optimization demands. Heterogeneous Systems provide more manifold resource characteristics, which additionally increase optimization complexity and the solution space. All resources are constrained and have to be efficiently utilized in order to satisfy the multiobjective optimization goals.

One effective and well-known optimization strategy for multiobjective goals is the usage of evolutionary algorithms that can explore the solution space effectively. In this section, a multiobjective optimization tool based on evolutionary algorithms is presented. This tool is integrated in the MNEMEE tool flow and has two optimization goals: the reduction of runtime (increase of performance) and the reduction of the energy consumption. These goals are integrated in the memory-aware mapping optimization process, which has to find an efficient thread to processor mapping. This thread to processor mapping is combined with the efficient mapping of the memory objects of the threads to the available memories in the memory hierarchy. In this way, the requirements of the application are matched to the available resources of the system in order to obtain an effective utilization even for complex homogeneous and heterogeneous architectures.

Related Work

There are several frameworks for design space exploration for multiprocessor systems with multiobjective goals, as Daedalus [77], SystemCoDesigner [86], HOPES [75], or DOL [18]. All these frameworks use different evaluation environments, different strategies to search in the design space, different optimization criteria, design constraints, applications, architectures, mapping models, or abstraction levels. These individual characteristics do not allow a comparison between all these frameworks.

The proposed technique in this chapter is constructed on top of the basic optimization in DOL (SHAPES). In particular, the DOL framework was chosen because of the modular structure and flexible models for performance analysis and search in the design space [114, 84]. As DOL provides the basic mapping optimization framework, it allows to investigate and develop the proposed novel memory mapping on top of it, while considering a multilevel on-chip memory hierarchy. To achieve this, several refinements were necessary within DOL, such as adaption to thread mapping

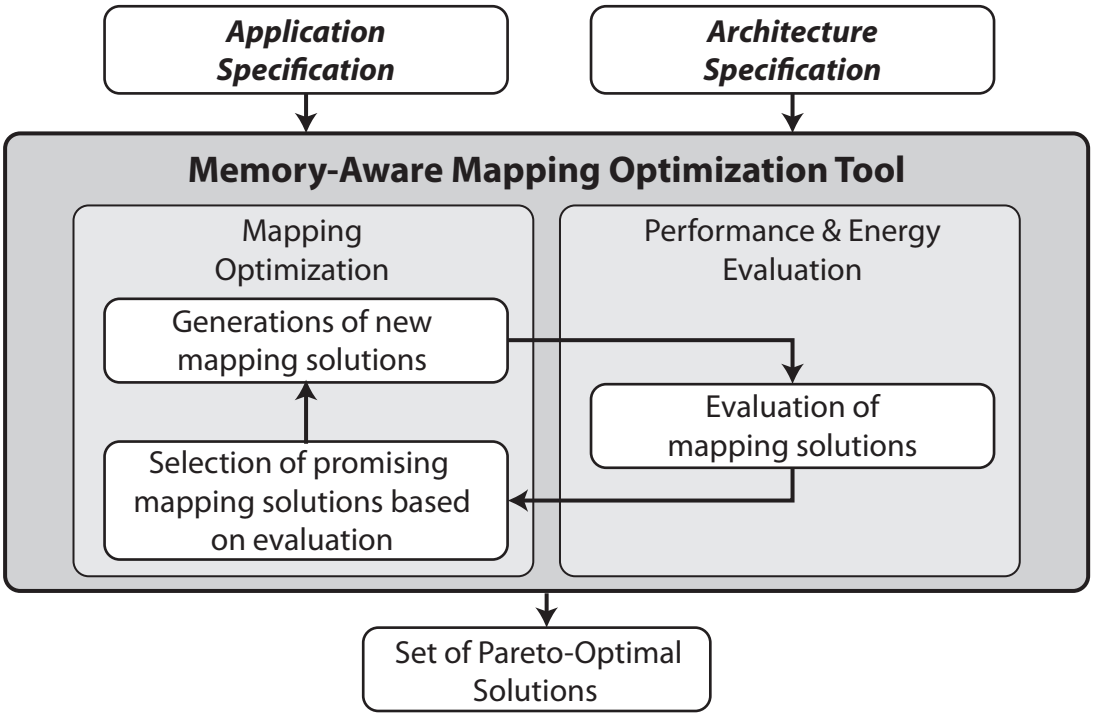

Figure 6.1: Overview over the *Memory-Aware Multiobjective Mapping Optimization Tool*

(instead of Kahn process-based mapping) and different performance analysis models for the runtime of threads. Furthermore, an energy model for the analysis and specification of the systems energy consumption was completely integrated, as well as the entire explicit memory modeling and memory optimization within specification, search, and analysis.

A more detailed overview of all related work is presented in Section 2.4.6.

6.2 Tool Overview

The *Memory-Aware Multiobjective Mapping Optimization Tool* (shortly: MAMMOT) is integrated as a MACCv2 tool in the MNEMEE tool flow. Figure 6.1 illustrates the structure as well as the input and output of this tool. The input for the tool is generated by the foregoing tools in the MNEMEE tool flow, i.e. the parallelization and synchronization tools and the *Thread Model Extraction Tool*. These tools provide the parallelized and synchronized source code and a flat thread graph as input for the mapping tool. The architecture database in MACCv2/MNEMEE provides all architecture relevant information. A detailed description of the underlying models is given in Section 2.3. The DOL tool is based on Java while all tools in the MNEMEE framework are based on C and C++. The DOL framework was fully integrated in the *Memory-Aware Multiobjective Mapping Optimization Tool*. It

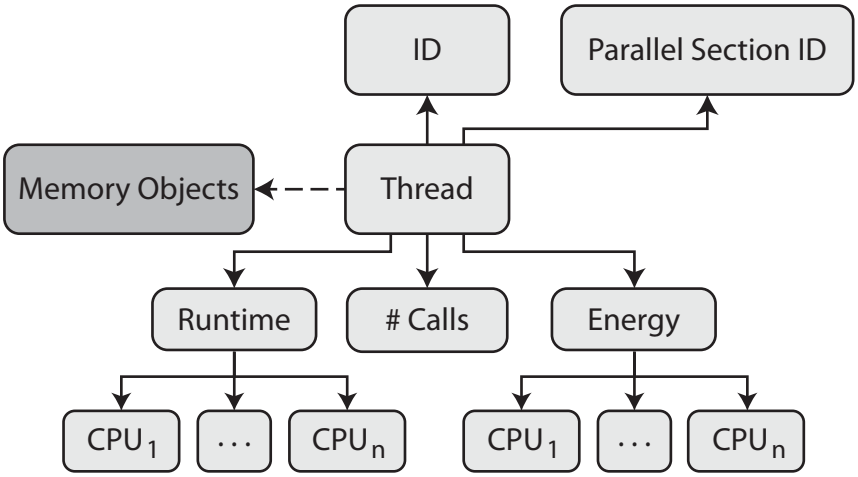

Figure 6.2: Thread specification

generates the input for the DOL framework and calls this framework, providing all files and parameters, within the MNEMEE tool flow.

6.2.1 Application specification

The application specification is described in XML. As a preprocessing step, the *Memory-Aware Multiobjective Mapping Optimization Tool* reads the flat thread graph which was generated by the *Thread Model Extraction Tool*. Afterwards, it generates an application specification file, which integrates all information of the flat thread graph. The application specification lists all threads. Each thread is specified by its thread id and all parallel sections in which the thread is executed. The parallel sections are also defined as ids. Additionally, the numbers of accesses to the thread are listed. Since the execution time and energy consumption of a thread on a processor can differ for the individual processors, all these runtime and energy values are specified for each processor. These values represent the energy consumption and runtime for one thread call. They do not include the energy consumption and runtime for the accesses to the bus and memories. An overview of the thread specification is given in Figure 6.2.

Each thread specifies all memory objects that are accessed. Each memory object is defined by its id. Furthermore, it contains information about the number of read and write accesses, its size, its basic size (in case of an array), its access width and access type (i.e. instruction or data). The mapping of the memory object can also be given, i.e. shared memory objects and memory objects that have to be mapped to private memory are marked as "‘shared" or "’private"‘, respectively. A mapping of "’0"‘ specifies that the memory object can be mapped to any accessible memory by the optimization tool. The key word "’local"‘ can be used to map certain memory objects to the local memories, if required by the designer. All shared memory objects are listed under the main thread which has the id 0. An overview of the memory

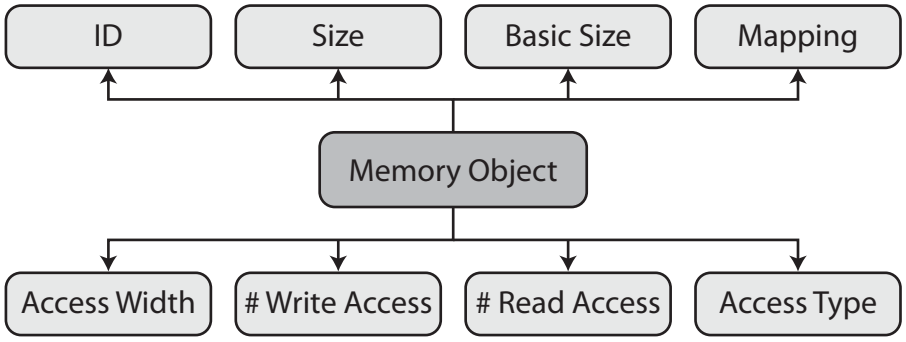

Figure 6.3: Memory object specification

object specification is given in Figure 6.3.

Additionally, the application specification lists all FIFO channels with source and target thread node, including the number of reads, writes and the data size in bytes. However, sometimes the designer would like to allocate threads to a specific processor like for instance special signal processing tasks on a DSP or critical code such as interrupt routines to a fast memory. In the application specification, it is possible for designers to manually enforce mappings of threads to specific processors by listing only one processor in the thread specification. By this, only this one processor will be considered in the thread to processor mapping step.

6.2.2 Architecture Specififcation

The architecture specification is also described in XML. The MACCv2 architecture database is read and the architecture specification XML file is generated. It lists all processors including their ids and their idle energy. Furthermore, all buses are specified including their frequency and bytes per cycle. All memories are listed including their ids, the type of the memory (scratchpad, private or shared), the size and the access type (i.e. instruction, data or unified). The overview of the memory specification is given in Figure 6.4. Additionally, each memory contains information about CPU access, i.e. the runtime and energy required for a CPU to

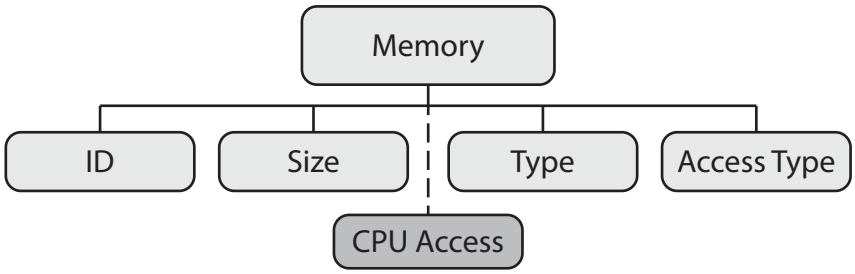

Figure 6.4: Memory specification

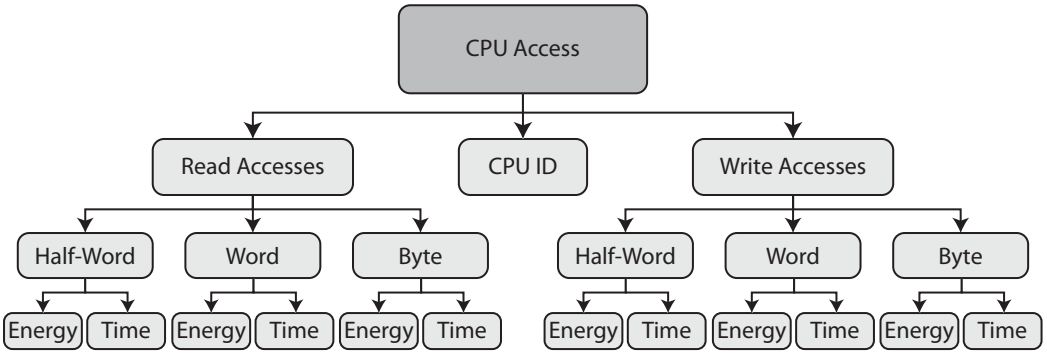

Figure 6.5: CPU specification

access this memory. Here, only CPUs are listed that have access to this memory. For each processor that has access to this memory, a distinction between read and write access is made. For each read and write access, the access time and energy consumption is specified for each access width (i.e. word, byte and half-word). Furthermore, for all connection between processor and memory, it is specified if it is a read or write path, including the number of cycles that is required for this access. Figure 6.5 gives an overview for the CPU accesses within the memory specification.

6.2.3 Mapping Optimization

The core part of the memory-aware mapping tool is an evolutionary algorithm (EA) which generates memory-aware mapping solutions. The application and architecture specification provide all information and constraints that is required in order to generate valid mapping solutions. Each mapping solution is evaluated for performance and energy consumption. For this, an analytical model is included in the tool. Based on this analysis, promising mapping solutions are selected and archived. Based on these solutions, new mappings are generated. The generation and evaluation of the mapping solutions are repeated in a design space exploration loop until an end criterion (i.e. maximum number of iterations) is reached. Since the mapping problem is multiobjective, there is no single optimal solution but a set of Pareto optimal solutions. The designer can decide which solution is most suitable for the design requirements of the underlying system. In some cases, it could happen that the optimization problem cannot be solved. For example, if the application memory requirements do not match the available memory capacity of the system. Then, the tool would stop with an appropriate message. A more detailed description of the evolutionary algorithm is presented in Section 6.4.

6.3 Optimization Objectives

This section provides a formal description of the optimization objective functions used in the *Memory-Aware Multiobjective Mapping Optimization Tool* for evaluating

the mapping solutions within the evolutionary algorithm. All values required in these objective functions are extracted from the input specification files.
Basically, two functions are considered to be optimized, i.e., system runtime and energy consumption.

Following notations are used for the definition of the equations in this section:

- the set of threads is defined by $T = 1, ..., k$, with thread $t \in T$ and
- the main thread $t_0 \in T$
- the set of processors is defined by $P = 1, .., p$, with processor $proc \in P$
- the set of parallel sections is defined as $S = 1, ..., s$, with a parallel section $ps \in S$
- the number of calls of a thread is specified with $nrCalls(t)$
- the set of memories with $M = 1, ..., m$, with memory $mem \in M$
- the set $M(proc)$ contains all memories mem that are accessible by processor $proc$
- MO is the set of memory objects, with memory object $mObj \in MO$
- $r(t, proc)$ represents the runtime of thread t on processor $proc$, without considering memory accesses
- $r(t, mem)$ indicates the time of memory accesses caused by the memory objects of thread t for access on memory mem, including the runtime to access buses
- $nrReadAcc(mObj)$ defines the number of read accesses to the memory object $mObj$
- $nrWriteAcc(mObj)$ defines the number of write accesses to the memory object $mObj$
- $accWidth(mObj)$ decribes the access width (i.e. Byte, half-word and word) of memory object $mObj$
- $accTimeRead(accWidth(mObj))$ represents the time required for a read access to memory object $mObj$ with access width $accWidth(mObj)$
- $accTimeWrite(accWidth(mObj))$ represents the time required for a write access to memory object $mObj$ with access width $accWidth(mObj)$
- $nrLoadAcc(accWidth(mObj))$ describes additional accesses to the memory object $mObj$ which are optionally required in order to obtain the full memory object size
- $e(t, proc)$ defines the energy that is consumed by thread t processor prc

- $e(t, mem)$ defines the energy consumed by thread t for accesses to memory mem
- $ActiveCycles(t)$ describe the number of cycles required for the execution of thread t
- $ActiveEnergyPerCycle(proc)$ describes the energy consumption for one cycle on processor $proc$
- $IdleEnergyPerCycle(proc)$ describes the energy consumption in idle mode for one cycle on processor $proc$
- $maxCycles_ps(t)$ represents the maximum number of cycles spend in the parallel section that thread t belongs to
- $energyRead(accWidth(mObj))$ represents the energy consumption required for a read access to memory object $mObj$ with access width $accWidth(mObj)$
- $energyWrite(accWidth(mObj))$represents the energy consumption required for a write access to memory object $mObj$ with access width $accWidth(mObj)$
- $MemAccessCycles(t)$ defines the number of cycles that are required for memory access for thread t

The total runtime for a thread is computed by the runtime of the thread on the processor it is mapped to, added to the total runtime for the bus and memory accesses that are performed to the appropriate memory object. The obtained value represents the runtime for one thread call and therefore has to be multiplied by the total number of thread calls. The overall runtime of the system includes the runtime of the main thread added to the runtime spend in the parallel sections. The runtime in a parallel section is specified by the threads that run in parallel in this section. The execution time of the parallel section is defined by the execution time of the thread with the maximum runtime in this parallel section.

The system runtime objective is given in Equation 6.1:

$$\begin{aligned} obj_1 = (nrCalls(t_0) * (r(t_0, proc) \ + \sum_{mem \in M(proc)} r(t_0, mem))) \\ + \sum_{ps \in \ S} (\max_{t \in ps} \{nrCalls(t) * (r(t, proc) + \sum_{mem \in M(proc)} r(t, mem))\}) \end{aligned} \tag{6.1}$$

The first line of Equation 6.1 represents the total execution time of the main thread t_0. Here, the number of calls to the main thread is multiplied by its runtime for one call. The runtime r includes the execution time on the mapped processor $r(t_0, proc)$ and the runtime for memory accesses $r(t_0, mem)$. The runtime for memory accesses is described in more detail in the next equation. However, the second

part iterates over all parallel sections $ps \in S$. The overall runtime of a parallel section is given by the most time consuming thread in this section. The memory access runtime of this thread is added to the overall runtime. For this, an iteration over the runtime of all memories *mem* that are accessible by the processor *proc* is performed.

More precisely, the memory access time caused by thread t is described with the following equation:

$$\begin{aligned} r(t, mem) = \sum_{\forall\ mObj \to mem} & (nrReadAcc(mObj) \\ & * accTimeRead(accWidth(mObj)) \\ & * nrLoadAcc(accWidth(mObj)) \\ & + nrWriteAcc(mObj) \\ & * accTimeWrite(accWidth(mObj)) \\ & * nrLoadAcc(accWidth(mObj)) \end{aligned}$$

Here, the equation iterates over all memory objects *mObj* that are mapped to ($\to$) memory *mem*. The total access time is obtained by the number of read and write accesses $nrReadAcc/nrWriteAcc$ to the memory object *mObj* multiplied by the time required for each access ($accTimeRead/accTimeWrite$). The different accesses cause distinctive energy and runtime values which also depend on the access width $accWidth(mObj)$ of the memory object *mObj*. Sometimes, additional accesses ($nrLoadAcc$) have to be performed in order to obtain the full memory object size.

The second objective function in the memory-aware mapping optimization represents the energy consumption of the system. For this, the energy, that is spend by each thread on the processor it is mapped to, is added to the energy that is consumed for the accesses to the memories and their underlying buses. The energy consumption for the memories is dependent on the memory objects of the threads and their mapping to the different memories in the memory hierarchy.

The second objective, which is representing the systems energy consumption, is defined as follows:

$$obj_2 = \sum_{proc \in P} \{ \sum_{\forall\ t \to proc} nrCalls(t) * (e(t, proc) + \sum_{\substack{\forall\ mem\ acc. \\ by\ proc}} e(t, mem)) \} \quad (6.2)$$

This equation iterates over all processors $proc \in P$. Here, for each thread t mapped ($\to$) to processor *proc*, the energy consumption has to be calculated. In detail, the number of calls $nrCalls(t)$ of thread t is multiplied by the energy $e(t, proc)$ consumed by thread t on processor *proc* added to the energy $e(t, mem)$ consumed by accesses to the memories *mem*.

The equation for the energy consumption of thread t, which is represented by $e(t, proc)$, is specified as follows:

$$\begin{aligned} e(t, proc) = & (ActiveCycles(t) * ActiveEnergyPerCycle(proc)) \\ & + ((maxCycles_p s(t) - ActiveCycles(t)) * IdleEnergyPerCycle(proc)) r45 \end{aligned}$$

The number of cycles for a thread t, i.e. $ActiveCycles(t)$, is multiplied by the energy per cycle consumed by the specific processor $ActiveEnergyPerCycle(proc)$. Thus, the first part of the equation represents the energy consumed in the active mode of the processor. The second part represents the energy consumed in the processor's idle mode. For this, the maximum cycles $maxCycles_ps(t)$ spend in the parallel section ps, that thread t belongs to, has to be considered.

In detail, the parallel section consists of several threads that run in parallel. The maximum cycles spend in a parallel section is given by the thread which consumes the most cycles for its execution. These cycles include the active cycles for the execution of the thread and the cycles that are required for the access to the memories. For the other processors, where a mapped thread t requires less cycles, the rest of the cycles (until the parallel section ends) is spent in idle mode. The energy for the idle mode has also to be considered in the overall energy consumption.

If a thread t, that is mapped to processor $proc$, requires less cycles than the maximum cycles of a parallel section, the processor spends the rest of the time (cycles) in idle mode. The energy consumption in idle mode is given by the idle time. The idle time is given in cycles and multiplied with the idle energy consumption for one cycle $IdleEnergyPerCycle(proc)$ on processor $proc$.

The main thread t_0 does not belong to a parallel section. Thus, this thread is executed in active mode on the processors $proc$ it is mapped to. Additionally, the energy for idle mode has to be considered when memory access is performed. This is represented by the value $MemAccessCycles(t_0)$ The energy equation is represented by:

$$\begin{aligned} e(t_0, proc) = &(ActiveCycles(t_0) * ActiveEnergyPerCycle(proc)) \\ &+(MemAccessCycles(t_0) * IdleEnergyPerCycle(proc)) \end{aligned}$$

The equation for the energy consumption of memory accesses $e(t, mem)$ is defined as follows:

$$\begin{aligned} e(t, mem) = \sum_{\forall\ mObj \to mem} &(nrReadAcc(mObj) \\ &* energyRead(accWidth(mObj)) \\ &* nrLoadAcc(accWidth(mObj)) \\ &+ nrWriteAcc(mObj) \\ &* energyWrite(accWidth(mObj)) \\ &* nrLoadAcc(accWidth(mObj))) \end{aligned}$$

This equation represents the energy consumption for memory accesses to an individual memory mem accessed by thread t. An iteration over all memory objects $mObj$ that are mapped to memory mem is performed. Here, the number of accesses $nrReadAcc$ / $nrWriteAcc$ has to be multiplied by the energy consumed for a single access to memory mem. Again, a distinction is made between read and write accesses and the access width $accWidth(mObj)$ for both. These are given by

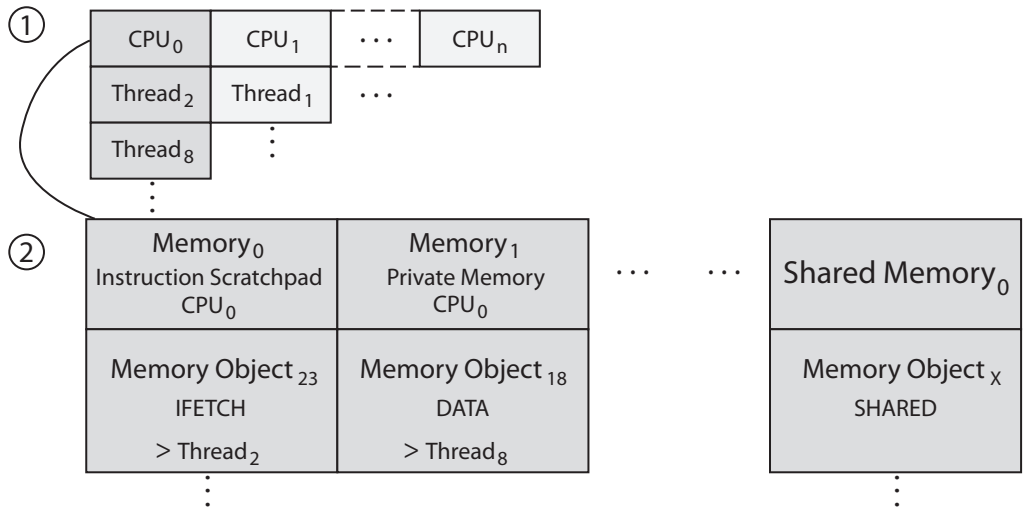

Figure 6.6: Individual representing a mapping solution candidate.

the values $energyRead$ ($accWidth(mObj)$) and $energyWrite$ ($accWidth(mObj)$). Finally, the number of additional accesses $nrLoadAcc$ ($accWidth(mObj)$), which is caused by the access width of the memory object, is multiplied by the overall read and write accesses, respectively. Please note that the energy consumed by bus accesses to memory, are already included in the $energyRead(accWidth(mObj)$ and $energyWrite(accWidth(mObj)$ values.

6.4 Evolutionary Algorithm

The evolutionary algorithm is used as a black-box optimization for solving the memory-aware mapping problem. In the first step, the evolutionary algorithm (EA) generates a population with several individuals. Each individual represents a mapping solution candidate. An example of an individual is illustrated in Figure 6.6. An individual is given by a complete thread to processor mapping and a complete mapping of the memory objects to the available memories. A fitness value is evaluated for each individual in the optimization loop of the evolutionary algorithm. This value represents the quality of an individual.

The mapping decisions are inspected randomly, and are generated in the following way: A thread and the corresponding memory objects are mapped randomly to accessible processors and their memories. Figure 6.6 (part 1) shows a possible distribution of threads to different CPUs.

After a processor is chosen, the memory objects of the threads are mapped to accessible memories. This is illustrated in Figure 6.6 (part 2). First, all predefined memory object mappings are performed, i.e., those memory objects which are marked as *shared*, *local* or *private* are mapped to the respective memory. Afterwards, the remaining memory objects of the threads are randomly mapped to the available

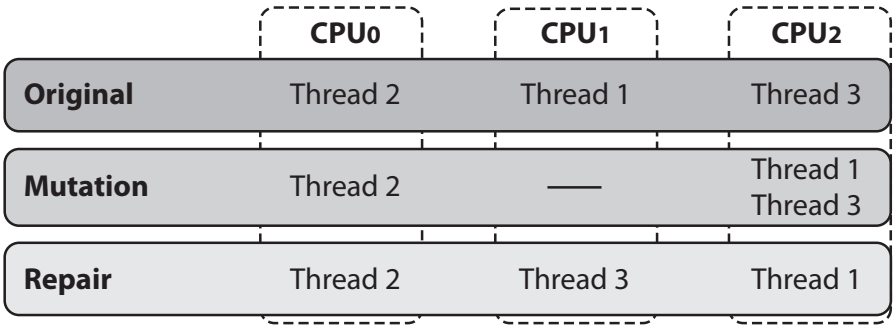

Figure 6.7: Mutation of genes

memories. This step is performed for all threads of the application. The algorithm takes care that only valid mappings are generated, i.e. the memory objects are mapped to memories that are accessible by the processor to which the thread was mapped. Furthermore, if the memory is specified as instruction or data only memory, only instructions or data memory objects are mapped to these memories. The evolutionary algorithm also verifies that the memory sizes are not exceeded during mapping. If it is not possible to extract valid mappings, because the memories are too small for all application code and data, the evolutionary algorithm stops with a corresponding message. If a memory at a lower level has no capacity anymore, the remaining memory objects will be mapped to a memory at a higher level in the memory hierarchy. If all memories are occupied, the remaining memory objects will be mapped to a larger, shared or a common main memory.

In the next step of the evolutionary algorithm, the generated individuals are evaluated. Here, a fast *performance and energy evaluation* is performed for each individual based on the objective functions defined in Section 6.3. After evaluation, the best individuals are saved in an archive. Based on the previous solutions, new individuals are generated. This is performed by random mutation or crossover on the solutions.

In the *mutation step*, a thread and its corresponding memory objects are switched to another processor, an example is shown in Figure 6.7. In this example, the mutation step decides that $Thread1$, which was originally mapped to CPU_1, is now mapped to CPU_2. After mutation, it is possible that two threads within a parallel section are mapped to one processor. For the case that the number of threads is smaller or equal to the number of CPUs, this could result in a suboptimal solution since the threads are supposed to run in parallel. This issue should be avoided since by maintaining these suboptimal solutions, the evolutionary algorithm requires more generations to find better solutions. Therefore, a so-called 'repair' step is required for this case, which checks for double assignment on the processors and remaps one thread to a free processor if available. Thus, in the example of Figure 6.7, *repair* remaps $Thread3$ and its corresponding memory objects to CPU_1. However, in some cases more than one thread could be mapped to a processor when the number of

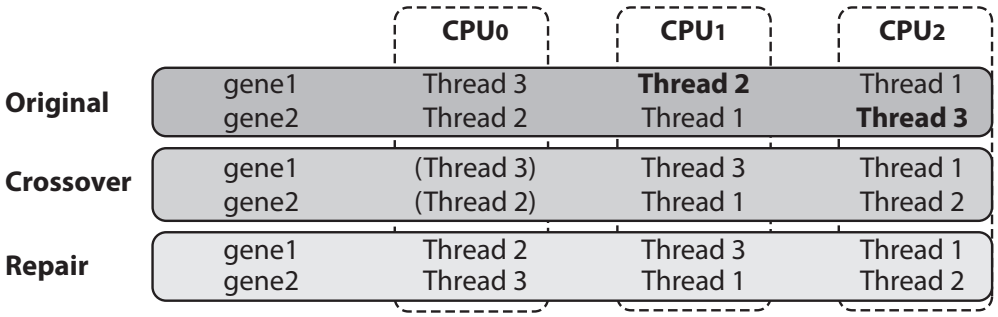

Figure 6.8: Crossover of genes

threads is larger than the number of processors. In this case, the algorithm calculates the maximum number of threads that are allowed to be mapped on a processor. For this, the number of threads is divided by the number of CPUs. This indicates the maximum number of threads on a CPU. If this division results with a rest, the maximum number of threads is increased by one. This solution is possible since the foregoing parallelization tool extracts threads with equal loads.

In addition to *mutation*, a *crossover* process can be performed. Here, two different mapping solutions (individuals) are considered, that are coded into so-called 'genes' During crossover, for each gene, a single random thread to processor mapping is chosen to be exchanged between individuals, as exemplified in Figure 6.8. Originally, in $gene1$, $Thread2$ was mapped to CPU_1 and in $gene2$, $Thread3$ was mapped to CPU_2. In the crossover process, a remap of $Thread3$ to CPU_1 in $gene1$ and in $gene2$ a remap of $Thread2$ to CPU_2 is performed. Now, a repair operation has to be performed again, because this mapping causes invalid mapping solutions, since $Thread2$ in $gene2$ and $Thread3$ in $gene1$ are mapped twice (shown in brackets). In the example, a remap of $Thread2$ to the processor, where $Thread3$ was originally mapped to, is performed. Again, remapping repair is performed (for $gene1$ and $gene2$). At the end, also all affected memory objects are remapped in order to obtain a valid solution or individual, respectively.

The *mutation* and *crossover* functions, typically, generate new individuals, which by design cover well the entire solution space. These new generated individuals are passed to the fitness evaluation again. The design space exploration, including optimization and performance / energy estimation in a loop, is running until solutions with a good fitness level have been found or until a maximum number of generations is reached. The maximum number of generations can be specified by the designer and should be dependent on the complexity of the application and the underlying considered architecture. The designer can choose the best trade-off for the design requirements within the generated solutions. An example for a set of generated mapping solutions is shown in Figure 6.9 for the *Edge Detect 8* benchmark. These solutions were extracted by the *Memory-Aware Multiobjective Mapping Optimization Tool* after 100 generations.

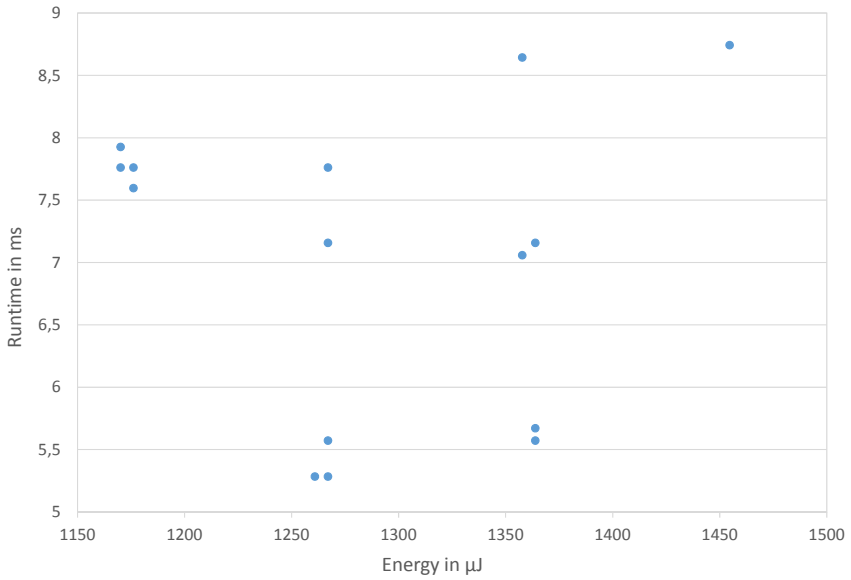

Figure 6.9: Generated solutions for the *Edge Detect 8* benchmark

The evolutionary algorithm was implemented in PISA [84] and as underlying multiobjective search algorithm, Strength Pareto Evolutionary Algorithm (SPEA2) [115] is used.

In particular, EXPO is the central module of the evolutionary algorithm framework, where individuals and their mutation/crossover process can be defined. As underlying multiobjective search algorithm, Strength Pareto Evolutionary Algorithm (SPEA2) [115] is used which communicates with EXPO via the PISA interface.

6.5 Evaluation

The experimental setup and evaluation environment is the same as for the ILP-based memory-aware approach in Section 5.4.1. It is based on the same heterogeneous architecture illustrated in Figure 5.2, which is implemented in the MACCv2 Framework and also implemented for the cycle-accurate instruction set simulator CoMET [16]. The parallelized benchmarks are described in Section 4.8.

6.5.1 Experimental Setup

For this evaluation, the parallelization tools and the *Thread Model Extraction Tool* were applied before the execution of the *Memory-Aware Multiobjective Mapping Optimization Tool*. They provide the parallelized and synchronized source code as well as the input for the *Memory-Aware Multiobjective Mapping Optimization Tool*, i.e. application and architecture specification.

In order to show the improvements of the EA-based memory-aware mapping tool,

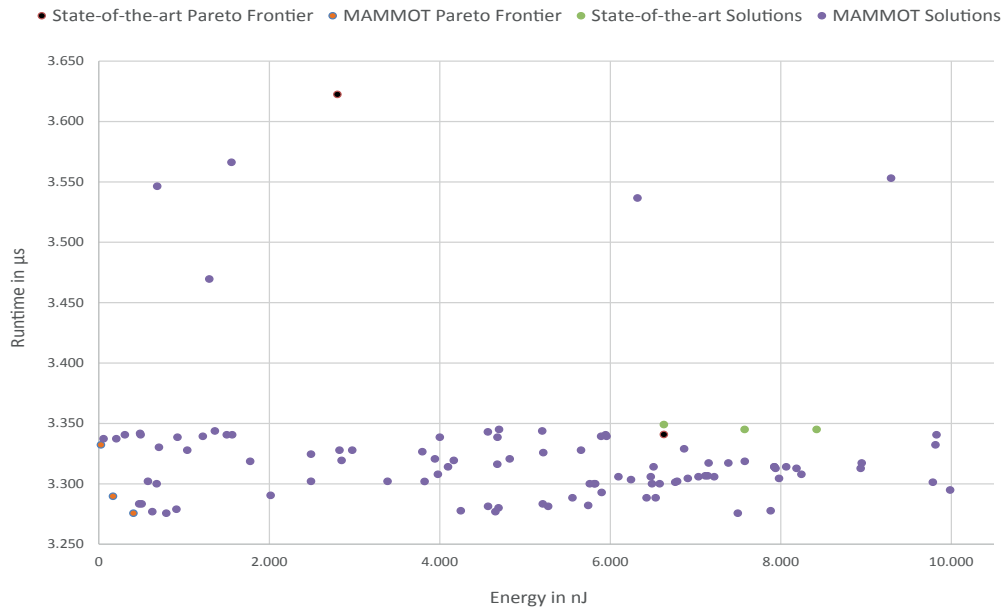

Figure 6.10: Generated solutions for the *Spectral* benchmark

we need an adequate comparison to other existing tools. However, a comparison to tools mentioned in Section 2.4.3 is infeasible due to the different underlying application models (e.g. process networks, data flow graphs), architecture specifications (i.e., no memory hierarchy) and different benchmarks with unequal parallelization of the benchmarks. To create a comparable reference, the EA-based memory-aware mapping tool is compared to a DOL mapping optimization, which performs mapping of threads to processors without considering memories. Here, DOL was adapted to have the same construction as the EA-based memory-aware mapping tool, i.e. the same underlying evolutionary algorithm performing a multiobjective-aware optimization for performance and energy and the same thread-based application model. Like other mapping tools, DOL does not consider nor exploit the underlying memory hierarchy in its optimization models and objective functions. This is usually the current practice in all mapping optimization tools from this class. In DOL, all memory objects are mapped to the private memory of a processor (except for shared memory objects). The output of the memory-aware and state-of-the-art optimization tools is a set of mapping solutions. However, evolutionary algorithms generate solutions randomly. Thus, the generated solutions are not deterministic. Executing the evolutionary algorithm multiple times with the same parameters (number of genes, generations, etc.), could produce different solutions. In order to compare the two mapping optimizations, each evolutionary algorithm was executed with 100 generations starting with a population of 100 genes. Next, the two extreme points from each Pareto frontier were selected and simulated with the cycle-accurate CoMET simulator. These two (extreme) points represent the best solutions for each optimization goal, i.e. one point contains the minimum energy consumption and the

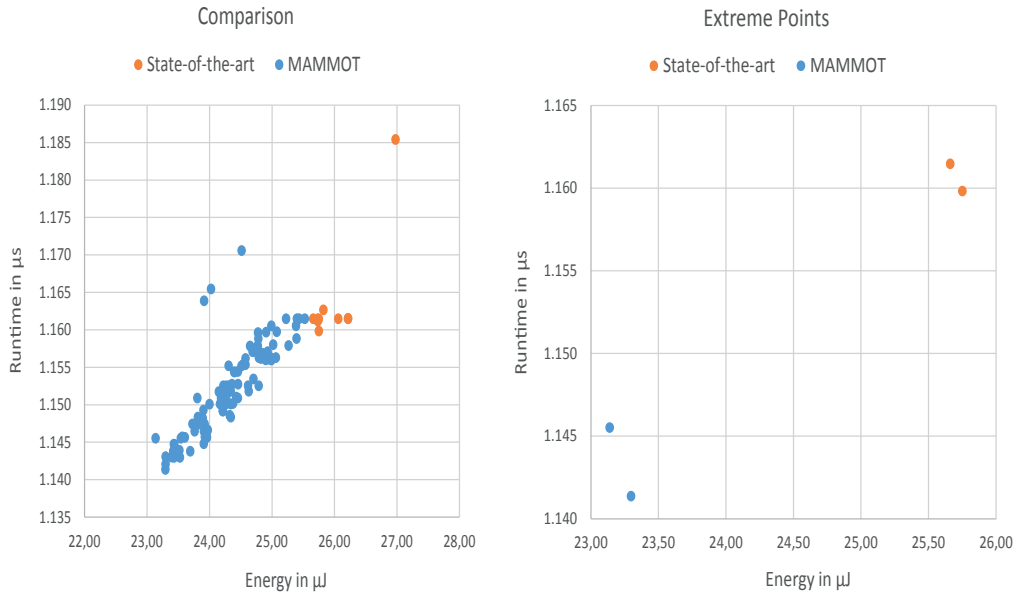

Figure 6.11: Generated solutions for the *H264-LBlock* benchmark

other point the minimum runtime. Since these optimization goals stand in conflict to each other, these two points differ. More clearly, a mapping solution is represented by a 2-tuple (x_i,y_i) with $i = 1, ..., n$ and n solutions. The value x represents the energy consumption and the value y represents the runtime. The best solution concerning the systems energy consumption is represented by (x_{min},y_i). Furthermore, the solution representing the best solution concerning the systems runtime is given by (x_i,y_{min}). These two solutions are part of the Pareto frontier. If additional Pareto optimal solutions exist (i.e. additional points within the Pareto frontier) they usually range between these two extreme points. The selection of these two extreme points and the corresponding simulation step were performed 25 times in order to obtain convincing average values.

Figures 6.10 and 6.11 show the extracted solutions from the *Memory-Aware Multiobjective Mapping Optimization Tool* for the benchmarks *Spectral* and *H264-LBlock*, respectively. Both figures illustrate all obtained solutions as well as the extreme points for the state-of-the-art mapping and memory-aware mapping optimization. These figures show that the *Memory-Aware Multiobjective Mapping Optimization Tool* (MAMMOT) extracts more efficient points than the state-of-the-art mapping.

6.5.2 Experimental results

This section compares the EA-based memory-aware mapping optimization with a state-of-the-art mapping optimization, which does not consider the memory subsystem. The following Figures represent the reduction that was obtained by the

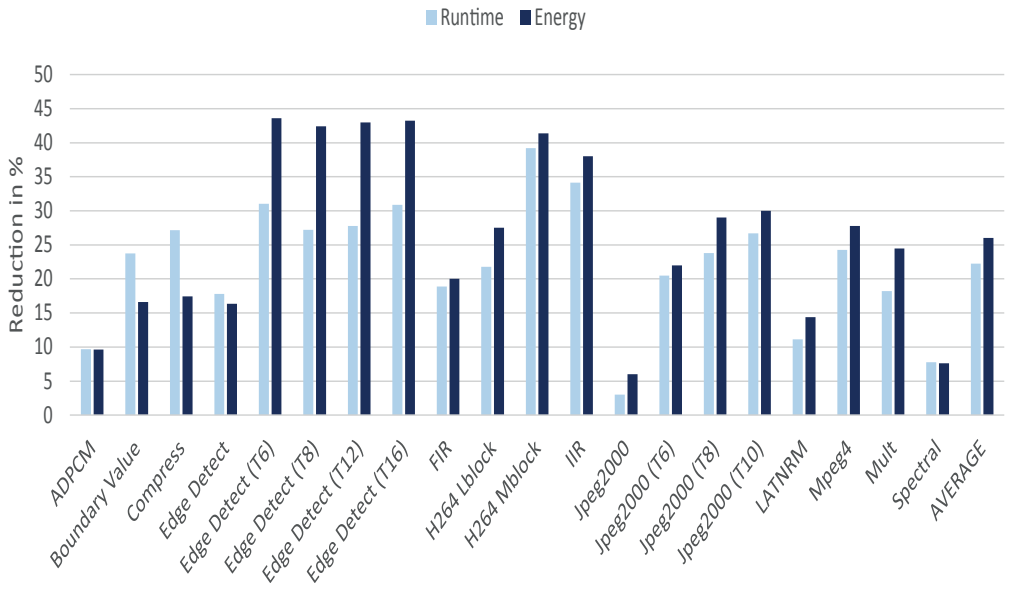

Figure 6.12: Optimization reduction achieved by the *Memory-Aware Multiobjective Mapping Optimization Tool* for a homogeneous architecture

Memory-Aware Multiobjective Mapping Optimization Tool compared to state-of-the-art mapping. The reduction is obtained by the extraction of the difference between the two extreme points of state-of-the-art mapping and the memory-aware mapping.

Figure 6.12 shows this comparison for a homogeneous platform. The highest reductions for energy consumption were obtained for the multiple parallelization of the *Edge Detect* benchmarks, i.e. *Edge Detect (T6) - T(16)*. Here, a reduction of energy was obtained by up to 43%. The benchmarks *IIR* and *H264-MBlock* showed also remarkable reduction in energy consumption by 38 to 41%. The best runtime reduction was achieved for the *H264-MBlock* with 39% and *IIR* with 34%. Overall, the system runtime was reduced by about 22% and the energy consumption was reduced by about 27%.

Figure 6.13 presents the results for a heterogeneous architecture. As in the homogeneous architecture, the largest benefit is obtained for the different parallelizations of the *Edge Detect* benchmarks as well as for *IIR* and *H264-MBlock*. The runtime was optimized by up to 39% and the energy consumption by to 44%. As mentioned in the last section, this large benefit is due to the fact that almost all memory objects of some benchmarks could be mapped to the faster and more energy-efficient local scratchpad memories. *JPEG2000* and *Spectral* gained benefits of 3% to 4.5%. These benchmarks work primarily with shared data memory objects, which are not allowed to be mapped to a higher memory level. However, the instruction code of the threads could be mapped to scratchpad memories and gains some benefit. On average, the *Memory-Aware Multiobjective Mapping Optimization Tool* optimized the runtime by about 21% and the energy consumption by about 26%.

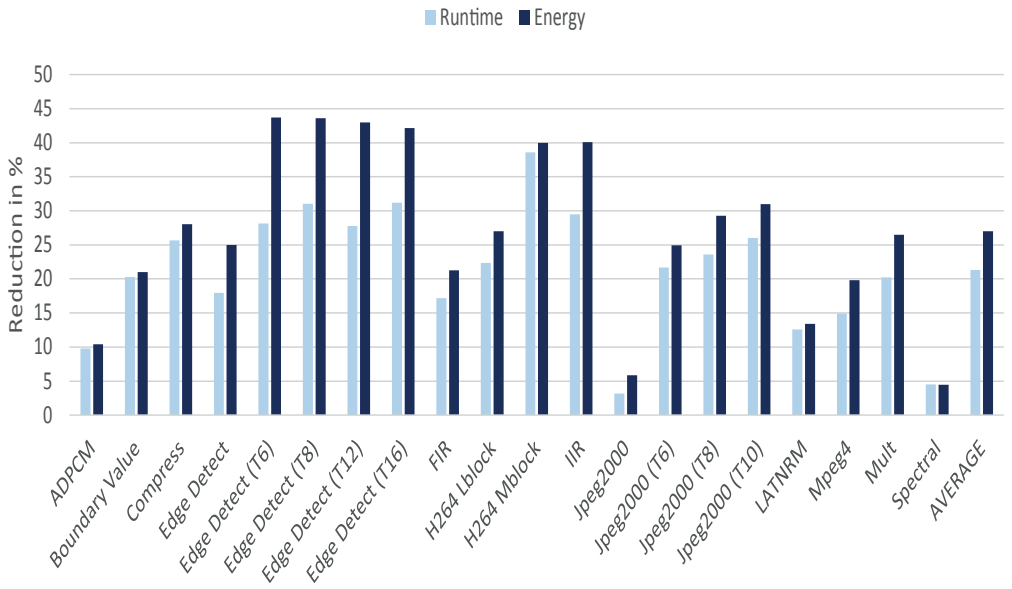

Figure 6.13: Optimization reduction achieved by the *Memory-Aware Multiobjective Mapping Optimization Tool* for a heterogeneous architecture

6.5.2.1 Comparison against cache-based systems

This section presents a comparison between cache based and a scratchpad based memory subsystem for the heterogeneous architecture that is described in Figure 5.2. Thus, the level-1 memories are either caches or scratchpad memories. Contrary to caches, the allocation of scratchpad memories has to be performed explicitly by the designer. This step is already included in the *Memory-Aware Multiobjective Mapping Optimization Tool*. The content of caches is loaded automatically. For this reason, caches occupy more die and consume more energy compared to a scratchpad. Please note, that the content of the caches can be exchanged dynamically during runtime, while the content of the scratchpads is static in this comparison. The loading of new content to memory usually results in an increase in runtime and energy consumption.

Figure 6.14 illustrates the energy consumption and runtime for cache and scratchpad based systems. The base line is a state-of-the-art thread to processor mapping without consideration of the memory subsystem. The memory-aware scratchpad allocation is presented by the *Memory-Aware Multiobjective Mapping Optimization Tool*, i.e. the evolutionary based optimization presented in this section. For most benchmarks, the *Memory-Aware Multiobjective Mapping Optimization Tool* outperforms the cache-based systems. As mentioned, this can result from the higher energy consumption of caches and from dynamic content exchange of memory objects during runtime. For *Edge Detect* and partly for *MPEG4*, the cache based system gains better results than the *Memory-Aware Multiobjective Mapping Optimization Tool*.

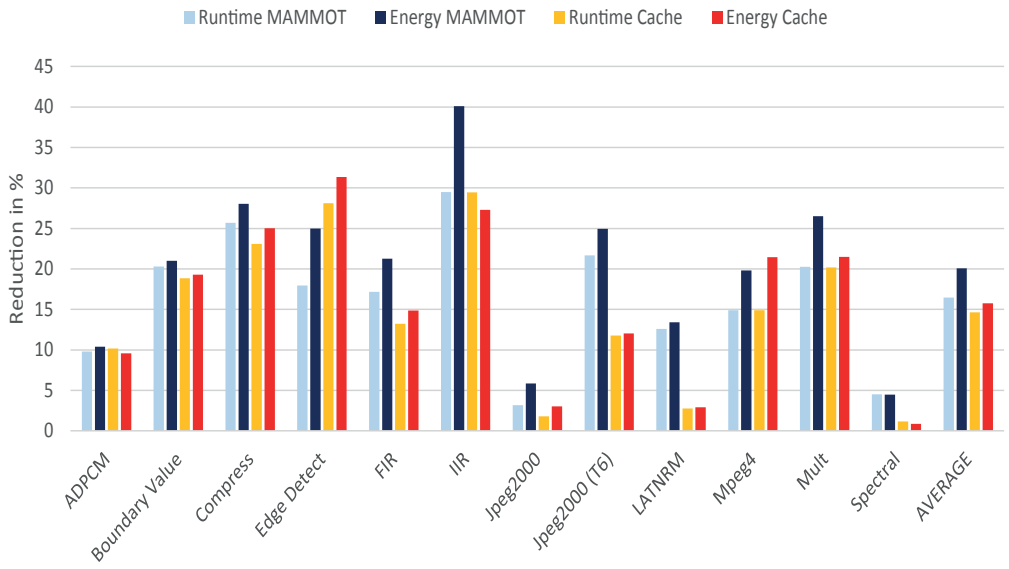

Figure 6.14: Comparison between Cache and the *Memory-Aware Multiobjective Mapping Optimization Tool* optimization for a heterogeneous architecture

Here, the benchmark can have several hot spots. In this case, it could be more efficient to dynamically change the memory content in order to consume less energy or to speed-up the runtime, reespectively.

6.5.2.2 Runtime of the memory-aware optimization

This section presents the time that was required for the execution of the memory-aware mapping optimization. Figure 6.15 illustrates the required time for the memory-aware mapping algorithm for homogeneous systems. The average runtime for compact thread graphs takes about 508 seconds up to 544 seconds. The average runtime for detailed thread graphs is increased due to more complex thread graphs for *IIR* and *MPEG4*. On the x-axis, the runtime is illustrated in seconds for an AMD Opteron 2.46 GHz. On the y-axis, all benchmarks and the average over all benchmarks is presented. *WC Detail* and *AC Detail* represent the runtime for a detailed thread graph containing the worst-case (WC) or average-case (execution time) of the underlying application benchmark. Thus, *WC Compact* and *AC Compact* represent a compact thread graph which contains the worst and average-case, respectively. The runtime represents the execution for the evolutionary algorithm for 100 generations.

The required runtime for the memory-aware optimization for heterogeneous systems is presented in Figure 6.16 The more complex benchmarks are also time-consuming for these platforms. Comparing the runtime for homogeneous and heterogeneous platforms, the runtime does not significantly differ for compact thread

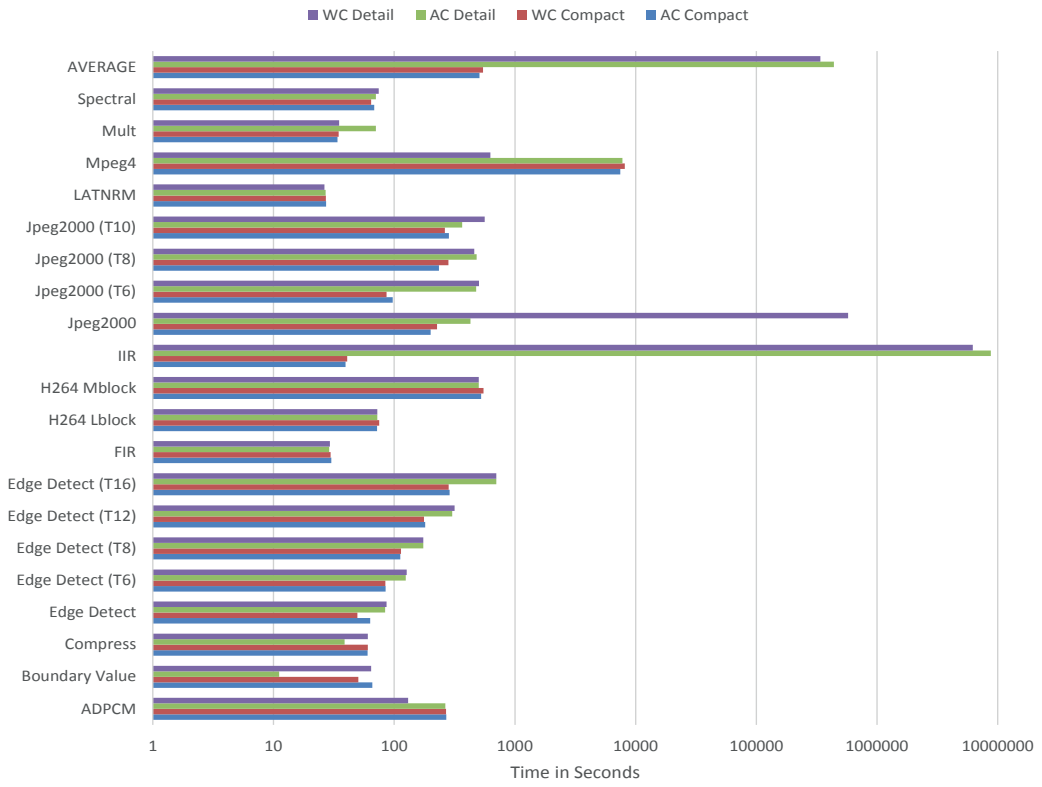

Figure 6.15: Runtime required for the memory-aware EA optimization - homogeneous architecture

graphs. For detailed thread graphs, it significantly differs for *IIR*, *Mult* and *MPEG4* benchmarks. This is due to the fact that these benchmarks are more complex and that their detailed thread graphs are also more complex since they contain more thread nodes and parallel sections than their compact counterpart. This shows that the memory-aware mapping optimization of these complex benchmarks is a time-consuming task due to the huge solution space.

6.5.3 Conclusions

This section introduced a memory-aware multiobjective mapping optimization, which considers MPSoC platforms with homogeneous and heterogeneous memory hierarchies. Since each application has different performance requirements, the optimization exploits the underlying memory hierarchy in order to efficiently match applications requirements with architecture capabilities. To evaluate the memory-aware mapping optimization, this approach is compared to a state-of-the-art mapping tool that does not consider the memory hierarchy. The effectiveness of the memory-aware mapping approach is presented for various benchmarks and we show that

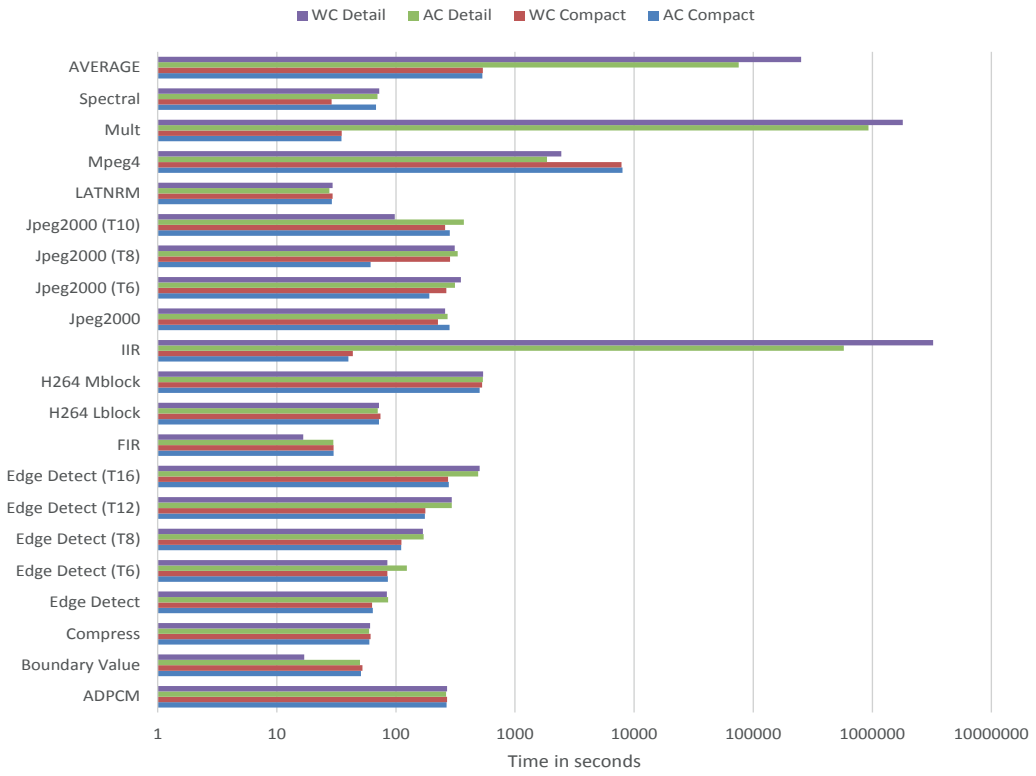

Figure 6.16: Runtime required for the memory-aware EA optimization - heterogeneous architecture

this tool outperforms state-of-the-art mapping. The evaluation was performed for a homogeneous and a heterogeneous MPSoC platform. On a homogeneous platform, the system runtime was reduced by about 22% and the energy consumption was reduced by about 27%. On a heterogeneous platform, the system performance was optimized by up to 21% and energy consumption by up to 26%. More reduction was gained for benchmarks which contain many memory objects that can be mapped to faster local memories.

The runtime of the memory-aware mapping tool lies by about 500 seconds. It increases for complex benchmarks. Furthermore, the designer can choose to increase the number of generations for individuals, i.e. searching for mapping solutions in the design space. This should also result in a higher runtime for this optimization.

Furthermore, a comparison of a cache based heterogeneous system to our memory-aware scratchpad based systems was performed for some benchmarks. The results show that the scratchpad-based systems usually outperform the cache-based systems. This could result from the higher energy consumption and runtime due to the additional die area of caches and dynamic loading of memory objects. Automatically performing all design steps, the memory-aware mapping optimization tool

flow can be used by designers to guide design decisions and to optimize energy and performance early in the design of MPSoC systems.

Chapter 7

Summary and Future Work

Contents

7.1 Summary and Conclusion

This thesis presented a *Thread Model Extraction Tool* and two approaches for memory-aware mapping optimization for homogeneous and heterogeneous embedded multi-core systems. One approach handles single objective optimizations and the second handles a multiobjective optimization. Both approaches reduce the systems runtime and energy consumption remarkably. All approaches work in an automated way and can be used within the MACCv2/MNEMEE tool flow. This alleviates the work of the designer and reduces design time significantly.

Chapter 2 gives an overview over the different embedded application and architecture models in embedded design. We also describe the mapping problem and its complexity. This includes the architecture and application model that is used in this thesis. All related work is described at the end of Chapter 2. This includes the mapping of memory objects to single core systems and multiprocessors system. The mapping of threads to processors showed that many different approaches exist. Some of them were already implemented in well-known design frameworks. All frameworks use different application and architecture models. A combined mapping of threads to processors and memory objects to memories in multicore systems is a new research field, which started to gain more attention in the last years.

The MNEMEE project is presented in Chapter 3. In this project, the MNEMEE partners developed a framework for embedded system design with focus on memory optimization. All tools of this framework are described in this Chapter. The framework starts with sequential C-code as input and performs parallelization, synchronization and optimizations for mapping of threads to processors and memory objects to memories. It is used as underlying infrastructure for implementing the memory-aware mapping optimizations.

The *Thread Model Extraction Tool* extracts a flat thread graph from parallelized and synchronized source code. Extracting thread graphs from source code was already handled in other publications, but these approaches considered other underlying systems and models. Thus, a problem description introduces the challenge of thread graph extraction for our underlying thread and communication model. A tool overview describes the single steps of the thread graph extraction approach. All pre-processing steps of this tool were implemented by other MNEMEE partners. Chapter 4.6. explains the actual thread graph extraction that is a part of this thesis. All steps and information handling of the model extraction as well as the constraints of this approach are discussed here. In addition, the extraction of architecture information is described. This is provided to the subsequent memory-aware mapping approach. Evaluation shows that the thread graphs (or models) were properly extracted from the source code. It provides the extracted number of threads, thread nodes, FIFO communication and parallel sections. Additionally, the designer is able to choose between detailed and compact thread graph model. In a detailed thread graph, all thread nodes and FIFO communications are unrolled when executed in a loop. By this, the exact timing is represented in the thread graph. In a compact model, the thread nodes and FIFO communications are not unrolled. Here, the designer can choose if the execution costs should represent the average case costs or (estimated) worst-case costs. The compact model suits better for the evaluation of complex thread graphs and can thus reduce the design time for this particular step. Furthermore, all benchmarks, which are also used in the next Chapters, are described here.

The memory aware mapping optimization for single objectives is discussed in Chapter 5. An ILP optimization for runtime and energy is presented which combines the mapping of threads to processors and memory objects to memories. The underlying model for runtime and energy estimation was chosen to be as precisely as possible. The runtime consists of the execution time for all threads on the processors and the access time for memory accesses. The access time for memories is also modeled in detail including bus access time and differentiation between read or write accesses and access width (byte, half-word, word). Additionally, the runtime for FIFO communication is included. The optimization is also aware of the memory type (i.e. instruction or data only memories) and maps only appropriate memory objects to these memories. For the optimization of energy consumption, the runtime model is included in order to obtain the proper cycles for idle and active cycles of the processors. The energy consumption is also considered as detailed as in the runtime optimization, i.e. energy consumption for bus accesses and read or write accesses for different access width. In addition, the energy consumed for FIFO communications is considered, as well as the idle energy of processors during memory access. The experimental setup uses a heterogeneous MPSOc architecture, which is related to an ARM big.little architecture and is thus very representative for state-of-the-art architectures. The evaluation is validated by the cycle-accurate COMET simulator. For the minimization of runtime, the experimental results show a reduction of 1% to 38.5% in runtime compared to an optimization which maps

only thread to processors without considering memory mapping. The average reduction over all benchmarks is given by 21%. Next to the reduction in runtime, it was observed that also the energy was reduced on average by 27% compared to a non-memory-aware approach. The experimental results for the optimization of energy consumption showed a reduction by 2.6% to 60% and an average reduction of 28%. Compared to the non-memory-aware mapping optimization an additional average reduction of 28% (homogeneous system) to 31% (heterogeneous system) was observed in runtime. It is shown that the reduction is dependent on the benchmark and the possibility of the utilization of its memory objects, i.e. how many memory objects are available to be moved on a faster memory. Another factor is the number of accesses to such memory objects and to shared memory objects. Accesses to shared memory consume a lot of time and energy and cannot be optimized in this case. However, a trade-off between energy and runtime is observed when optimizing for one of both optimization goals. The minimization of energy consumption increases the overall runtime by 31% compared to the runtime based memory-aware ILP optimization. Contrary, the energy is increased by 35% by the runtime based ILP optimization compared to the energy based ILP optimization.

A multiobjective memory-aware mapping optimization is discussed in Chapter 6. An evolutionary algorithm reduces runtime and energy consumption, which are conflictive objectives. The DOL framework is used as a basis framework for this optimization. An application model that is based on threads and their memory objects as well as an application model, which handles memories in more detail, were integrated in this framework for this thesis. Additionally, the objectives for the reduction for runtime and energy were implemented in DOL. The tool produces genes (i.e. mapping solutions) which are evaluated and generated in a design space exploration loop. This step evaluates the performance and energy consumption with an analytical model. The evolutionary algorithm works as a black box optimization, which generates new mapping solutions by crossover and mutation. As in the ILP-optimization, the runtime and energy model in the evolutionary algorithm is specified as detailed as possible including the differentiation of memory accesses (read/write), bus and FIFO communication accesses, etc. Since the mapping problem is multiobjective, a set of mapping solutions is provided to the designer. Furthermore, the designer can specify certain mappings (for threads to processors and/or memory objects to memories). The solution is validated for a homogeneous and heterogeneous platform in the cycle-accurate CoMET simulator. Our approach was compared to a state-of-the-art mapping tool that maps only threads to processors without considering memory mapping. The experimental results showed a reduction in runtime from about 3% to about 40%, with an average reduction of 21 to 22%. The energy consumption was reduced from about 4.5% up to about 44% with an average of 26 to 27%. Here, the reduction is also dependent on the benchmark and the possibility to allocate its memory objects to faster and more energy-efficient memories.

All presented optimization approaches can be used for homogeneous or heterogeneous platforms. Furthermore, a various set of multiple benchmarks, which rep-

resent a good average over complexity and utilization, are provided for input. Both, homogeneous and heterogeneous parallelization (i.e. number of threads equal or unequal to number of available processors in the system) is provided for evaluation. These different benchmarks showed that the utilization can be highly dependent on the individual benchmark. The most reduction in energy consumption and runtime was observed for benchmarks, which contain many allocatable instructions and data that can be mapped to the faster and more energy-efficient local memory. On the other side, some benchmarks spent the most time accessing shared memory objects. Therefore, little speed-up or energy reduction can be reached in these benchmarks.

Overall, the designer can choose an optimization according to the systems requirements. Furthermore, the designer can perform required adaptions (predefined mappings, detailed or compact thread graphs, etc.). The huge advantage is that the framework is working in a fully automated way, providing all optimizations and cycle-accurate validation to the designer. These optimizations can be used by designers to guide design decisions and to optimize energy and performance early in the design of MPSoC systems. Within the MNEMEE project, the industrial partners were able to achieve a reduction in design time by up to 76% by using the overall MNEMEE tool flow. This illustrates the effectiveness of automated optimization tools within embedded system design.

7.2 Future Work

7.2.1 Memory-Aware Mapping

Several improvements or extensions are possible for the memory-aware mapping tool.

For the memory-aware ILP optimization, a heuristic is required when more complex benchmarks are considered or when multiple applications have to be run on a homogeneous or heterogeneous architecture.

The ILP-based memory-aware mapping optimization for the reduction of energy consumption could be improved by switching off processors in order to save energy. A more detailed analysis should determine at which point of time it could be efficient to switch off the processors. The analysis should also include the time and energy consumption for the wake-up of the processors. Adding this requirement to the optimization would increase the complexity. Nevertheless, it should be feasible to integrate this constraint into a static memory-aware mapping optimization.

Another step for an extension of memory-aware mapping optimization would be the integration of stack or heap data or large arrays into the memory mapping of the memory-aware mapping optimization These kind of optimizations were already evaluated for scratchpad allocation in single-processor systems (as discussed in Chapter 2.4.2). The integration of stack, heap and large arrays require a more complex analysis of the applications memory requirements. In addition, it results in a more complex optimization since e.g. for large arrays a proper splitting point could be necessary. A splitting point (e.g. pointer) could also be required for stack

or heap allocation.

Another challenge is the consideration of dynamic overlays for the allocation of level-1 memories in the memory-aware mapping optimization. Target applications should be complex benchmarks, which have a large portion of memory objects that could be allocated to level-1 and/or level-2 memories. These applications usually have several memory hotspots during their execution, which could be dynamically allocated to a level-1 memory. For this optimization, a lifetime analysis of variables is required, as well as the consideration of runtime overhead for the copy function which exchanges the memory content during runtime.

The next challenge would be an integration of a dynamic thread and memory mapping, i.e. the thread is mapped to another processor during runtime including the mapping of all its non-shared memory objects to other memories. For this, a lot of pre-processing could be required, which can be performed offline in order to manage fast decisions during runtime. This could be an interesting scenario for the concurrent execution of multiple benchmarks. State-of-the-art task to processor mapping optimization already evaluate this kind of optimization. The key would be the integration of memory mapping into this optimization.

The parallelization of sequential code could be combined with the mapping process. Dependent on the number of processors and the type of processors, the parallelization tool decides about the generations of threads (i.e. number and workload). This is already performed for homogeneous architecture, as the parallelization optimization in the MNEMEE toolflow [94]. If the parallelization analysis is extended to include the memory hierarchy in the optimization, it could eventually generate threads, which fit to the processor and memory resources in the architecture. A starting point could be to parallelize for a given homogeneous platform with homogeneous processors and heterogeneous memories (per processor). This optimization could be extended to a heterogeneous architecture with different processors and memory sizes on different levels. This optimization could be provided for complex benchmarks or for systems, which execute multiple benchmarks. For this, the parallelization optimization should be able to extract heterogeneous threads. However, extensive analysis would be required in order to decide if this kind of optimization is feasible and efficient.

Another problem, which showed up in all optimizations, are benchmarks that spent the most time accessing shared memory objects. Only little speed-up or energy reduction can be reached for these benchmarks. An idea could be to combine parallelization with a memory optimization, which splits up data that is not accessed at the same time. This combination could be used with a memory-aware mapping.

7.2.2 Thread Graph Extraction

This tool is tailored towards the MNEMEE tool flow and can be used only inside this tool flow. A tool of this class always has to be tailored towards some API for parallelization and synchronization (e.g. Open-MP, MPI, MPMH). It could be improved by handling more parallelization and synchronization APIs. It has to be

extended to handle more profiling and analysis. By this, the thread graph extraction would have to be improved to handle a more common kind of thread graph. By this, it could be used as a common tool for other optimizations.

7.2.3 Design Frameworks

Finally, the author would like to discuss the usage of design frameworks in research. Nowadays in research, many basis tools are required in order to have the possibility to concentrate on research challenges and new optimizations. A lot of work is required for pre-processing or post-processing steps (e.g. setting up a good simulation environment for a meaningful evaluation of an optimization). Thus, the portion or percentage of work that has been investigated for research has been shifted. Depending on the optimization, more and more work is required for the optimization environment. Furthermore, a researcher has to face and choose between many standards, as for example communication protocols (openMP, MPI, etc.), application models (process networks, thread model, etc.) or architecture models (homogeneous or heterogeneous systems, network on chips, massive parallel, etc.). It is not always easy to choose between the models. In addition, the evaluation of several models for an optimization seems an unreachable request.

Recent research started to focus on building up design frameworks for embedded system design as described in Chapter 2.4.5 and as developed in the MNEMEE project. These frameworks are urgently required for the development of new and complex optimizations in research. In many cases, they build up the basis for them. A researcher who would like to develop an optimization, which requires many pre-processing steps, could end up in the development of months to years in order to obtain the desired pre-processing. Usually, this pre-work will not be honored in his/her thesis or is not utilizable for research respectively. As an example: The author of this thesis would had to implement parallelization, synchronization and post-processing steps (setup of cycle-accurate simulation environment including OS) in order to have the possibility to develop a memory-aware mapping optimization. In the MNEMEE project, several persons were required for this setup. They worked for about 2 years on this framework.

Therefore, design frameworks are very welcome. Nevertheless, it is still a difficult situation for new researchers. Since several application and architecture models exist, it is not always possible to find a proper underlying framework. Research started to go into the right direction by developing these kind of frameworks, but there is still a lot of work. One problem is that optimizations cannot be compared to each other due to different underlying application, architecture models and different benchmarks.

Ideally, a design framework should offer the possibility to

- provide several application and communication models
- dependent on the underlying models, parallelization and synchronization is provided

- provide interfaces and tool templates for the integration of new optimization into the tool flow
- provide profiling and analysis for a broad mass of architecture and application information extraction
- provide the possibility to choose between fine-grained and coarse-grained design space exploration (e.g. cycle-accurate or high-level) or analysis model (e.g. worst case model or average case model)
- provide the possibility to choose an architecture or build up an own architecture where all elements as processors memories, buses etc. can be build up easily
- provide the possibility to choose hardware partitioning
- benchmarks and a simulation environment for systems, which execute more than one benchmark at a time

Of course, it should be discussed if it is even feasible to develop such an ideal framework. However, some skeleton has to be provided where the integration of optimization is easier to manage in order to give researcher a better chance for developing optimizations that are more complex. A good starting point for this is the MACCv2 [14] framework. Nowadays, different design frameworks with different application model exist. There should be a possibility to compare the results of different frameworks by setting up some basis (e.g. the same benchmarks). This kind of discussion already started in some research community (MAP2MPSOC workshop 2011/2012 - internal discussion [116]).

Bibliography

[1] H. E. Schaefer, *Nanoscience: The Science of the Small in Physics, Engineering, Chemistry, Biology and Medicine (Google eBook)*. Springer, 2010. (Cited on page 1.)

[2] Intel, "The Intel Xeon-Phi Coprocessor," March 2016. (Cited on page 1.)

[3] G. Moore, "Cramming More Components Onto Integrated Circuits," *Proceedings of the IEEE*, vol. 86, no. 1, pp. 82–85, 1998. (Cited on page 1.)

[4] International Technology Roadmap for Semiconductors - ITRS, July 2013. (Cited on page 1.)

[5] International Technology Roadmap for Semiconductors - ITRS, "More Moore Roadmap - ITRS 2.0 White Paper," March 2016. (Cited on page 1.)

[6] P. Marwedel, *Embedded Systems Design - Embedded Systems Foundations of Cyber-Physical Systems*. Springer, 2011. (Cited on pages 2, 11, 15, 16, 17, 20, 21, 22, 23, 41 and 147.)

[7] E. A. Lee, "Cyber Physical Systems: Design Challenges," in *International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC)*, May 2008, invited Paper. [Online]. Available: http://chess.eecs.berkeley.edu/pubs/427.html (Cited on page 2.)

[8] K. Hoi-Jun Yoo and J. K. K. Lee, *Low-Power NoC for High-Performance SoC Design*. CRC PR INC, 2008. (Cited on page 3.)

[9] P. Machanick, "Approaches to Addressing the Memory Wall," School of IT and Electrical Engineering, Tech. Rep., 2002. (Cited on page 4.)

[10] R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, and P. Marwedel, "Scratchpad Memory: Design Alternative for Cache On-Chip Memory in Embedded Systems," in *Proceedings of the tenth international symposium on Hardware/software codesign*, 2002. (Cited on page 5.)

[11] Texas Instruments, "Omap 5912," March 2016. (Cited on pages 5, 6, 13 and 147.)

[12] A. Mallik, S. Mamagkakis, C. Baloukas, L. Papadopoulos, D. Soudris, S. Stuijk, O. Jovanovic, F. Schmoll, D. Cordes, R. Pyka, P. Marwedel, F. Capman, S. Collet, N. Mitas, and D. Kritharidis, "MNEMEE - An Automated Toolflow for Parallelization and Memory Management in MPSoC Platforms," *48th Design Automation Conference (DAC), San Diego, California, USA*, June 2011. (Cited on pages 9, 10, 46 and 54.)

[13] C. Baloukas, L. Papadopoulos, D. Soudris, S. Stuijk, O. Jovanovic, F. Schmoll, D. Cordes, R. Pyka, A. Mallik, S. Mamagkakis, F. Capman, S. Collet, N. Mitas, and D. Kritharidis, "Mapping Embedded Applications on MPSoCs: The MNEMEE Approach," in *Proceedings of the 2010 IEEE Annual Symposium on VLSI*, July 2010, pp. 512–517. (Cited on pages 9, 10, 46 and 54.)

[14] R. Pyka, F. Klein, P. Marwedel, and S. Mamagkakis, "Versatile System-Level Memory-Aware Platform Description Approach for Embedded MPSoCs," in *Proceedings of the ACM SIGPLAN/SIGBED 2010 Conference on Languages, Compilers, and Tools for Embedded Systems*, 2010. (Cited on pages 9, 47, 48, 70, 133 and 147.)

[15] O. Jovanovic, N. Kneuper, P. Marwedel, and M. Engel, "ILP-based Memory-Aware Mapping Optimization for MPSoCs," in *The 10th IEEE/IFIP International Conference on Embedded and Ubiquitous Computing*, Paphos, Cyprus, December 2012. (Cited on page 10.)

[16] CoMET, "Synopsys, Inc." May 2016. (Cited on pages 10, 94 and 118.)

[17] O. Jovanovic, P. Marwedel, I. Bacivarov, and L. Thiele, "MAMOT: Memory-Aware Mapping Optimization Tool for MPSoC," in *15th Euromicro Conference on Digital System Design (DSD 2012)*, September 2012. (Cited on page 10.)

[18] L. Thiele, I. Bacivarov, W. Haid, and K. Huang, "Mapping Applications to Tiled Multiprocessor Embedded Systems," in *Proceedings of the 7th International Conference on Application of Concurrency to System Design*, 2007. (Cited on pages 10, 39, 41 and 106.)

[19] W. Wolf, A. A. Jerraya, and G. Martin, "Multiprocessor System-on-Chip (MPSoC) Technology," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 27, pp. 1701–1713, 2008. (Cited on pages 12 and 13.)

[20] Samsung, "big.LITTLE Technology," March 2016. (Cited on pages 14 and 93.)

[21] Samsung, "big.LITTLE Technology: The Future of Mobile," March 2016. (Cited on pages 14 and 93.)

[22] Samsung, "Exynos 5 Octa," March 2016. (Cited on page 14.)

[23] Samsung, "Exynos 5420," March 2016. (Cited on page 14.)

[24] Samsung, "Exynos 5410 Development Board," March 2016. (Cited on page 14.)

[25] Samsung, "Exynos 8 Octa," March 2016. (Cited on page 14.)

[26] Qualcomm Technologies, Inc., "Snapdragon 808 Processor," March 2016. (Cited on page 14.)

[27] Qualcomm Technologies, Inc., "Snapdragon 810 Processor," March 2016. (Cited on page 14.)

[28] NVIDIA Corporation, "NVIDIA TEGRA X1," March 2016. (Cited on page 14.)

[29] V-Model XT Authors, December 2012. (Cited on page 16.)

[30] D. D. Gajski and R. Kuhn, "Guest Editor's Introduction: New VLSI Tools," *IEEE Computer*, December 1983. (Cited on page 16.)

[31] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*. San Mateo, CA: Morgan Kaufmann, 1990. (Cited on pages 17 and 19.)

[32] G. Kahn, "The Semantics of a Simple Language for Parallel Programming," in *Information processing*, Stockholm, Sweden, August 1974, pp. 471–475. (Cited on page 21.)

[33] E. A. Lee and T. M. Parks, "Dataflow Process Networks," in *Proceedings of the IEEE*. Kluwer Academic Publishers, 1995, vol. 83, pp. 773–799. (Cited on page 21.)

[34] M. Geilen and T. Basten, "Requirements on the Execution of Kahn Process Networks," in *Proceedings of the 12th European conference on Programming*, 2003, pp. 319–334. (Cited on page 21.)

[35] E. Lee and D. Messerschmitt, "Synchronous Data Flow," *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235 – 1245, September 1987. (Cited on page 21.)

[36] C. A. Petri, "Kommunikation mit Automaten," Ph.D. dissertation, Darmstadt University of Technology, Germany, 1962. (Cited on page 21.)

[37] OpenMP, "The OpenMP ARB," May 2016. (Cited on page 22.)

[38] H. Yang and S. Ha, "Pipelined Data Parallel Task Mapping/Scheduling Technique for MPSoC," in *Proceedings of the Conference on Design, Automation and Test in Europe*, 2009, pp. 69–74. (Cited on page 26.)

[39] R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, and P. Marwedel, "Scratchpad Memory: Design Alternative for Cache On-Chip Memory in Embedded Systems," in *Proceedings of the tenth international symposium on Hardware/software codesign*, 2002, pp. 73–78. (Cited on page 28.)

[40] S. Steinke, L. Wehmeyer, B. Lee, and P. Marwedel, "Assigning Program and Data Objects to Scratchpad for Energy Reduction," in *Proceedings of the conference on Design, automation and test in Europe*, 2002. (Cited on pages 28, 32 and 94.)

[41] M. Verma, S. Steinke, and P. Marwedel, "Data Partitioning for Maximal Scratchpad Usage," in *ASPDAC 2003*, January 2003. (Cited on pages 28 and 32.)

[42] M. Verma, L. Wehmeyer, and P. Marwedel, "Dynamic Overlay of Scratchpad Memory for Energy Minimization," in *Proceedings of the 2nd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, 2004, pp. 104–109. (Cited on pages 29 and 32.)

[43] M. Verma, L. Wehmeyer, and P. Marwedel, "Efficient scratchpad allocation algorithms for energy constrained embedded systems," in *Proceedings of the Third international conference on Power - Aware Computer Systems*, 2003, pp. 41–56. (Cited on pages 29 and 32.)

[44] M. Verma, K. Petzold, L. Wehmeyer, H. Falk, and P. Marwedel, "Scratchpad Sharing Strategies for Multiprocess Embedded Systems: A First Approach," in *IEEE 3rd Workshop on Embedded System for Real-Time Multimedia (ESTIMedia)*, 2005, pp. 115–120. (Cited on pages 29 and 32.)

[45] S. Steinke, N. Grunwald, L. Wehmeyer, R. Banakar, M. Balakrishnan, and P. Marwedel, "Reducing Energy Consumption by Dynamic Copying of Instructions onto Onchip Memory," in *Proceedings of the 15th international symposium on System Synthesis*, 2002, pp. 213–218. (Cited on pages 29 and 32.)

[46] R. A. Ravindran, P. D. Nagarkar, G. S. Dasika, E. D. Marsman, R. M. Senger, S. A. Mahlke, and R. B. Brown, "Compiler Managed Dynamic Instruction Placement in a Low-Power Code Cache," in *Proceedings of the international symposium on Code generation and optimization*, 2005, pp. 179–190. (Cited on pages 29 and 32.)

[47] O. Avissar, R. Barua, and D. Stewart, "Heterogeneous Memory Management for Embedded Systems," in *Proceedings of the 2001 international conference on Compilers, architecture, and synthesis for embedded systems*, 2001, pp. 34–43. (Cited on pages 29 and 32.)

[48] M. Verma, L. Wehmeyer, and P. Marwedel, "Cache-Aware Scratchpad-Allocation Algorithms for Energy-Constrained Embedded Systems," *IEEE Trans. on CAD of Integrated Circuits and System (TCAD)*, vol. 25, no. 10, pp. 2035–2051, 2006. (Cited on pages 30 and 32.)

[49] S. Udayakumaran and R. Barua, "Compiler-Decided Dynamic Memory Allocation for Scratch-Pad Based Embedded Systems," in *Proceedings of the 2003 international conference on Compilers, architecture and synthesis for embedded systems*, 2003, pp. 276–286. (Cited on pages 30 and 32.)

[50] A. Dominguez, S. Udayakumaran, and R. Barua, "Heap Data Allocation to Scratch-Pad Memory in Embedded Systems," *J. Embedded Comput.*, vol. 1, no. 4, pp. 521–540, 2005. (Cited on pages 30 and 32.)

[51] R. Pyka, C. Fassbach, M. Verma, H. Falk, and P. Marwedel, "Operating System Integrated Energy Aware Scratchpad Allocation Strategies for Multiprocess Applications," in *10th International Workshop on Software & Compilers for Embedded Systems (SCOPES)*, 2007, pp. 41–50. (Cited on pages 30 and 32.)

[52] R. Szymanek, F. Catthoor, and K. Kuchcinski, "Data Assignment and Access Scheduling Exploration for Multi-Layer Memory Architectures," in *IEEE Symposium on Embedded Systems for Real-Time Multimedia*, 2004. (Cited on page 31.)

[53] F. Angiolini, L. Benini, and A. Caprara, "An Efficient Profile-Based Algorithm for Scratchpad Memory Partitioning," *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, vol. 24, no. 11, pp. 1660–1676, 2006. (Cited on page 31.)

[54] A. Marongiu and L. Benini, "Efficient OpenMP Support and Extensions for MPSoCs with Explicitly Managed Memory Hierarchy," in *Proceedings of the Conference on Design, Automation and Test in Europe*, 2009, pp. 809–814. (Cited on page 33.)

[55] A. Marongiu and L. Benini, "An OpenMP Compiler for Efficient Use of Distributed Scratchpad Memory in MPSoCs," *IEEE Trans. Comput.*, vol. 61, no. 2, pp. 222–236, 2012. (Cited on page 33.)

[56] P. Francesco, P. Antonio, and P. Marchal, "Flexible Hardware/Software Support for Message Passing on a Distributed Shared Memory Architecture," in *Proceedings of the conference on Design, Automation and Test in Europe - Volume 2*, 2005, pp. 736–741. (Cited on page 33.)

[57] W. Che, A. Panda, and K. S. Chatha, "Compilation of Stream Programs for Multicore Processors that Incorporate Scratchpad Memories," in *Proceedings of the Conference on Design, Automation and Test in Europe*, 2010, pp. 1118–1123. (Cited on page 33.)

[58] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy, "Introduction to the Cell Multiprocessor," *IBM Journal of Research and Development*, vol. 49, no. 4/5, pp. 589–604, 2005. (Cited on page 33.)

[59] M. Kandemir, I. Kadayif, A. Choudhary, J. Ramanujam, and I. Kolcu, "Compiler-Directed Scratch Pad Memory Optimization for Embedded Multiprocessors," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 12, no. 3, pp. 281–287, 2004. (Cited on page 34.)

[60] L. A. D. Bathen, N. D. Dutt, D. Shin, and S.-S. Lim, "SPMVisor: Dynamic Scratchpad Memory Virtualization for Secure, Low Power, and High Performance Distributed On-Chip Memories," in *Proceedings of the seventh IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, 2011, pp. 79–88. (Cited on page 34.)

[61] L. A. Bathen and N. Dutt, "HaVOC: A Hybrid Memory-Aware Virtualization Layer for On-Chip Distributed ScratchPad and Non-Volatile Memories," in *Proceedings of the 49th Annual Design Automation Conference*, 2012, pp. 447–452. (Cited on page 34.)

[62] G. Sun, X. Dong, Y. Xie, J. Li, and Y. Chen, "A Novel Architecture of the 3D Stacked MRAM L2 Cache for CMPs," in *HPCA*, 2009, pp. 239–249. (Cited on page 34.)

[63] J. Hu, C. J. Xue, Q. Zhuge, W.-C. Tseng, and E. H.-M. Sha, "Towards Energy Efficient Hybrid On-Chip Scratch Pad Memory with Non-Volatile Memory," in *Proceedings of the conference on Design, Automation and Test in Europe*, 2011, pp. 746–751. (Cited on page 34.)

[64] L. Zhang, M. Qiu, W.-C. Tseng, and E. H.-M. Sha, "Variable Partitioning and Scheduling for MPSoC with Virtually Shared Scratch Pad Memory," *J. Signal Process. Syst.*, vol. 58, no. 2, pp. 247–265, 2010. (Cited on page 34.)

[65] I. Issenin, E. Brockmeyer, B. Durinck, and N. Dutt, "Multiprocessor System-on-Chip Data Reuse Analysis for Exploring Customized Memory Hierarchies," in *Proceedings of the 43rd annual Design Automation Conference*, 2006, pp. 49–52. (Cited on page 35.)

[66] V. Suhendra, A. Roychoudhury, and T. Mitra, "Scratchpad Allocation for Concurrent Embedded Software," *ACM Transactions on Programming Languages and Systems*, vol. 32, no. 4, pp. 13:1–13:47, 2010. (Cited on page 35.)

[67] A. Goens, J. Castrillon, M. Odendahl, and R. Leupers, "An Optimal Allocation of Memory Buffers for Complex Multicore Platforms," *Journal of Systems Architecture*, vol. 66-67, pp. 69–83, May 2016. (Cited on page 35.)

[68] D. Wu, B. M. Al-Hashimi, and P. Eles, "Scheduling and Mapping of Conditional Task Graphs for the Synthesis of Low Power Embedded Systems," in *Proceedings of the conference on Design, Automation and Test in Europe*, 2003. (Cited on page 36.)

[69] S. Manolache, P. Eles, and Z. Peng, "Task Mapping and Priority Assignment for Soft Real-Time Applications under Deadline Miss Ratio Constraints," *ACM Transactions on Embedded Computing Systems*, vol. 7, pp. 19:1–19:35, January 2008. (Cited on page 36.)

[70] E. W. Briao, D. Barcelos, and F. R. Wagner, "Dynamic Task Allocation Strategies in MPSoC for Soft Real-Time Applications," in *Proceedings of the conference on Design, Automation and Test in Europe*, 2008. (Cited on page 36.)

[71] M. Schmitz, B. Al-Hashimi, and P. Eles, "Energy-Efficient Mapping and Scheduling for DVS Enabled Distributed Embedded Systems," in *Proceedings*

of the conference on Design, Automation and Test in Europe, 2002. (Cited on page 36.)

[72] A. J. Page, T. M. Keane, and T. J. Naughton, "Multi-Heuristic Dynamic Task Allocation using Genetic Algorithms in a Heterogeneous Distributed Systems," *J. Parallel Distrib. Comput.*, vol. 70, no. 7, pp. 758–766, 2010. (Cited on page 36.)

[73] B. Arafeh, K. Day, and A. Touzene, "A Multilevel Partitioning Approach for Efficient Tasks Allocation in Heterogeneous Distributed Systems," *J. Syst. Archit.*, vol. 54, no. 5, pp. 530–548, 2008. (Cited on page 36.)

[74] F. Zamfirache, M. Frîncu, and D. Zaharie, "Population-Based Metaheuristics for Tasks Scheduling in Heterogeneous Distributed Systems," in *Proceedings of the 7th international conference on Numerical methods and applications*, 2011, pp. 321–328. (Cited on page 36.)

[75] S. Ha, "Model-Based Programming Environment of Embedded Software for MPSoC," in *Proceedings of the 2007 Asia and South Pacific Design Automation Conference*. IEEE, 2007, pp. 330–335. (Cited on pages 36, 41 and 106.)

[76] A. Gerstlauer, C. Haubelt, A. D. Pimentel, T. P. Stefanov, D. D. Gajski, and J. Teich, "Electronic System-level Synthesis Methodologies," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 28, no. 10, pp. 1517–1530, October 2009. (Cited on pages 37 and 38.)

[77] H. Nikolov, M. Thompson, T. Stefanov, A. D. Pimentel, S. Polstra, R. Bose, C. Zissulescu, and E. F. Deprettere, "Daedalus: Toward Composable Multimedia MPSoC Design," in *Proceedings of the 45th Annual Design Automation Conference*, 2008, pp. 574–579. (Cited on pages 37, 41 and 106.)

[78] S. Verdoolaege, H. Nikolov, and T. Stefanov, "PN: A Tool for Improved Derivation of Process Networks," *EURASIP J. Embedded Syst.*, vol. 2007, no. 1, pp. 19–19, January 2007. (Cited on page 37.)

[79] C. Erbas, S. Cerav-erbas, and A. D. Pimentel, "Multiobjective Optimization and Evolutionary Algorithms for the Application Mapping Problem in Multiprocessor System-on-Chip Design," *IEEE Transactions on Evolutionary Computation*, pp. 358–374, 2006. (Cited on page 37.)

[80] M. Thompson, H. Nikolov, T. Stefanov, A. D. Pimentel, C. Erbas, S. Polstra, and E. F. Deprettere, "A Framework for Rapid System-Level Exploration, Synthesis, and Programming of Multimedia MP-SoCs," in *Proceedings of the 5th IEEE/ACM international conference on Hardware/software codesign and system synthesis*, 2007, pp. 9–14. (Cited on pages 38 and 147.)

[81] J. Keinert, M. Streubner, T. Schlichter, J. Falk, J. Gladigau, C. Haubelt, J. Teich, and M. Meredith, "SystemCoDesigner, an automatic ESL synthesis

approach by design space exploration and behavioral synthesis for streaming applications," *ACM Transactions on Design Automation of Electronic Systems*, vol. 14, no. 1, pp. 1–23, January 2009. (Cited on page 38.)

[82] Forte Cynthesizer, "Forte Design Systems," February 2013. (Cited on page 39.)

[83] E. Zitzler and L. Thiele, "Multiobjective Evolutionary lgorithms: A Comparative Case Study and the Strength Pareto Approach," *IEEE Transactions on Evolutionary Computation*, vol. 3, no. 4, November 1999. (Cited on page 40.)

[84] S. Bleuler, M. Laumanns, L. Thiele, and E. Zitzler, "PISA: a Platform and Programming Language Independent Interface for Search Algorithms," in *Proceedings of the 2nd international conference on Evolutionary multi-criterion optimization*, 2003. (Cited on pages 40, 106 and 118.)

[85] L. Thiele, S. Chakraborty, M. Gries, and S. Kuenzli, "Design Space Exploration of Network Processor Architectures," in *In Network Processor Design: Issues and Practices, Volume 1.* Morgan Kaufmann Publishers, 2002, pp. 30–41. (Cited on page 40.)

[86] C. Haubelt, T. Schlichter, J. Keinert, and M. Meredith, "SystemCoDesigner: Automatic Design Space Exploration and Rapid Prototyping from Behavioral Models," in *ACM Transactions on Design Automation of Electronic Systems*, 2008. (Cited on pages 41 and 106.)

[87] M. M. Baskaran, U. Bondhugula, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan, "Automatic Data Movement and Computation Mapping for Multi-Level Parallel Architectures with Explicitly Managed Memories," in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, 2008, pp. 1–10. (Cited on page 41.)

[88] R. Szymanek and K. Kuchcinski, "Task Assignment and Scheduling under Memory Constraints," in *EUROMICRO Conference on Software Engineering and Advanced Applications*, 2000. (Cited on pages 41, 42 and 43.)

[89] R. Szymanek and K. Kuchcinski, "A Constructive Algorithm for Memory-Aware Task Assignment and Scheduling," in *Proceedings of the 9th international symposium on Hardware/software codesign*, 2001. (Cited on pages 42 and 43.)

[90] R. Szymanek and K. Krzysztof, "Partial Task Assignment of Task Graphs under Heterogeneous Resource Constraints," in *Proceedings of the 40th annual Design Automation Conference*, 2003. (Cited on pages 42 and 43.)

[91] V. Suhendra, C. Raghavan, and T. Mitra, "Integrated Scratchpad Memory Optimization and Task Scheduling for MPSoC Architectures," in *Proceedings*

of the 2006 international conference on Compilers, architecture and synthesis for embedded systems, 2006. (Cited on pages 43, 80 and 88.)

[92] Y. Kim, J. Lee, A. Shrivastava, J. Yoon, and Y. Paek, "Memory-Aware Application Mapping on Coarse-grained Reconfigurable Arrays," in *Proceedings of the 5th International Conference on High Performance Embedded Architectures and Compilers*, 2010, pp. 171–185. (Cited on page 43.)

[93] C. Baloukas, J. L. Risco-Martin, D. Atienza, C. Poucet, L. Papadopoulos, S. Mamagkakis, D. Soudris, J. Ignacio Hidalgo, F. Catthoor, and J. Lanchares, "Optimization Methodology of Dynamic Data Structures Based on Genetic Algorithms for Multimedia Embedded Systems," *Journal of Systems and Software*, vol. 82, no. 4, pp. 590–602, April 2009. (Cited on page 48.)

[94] D. Cordes, P. Marwedel, and A. Mallik, "Automatic Parallelization of Embedded Software Using Hierarchical Task Graphs and Integer Linear Programming," in *Proceedings of the 8th IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, 2010, pp. 267–276. (Cited on pages 49, 50, 94, 131 and 147.)

[95] Y. Iosifidis, A. Mallik, S. Mamagkakis, E. De Greef, A. Bartzas, D. Soudris, and F. Catthoor, "A Framework for Automatic Parallelization, Static and Dynamic Memory Optimization in MPSoC platforms," in *Proceedings of the 47th Design Automation Conference*, 2010, pp. 549–554. (Cited on page 49.)

[96] S. Stuijk, M. Geilen, and T. Basten, "A Predictable Multiprocessor Design Flow for Streaming Applications with Dynamic Behaviour," in *Digital System Design: Architectures, Methods and Tools (DSD), 2010 13th Euromicro Conference on*, 2010, pp. 548–555. (Cited on page 52.)

[97] "OMAP-L137 Reference Manual," April 2013. (Cited on page 53.)

[98] "MSC8144 Reference Manual," April 2013. (Cited on page 53.)

[99] M. D. Dikaiakos, A. Rogers, and K. Steiglitz, "FAST: A Functional Algorithm Simulation Testbed," in *In International Workshop on Modelling, Analysis and Simulation of Computer and Telecommunication Systems*, 1994, pp. 142–146. (Cited on pages 56 and 58.)

[100] M. Cosnard and M. Loi, "Automatic Task Graph Generation Techniques," in *Proceedings of the 28th Hawaii International Conference on System Sciences*, 1995, pp. 113–. (Cited on pages 56, 57 and 58.)

[101] V. S. Adve and R. Sakellariou, "Compiler Synthesis of Task Graphs for Parallel Program Performance Prediction," in *Proceedings of the 13th International Workshop on Languages and Compilers for Parallel Computing-Revised Papers*, 2001, pp. 208–226. (Cited on pages 56 and 57.)

[102] K. S. Vallerio and N. K. Jha, "Task Graph Extraction for Embedded System Synthesis," in *Proceedings of the 16th International Conference on VLSI Design*, 2003. (Cited on pages 56, 59 and 60.)

[103] E. Saad, M. El Adawy, H. A. Keshk, and S. Habashy, "Task Graph Generation," in *Radio Science Conference, Proceedings of the Twenty Third National*, vol. 0, 2006, pp. 1–9. (Cited on pages 56, 59 and 60.)

[104] R. Namballa and N. R. A. Ejnioui, "Control and Data Flow Graph Extraction for High-Level Synthesis," in *VLSI, 2004. Proceedings. IEEE Computer society Annual Symposium on*, 2004, pp. 187–192. (Cited on page 60.)

[105] K. Ganeshpure and S. Kundu, "On Run Time Task Graph Extraction of SoC," in *SoC Design Conference (ISOCC), 2010 International*, 2010, pp. 380–383. (Cited on page 60.)

[106] R. Baert, E. Brockmeyer, S. Wuytack, and T. J. Ashby, "Exploring Parallelizations of Applications for MPSoC Platforms using MPA," in *Proceedings of the conference on Design, Automation and Test in Europe*, 2009. (Cited on pages 61 and 94.)

[107] yWorks, "yEd Graph Editor," May 2016. (Cited on pages 68 and 72.)

[108] C. G. Lee, "UTDSP Benchmark Suite," February 2013. (Cited on page 70.)

[109] S. H.-e. Peng, "UTDSP, a VLIW Programmable DSP Processor," Master's thesis, University of Toronto, Canada, 2000. (Cited on page 70.)

[110] R. Kumar, D. M. Tullsen, P. Ranganathan, N. P. Jouppi, and K. I. Farkas, "Single-ISA Heterogeneous Multi-Core Architectures for Multithreaded Workload Performance," in *Proceedings of the 31st annual international symposium on Computer architecture*. IEEE Computer Society, 2004, pp. 64–75. (Cited on page 93.)

[111] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen, "Single-ISA Heterogeneous Multi-Core Architectures: The Potential for Processor Power Reduction," in *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2003, p. 81. (Cited on page 93.)

[112] CPLEX Optimizer, "IBM Corporation," May 2016. (Cited on page 94.)

[113] J. H. A. S. Thoziyoor, N. Muralimanohar and N. P. Jouppi, "Technical Report HPL-2008-20, CACTI 5.1, HP Laboratories, 2008," May 2016. (Cited on page 94.)

[114] W. Haid, M. Keller, K. Huang, I. Bacivarov, and L. Thiele, "Generation and Calibration of Compositional Performance Analysis Models for Multi-Processor Systems," in *Proceedings of the 9th international conference on Systems, architectures, modeling and simulation*, 2009. (Cited on page 106.)

[115] E. Zitzler, M. Laumanns, and L. Thiele, "SPEA2: Improving the strength pareto evolutionary algorithm for multiobjective optimization," in *Evolutionary Methods for Design Optimization and Control with Applications to Industrial Problems*, 2001. (Cited on page 118.)

[116] Workshop on Mapping of Applications to MPSoCs, June 2012. (Cited on page 133.)

List of Figures

List of Tables