ILP-based Memory-Aware Mapping Optimization for MPSoCs

Olivera Jovanovic, Nils Kneuper, Michael Engel and Peter Marwedel

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

***Abstract*—The mapping of applications onto multiprocessor system-on-chip (MPSoC) devices is an important and complex optimization task. The goal is to efficiently distribute application tasks to available processors while optimizing for energy or runtime. Unfortunately, the influence of memories or memory hierarchies is not considered in existing mapping optimizations so far, even though it is a well-known fact that memories have a drastic impact on runtime and energy consumption of the system. In this paper, we address the challenge of finding an efficient application to MPSoC mapping while explicitly considering the underlying memory subsystem and an efficient mapping of task's memory objects to memories. For this purpose, we developed a memory-aware mapping tool based on ILP optimization. Evaluations on various benchmarks show that our memory-aware mapping tool outperforms state-of-the-art mapping optimizations by reducing the runtime up to 18%, and energy consumption up to 21%.**

I. Introduction

Multiprocessor system-on-chip (MPSoC) devices integrate several components on a chip in order to increase the available processing capabilities and decrease the power consumption. The mapping of applications onto these systems is a system-level optimization, which aims to exploit the resources of these systems efficiently. This important and complex task was explored intensely in the past years resulting in various optimizations with the goal to efficiently distribute application tasks to available processors while optimizing for a particular objective, e.g. performance or energy consumption. However, not all available system resources are considered in mapping optimizations. Most optimizations are not fully exploiting or are not even considering the memory subsystem at all.

However, the memory subsystem has a drastic influence on the system's runtime and energy consumption due to the still existing *memory wall problem* [1]. This problem describes the huge gap between the speed of processors and the speed of memories. 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 in on-chip memories and thereby to reduce runtime and energy consumption.

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 [2]. The content of these memories is known in advance, they are predictable with respect to runtime and energy consumption. But unlike caches, they have to be explicitly allocated by the application designer. Also, they consume less energy than caches since no additional hardware is required for the management of its content. For these reasons, we explicitly consider scratchpad memories instead of caches.

Due to the drastic influence of the memory wall problem on runtime and energy, it is crucial to consider the underlying memory hierarchy in the application to architecture mapping as well. In this paper, we address this challenge. Our memory-aware mapping optimization is able to consider both homogeneous and heterogeneous systems. Furthermore, since we consider scratchpad memories, an allocation of promising instruction and data memory objects is automatically performed. Fast evaluation is achieved due to analytic estimation models for runtime and energy consumption. The definition of an automatic, fast and flexible memory-aware mapping optimization framework has the aim to support MPSoC designers in their design decision. An integer linear programming (ILP) optimization is defined for the overall performance or energy consumption aiming at optimal solutions. Our optimization is validated on a cycle-accurate simulator. A reduction of runtime by up to 18% and a reduction of energy consumption by up to 21% is achieved compared to state-of-the-art mapping optimizations. This shows that ignoring memory resources in a system leads to wasted optimization potential.

The rest of the paper is organized as follows: Related work is investigated in Section 2 and a problem specification is given in Section 3. The memory-aware mapping optimization engine is described in Section 4. Section 5 describes the experimental setup and then the evaluation is illustrated in Section 6. Section 7 concludes the paper.

II. Related Work

Various approaches exist for mapping tasks to processors on different architectures (homogeneous and heterogeneous). Different configurations for power management like dynamic voltage scaling (DVS) [6], real-time requirements [8], dynamic scheduling/mapping [3], as well as a combination of these requirements [7] were considered. However, these approaches do not consider memory mapping or the integration of the memory hierarchy in their objectives. The general assumption

978-0-7695-4914-9/12 $26.00 © 2012 IEEE
DOI 10.1109/ICCSE.2012.64

is that the system has enough memory to cover all memory requirements of given tasks, or the influence of memories on the execution time or energy of tasks are abstracted.

Also, various optimizations for mapping instruction or data memory objects to scratchpad memories exist for single core SoC [10], [9] and recently also for MPSoC devices [11], [12]. But, in MPSoC devices these optimizations are decoupled from the application to processor mapping optimization. They are usually performed *after* this step. However, Steinke et al. [10] statically assign code and data objects to scratchpad memories. Verma et al. [9] use overlay techniques for scratchpad memories in order to dynamically copy code and data memory objects. Both optimizations work only for single core SoC devices. Che et al. [11] utilize on-chip memory for code overlay and communication overhead caused by FIFOs. All on-chip memories are accessible by all other processors. He et al. [12] combine task scheduling with memory access optimization for MPSoCs on a two-level-memory hierarchy. They consider task graphs with multiple entry and exit points. Here, only the scheduling of tasks is optimized and only data memory objects are mapped to on-chip memories. None of these publications integrate their optimizations in an application-to-architecture mapping and they consider only fully homogeneous architectures where all local memories have the same size.

However, some publications integrate memories in the mapping step. Szymanek et al. [13] perform system synthesis with task assignment, scheduling and assignment of data with constraint logic programming. This work targets a different optimization level and can not be compared to our work. Suhendra et al. [14] suggest ILP to find task mapping, pipelined scheduling, and scratchpad memory partitioning for MPSoC. Partitioning means that each processor has access to the scratchpad memories of the other processors. If a task requires more local memory, the scratchpad memory of another processor will be partitioned and used. Here, only data memory objects are allocated to the local memory.

Compared to [14], we consider the entire memory hierarchy with different memory types and sizes. Also, our optimization allocates both instruction and data memory objects to memories. We consider a heterogeneous architecture model, where partitioning of scratchpad memories is not performed (i.e. processors do not have access to the scratchpad memories of other processors). The combined mapping of threads to processors with the mapping of memory objects to their underlying memories are integral parts of our work. Our ILP optimization is able to obtain an application-to-architecture mapping for homogeneous and heterogeneous systems.

III. Problem specification

This section describes the underlying architecture and application models. Furthermore, the complexity of the integration of memory-awareness into the application to architecture mapping optimization is described in more detail.

Application model. Here, we assume that an already parallelized application is represented as an acyclic task graph. A thread-based application model as depicted in Figure 1 is used. Here, a main thread accomplishes computation, creates new threads and hereby initiates the parallel execution. The 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 1. Here, the application has two parallel sections with four threads in parallel section 1 and three threads in parallel section 2. Each 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 IV-B.

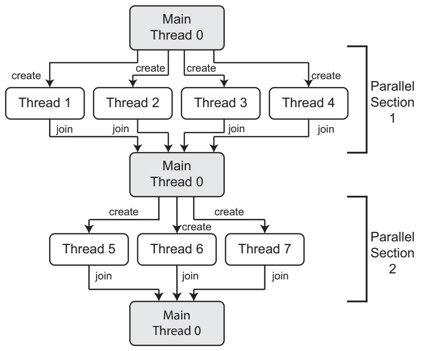

Fig. 1. Thread-based application model

Each thread consists of memory objects, which are either instruction code or data. They are mainly characterized by their size and the frequency of accesses and 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.

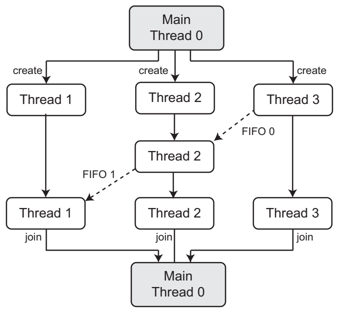

Fig. 2. Task graph including FIFO communication

Based on the underlying characteristics of the application, our model can also contain FIFO queues (First-in First-out) for inter-thread communication as depicted in Figure 2. In this case, a thread can be composed of several thread nodes. The edges between the different thread nodes illustrate either FIFO (i.e., communication) edges or control flow edges. This specification ensures the precise determination of runtime since a thread node cannot be executed until all predecessor nodes have finished their execution or communication, respectively.

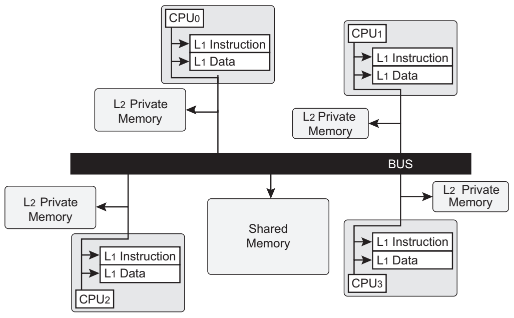

Fig. 3. Heterogeneous MPSoC architecture with memory hierarchy

Architecture model. We are considering an architecture with *n* processors, which can be homogeneous or heterogeneous, i.e. differ in type and/or clock frequency. Each processor can have a distinct memory hierarchy with different hierarchy levels and different sizes on each level. Figure 3 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. Further, the architecture has at least one shared memory which is accessible by all processors and used for inter-thread communication and synchronization. The shared memory can be accessed through an on-chip network (or bus). The local memories can differ in the memory type (instruction only / data only / unified). All memories can differ in size. Each memory has different energy and runtime values which depend on type, level and size of the memory.

Based on the information specified in the application and architecture model, the ILP-based memory-aware mapping optimization provides an optimal solution. In the evaluation section, we will validate our solutions on a cycle-accurate simulator.

Mapping. State-of-the-art application to MPSoC mapping tools perform the mapping of threads to available processors, while optimizing for an objective. The mapping optimization problem is known to be NP-complete. Considering the thread to processors mapping step, the number of options is significant. The available options and problem complexity is further increased by including the memory hierarchy with an explicit mapping of memory objects to memories. Now, next to the processor also the memories are crucial for the decision, where different combinations are possible. If several threads are mapped to one processor, several constraints have to be checked, i.e. memory capacity shouldn't be exceeded or the examination of the memory type. A combination of processor frequencies and the capacity and speed of their underlying memory hierarchy has to be adequate for the threads and their individual resource requirements (i.e., memory objects). Also, the allocation of the proper memory objects of a thread is crucial. If a small memory has to be shared among several threads, the decision has to be taken, which memory objects of which thread gain the most benefit in terms of runtime or energy and should be placed in local memories? To sum up: Which 'processor/memory' pair is most suitable for which combination of possible 'threads/memory objects' mapping?

We solve this complex mapping problem with integer linear programming (ILP). Based on the system requirements, the designer can choose between an ILP for the minimization of the overall energy consumption and an ILP for the minimization of runtime. Our approach will be described in detail in Section IV-C.

IV. Memory-Aware Optimization Engine

This section describes our ILP-based memory-aware mapping optimization approach which is also illustrated in Figure 4. First, an overview over the fully automated tool-flow is given. Nevertheless, the designer is still able to perform steps manually (i.e. manual parallelization or mapping).

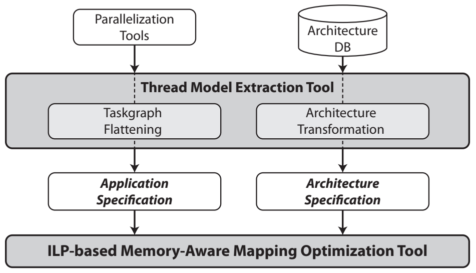
Parallelization Tools
Architecture DB
Thread Model Extraction Tool
Taskgraph Flattening
Architecture Transformation
Application Specification
Architecture Specification
ILP-based Memory-Aware Mapping Optimization Tool

Fig. 4. Memory-aware Mapping Optimization: Tool-Flow Overview

All tools are implemented within the MACC-framework [20] which provides an architecture database and profiling and analysis tools. The input of the tool-flow is sequential ANSI-C application code which has to be parallelized first by a parallelization tool [21]. In the next step, the parallelization is implemented by adding synchronization and communication mechanisms (i.e. FIFOs, etc.), by using the MPSoC Parallelization (MPA) tool [22]. At this point, annotated application code is provided which defines parallel sections with communication and synchronization. Based on this output, the *Thread Model Extraction Tool* extracts a flat task graph and annotates additional profiling information (i.e, number of calls of threads and memory objects, etc.). The output of this tool is passed to our ILP-based memory-aware mapping optimization tool which processes all given information and performs the ILP optimization. A description of architecture and application specifications will be explained next. All this information contributes to the ILP optimization described in Section IV-C.

A. Architecture Specification

The architecture specification is fed with information from an architecture database within the MACC-framework [20]. This framework provides different predefined architectures, as well as the means to model further architectures. In general, the architecture specification contains all information about processors, memories and the interconnections between them. Further, energy and runtime information about all these components is specified (for example active/idle energy for processors, memories, interconnections). All memories and their characteristics are retrieved, which are the *size*, *type* (i.e. scratchpad, shared, private), and *access type* (i.e. unified, data or instruction only) of the memory. For each memory, it is defined which processor has access to it and the energy and runtime values caused by these different type of accesses (i.e. access width and read/write).

B. Application Specification

The ILP Optimization Tool reads the annotated task graph, which was generated by the *Thread Model Extraction Tool*. The important components of the application and its specification are the threads, their interconnection (i.e. control-flow, FIFO), thread nodes and their memory objects. Next to profiling information as runtime/energy on each processor (without memory delay), all memory objects belonging to the thread are defined with all important information (i.e. number of accesses, size, access width and type (instruction, data, shared), etc.). If a thread is not allowed to be executed on a processor, this information is also specified properly. The designer is allowed to manually specify mappings for memory objects to designated memories and threads to designated processors.

C. ILP Optimization

This section presents the integer linear programming (ILP) equations for an 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.

1) Optimization for Runtime: Here, the ILP formulation for the optimization of the overall runtime is presented.

The objective function minimizes the runtime of the given application. 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. The goal is to minimize the completion time of the last thread which is the exit node of the application:

$$min\,(EndThrNode_{Exit}) \tag{1}$$

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 $ExecTimeNode_{i,j}$:

$$EndThrNode_{i,j} \geq StartThrNode_{i,j} + ExecTime_{i,j} \tag{2}$$

The value of $StartThrNode_{i,j}$ is dependent on the predecessors of thread node (i,j) and will be explained later.
Below, additional constraints for the ILP are defined.

a) Overall runtime of a thread: The overall runtime of a thread is composed of the execution time of the thread on a processor and the time required for memory accesses.

$$ExecTime_{i,j} = ProcTime_{i,j} + AccTime\_memories_{i,j} \tag{3}$$

The execution time on a processor $ProcTime_{i,j}$ and the access time $AccTime\_memories_{i,j}$ to the memories depend on the mapping decision of this optimization.

The set of threads is defined by $T = 1, 2, .., k$ and the number of processors with $P = 1, .., p$. $X_{i,p}$ is a binary decision variable and has the value 1 if thread i is mapped to processor p, otherwise it has value 0.

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

$$ProcTime_{i,j} = \sum_{p=1}^{P}(X_{i,p} * ExecTimeProc_{i,j,p}) \tag{4}$$

The value $ExecTimeProc_{i,j,p}$ represents the runtime of thread node i,j on processor p.

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 equation is defined for each thread i:

$$\sum_{p=1}^{P} X_{i,p} = 1 \tag{5}$$

Next, the overall access time to memories $AccTime\_memories_{i,j}$ caused by thread node (i,j) is defined by:

$$\begin{aligned} &AccTime\_memories_{i,j} = \\ &\sum_{m_{obj,i} \in MO_{i,j}} (\sum_{m=1}^{M} ProcVarMem_{p,m,m_{obj,i}} \\ &\quad * (\ NrReadAcc_{m_{obj,i}} * AccTimeRead_{p,m,m_{obj,i}} \\ &\quad + NrWriteAcc_{m_{obj,i}} * AccTimeWrite_{p,m,m_{obj,i}}\ )) \end{aligned} \tag{6}$$

Let MO_i be the set of memory objects for thread i and M the set of memories. The set $MO_{i,j}$ is a subset of MO_i and contains all memory objects that are accessed by thread node (i,j). This equation iterates over all memory objects $m_{obj,i}$ of thread i which are elements of the set $MO_{i,j}$ and sums up the time that is required for the access to each memory object. The access time depends on the memory to which $m_{obj,i}$ is mapped. The binary decision variable $ProcVarMem_{p,m,m_{obj,i}}$ has value 1 when memory object $m_{obj,i}$ is mapped to memory m which is accessible by processor p. Otherwise its value is 0. The overall access time spent on a memory m is composed of read and write accesses of memory object $m_{obj,i}$. The number of read accesses $NrReadAcc_{m_{obj,i}}$ is multiplied with the access time $AccTimeRead_{p,m,m_{obj,i}}$ required for one read access. The same is defined for the write accesses $NrWriteAcc_{m_{obj,i}}$ to the memory object $m_{obj,i}$. As mentioned before, read and write accesses can cause different access times which also depend on size and access width of the memory object. Our 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. Also, only valid combinations of $ProcVarMem_{p,m,m_{obj,i}}$ are generated with memories m which are accessible by processor p and which are capable of holding the appropriate memory objects (i.e. a data memory object can not be mapped to an instruction memory). Please note that this access time also includes the time for the access to the underlying buses.

b) Memory constraints: The next constraint ensures that the size of memories are not exceeded by the memory objects mapped to the memories. Following equation is defined for each memory:

$$size_m \geq \sum_{i=1}^{T} \sum_{m_{obj,i}=1}^{MO} (ProcVarMem_{p,m,m_{obj,i}} * size_{m_{obj,i}}) \tag{7}$$

The capacity of a memory m is given by $size_m$. The size of a memory object $m_{obj,i}$ is given by $size_{m_{obj,i}}$. The right hand side of this equation iterates over all threads i and all their memory objects $m_{obj,i}$ and sums up the size of all memory objects mapped to memory m.

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 i is mapped to a processor p, then the memory objects $m_{obj,i}$ of this thread i can only be mapped to a memory m that can be accessed by processor p. This is defined in the next equation:

$$X_{i,p} = \sum_{m=1}^{M} ProcVarMem_{p,m,m_{obj,i}} \tag{8}$$

This equation also ensures that a memory object $m_{obj,i}$ can be mapped to only one memory m in the system.

c) Dependencies in the task graph: A thread node can start its execution when all predecessors have finished their execution. This is defined by following equation:

$$\begin{aligned}
&\forall pred_{i,j} : \\
&if(i = h) : \\
&StartThrNode_{i,j} \geq EndThrNode_{h,l} + 1 \\
&if(i \neq h) : \\
&StartThrNode_{i,j} \geq EndThrNode_{h,l} + 1 + FIFO_{i,h}
\end{aligned} \tag{9}$$

For all predecessors $pred_{i,j}$ of thread node j following is defined: the start time $StartThrNode_{i,j}$ is greater than the finishing time of $EndThrNode_{h,l}$ of its predecessor node l of thread h added to the time required for communication over FIFOs between both tasks. Here, the predecessor node can belong to the same thread as the successor node and represents a control flow edge, i.e. for the case if $i = h$. Or, if FIFO communication is performed, the predecessor belongs to a different thread node, i.e. for the case $i \neq h$. Depending on the application and its parallelization, the FIFO communication is optional. Here, the number 1 represents on time unit.

d) Time for FIFO Communication: If FIFO communication is defined, the thread consists of more than one node. One of these nodes has an incoming FIFO edge (FIFO read). The source node of the incoming FIFO edge must first complete its execution before the target node can proceed with its execution.

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

$$FIFO_{i,h} = \frac{NrElements * SizeElement * NrAccesses}{BusSpeed} \tag{10}$$

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

e) Processor assignment: The following 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. This equation is taken from [14] and was adapted to our thread model.

Let $L_{h,i}$ be a decision variable which has value 1 when thread h and thread i are mapped onto different processors, and else value 0, if they are mapped onto the same processor j. Here, only different threads are considered, i.e. $h \neq i$.

$$\forall p : 1...P, \quad L_{h,i} \leq 2 - X_{h,p} - X_{i,p} \tag{11}$$

$$\forall p : 1...P, \forall q : 1...P, p \neq q, \quad L_{h,i} \geq X_{h,p} + X_{i,q} - 1 \tag{12}$$

The decision variable $L_{h,i}$ is set to 0 or 1 by the definition of the decision variables $X_{h,p}$ and $X_{i,q}$. Also, $L_{h,i}$ is required for all combinations of threads in the parallel section. There must be one variable $L_{h,i}$ for the sequence that thread h is executed before i and one variable $L_{i,h}$ for the sequence where thread i is executed before h.

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. $h \neq i$ which belong to the same parallel section. Thread node (h, l) belongs to thread h and thread node (i, j) belongs to thread i. Here, variable $B_{h,i}$ is set to 1 if thread node (h, l) and thread node (i, j) are mapped onto the same processor and thread h is executed after thread i, else it is set to 0.

$$B_{h,i} + B_{i,h} - L_{h,i} = 1 \tag{13}$$

$$StartThrNode_{h,l} \geq EndThrNode_{i,j} - \infty * B_{h,i} + 1 \tag{14}$$

$$StartThrNode_{i,j} \geq EndThrNode_{h,l} - \infty * B_{i,h} + 1 \tag{15}$$

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 9.

f) Restrictions of ILP model: Here, the restrictions of this ILP model is described. First, the accesses to memories can not always be modeled as precisely as in real memories. For example, due to the abstraction level of this model, fast accesses which are performed in blocks are not covered. Also, due to the processor pipeline, access time to memories can be decreased. A precise model covering processor pipelines is not included here.

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 maintained 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 provided.

For the minimization of the overall energy consumption of the system, the objective function is defined in equation 16:

$$min\,(CPUEnergy + MemEnergy) \tag{16}$$

The goal is to minimize the total energy consumption which consists of the energy consumed by all processors and all memory accesses. The memory accesses also include the energy spent for the underlying buses.

a) 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 17:

$$\begin{aligned} &CPUEnergy = \\ &(\sum_{p=1}^{P} IdleEnergy_{Proc_p} * EndThread_{Exit}) \\ &+ (\sum_{i=1}^{T}\sum_{p=1}^{P} CPUEnergy_{i,p} * X_{i,p}) \end{aligned} \tag{17}$$

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

Here, $CPUEnergy_{i,p}$ defines the difference between the energy for the active mode and idle mode:

$$CPUEnergy_{i,p} = Cycles_i * \left(Active_{Proc_p} - Idle_{Proc_p}\right) \tag{18}$$

Only the execution cycles that are required for the computation of thread i are multiplied with the difference between active energy $Active_{Proc_p}$ and idle energy $Idle_{Proc_p}$.

b) Energy consumption for memory accesses: Here, the energy consumption for all memory accesses are defined:

$$\begin{aligned} &MemEnergy = \\ &\sum_{i=1}^{T}\sum_{m=1}^{M}\sum_{m_{obj,i}\in MO_{i,j}} ProcVarMem_{p,m_{obj,i},m} \\ &\quad * NrReadAcc_{i,m_{obj,i}} * AccEnergyRead_{p,m_{obj,i},m} \\ &\quad + NrWriteAcc_{i,m_{obj,i}} * AccEnergyWrite_{p,m_{obj,i},m} \end{aligned} \tag{19}$$

This equation iterates over all threads i, over all memories m and over all memory objects $m_{obj,i}$ in the system. If a memory object $m_{obj,i}$ is mapped to a memory m (i.e. $ProcVarMem_{p,m_{obj,i},m}$ 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 $NrReadAcc_{i,m_{obj,i}}$ for the memory object $m_{obj,i}$ of thread i is multiplied by the energy $AccEnergyRead_{p,m_{obj,i},m}$ that is required for one read access. The energy consumption for the write accesses to $m_{obj,i}$ 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.

V. Experimental Setup and Evaluation

A. Experimental Setup

The benchmarks used in this evaluation are taken from the UTDSP suite [28] and real-life benchmarks (MPEG4). The code size of benchmarks ranges from 6.5 kB up to 3 MB with an average code size of 50 kB per benchmark.

In a first step, parallelization tools [22], [21] 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 I.

The *Spectral* and *Mpeg4* are the most complex benchmarks in this setup. The *Spectral* benchmark has 6 parallel sections and 21 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. We believe that these benchmarks represent a good average over mapping complexity and computational workload.

The commercial ILP solver CPLEX [24] is used for the minimization of the objective functions in our ILP formulation. Energy values are computed according to the model defined by Steinke et. al. [10] while the memory models are provided by CACTI [25]. For the validation of our solutions generated by the ILP-optimization, the cycle-accurate instruction set simulator CoMET [29] is used.

The core characteristics of the architecture considered in this evaluation is comparable to the *single-ISA heterogeneous multi-core architecture* suggested in [27], [26]. 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, raw execution bandwidth, or other characteristics. In [27], 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. A similar strategy is used in the big.Little system from ARM where a high-performance Cortex-A15 processor is paired with an energy efficient Cortex-A7 processor [30]. Architectures like these seem to become more common.

Based on these assumptions, the architecture illustrated in Figure 5 was implemented for the CoMET simulator. 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. The system also contains one DRAM shared memory with 512 MB for shared data and instructions.

Benchmark	Boundary	Compress	Edge Detect	FIR	IIR	LAT	Mpeg4	Spectral
Nr. Threads	5	5	9	5	5	3	4	21
Nr. Parallel Sections	1	1	1	1	1	1	1	6

TABLE I
PARALLELIZATION OF BENCHMARKS.

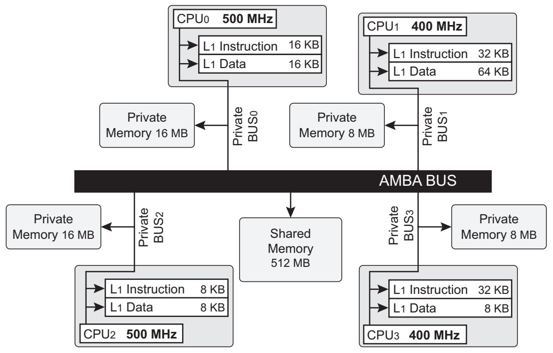

Fig. 5. Heterogeneous MPSoC Architecture for Evaluation

B. Evaluation

In this section, the memory-aware mapping optimization is compared to a 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 neither includes memory-awareness nor mapping of memory objects to on-chip memories. 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. In this way, we create a comparable reference to the common state-of-the-art mapping optimizations.

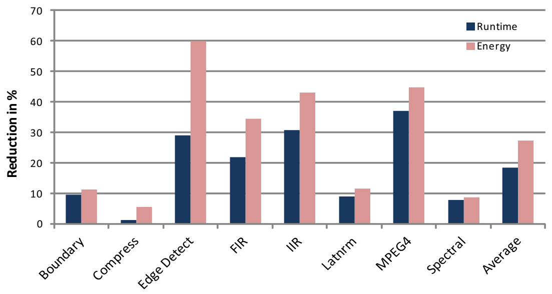

Fig. 6. Reduction in runtime and energy achieved by Runtime-ILP

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. The results for the ILP-based optimization with the goal to minimize runtime are shown in Figure 6. 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 which minimizes runtime. Although the goal of the ILP is to minimize the runtime, the resulting system's energy consumption is also illustrated in this figure. For the memory-aware ILP-based mapping optimization, the benchmarks *MPEG4* and *IIR* showed the most reduction of runtime with 37% and 30%, respectively. 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. The memory-aware ILP optimization maps the threads to suitable processors where the most efficient memory objects are allocated to the on-chip memories and thus it gains the largest benefit. The allocatable memory objects for the *IIR* benchmark are all allocated on the on-chip memories. Also, the time spent for the execution of the instructions is large in this benchmark. However, *Compress* and *Boundary* reached a reduction of 1.3% and 9.5%. In the *Compress* benchmark, the most time is spent on accessing shared memory objects, which have to be mapped to shared memory and therefore no speed-up can be reached here. The average reduction for runtime is about 18%. Concerning the resulting overall energy consumption of the system, over all benchmarks the energy was also reduced by a remarkable amount of 27%.

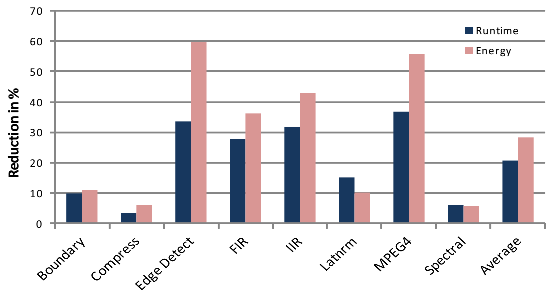

Fig. 7. Reduction in energy and runtime achieved by Energy-ILP

The evaluation results for the energy based ILP memory-aware mapping optimization are shown in Figure 7. Please note, that the base line here is not the same as in Figure 6. The underlying baseline of Figure 6 is a state-of-the-art ILP-based runtime optimization. In Figure 7, 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. However, for the energy-based ILP optimization, the resulting runtime of the benchmarks is also illustrated in this figure. Here, again the benchmarks *MPEG4* and *IIR* reached the most reduction of energy by 37% and 32% respectively. However, *Compress* and *Spectral* reached a reduction of about 3.6% and 6%. The average energy reduction is about 21%. Compared to the overall runtime of the state-of-the-art mapping, a reduction of about 28% in average is reached.

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.

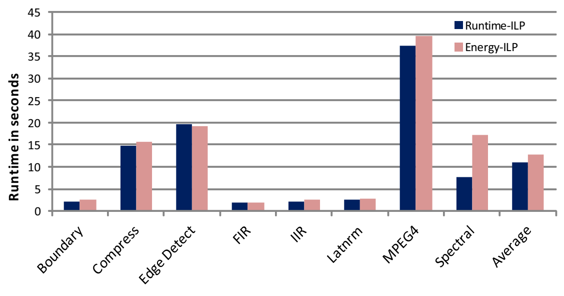

Fig. 8. Runtime of ILP-optimization

Figure 8 illustrates the time that was required for the execution of the memory-aware ILP-based mapping optimization. The runtime is illustrated in seconds for an AMD Opteron 2.46 GHz. The optimization of the *Mpeg4* benchmark consumes the most time and takes about 37 seconds. This can be explained by the huge amount of allocatable memory objects which expands the solution space. Over all benchmarks, an average runtime of 11 seconds was achieved.

VI. Conclusions

This paper presents a novel ILP-based memory-aware mapping optimization for homogeneous and heterogeneous MPSOC devices with memory hierarchies. Next to the mapping of application threads to processors, our 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.

We evaluated our memory-aware mapping optimization by comparing our approach to a state-of-the-art mapping optimization which neither includes the consideration of the memory hierarchy nor the mapping of instruction and data memory objects to memories. For a heterogeneous platform the overall performance is optimized by 18% and energy consumption was optimized by 21% over various benchmarks.

Acknowledgment

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

References

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

[2] 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.

[3] 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.

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

[5] 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.

[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] M. Verma and P. Marwedel. Overlay Techniques for Scratchpad Memories in Low Power Embedded Processors. In IEEE TVLSI, 2006.

[10] S. Steinke, L. Wehmeyer, B. Lee and P. Marwedel. Assigning Program and Data Objects to Scratchpad for Energy Reduction. In *Proc. of DATE*, Paris,France, 2002.

[11] W. Che, A. Panda, and K. S. Chatha. Compilation of stream programs for multicore processors that incorporate scratchpad memories. In *Proc. of DATE*, Dresden, Germany, 2010.

[12] Y. He, C. J. Xue, C. Xu, and E. H.-M. Sha. Co-optimization of memory access and task scheduling on MPSoC architectures with multi-level memory. In *Proc. of ASPDAC*, Taipei, Taiwan ,2010.

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

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

[15] 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.

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

[17] 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.

[18] K. Huang, W. Haid, I. Bacivarov, M. Keller and L. Thiele Embedding Formal Performance Analysis into the Design Cycle of MPSoCs for Real-time Streaming Applications. In *ACM TECS*, 2012.

[19] 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.

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

[21] 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.

[22] 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.

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

[24] CPLEX Ltd. http://www.cplex.com, 2012.

[25] S. Thoziyoor, N. Muralimanohar, J. H. Ahn and N. P. Jouppi. CACTI 5.1. *Technical Report HPL-2008-20*, HP Laboratories, 2008.

[26] 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 *Proc. of MICRO*, San Diego, USA 2003.

[27] 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 *Proc. of ISCA*, Munich, Germany, 2004.

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

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

[30] ARM, Ltd. White Paper: Big.LITTLE Processing with ARM Cortex-A15 & Cortex-A7. http://www.arm.com/files/downloads/big.LITTLE_Final.pdf, 2012.