MAMOT: Memory-Aware Mapping Optimization Tool for MPSoC

Olivera Jovanovic and Peter Marwedel
Design Automation for Embedded Systems
University of Dortmund, Germany
Email: firstName.lastName@tu-dortmund.de

Iuliana Bacivarov and Lothar Thiele
Computer Engineering and Networks Laboratory
ETH Zuerich, Switzerland
Email: firstName.lastName@tik.ee.ethz.ch

***Abstract*—Mapping applications onto multiprocessor system-on-chip (MPSoC) devices is a well-known and complex optimization problem, receiving large interest in recent years. Diverse frameworks for mapping optimization have been developed, that typically distribute application tasks to available processors while optimizing one or more objectives such as performance and energy consumption. However, even though the memory subsystem is a resource that contributes drastically to the overall performance and energy consumption of the MPSoC, no mapping optimization framework is considering it so far. This paper addresses this challenge, and investigates the effect of memory hierarchies for MPSoC mapping. A memory-aware mapping optimization tool (that we refer to as MAMOT) has been developed for this purpose. We primarily focus on heterogeneous MPSoCs with heterogeneous memory hierarchies. Our evaluations show that considering memory mapping during MPSoC mapping optimization may reduce the application runtime up to 29.5% and its energy consumption up to 40%.**

***Keywords*-task mapping; memory hierarchy mapping; multiobjective optimization;**

I. Introduction

Multiprocessor system-on-chip (MPSoC) devices integrate multiple, often heterogeneous, components on a single chip to cope with the high performance requirements of today's applications and their increasing power consumption. But in order to optimally exploit MPSoCs, the mapping of applications onto these platforms has to match the available resource potential. Mapping optimization is a well-known and complex task, and several variants have been explored in the past years [1]–[3]. Even if they include complex search and optimization strategies, most of these tools are not fully exploiting or completely ignoring the memory subsystem.

But why are memories so important? Despite current remarkable technological advancements, there is still a gap between the speed of processors and speed of the memory, known as the memory wall problem [4]. Memory accesses tend to slow down the performance, while large memories consume a huge amount of energy. To cope with this problem, recent architectures are expanded with memory hierarchies, where the basic idea is to place small and fast memories close to processors (i.e., on-chip memories) and in this way access times can be reduced and energy consumption decreased. Moreover, in the design of real-time embedded systems, scratchpad memories are intensely used in on-chip memory hierarchies [5]. Scratchpad memories are predictable in terms of energy and performance, and consume less energy than caches because they do not require additional hardware for management. But unlike caches, scratchpads have to be explicitly allocated by the application designer. Considering all these issues and to alleviate design complexity, the memory wall problem and memory allocation have to be addressed already at system-level.

Memory-awareness adds one more dimension to the already complex application to processor mapping optimization problem, resulting in a huge design space to explore. In this paper, we address this challenge and investigate the effect of memory mapping on the overall mapping performance. In this sense, we have developed a novel and fully automatic memory-aware mapping optimization tool for MPSoCs (which we refer to as MAMOT). The framework employs multiobjective-aware optimizations, based on evolutionary algorithms. Analytic estimation models for runtime and energy consumption are included for fast evaluations. Moreover, as we explicitly consider scratchpad memories, mapping of promising instruction and data memory objects to scratchpad memories is automatically performed. Our experiments indicate runtime reduction up to 29.5% and energy consumption reduction by up to 40% compared to state-of-the-art mapping optimizations that don't include explicit memory mapping. This indicates that ignoring memory resources during system-level optimization leads to wasted optimization potential.

The rest of the paper is organized as follows: After investigating related work in section 2, the mapping problem is specified in section 3. Section 4 describes our memory-aware mapping optimization tool (MAMOT). Extensive evaluations are presented in section 5 and section 6 concludes the paper.

II. Related Work

Mapping tasks to processors in MPSoCs is a topic vastly explored in recent years, and various approaches for different architectures and targeting different objectives were proposed. Homogeneous or heterogeneous platforms were considered, with different configurations for power management like dynamic voltage scaling (DVS) [6], real-time requirements [8], dynamic scheduling/mapping [1], as well as a combination of those [7]. Heuristics, stochastic methods, or evolutionary algorithms are handling the complex system optimizations. 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 has been abstracted.

Szymanek et al. [9] use constraint logic programming for system synthesis in order to perform task assignment and to minimize the size of local memory. Suhendra et al. [10] use integer linear programming to find task mapping, pipelined scheduling, and scratchpad memory partitioning for MPSoC. During scratchpad partitioning, memory requirements for each task are calculated. If a task requires more local memory, the scratchpads of other processors are partitioned and used. Contrary to [9] and [10], we do not perform system synthesis nor scratchpad partitioning. Compared to these approaches, we are able to consider the entire memory hierarchy with several levels, different memory types and sizes. Further, automatic design space exploration and multiobjective memory-aware optimization are integral parts of our work.

Several frameworks for MPSoC design space exploration with multiple objectives exist, like for instance Daedalus [3], SystemCoDesigner [11], HOPES [12], or DOL [2]. 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. They have their own characteristics and cannot be compared to each other. Our work is constructed on top of the basic mapping optimization in the distributed operation layer (DOL), by including all memory-aware design concerns mentioned above. In particular, we use DOL because of the modular structure and flexible models for performance analysis and search in the design space [13], [14]. As DOL provides the basic mapping optimization framework, it allows us to investigate and develop our novel memory mapping, while considering a multilevel on-chip memory hierarchy. To achieve this, several refinements were necessary within DOL, such as adaption to thread mapping (DOL is considering process-based mapping), different performance analysis models for energy and runtime of threads, as well as the entire explicit memory modeling and memory optimization within specification, search, and analysis.

III. Problem Specification

Application model. We use a thread-based application model as depicted in Figure 1, where the application is supposed to be already parallelized and specified as a taskgraph. In this model, there is a main thread which is responsible for the control-flow. This thread initiates parallel computation by creating other threads that run in parallel as long as they have to accomplish computation and at the end they are all joined by the main thread that continues its execution. An application can have one or several parallel sections that contain a variable number of threads, as illustrated in Figure 1. The threads are the units that will be mapped onto the processors of the architecture. The advantage of this application model is that it explicitly exposes data parallelism and it can be automatically generated from existing sequential applications. More details on the parallelization tools are given in Section IV-A.

Each thread is composed of memory objects, i.e., instruction code and data. These memory objects represent resource requirements of a thread and are characterized by size and number of accesses. They directly influence performance and energy consumption of the system and have to be considered during memory-aware mapping optimization.

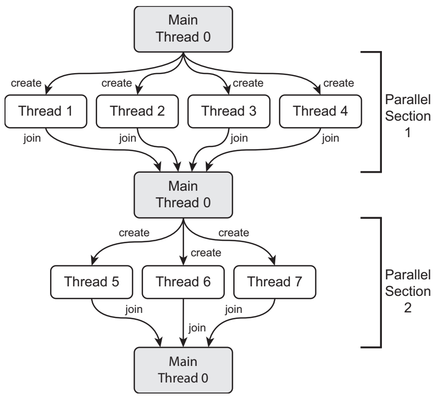

Figure 1. Thread-based application model.

Architecture model. The architecture we consider consists of *n* heterogeneous processors, e.g., different types and/or clock frequencies. Each processor has its own memory hierarchy, with different possible hierarchal levels and/or different sizes. An example architecture with four processors is illustrated in Figure 2. Each processor has separated instruction and data scratchpad memories on level 1, and a private memory on level 2, having exclusive access to these level 1 and 2 memories. The shared memory is the only memory accessible by all processors, via the on-chip network (or bus), and is used for inter-thread communication. Based on size, type, and hierarchical level, each memory causes different energy and runtime costs per access. After the specification of the architecture and application, the mapping optimization process can start.

Mapping. State-of-the-art application to MPSoC mapping tools perform the mapping of threads to available processors, while optimizing one or several objectives. The mapping optimization problem is known to be NP-complete. Even when considering just the threads to processors mapping, the number of options is significant. Considering in addition

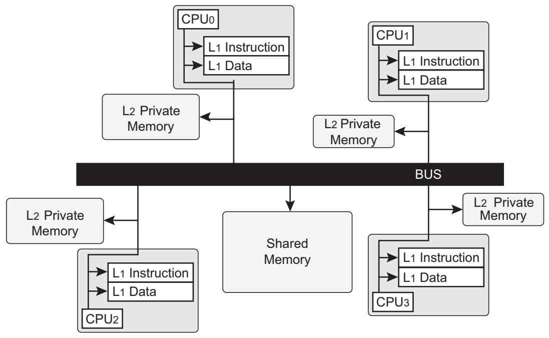

Figure 2. Heterogeneous MPSoC architecture with memory hierarchy.

the memory hierarchy, and therefore explicitly mapping memory objects to memories, increases further the options available and the problem complexity. Now, not only the processors but also the memories are crucial for deciding the mapping of a thread, and different combinations are possible. Mapping several threads to one processor, for instance, comes with additional constraints such as checking if memory capacity is enough for all the mapped threads. Not only the processor frequency but the combination of processor frequencies and the capacity and speed of their 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 have to be taken during memory-aware mapping optimization, such as, where should memory objects of a thread be placed? If the memory is small and shared among several threads, which memory objects of which thread gain the most benefit in terms of runtime or energy, for instance, and should be placed in local memories? Briefly, which 'processor/memory' pair is most suitable for which combination of possible 'threads/memory objects' mapping?

We solve this complex mapping problem with an evolutionary algorithm which is optimizing for performance and energy. Our approach is described in the next section.

IV. Memory-Aware Optimization Engine

This section describes our memory-aware mapping optimization approach, illustrated in Figure 3. The inputs are the application and architecture specifications, both described in a dedicated XML format. These XML files are automatically extracted by parallelization tools for the application, and from an external database for the architecture, respectively.

The 'core' part of the *memory-aware mapping optimization tool* (MAMOT) is an evolutionary algorithm (EA) that, based on the input data, generates memory-aware mapping solutions. Each solution is evaluated for performance and energy, with analytical models included in the tool, in a design space exploration loop. Since the mapping problem is multiobjective, there is no single optimal solution, but a set of Pareto-optimal solutions. The designer finally decides which solution is most suitable for his/her design requirements.

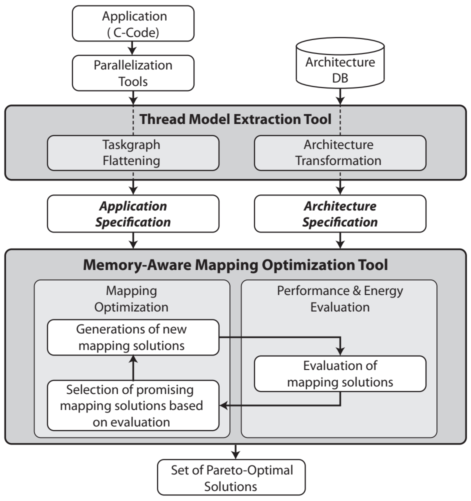
Application (C-Code)
Parallelization Tools
Architecture DB
Thread Model Extraction Tool
Taskgraph Flattening
Architecture Transformation
Application Specification
Architecture Specification
Memory-Aware Mapping Optimization Tool
Mapping Optimization
Performance & Energy Evaluation
Generations of new mapping solutions
Evaluation of mapping solutions
Selection of promising mapping solutions based on evaluation
Set of Pareto-Optimal Solutions

Figure 3. Complete toolflow for memory-aware mapping optimization.

A. Application Specification

To express the thread-based application model used as input by MAMOT, a new specification library has been developed, on top of DOL. Figure 3 shows the *parallelization toolchain* that generates the MAMOT application specification file. First, sequential ANSI-C application code is parallelized by using the tool proposed in [15]. Afterwards, the MPSoC Parallelization (MPA) tool [16] implements the parallelization by adding synchronization and communication mechanisms. The output is annotated application code, which contains parallel sections and communication/synchronization. This output is passed to the *thread model extraction tool*, which based on the annotated application code, extracts the application taskgraph. The taskgraph is then annotated with profiling information, such as runtime and energy consumption of the threads with the aid of the MACC-framework [17]. Finally, the *thread model extraction tool* generates the application specification file for MAMOT. All these steps are part of a fully automated toolchain.

In the following, we describe the content of the application specification file, where the main components are the individual threads and their memory objects. The specification structure of a thread is depicted in Figure 4. Each thread is defined by its *id* and the *id* of the *parallel section* it belongs to. Further, specific information such as *number of calls* of a thread, or *runtime* and *energy* consumption of a thread on the different processors of the architecture are added. These annotated energy/runtime figures do not include energy consumption or runtime required for buses and memory accesses, which are specified separately in the architecture file.

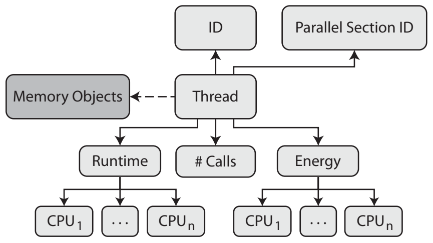

Figure 4. Thread specification.

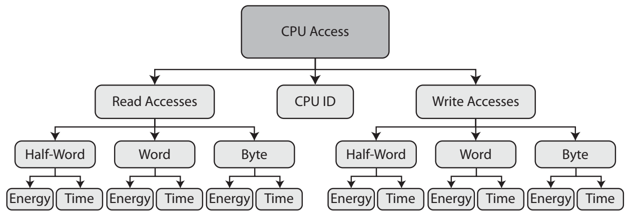

Figure 5. CPU access information for each memory.

Memory objects are specified within each single thread specification and they are primarily defined by their *id* and *size*. Next, the number of *read* and *write accesses* to the memory object has to be captured, since they can produce different runtime or energy consumptions. Further, it is important to know if the memory object is an instruction or data memory object, to determine a valid mapping to a specific memory type, such as data only, instruction only, or unified memory. Next, the *access width* of a memory object is important to be known, e.g., if it is accessed by the processor as word, half-word, or byte since it also implies different energy and runtime values. Finally, we also define the *mapping* of a memory object to a specific level in the memory hierarchy, by distinguishing among three possibilities. First, the memory object can be mapped anywhere in the memory hierarchy. Second, the memory object is mapped to the private memory of the processor. For memory objects with a size equal to or smaller than 4 bytes and frequently used, the compiler optimization usually maps them automatically to a register. The third *mapping* specification is reserved for shared memory objects. These memory objects are responsible for synchronization and communication between threads. Since they have to be accessed by all threads, they are automatically mapped to the shared memory.

In some cases, the designer wants 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 our tool, it is possible for designers to manually enforce mappings of threads to specific processors or mapping of certain memory objects to a specific location in the memory hierarchy.

B. Architecture Specification

The architecture specification file contains information about the resources of the architecture, i.e., processors and their frequency, memories and their contribution to energy consumption and runtime. As illustrated in Figure 3, all architecture information is extracted from an architecture database within MACC [17] framework. There, the designer can specify and choose among different architectures. The *thread model extraction tool* acts as an interface with this database, processing all required information and automatically generating the MAMOT architecture specification file.

As mentioned in Section II, DOL provides a basic architecture specification library. This library describes processors, simplified memory structures, and interconnections between them. For our memory-aware mapping optimization, we require more detailed memory specification, modeling a multilevel memory hierarchy and interconnect characteristics such as access time or energy consumption. For this scope, we have developed the respective specifications and added them to the DOL libraries.

First, the memory model has to include the system memory hierarchy, with all characteristics. 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, while *access type* defines if it is a unified access, access to data or instruction only memory objects.

Second, the interconnections between processors and memories are important. They specify which processor has access to a memory or how much does an access to a specific memory cost in terms of runtime and energy. These performance figures are different for each processor and memory type, and are important in order to determine the overall runtime and energy costs. Figure 5 illustrates the required specification. This specification is set up for each processor having access to a specific memory. Here, we first specify the *CPU id* together with access runtimes and energy information. We distinguish between *read* and *write accesses*, and also between different access widths such as *byte*, *halfword*, and *word*, because runtime and energy can vary for different accesses. This specification data is further used in the objective functions for energy and performance, during mapping optimization.

C. Optimization Objectives

This section provides a formal description of the optimization objective functions used in MAMOT for evaluating the mapping solutions within the evolutionary algorithm. All values required in these objective functions are extracted from the specification files.
Basically, we consider two functions to be optimized, i.e., system runtime and energy consumption. System runtime objective is given in Equation 1:

$$obj_1 = (nrCalls(t_0) * (r(t_0, p) \; + \sum_{m \in M(p)} r(t_0, m)))$$
$$+ \sum_{ps \in \; S} (\max_{t \in ps} \{nrCalls(t) * (r(t, p) + \sum_{m \in M(p)} r(t, m))\}) \qquad (1)$$

In the formal notation, $t \in T$ represents a thread (t_0 being the *main thread*), p represents a processor, and S is the set of parallel sections. The first line of Equation 1 represents the total execution time of the main thread t_0, while the second part iterates over all parallel sections $ps \in S$, where the parallel section runtime is given by the most time consuming thread. $nrCalls(t)$ defines the number of thread calls for thread t. $r(t, p)$ represents the runtime of thread t on processor p, without considering memory accesses, while $r(t, m)$ indicates the time of memory accesses caused by the memory objects in t for the specific memory m. The overall memory access runtime is obtained by iterating over all memories m in the set $M(p)$. $M(p)$ contains all memories m that are accessible by processor p to which the thread t is mapped to.

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

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

Here, we iterate over all memory objects $mObj$ that are mapped to ($\to$) memory m. The total access time is obtained with the number of accesses $nrReadAcc$ / $nrWriteAcc$ for the memory object $mObj$ multiplied by the time required for each access $accTimeRead$ / $accTimeWrite$. Different accesses cause different energy and runtime values. Furthermore, each read and write access also differs for the specified access width $accWidth(mObj)$. The access width also defines how many 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 and is defined as follows:

$$obj_2 = \sum_{p \in P} \{ \sum_{\forall \; t \to p} nrCalls(t) * (e(t, p) + \sum_{\substack{\forall \; m \; acc. \\ by \; p}} e(t, m)) \; \} \quad (2)$$

We iterate over all processors $p \in P$, and for each thread t mapped ($\to$) to processor p, the energy consumption has to be calculated. In detail, the number of calls $nrCalls(t)$ for thread t is multiplied by the energy $e(t, p)$ consumed by t on processor p plus the energy $e(t, m)$ consumed by accesses to the memories m. The equation for $e(t, m)$ is defined as follows:

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

This equation represents the energy consumption for memory accesses to an individual memory m accessed by thread t. We iterate over all memory objects $mObj$ that are mapped to memory m. Here, the number of accesses $nrReadAcc$ / $nrWriteAcc$ has to be multiplied by the energy consumed for a single access to memory m. Again, we distinguish between read and write accesses and the access width $accWidth(mObj)$ for both. These are given by the values $energyRead$ $(accWidth(mObj))$ and $energyWrite$ $(accWidth(mObj))$. Finally, we also multiply the number of additional accesses $nrLoadAcc$ $(accWidth(mObj))$ caused by the access width of the memory object to the overall read and write accesses, respectively.

D. Evolutionary Algorithm

We use the evolutionary algorithm as a black-box optimization for solving our mapping problem. In the first step, the evolutionary algorithm (EA) generates a population with several individuals, where each individual represents a mapping solution candidate, see Figure 6. Mapping decisions are then inspected randomly, and are generated in the following way: A thread, and the corresponding memory objects are mapped randomly to accessible processors. Figure 6.1 shows a possible distribution of threads to different CPUs. After a processor is chosen, memory objects of the threads

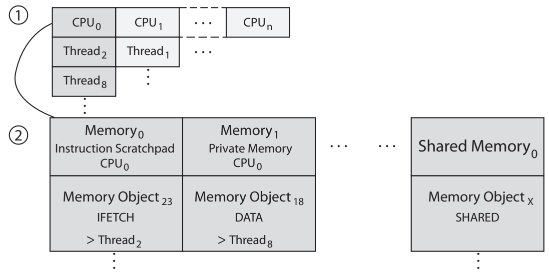

Figure 6. Individual representing a mapping solution candidate.

are mapped to accessible memories. This is illustrated in Figure 6.2. First, all predefined memory object mappings are performed, i.e., those marked with *shared* are mapped to the shared memory. Afterwards, the remaining memory objects of the threads are randomly mapped to the available memories. This step is performed for all threads of the application. Since attention is paid that a processor and its memories are accessible for a thread and its memory objects, only valid mappings are generated. Furthermore, if the memory is specified as instruction or data only memory, only instructions or data memory objects are mapped there, respectively. The EA 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 EA 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 common main memory.

In the next step in EA, 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 IV-C. 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.

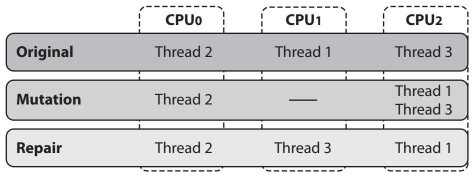

Figure 7. Mutation of genes.

In the *mutation step*, a thread and its corresponding memory objects are switched to another processor, an example being shown in Figure 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 are 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 EA 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 7, *repair* remaps $Thread3$ and its corresponding memory objects to CPU_1.

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 8. Originally, in $gene1$, $Thread2$ was mapped to CPU_1 and in $gene2$, $Thread3$ was mapped to CPU_2. In the crossover process, we remap $Thread3$ to CPU_1 in $gene1$ and in $gene2$ we remap $Thread2$ to CPU_2. Now, we again have to perform a repair operation, because this mapping causes invalid mapping solutions, since $Thread2$ in $gene2$ and $Thread3$ in $gene1$ are mapped twice (shown in brackets). In the example, we remap $Thread2$ to the processor where $Thread3$ was originally mapped to. Again, remapping repair is performed (for $gene2$). At the end, all affected memory objects are remapped.

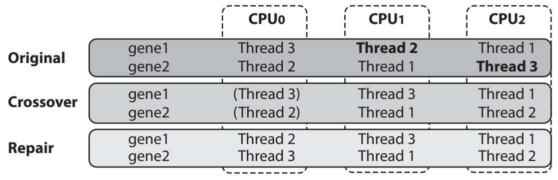

Figure 8. Crossover of genes.

The *mutation* and *crossover* functions, typically, generate new individuals, which by design cover well the entire solution space. 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 a maximum number of generations is reached. The result is a set of Pareto-optimal solutions where the designer can choose the best trade-off for his/her design requirements.

The evolutionary algorithm of our tool was implemented in PISA [14] and as underlying multiobjective search algorithm, Strength Pareto Evolutionary Algorithm (SPEA2) [18] is used.

V. Experimental Setup and Evaluation

This section presents the experimental setup and evaluations for MAMOT. Section V-A gives an overview over the used benchmarks and presents the heterogeneous architecture used for evaluations. In the evaluation Section V-B we discuss the results in terms of system performance and energy optimization by comparing our tool to a state-of-the-art mapping optimization.

A. Experimental Setup

For the validation of the memory-aware mapping optimization, several ANSI-C benchmarks from the UTDSP suite [19] are chosen, together with real-life benchmarks, and an illustrative industrial benchmark used in INTRACOM TELECOM S.A.'s Wimax system. The code sizes of the benchmarks range from 1 kB up to 328 kB with an average code size of 50 kB per benchmark. To parallelize and synchronize these application benchmarks, the tools described in Section IV-A have been used [15], [16]. Table I shows the number of extracted threads per benchmark.

The *Spectral* and *Wimax* benchmarks are the most complex benchmarks in our setup. They have 4 and 6 parallel sections and a total of 21 and 17 threads. Compared to *Wimax*, *Spectral* is smaller in terms of computation load

Benchmark	ADPCM	Boundary	Compress	Edge Detect	FIR	IIR	Jpeg	LAT	Mpeg4	Mult	Spectral	Wimax
Nr. Threads	4	5	5	9	5	5	7	3	4	6	21	17
Nr. Parallel Sections	1	1	1	1	1	1	1	1	1	1	6	4

Table I
PARALLELIZATION OF BENCHMARKS.

and complexity. Although other benchmarks hold only one parallel section, most of them still have complex threads in terms of computation workload, i.e., *Mpeg4*. Since we evaluate on an architecture with four processors, a mapping of 17 (WIMAX) or 21 (Spectral) threads with all corresponding memory objects is a complex task. Thus, we believe that these benchmarks represent a good average over mapping complexity and computation workload.

The solutions of the memory-aware mapping optimization are generated based on the objective functions presented in Section IV-C, and validated via the cycle-accurate instruction set simulator CoMET [20]. Therefore, the architecture depicted in Figure 9 is implemented in CoMET. The proposed architecture consists of four ARM11 processors with clock frequencies of 400 or 500 MHz. It has a heterogeneous memory architecture, where each processor has one instruction and one data scratchpad memory, each differing in size for the considered processors. Each processor has also a private memory of 8 or 16 MB. The system includes one DRAM shared memory of 512 MB for shared data and instruction.

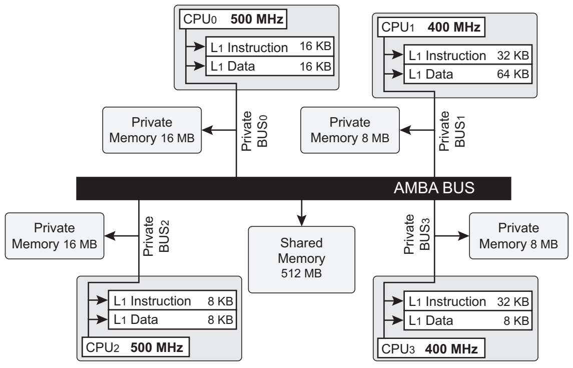

Figure 9. Heterogeneous MPSoC architecture for evaluation.

B. Evaluation

In order to show the improvements of MAMOT, we need an adequate comparison to other existing tools. But a comparison to tools mentioned in Section II is infeasible due to the different underlying application models (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, we compare MAMOT 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 MAMOT, 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 exploits the underlying memory hierarchy in its optimization models and objective functions. This is 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 both tools is a set of Pareto-optimal solutions, but how EAs generate solutions randomly, the generated solutions are not deterministic. Executing the evolutionary algorithm multiple times with the same parameters (number of genes, generations, etc.), could produce different Pareto solutions. In order to compare the two mapping optimizations, for each individual benchmark, each evolutionary algorithm was executed with 100 generations starting with a population of 100 genes. To compare the solutions, one (randomly chosen) point from each Pareto-front was selected and simulated with the cycle-accurate CoMET simulator. This step was performed 50 times in order to obtain convincing average values.

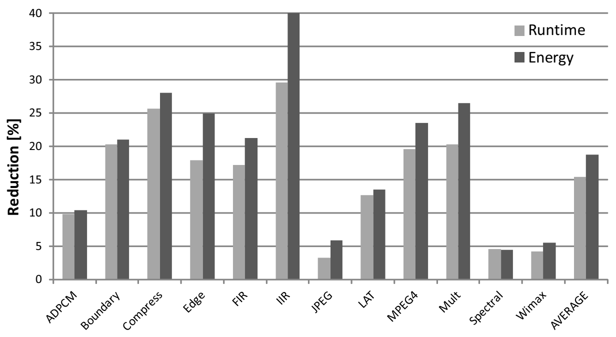

Figure 10. Optimization reduction achieved by MAMOT.

Figure 10 compares MAMOT with DOL, for all benchmarks, showing the additional optimization reduction obtained by MAMOT compared to solutions of state-of-the-art tools. The largest benefit is obtained for the *IIR* benchmark, where energy consumption was optimized by 40% and runtime by 29.5%. This large benefit is due to the fact that all memory objects of this benchmark could be mapped to the faster and more energy-efficient local scratchpad memories, i.e., except of those who are marked as private or shared. *ADPCM*, *Wimax* and *Spectral* gained benefits of 5% to 10%. 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 the threads could be mapped to local scratchpad memories and gain some benefit. In average, our tool optimized the runtime by 15.58% and the energy by 19%.

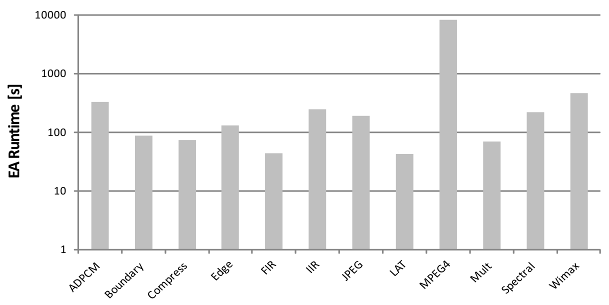

Figure 11. Runtime required for the evolutionary algorithm.

Figure 11 illustrates the time required for the execution of our evolutionary algorithm for 100 generations. Runtimes are given in seconds for an AMD Opteron 2.46 GHz. Most time was consumed for *Mpeg4* and *Wimax*. This shows, that even though *Mpeg4* has 4 threads and only one parallel section, the memory-aware mapping optimization of this complex benchmark is a time consuming task. This benchmark has about 46 kB of instruction memory objects and 262 kB of data memory objects, which can be mapped to local memory. Thus, the solution space is huge and the allocation step for memory objects is time consuming.

VI. Conclusions

This paper introduces a memory-aware multiobjective mapping optimization which considers MPSoC platforms with heterogeneous memory hierarchies. Since each application has different performance requirements, our optimization exploits the underlying memory hierarchy in order to efficiently match applications' requirements with architecture capabilities. To evaluate our memory-aware mapping optimization, we compared our approach with a state-of-the-art mapping tool that does not consider the memory hierarchy. The effectiveness of our approach is presented over various benchmarks and shows that our tool outperforms state-of-the-art mapping. On a heterogeneous platform, the system performance was optimized by up to 29.5% and energy up to 40%. 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.

Acknowledgment

This work was kindly supported by the SFB 876 research project of the Deutsche Forschungsgemeinschaft (DFG).

References

[1] E. W. Brião, D. Barcelos and F. R. Wagner. *Dynamic task allocation strategies in MPSoC for soft real-time applications*. In *Proc. of DATE*, Munich, Germany, 2008.

[2] L. Thiele, I. Bacivarov, W. Haid and K. Huang. *Mapping Applications to Tiled Multiprocessor Embedded Systems*. In *Proc. of ACSD*, Bratislava, Slovak Republic, 2007.

[3] H. Nikolov, M. Thompson, T. Stefanov, A. Pimentel, S. Polstra, R. Bose, C. Zissulescu, and E. Deprettere. *Daedalus: toward composable multimedia MP-SoC design*. In *Proc. of DAC*, Anaheim, USA, 2008.

[4] P. Machanick. *Approaches to Addressing the Memory Wall*. Technical report, School of IT and Electrical Engineering, University of Queensland, 2002.

[5] R. Banakar, S. Steinke, B. Lee, M. Balakrishnan, and P. Marwedel. *Scratchpad memory: design alternative for cache on-chip memory in embedded systems*. In *Proc. of CODES*, New York, USA, 2002.

[6] 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 *Proc. of DATE*, Munich, Germany, 2003.

[7] M. Schmitz, B. Al-Hashimi and P. Eles. *Energy-Efficient Mapping and Scheduling for DVS Enabled Distributed Embedded System*. In *Proc. of DATE*, Paris, France, 2002.

[8] S. Manolache, P. Eles and Z. Peng. *Task mapping and priority assignment for soft real-time applications under deadline miss ratio constraints*. *ACM TECS*, 7:19:1–19:35, 2008.

[9] R. Szymanek and K. Kuchcinski. *A constructive algorithm for memory-aware task assignment and scheduling*. In *Proc. of CODES*, Copenhagen, Denmark, 2001.

[10] V. Suhendra, C. Raghavan and T. Mitra. *Integrated Scratchpad Memory Optimization and Task Scheduling for MPSoC Architectures*. In *Proc. of CASES*, Seoul, Korea, 2006.

[11] C. Haubelt, T. Schlichter, J. Keinert and M. Meredith. *SystemCoDesigner: automatic design space exploration and rapid prototyping from behavioral models*. In *Proc. of DAC*, Anaheim, California, 2008.

[12] S. Ha. *Model-based Programming Environment of Embedded Software for MPSoC*. In *Proc. of ASP-DAC*, Yokohama, Japan, 2007.

[13] W. Haid, M. Keller, K. Huang, I. Bacivarov and L. Thiele. *Generation and Calibration of Compositional Performance Analysis Models for Multi-Processor Systems*. In *Proc. of SAMOS*, Samos, Greece, 2009.

[14] S. Bleuler, M. Laumanns, L. Thiele and E. Zitzler. *PISA - A Platform and Programming Language Independent Interface for Search Algorithms*. In *Proc. of EMO*, Faro, Portugal, 2003.

[15] D. Cordes, P. Marwedel and A. Mallik. *Automatic parallelization of embedded software using hierarchical task graphs and integer linear programming*. In *Proc. of CODES/ISSS*, Scottsdale, USA, 2010.

[16] R. Baert, E. Brockmeyer, S. Wuytack an T. J. Ashby. *Exploring parallelizations of applications for MPSoC platforms using MPA*. In *Proc. of DATE*, Nice, France, 2009.

[17] R. Pyka, F. Klein, P. Marwedel and S. Mamagkakis. *Versatile System-Level Memory-Aware Platform Description Approach for Embedded MPSoCs*. In *Proc. of LCTES*, Stockholm, Sweden, 2010.

[18] E. Zitzler, M. Laumanns and L. Thiele. *SPEA2: Improving the Strength Pareto Evolutionary Algorithm for Multiobjective Optimization*. In *Proc. of EUROGEN*, Athens, Greece, 2001.

[19] C. G. Lee. *UTDSP Benchmark Suite*. http://www.eecg.toronto.edu/~corinna/DSP/infrastructure/UTDSP.html, 2012.

[20] Synopsys. CoMET. *Virtual Prototyping Solution*. `http://www.synopsys.com`, 2012.