Compilation and Simulation Tool Chain for Memory Aware Energy Optimizations

Manish Verma[1], Lars Wehmeyer[1], Robert Pyka[1], Peter Marwedel[1], Luca Benini[2]

[1] Department of Computer Science XII, University of Dortmund, 44221 Dortmund, Germany
[2] DEIS, University of Bologna, 40136 Bologna, Italy

Abstract. Memory hierarchies are known to be the energy bottleneck of portable embedded devices. Numerous memory aware energy optimizations have been proposed. However, both the optimization and the validation is performed in an ad-hoc manner as a coherent compilation and simulation framework does not exist as yet. In this paper, we present such a framework for performing memory hierarchy aware energy optimization. Both the compiler and the simulator are configured from a single memory hierarchy description. Significant savings of upto 50% in the total energy dissipation are reported.

1 Introduction

Contemporary portable devices are experiencing an ever-increasing spiral of feature enhancement and device convergence. Today's mobile devices, besides acting as phones, also serve as PDA, MP3 player, digital camera and also as a video game console. Fast processors, large memories and aggressive energy optimization techniques are required to support all the aforementioned features in a portable device. It is expected that future devices will have even faster processors and larger memories, both of which are extremely power hungry. As a consequence, a lot of research effort is being directed towards energy optimizations.

The memory subsystem has been identified as the energy and performance bottleneck of the entire system. This problem is expected to aggravate in the future as the performance gap between the processor and the memory is growing. This phenomenon is also known as the "Memory Wall Problem" [1]. Memory hierarchies are constructed to improve the energy dissipation and the performance of the memory subsystem. In addition, the application is optimized to efficiently utilize the memory subsystem.

In order to perform a fast and efficient design space exploration, a coherent framework for code-optimization and system simulation is required: A framework which can optimize the application code for a given memory hierarchy and also evaluate the optimization by simulating the optimized executable on the same memory hierarchy. Unfortunately, most contemporary memory optimizations are performed at the source-level with a complete disregard to the compiler generating the executable. Often, the simulation framework is also a stopgap solution such that every new memory hierarchy requires manual intervention and recompilation of the entire simulation framework. In this paper, we present a coherent framework called Memory Aware C Compilation (MACC) framework.

Figure 1 presents the workflow of the MACC framework. The MACC framework is partitioned into the compilation and the simulation framework, both of which share the

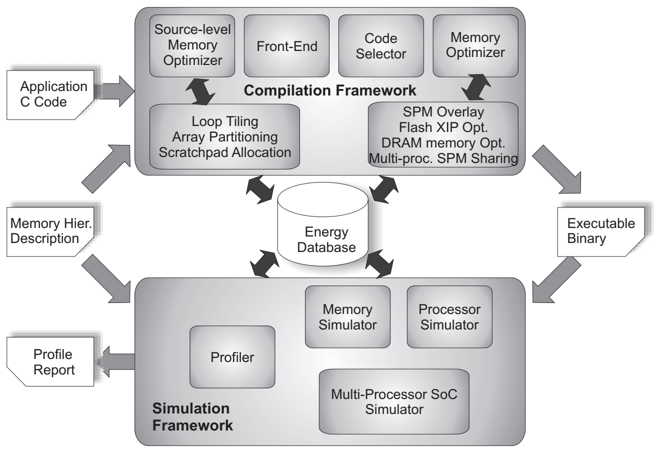

Fig. 1: Workflow of MACC Framework

memory hierarchy description and the energy database. The compilation framework, depending upon the memory hierarchy, optimizes the application code and generates the executable binary. This binary is then simulated and profiled by the simulation framework and the system statistics are reported. These statistics are used to evaluate the system and the optimizations. The main advantages of the MACC framework are that it includes the only energy optimizing compilation framework known to the research community and a highly configurable processor independent memory hierarchy simulator. We start by explaining the compilation framework of the MACC framework.

2 Compilation Framework

The compilation framework shown in the upper half of Figure 1 provides the user with a rich set of memory optimizations. It optimizes the energy consumption of the system by efficiently utilizing the memory hierarchy. Besides the application source file, it requires the memory hierarchy description file, an XML representation of the memory hierarchy, as input. The memory hierarchy can also be designed by the user with the help of a GUI (*cf.* Section 4.1 for additional details). In addition, the compilation framework has access to the energy database which contains the instruction-level energy model of the processor as well as the energy and timing models of various memories. The compilation framework supports the ARM7 processor and includes numerous memory optimizations supporting memories *viz.*, SRAM/Scratchpad, Cache, Loop Cache, DRAM and Flash. The framework is being extended to support an ARM based Multi-processor SoC [2]. The compilation framework is divided into the following tools:

1. *Source-Level Memory Optimizers:* performs memory related optimization at the source code level.
2. *Front-End:* converts the application source code into an intermediate representation (IR) and performs several traditional optimizations (*e.g.* dead code elimination)
3. *Code Selector:* converts the IR into the assembly code of the application and performs several low-level optimizations (*e.g.* peephole optimization).
4. *Memory Optimizer:* performs most of the memory optimizations in cooperation with the linker to generate the executable binary of the application.

The memory aware compilation framework started as a research endeavor and has matured into a fairly stable framework. It is based upon ICD-C [4] and Lance [5] compilation frameworks. It supports all ANSI-C data types and can compile and optimize applications from Mediabench, MiBench, DSPStone and UTDSP benchmark suites. The framework includes the a large number of standard compiler optimization, some of them are: Dead Code Elimination (DCE),Common Subexpression Elimination (CSE), Copy Propagation (CP), Constant Folding (CF), etc.

The objective of the compilation framework is to generate energy efficient code. Therefore, the compiler generates 16-bit Thumb mode code for ARM processor. Authors in [6] demonstrated that the 16-bit Thumb assembly code is 30% more energy efficient than the usual 32-bit mode code. The framework provides a rich selection of memory aware energy optimizations. All optimizations are performed on the set of memory objects (*MO*). A memory object is a part of the application program (*e.g.* variable, array tile, function, basic block etc.) whose mapping onto the memory hierarchy enables various memory optimizations. Subsections 2.1 and 2.2 describe the source-level and the backend memory optimizations, respectively.

2.1 Source-Level Memory Optimizer

The source-level memory optimizer is the highest level optimization phase of the compilation framework. It includes several optimizations also present in the backend memory optimizer, albeit at a coarser granularity level. The main benefit of the source-level optimizer is its inherent retargetability. The optimized application can be compiled for any other processor resulting in similar gains. Besides scratchpad allocation strategies, two new optimizations included in the optimizer are *array partitioning* and *array tiling*. The array partitioning approach partitions the large arrays found applications into two smaller arrays such that one of them allocated to the scratchpad. The array tiling approach generates *tiles* (sub-arrays of equal size) and overlays them onto the scratchpad.

The source-level memory optimizer is based primarily on the ICD-C framework. It features a lossless object oriented intermediate representation for C programs. Most important for our optimizer is the capability to write out the IR to a file conforming to the C standard. The set of memory objects considered by the optimizations consists of global variables and functions. The memory optimizer includes the following optimizations:

1. Non-Overlayed Scratchpad Allocation [7]
2. Scratchpad Overlay (with support for DMA) [8]
3. Array Partitioning [9]
4. Array Tiling

The first two approaches are also present in the backend memory optimizer and will be presented in detail in the following section. The focus of *array partitioning* approach are applications containing large arrays which are accessed through irregular index functions. These arrays cannot be allocated onto small and energy efficient scratchpad memories. Consequently, the *array partitioning* approach divides the large array into two smaller partitions such that the allocation of one of the two partitions to the scratchpad memory is guaranteed. Additionally, the application source code is modified such that the irregular index functions correctly access the two array partitions. If the application under consideration contains only arrays with affine index functions, the *array tiling* optimization can be used. It generates several equal sized partitions or tiles of the arrays. These tiles are then swapped in and out of the scratchpad memory at runtime, based upon their live-ranges.

2.2 Backend Memory Optimizer

The backend memory optimizer includes numerous optimizations for various memories. Unlike most of the current approaches, the optimizations consider both the data and instructions for optimization. The memory optimizer includes optimizations for scratchpad, instruction cache, loop-cache, DRAM and Flash memory. Scratchpad allocation approaches reduces the energy dissipation of the system through the improved utilization of the scratchpad memory. Trace generation based instruction cache optimization is used to improve the spatial locality of the application.

The backend memory optimizer is the last optimization step of the compilation framework. It works in conjunction with the assembler and the linker and produces the optimized executable by mapping all the memory objects to the assigned memories. The optimizations are performed at a finer level than in the source level memory optimizer as the set of memory objects is composed of global variables, basic blocks and the stack. Some of the important memory optimizations are enumerated below:

1. Non-Overlayed Scratchpad Allocation [7]
2. Partitioned Scratchpad Allocation [10]
3. Scratchpad Overlay (with DMA support) [8]
4. Instruction Cache Optimization [11]
5. Pre-loaded loop cache Optimization [11]
6. DRAM memory optimization [12]
7. XIP Flash Memory Optimization [12]

Non-overlayed scratchpad allocation [7] maps the best set of memory objects onto the scratchpad memory which remain allocated onto the scratchpad for the entire execution time of the application. Each memory object *mo* has two parameters: (a) $E_{profit}(mo)$ quantifies the energy reduction that can be achieved by assigning the memory object on the scratchpad memory and (b) $size(mo)$ returns the size of the memory object. The best set of memory objects is chosen such that the total energy benefit is maximized and the aggregate size of the memory objects in the best set is less than the scratchpad size. The allocation problem can be formulated as the following:

$$\text{Maximize:} \qquad E_{profit}^{Total} = \sum_{mo} E_{profit}(mo) * x_{mo}$$
$$x_{mo} * size(mo) \leq ScratchpadSize \;\; \forall mo \in MO \;\; x_{mo} \in \{0,1\}$$

It can be easily seen that the non-overlayed scratchpad allocation is the well-known knapsack problem. If a number of partitioned scratchpad memories are being used instead of one single scratchpad, additional savings are possible since smaller memories are faster and consume less energy per access. The above equations have to be reformulated to take into account the increased freedom of allocating the memory objects to a number of scratchpad memory partitions. Additionally, the leakage energy dissipation of a large number of scratchpad memories was also studied in our experiments to let the compiler choose those memory partitions that are most profitable in order to minimize the overall system energy dissipation.

The *scratchpad overlay* optimization uses the fact that a memory object is not required by the application for its entire execution time. In other words, memory objects also have *live-ranges*. Therefore, memory objects with non-conflicting live-ranges can be assigned to the same location onto the scratchpad. The approach also takes into account the spilling of memory objects to the main memory in order to maximize the total energy reduction. The overlay approach [8] was found be similar to the *global register allocation* approach and both optimal and near-optimal solutions were presented

The memory optimizations [11] for a cache based architecture are also present in the backend memory optimizer. The included approaches improve the spatial locality of the application code by generating *traces*. Additionally, scratchpad and loop cache allocation approaches are also included. These approaches utilize a scratchpad or a loop-cache as an instruction buffer and map the instruction sequences to minimize the number of cache misses and the total energy dissipation of the system.

If a DRAM main memory is used in the system, considering per-access costs for memory accesses is insufficient due to the state-dependent behavior of a dynamic RAM. A corresponding energy model is integrated into our evaluation framework. It also supports the power management features commonly found in DRAM chips today. This can be exploited in an optimization that allocates memory objects to a scratchpad memory in order to maximize the time that the main DRAM memory can be kept in the power down state.

Most embedded systems today carry Flash memories to permanently store configuration information or the application's binary code. In contrast to the prevailing "Store-and-Download" approach, where code and data is first copied and then accessed from the faster main memory, the "eXecute-In-Place" (XIP) feature allows the memory objects to be accessed directly from the Flash memory. The corresponding optimization determines a trade-off between the copy costs and the slower Flash memory access times. The main benefit of this optimization is that it significantly reduces the main memory requirements of the system.

3 Energy Database

A fine-grained, accurate and exhaustive energy database is an essential component of the entire MACC framework. An evaluation board (AT91EB01) [13] featuring an ARM7 processor was chosen to generate an accurate energy database. Current measurements were performed on the board to determine an instruction level energy model for the ARM7 processor. A measurement based energy model was also determined for the SRAM main memory of the board. The energy model for the processor and the memory was found to be 98% accurate [14]. Behavioral energy models for memories have also been found to be very accurate. Consequently, we used behavioral models for the memories whose current consumption could not be measured. The accurate energy model for the MPSoC, accounting for the processors, memories and the interconnect, was obtained from ST Microelectronics.

4 Simulation Framework

The presented simulation framework allows simulation of a system consisting of an ARM7 processor attached to a customizable memory hierarchy. The processor simulator provided by ARM Ltd. is used to generate the instruction trace. The instruction trace

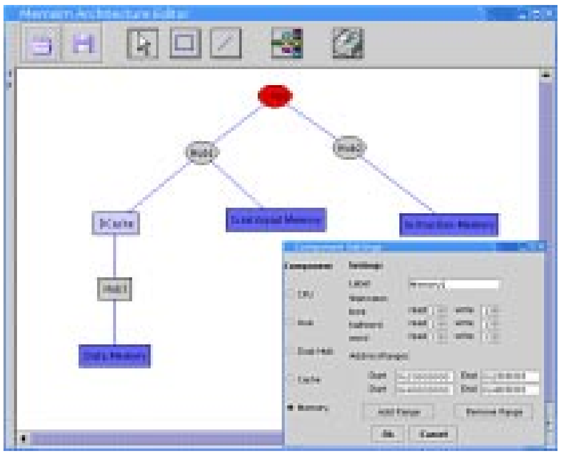

Fig. 2: Example MEMSIM memory hierarchy configuration

is fed into the memory simulator which simulates the specified memory hierarchy. The profiler accesses the instruction trace, the memory simulator and the energy database to compute the system statistics (*e.g.* execution time in CPU cycles and energy dissipated by the processor and the memory hierarchy). In addition, it computes the application statistics (*e.g.* number and type of accesses to each global variable). Currently, we are integrating the ARM-based MPSoC [2] into the simulation framework.

4.1 Memory Simulator

In order to efficiently simulate different memory hierarchy configurations, a flexible memory hierarchy simulator (MEMSIM) was developed. Memory regions with different access characteristics, a number of different cache parameters, loop caches and scratchpad memories are currently supported. MEMSIM reads the XML description of the memory hierarchy and a memory access trace of a program. The development of MEMSIM enabled us to overcome limitations of pure instruction set simulation and also of the currently available cache and memory simulation frameworks. The technical requirements for MEMSIM, e.g. cycle true simulation, flexibility and configurability were achieved by using object oriented design principles in the design and implementation phase. All components of the memory hierarchy are derived from one uniform base class, which enables the easy and seamless integration of memory models into the simulation framework.

While a variety of cache simulators are available, none of them seemed suitable for an in-depth exploration of the design space of a memory hierarchy. In addition, scratchpad memories, loop caches and DRAM memories should also be considered. This flexibility is missing in previously published memory simulation frameworks which tend to focus on one particular component of the memory hierarchy. Therefore, the development of a new memory simulator, called MEMSIM, was necessary. To avoid the high complexity of implementing a cycle-true instruction set simulator for a particular processor, MEMSIM runs as a post-pass to processor simulation. The sequence of executed instructions and memory accesses is fed into MEMSIM and the accesses to each memory is computed accordingly. By encapsulating the trace reader functions in classes of their own with a defined interface, it is possible to use a variety of available processor simulators by only adjusting the internal implementation of the trace reader functions.

All components of the simulated memory hierarchy are implemented as abstract components. All instantiated components inherit from this virtual base class and implement the functionality required to perform as a part of the memory hierarchy. To connect the different components to each other, the concept of so-called hubs is used. Using hubs, it is only necessary to consider the connection of each memory component to its neighboring hub, which in turn connects to other memory components. An example memory hierarchy is shown in Figure 2. A graphical user interface is provided so that the user can comfortably select the components that should be simulated in the memory hierarchy. The GUI generates a description of the memory hierarchy in the form of an XML file which is then processed by MEMSIM in order to instantiate the memory components, connect and simulate them.

4.2 MPSoC Simulator

The MPSoC simulation framework, presented in Figure 3, is a SystemC based cycle true simulator. It is capable of simulating a runtime configurable number of processing units, which are connected through a single bus to memories and I/O devices. The most common setup is to use a simulation of an ARM7 core for the processing unit, and an AMBA bus simulation for the interconnection. There are also other combinations of buses (i.e. STbus) and processing units available.

As shown in the figure, each ARM-based processing unit has its own private memory, which can be a unified cache or separate caches for data and instructions. A wide range of parameters may be configured, including the size, associativity and the number of wait states. Beside the cache, a scratchpad memory of configurable size can be attached to each processing unit. The recent development of the simulator targets the hardware requirements in streaming media applications, therefore offering “smart memories” which are basically scratchpad memories accompanied by DMA units.

The MPSoC simulator does not support a configurable multilevel memory hierarchy. The memory hierarchy consists of caches, scratchpads and the shared main memory. Currently, an effort is being made to integrate MEMSIM into the simulator. Finally, it provides a number of semaphores which may be used to synchronize inter-processor communications.

The simulator offers various reporting and tracing facilities. At the lowest level it may report waveform diagrams of the performed bus actions. It is further able to report memory access traces. This feature is most important for the integration into the MACC simulation framework. Beside the simple access based trace files, sophisticated statistic may be generated, including precise information about the amount of cycles spent in bus actions, processing and waitstates.

4.3 Profiler

The profiler uses trace file, memory hierarchy simulator and computes the access to each memory in the memory hierarchy. These accesses are then mapped to timing and energy models of the processor and the memories to compute the execution time and the

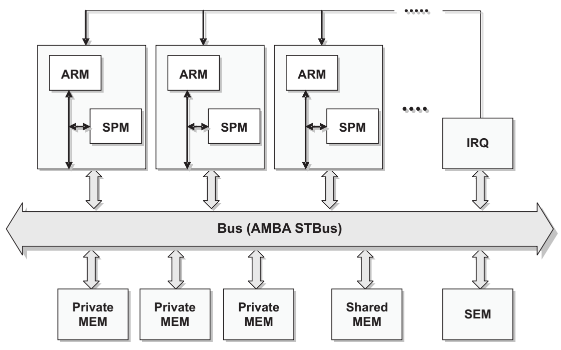

Fig. 3: Multi-process SoC (MPSoC) Simulator

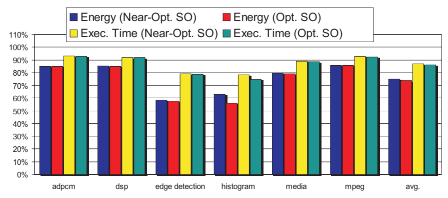

Fig. 4: SPM overlay vs. Non-Overlayed allocation

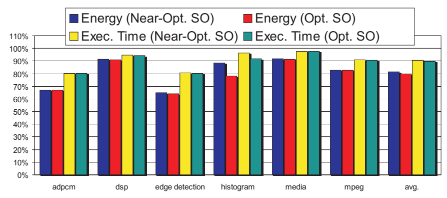

Fig. 5: SPM overlay vs. Cache based system

energy dissipation of entire system during the execution a given application. The profiler also contains a mapping of the memory objects present in the application source code and their addresses in the application executable. Therefore, the profiler is able to back-annotate each fetched or executed address within the system to the corresponding memory object. This enables the profiler to gain in-depth knowledge about the application and the system under simulation. This extensive information as a tabulated report file is then presented back to the users. Some of the contents of the report file are enumerated below:

1. Energy consumption, number of accesses and size of every function and basic block
2. Energy consumption, number of accesses and size of every variable
3. Execution order of the basic blocks
4. Energy Consumption and number of accesses to each memory
5. Energy Consumption of the processor
6. Number of executed instruction and execution time (CPU Cycles)

5 Experimental Results

In this section, we first compare the scratchpad overlay and the non-overlayed scratchpad allocation approaches. The values shown in Figure 4 are average values over varying scratchpad size in the range of 128 to 1024 bytes. According to the figure, applying

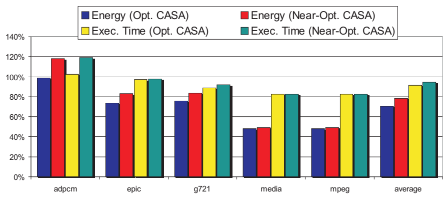

Fig. 6: Instruction Cache optimization vs. Preloaded loop cache optimization

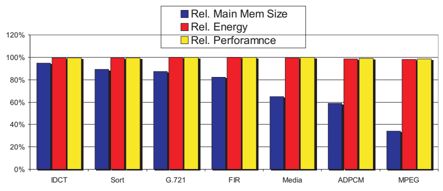

Fig. 7: Results for XIP Optimization

a more sophisticated allocation strategy, which takes into account the temporal distribution of memory object usages, results in a significant reduction in energy consumption and execution time. Average reductions of more than 40% and 20% in the energy dissipation and the execution time, respectively, of the *edge detection* benchmark are reported.

Furthermore a comparison of the scratchpad overlay memory optimization technique included in the MACC against a cache based system is presented in Figure 5. Similar to the previous figure, these are average values for each benchmark obtained by varying scratchpad and cache sizes in the range of 128 to 1024 bytes. In this case as well, the scratchpad overlay approach demonstrates energy savings of over 30% for the *adpcm* benchmark. Average performance improvement of about 20% is also reported for the same benchmark.

The memory optimization using the scratchpad as an instructions buffer achieves significant energy and runtime savings compared to the preload loop cache optimization. The memory optimizations reduces the number of conflict cache misses while the loop cache optimization buffers frequently executed instructions. In this particular case, we achieve energy savings of about 50% and execution time reductions of close to 20% for the *mpeg* benchmark. The results are illustrated in Figure 6.

Exploiting the reduced energy dissipation of a DRAM memory in the power down state leads to significant energy savings. By allocating memory objects to a scratchpad memory in such a way as to maximize the power down periods of the DRAM main memory, substantial energy savings of up to 80% compared to a system without scratchpad memory and no power down times for the used DRAM were achieved. Since both allocation results and obtainable savings are similar to the static scratchpad allocation scheme that uses energy per access as the cost function (in contrast to maximizing the power down time of the main memory), no specific results are given for the sake of brevity.

Figure 7 shows the obtained results when the Flash memory used in an embedded system is capable of executing code using the XIP technique. An intrapage access time of 20*ns* is assumed for the used Flash memory. The leftmost bar shows the percentual amount of DRAM main memory that is still required despite also using Flash memory to execute instructions: for the *mpeg* benchmark, 65% of the previously used DRAM main memory is not required when XIP is being used. The gains concerning energy and performance of XIP compared to an execution from the faster DRAM are marginal, as shown by the second and third bars in the figure. For the *mpeg* benchmark, both the energy dissipation and the number of executed cycles are reduced by a maximum of about 2%. This shows that the large savings with respect to the required main memory capacity do not incur any overhead concerning energy or performance for the considered setup. Taking into account that the amount of main memory is an important cost factor for embedded systems, the exploitation of XIP functionality should be considered during the design and optimization of embedded systems.

6 Conclusions & Future Work

In this paper, we presented the MACC framework, a coherent compilation and simulation framework for performing and evaluating memory aware energy optimizations. The framework features an energy optimizing compilation framework for the Uni- or Multi-process ARM SoCs and a highly configurable simulation framework. In addition to optimization of the application for a given memory hierarchy, the framework enables fast and efficient memory hierarchy design space exploration. In the future, we would like to extend the MACC framework for homogeneous and heterogeneous MPSoCs.

Additional information and a complete list of publications concerning MACC can be found at [3].

References

1. Wulf, W.A., McKee, S.A.: Hitting the Memory Wall: Implications of the Obvious. IEEE Computer Architecture News **23**(1) (1995)
2. Francesco, P., Marchal, P., Atienza, D., Benini, L., Catthoor, F., Mendias, M.: An Integrated Hardware/Software Approach for Run-Time Scratchpad Management. In: Proc. of DAC, San Deigo, CA, USA, DAC (2004)
3. Department of Computer Science XII, University of Dortmund: MACC: Memory Aware C Compilation Framework, (http://ls12.cs.uni-dortmund.de/research/macc/)
4. ICD: Informatik Centrum Dortmund e.V.: ICD-C Compiler Development Framework, (http://www.icd.de/es/icd-c/icd-c.html)
5. ICD: Informatik Centrum Dortmund e.V.: LANCE Retargetable C Compiler, (http://www.lancecompiler.com/)
6. Krishnaswamy, A., Gupta, R.: Profile Guided Selectino of ARM and THUMB Instructions. In: Proc. of LCTES'02/SCOPES'02, ACM (2002)
7. Steinke, S., Wehmeyer, L., Lee, B.S., Marwedel, P.: Assigning program and data objects to scratchpad for energy reduction. In: Proc. of DATE, Paris, France (2002)
8. Verma, M., Wehmeyer, L., Marwedel, P.: Dynamic Overlay of Scratchpad Memory for Energy Minimization. In: Proc. of CODES+ISSS, Stockholm, Sweden (2004)
9. Verma, M., Steinke, S., Marwedel, P.: Data Partitioning for Maximal Scratchpad Usage. In: Proc. of ASPDAC. (2003)
10. Wehmeyer, L., Helmig, U., Marwedel, P.: Compiler-optimized Usage of Partitioned Memories. In: Proceedings of the 3rd Workshop on Memory Performance Issues (WMPI2004), (ACM International Conference Prodeedings Series, ISBN: 1-59593-040)
11. Verma, M., Wehmeyer, L., Marwedel, P.: Cache-aware Scratchpad Allocation Algorihm. In: Proc. of DATE, Paris, France (2004)
12. Wehmeyer, L.: Fast, Efficient and Predictable Memory Accesses – Optimization algorithms for memory architecture aware compilation. Ph.D.-thesis, unpublished (2005)
13. ATMEL: Atmel Corporation, (http://www.atmel.com)
14. Steinke, S., Knauer, M., Wehmeyer, L., Marwedel, P.: An Accurate and Fine Grain Instruction-Level Energy Model Supporting Software Optimizations. In: Proc. of PATMOS, Yverdon-Les-Bains, Switzerland (2001)