Fast, predictable and low energy memory references through architecture-aware compilation[1]

Peter Marwedel[1], Lars Wehmeyer[1], Manish Verma[1], Stefan Steinke[2], and Urs Helmig[1]

[1] University of Dortmund, Germany
[2] Kostal GmbH & Co KG, Lüdenscheid, Germany

Abstract. The design of future high-performance embedded systems is hampered by two problems: First, the required hardware needs more energy than is available from batteries. Second, current cache-based approaches for bridging the increasing speed gap between processors and memories cannot guarantee predictable real-time behavior. A contribution to solving both problems is made in this paper which describes a comprehensive set of algorithms that can be applied at design time in order to maximally exploit scratch pad memories (SPMs). We show that both the energy consumption as well as the computed worst case execution time (WCET) can be reduced by up to to 80% and 48%, respectively, by establishing a strong link between the memory architecture and the compiler.

Keywords: Embedded system, compiler, energy efficiency, low power, WCET, scratch-pad, memory access

1 Introduction

The design of embedded systems is very much driven by applications. It is expected that future applications will require significantly more processing power, due to audio and video applications and high computational demands for channel coding [7]. As a result, more powerful processors have to be used in embedded systems. However, the electrical energy available in embedded systems (especially in portable systems) is strictly limited. This has been seen as the most important constraint in the design of future embedded systems [8]. A significant amount of research on low-power design techniques has been performed, but the 100 to 1000 fold improvement demanded by De Man [7] has not yet been achieved, making additional techniques necessary.

Increased processor speeds will also bring a problem to embedded systems which has so far mainly affected the design of PCs and mainframes: the speed gap between high end processors and memories is widening. While processor speeds are currently improving between 50 and 100% per year, the speed of memories is only increasing at 7% per year. Accessing main memory will soon cost as many cycles as a page miss did in the first computer using virtual memory [21].

For any given technology, access times as well as the energy required per memory access are a function of the memory size: The larger the memory, the larger the access

(1) This work has been partially supported by grants from the German research foundation (DFG) under contracts Ma 943/6-3 and Ma 943/8-2.

times and the energy consumed per access. Note that the energy consumption per access shown in fig. 1 differs by a factor of up to $\alpha = 24$, whereas the access time differs by a value of up to $\beta = 2.7$. The increasing energy consumption and access times for larger memories can be confirmed using the CACTI tool [4, 34]. In general, due to the increasing sizes of applications and the corresponding memory sizes, α and β can be expected to become even larger in the future. Due to these facts, it does make sense to map hot spots in applications to smaller memories instead of using just one large, homogeneous memory.

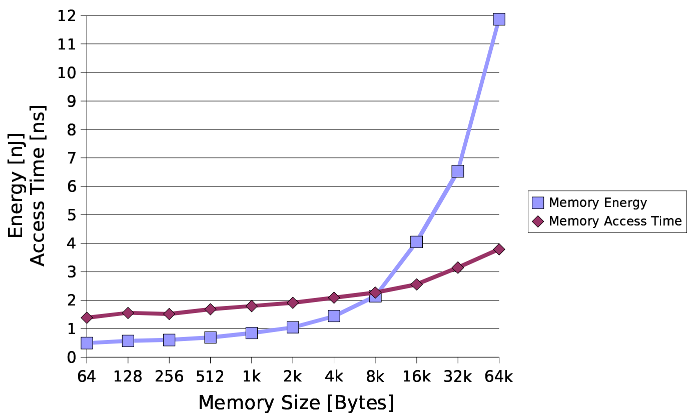

Fig. 1. Energy consumption and access time as a function of the memory size

Caches have been established as the key solution to ease the problem for PCs and mainframes. Unfortunately, currently available cache technologies have mostly been designed to improve the average case timing behavior, not the worst case timing behavior. Furthermore, a certain percentage of main memory references is still required and may be the limiting factor for future technologies. This potential future problem became known as the memory wall [35].

Many embedded systems are real-time systems. For such systems, it is necessary to guarantee meeting real-time constraints. Accordingly, worst case execution times (WCETs) must be derived in order to prove a real-time system to have required properties. In some cases, it is possible to prove that caches improve the worst case execution time. However, many caches contain features which make this difficult. For example, it is difficult to model conflicts between instruction and data references for unified caches. Also, the effect of some (e.g. random) replacement policies is difficult to predict at design time. Puschner and Burns [25] provide an overview of the state of the art in WCET prediction. Systems which allow the derivation of WCETs that are close to the actual execution times are said to have a *predictable* timing behavior. The design of predictable high-performance embedded systems is a major challenge of the future. This was stated by Xu and Parnas [36] as follows:

For satisfying timing constraints in hard-real-time systems, predictability of the system's behavior is the most important concern; pre-run-time scheduling is often the only practical means of providing predictability in a complex system.

Accordingly, predictable real-time operating systems sometimes follow the time-triggered approach suggested by Kopetz [15] instead of the usual event-triggered approach. The key idea of this approach is to remove the unpredictability that is caused by many of the schedulers in operating systems.

One of the ideas of this paper is to remove the unpredictable cache access times by using scratch pad memories (SPMs) and by letting compilers compute memory access times at design time. Scratch pad memories are small memories that are mapped into the address space of the hardware. They are accessed whenever addresses in the corresponding address range are used. Tags (as needed for caches) are not required. Accordingly, the energy per access to a SPM is lower than the energy for an access to a cache. Fig. 2 compares the energies of accesses to different types of caches against an SPM. It is obvious that the SPM consumes the least amount of energy per access, even compared to the simple direct mapped cache organization. Direct mapped caches are not very well suited for caching data since their simplicity tends to provoke cache thrashing.

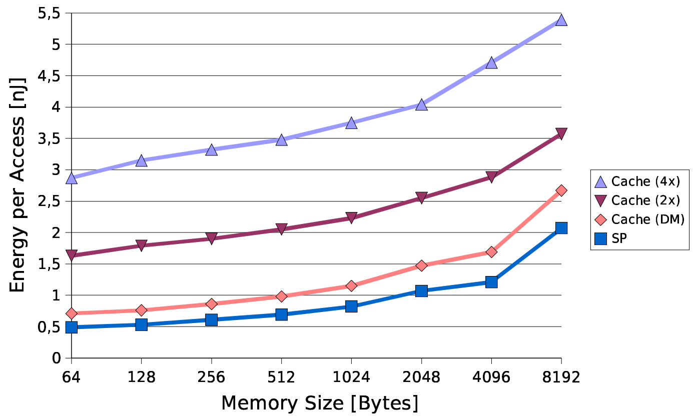

Fig. 2. Comparison between the energy consumption of caches and SPMs

SPMs (also called 'Tightly Coupled Memories' or TCMs) are available with some processor cores [1] and they are being used in industry. However, there is no comprehensive tool support which maximizes the benefits that can be achieved with SPMs. This paper describes a research effort that aims at providing a comprehensive set of tools required to exploit the presence of SPMs.

The remainder of this paper is structured as follows: a description of related work is provided in section 2. Section 3 describes the different contributions that were made by our group, putting previously published work into perspective and also providing new results (especially on predictability issues). Results are described together with each contribution. The paper closes with a description of future work and a conclusion.

2 Related work

Most of the previous work on SPM usage is restricted to storing data elements, such as arrays accessed in innermost loops, in the SPM. In [24], an architecture containing both cache and scratch pad is assumed. Arrays that are too large to fit in the SPM are therefore kept in main memory and are accessed through the data cache. A generalized memory hierarchy where each level has a cache and a scratch pad memory is also considered. The authors of [6] use the SPM (or, as it is called in their publication, compiler controlled memory) as a cheap alternative to spilling register values to main memory. An optimal algorithm to statically distribute data among several memory partitions based on profiling of applications and solving a binary linear equation system is presented in [3]. The possibility of distributing the stack to different memories is also investigated. The authors of [12] use so-called Presburger formulas to determine which set of data should be kept in the SPM. In contrast to earlier work, they not only consider a static allocation of elements to the different levels of the memory hierarchy, but also consider copying data elements from e.g. main memory to the SPM at runtime. This is also true for [33], which uses a combination of well-known loop optimizations with the consideration of limited scratch pad capacity in order to minimize data traffic between the SPM and the main memory during runtime.

An approach to store both data and instructions on the SPM was first presented in [29]. This work was extended to also consider copying instructions at runtime [28]. Using graph coloring (as in register allocation) [23] to solve this problem is possible, but not straightforward, since the performance of graph coloring is poor when assigning many values to a small storage location. Also, the scope of register allocation usually does not exceed the function level, whereas the scratch pad allocation problem has to consider global data and function-call relations. In order to allow a maximum number of instructions and data objects to be placed on the SPM, large arrays are partitioned in [31]. The arrays to be partitioned along with their respective splitting points are determined and the most beneficial objects are moved to the SPM. Detailed information about these approaches will be presented in section III.

Several approaches for WCET estimation frameworks have been proposed. They range from straightforward models to nearly completely automated estimation frameworks taking into account information about the software as well as architectural hardware features.

In order to compute the WCET of an application, the programmer usually needs to provide information about the number of loop iterations and feasible control paths. Information about the execution time of each assembly instruction is also required. Using a compiler, the high level information has to be transformed down to the assembly code level. Optimizing compilers can complicate this mapping. Some approaches have dealt with this problem [13], while others only consider the assembly level in their analysis [5, 22].

More sophisticated WCET analysis tools take into account detailed information concerning the hardware, including pipelines [20] and caches [10]. In general, it can be stated that modern complex processors are difficult to handle with respect to WCET computation. This is partly due to the fact that the market demands an increase in average (not worst case) performance. On the other hand, hardware vendors are trying to protect their intellectual property by not providing detailed information about the hardware architecture and exact timing to customers [14].

Pipelines are being used to speed up the execution by interleaving e.g. instruction prefetch and execution phases in two pipeline stages. However, WCET computation has to consider the possibility of e.g. branch penalties. If the currently executed instruction leads to a change of control flow, then the already fetched instruction has to be discarded and a new instruction fetch access to memory is required. Considering this and other effects in WCET computation is not trivial [37].

Other architectural features often used in processors are instruction and/or data caches. In order to avoid a loose WCET bound by assuming all cache accesses to be misses, more complicated analyses have to be included in the WCET framework. Information concerning conflicting objects and the cache organization needs to be considered to determine a more realistic, yet safe upper bound for the execution time of an application [18]. Methods for instruction caches have considered different cache organizations, e.g. direct mapped and set associative [19]. Data caches require even more complex analyses since the data address referenced in one assembly instruction usually changes over time (e.g. by using register-offset addressing). This makes the hit/miss ratio of data caches hard to determine. Some of these issues have also been tackled in [19].

In this context, scratch pad memories [24] not only reduce execution times, but also simplify WCET computation. Since no misses can occur, all accesses to the SPM can be treated like regular memory accesses, albeit with a considerable reduction in wait cycles due to the short access times of small scratch pad memories. This improved performance has a direct impact on the execution time as well as the WCET estimate, which comes at no extra cost during analysis. The fact that engineers of safety-critical real-time systems will have to take care to assure a predictable and robust timing behavior in the future [14] makes the integration of SPMs into such embedded systems a very promising approach.

3 Compiler support for scratch pads

3.1 The knapsack model

In order to simplify the discussion, we will initially consider the case of a single SPM and a single main memory. We will try to identify those locations of a program that should be allocated to the SPM.

We can model instructions and data in a consistent manner if we define memory segments to be contiguous blocks of memory locations holding either variables or instructions. In the case of variables, each variable forms its own segment. In the case of instructions, we first consider only complete functions and the corresponding code blocks as segments.

Now, for each of the segments, we can compute the energy gain E_{g_i} resulting from the allocation of segment i to the SPM. Let us assume that the energy required for an access to the main memory is E_m and that the energy for an access to the SPM is E_s. Furthermore, assume that we know from static analysis or profiling that we have n_i memory accesses for memory segment i. Then, the gain resulting from the allocation of i to the scratch pad is $E_{g_i} = (E_m - E_s) * n_i$. Assume that the size of memory segment i is s_i and that the size of the SPM is K. In order to compute the set of segments mapped to the SPM, we introduce decision variables x_i for each segment i, with $x_i = 1$ if segment i is mapped to the SPM, and $x_i = 0$ otherwise. Then, minimization of the energy consumption can be expressed as the problem of maximizing the gain:

$$G = \sum_i x_i * E_{g_i} \tag{1}$$

while respecting the size constraint

$$\sum_i x_i * s_i \leq K \tag{2}$$

This problem is a special instance of the knapsack problem [26]. The knapsack model can also be used to minimize the execution time. In the above model, we just have to replace the energy gain by the corresponding execution time gain T_{g_i}. Let T_m be the number of wait cycles for accesses to the main memory and let T_s be the number of wait cycles for accesses to the SPM. Then, T_{g_i} is equal to $(T_m - T_s) * n_i$.

There are many algorithms for solving knapsack problems. In our work, we have mapped the knapsack problem to an integer programming (IP) problem. Equations 1 and 2 are indeed also a special case of an IP problem. A key advantage of IP models is that they can be easily extended to more general cases, which will be shown in the following section. The number of IP variables corresponds to the number of functions and variables. These numbers are small enough to avoid any run-time problems for IP solvers.

Fig. 3 shows the energy savings and the performance gain that can be obtained using various scratch pad sizes for the *multi_sort* benchmark which includes some frequently-used sorting algorithms in one application(1). It can be observed that the algorithm is not capable of taking advantage of a very small 64 byte SPM, since all of the functions and data elements (e.g. arrays) in the benchmark are too large to fit.

3.2 Extensions: Migration of basic blocks, sets of adjacent basic blocks, the run-time stack and fractions of arrays

The model described so far can be extended by also considering basic blocks and sets of adjacent basic blocks as segments. In this case, the optimization problem becomes slightly more complex: if individual basic blocks are moved to the SPM, additional jumps have to be generated for branching into and out of the address space of the SPM. The additional cost of these jumps must be considered in the cost model [29]. However,

(1) This benchmark was used as a running example throughout this paper

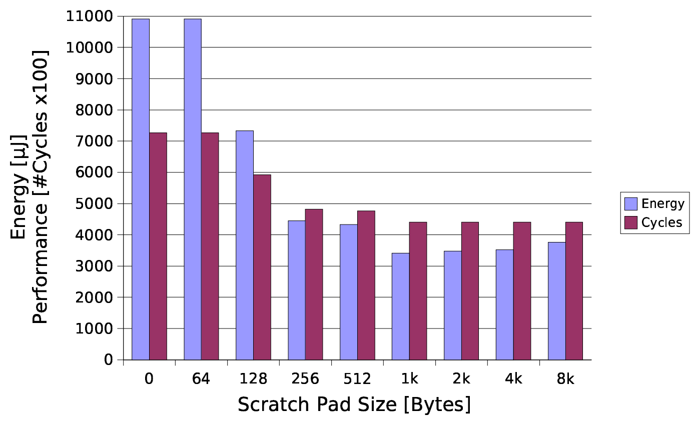

Fig. 3. Energy consumption and performance vs. the size of the scratch pad

no additional jumps are required between two adjacent blocks that are both moved to the SPM.

The run-time stack can contain local variables, function parameters, the current function's return address as well as spilled register values. It can also be migrated to the SPM, provided a safe upper bound on its maximum size can be derived. Since access to the stack is realized through stack-pointer relative addressing, moving the stack area to the SPM only requires setting the stack pointer register to an SPM address [27]. Keeping only parts of the stack in the SPM (i.e. manually modifying the stack pointer at runtime) is also possible.

As an example, we have used the AT91EB01 evaluation board by ATMEL Corp. [2] containing an ARM7TDMI processor and an onchip SPM. For this board, the energy required for a 32-bit load instruction of the THUMB instruction set can be reduced by a factor of about $\alpha = 7$, if both instructions and data are stored in an onchip SPM [30]. This means that ideally, the energy consumption of an application could be reduced to about 14% of the energy required for an architecture without SPM. In practice, reductions to about 20% have been observed for the algorithms described so far. This means that it is possible to get close to the optimum energy reduction. The algorithms described below aim at getting closer to the optimum and to do this for a larger set of applications.

Fig. 4 shows the gain obtained by allocating not only functions and data, but also considering basic blocks, sets of adjacent basic blocks and the stack to the SPM. Even for the smallest scratch pad size of 64 bytes, a 30% reduction in energy can be observed, in contrast to the case where only functions were being moved. Especially for small SPMs, the granularity of considered objects does make a big difference.

Only being able to move arrays as a whole into the SPM is a restriction that can hamper maximum utilization of the SPM. It is therefore desirable to also move as much as possible of an array into the SPM. An algorithm for the required partitioning of arrays is presented in [31]. The algorithm first chooses a candidate array A, then determines whether it is worthwhile to partition A. At the same time, all possible splitting points for

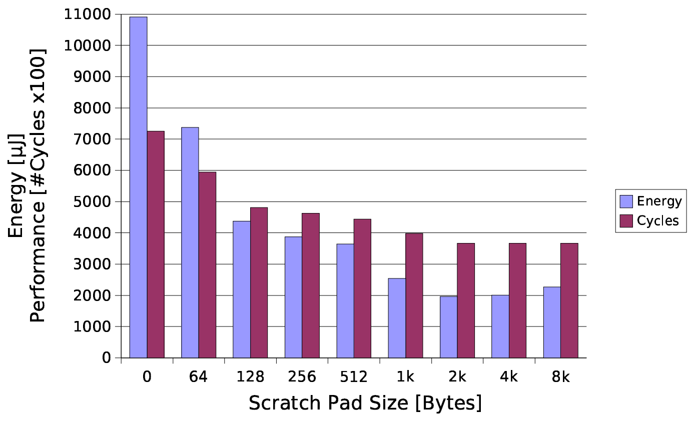

Fig. 4. Performance in number of cycles vs. the size of the scratch pad

array *A* are considered. If the array is to be split, then the original program is modified in such a way that one of the array partitions (along with the selected basic blocks) is moved to the SPM and accesses to the split array are redirected to one of the two array partitions, depending on the access index. This of course adds some overhead to the code. Despite this overhead, improvements in energy consumption of up to 17% with an average of 10% compared to the approach described in section III A are reported. In order to reduce the overhead and further improve performance, post-pass high-level optimizations were performed which help normalize the control flow when split arrays are accessed from within nested loops.

For the model considered so far, there is no copying of segments in and out of the SPM at run-time. We call this the static case.

3.3 Worst case execution times and the scratch pad

In addition to providing fast and low energy memory references, SPMs also improve the worst case execution time. This can be demonstrated for the bubble-sort program that has been used in the literature on worst case execution time bounds [13]:

```
#define N_EL 10
int arr[] = { ... }
int main(void){
 int x;  int i, j, temp;
 for (i=N_EL; i>1; i--) {
   for (j=2; j<=i; j++)  {
     if (arr[j-1] < arr[j]) {
       temp=arr[j-1]; arr[j-1]=arr[j];
       arr[j]=temp;
 }}} return(0); }
```

The worst case input pattern can be easily determined for this example: it is an input array sorted in ascending order whereas the resulting array is to be sorted in descending order. The method for computing WCETs is that of Li et al. [17]. We computed the corresponding WCETs for an ARM7TDMI using three memory architectures:

1. an architecture with just a main memory,
2. an architecture with cache and main memories and
3. an architecture with an SPM and a main memory.

For architecture 1, we are using the CPU cycles from the ARM documentation. For the memory, we assume 2 wait-states (note that future processors are likely to have a much higher number of wait-states). The resulting number of cycles is shown in table 1.

Table 1. Cycles for architecture 1

Instruction	CPU	IF	DF	Total cycles
LDR	3	2	2	7
STR	2	2	2	6
arithm./log.	1	2	0	3
Pipeline stall overhead: 6 cycles				

The resulting WCET is 5,197 cycles, whereas the actual execution time in simulations is 4,676 cycles. The difference can be attributed to the pessimistic assumption that pipeline stalls occur at every basic block entry.

For architecture 2, we are using a 64 byte unified cache. The corresponding number of cycles is shown in table 2. Since instruction and data references may interfere, and since ARM cores also supports caches with a random replacement policy, we assume misses on all fetches. The time required to fill a cache line is assumed to be 12 cycles for 16 bit THUMB instructions and 6 cycles for 32 bit data.

Table 2. Cycles for architecture 2

Instruction	CPU	IF	DF	Total cycles
LDR	3	12	6	21
STR	2	12	3	17
arithm./log.	1	12	0	13
Pipeline stall overhead: 6 cycles				

The resulting WCET is 19,737 cycles, whereas the actual execution time in simulations is 4,257 cycles for the actual run and 16,881 cycles if all misses are assumed. The relatively poor performance of the cache is a result of the small cache size which results in a large number of conflict misses, the large penalty for cache line fills and the

Table 3. Cycles for architecture 3

Instruction	CPU	IF	DF	Total cycles
LDR	3	0	2	5
STR	2	0	2	4
arithm./log.	1	0	0	1
Pipeline stall overhead: 2 cycles				

fact that no split cache is used, which leads to additional misses due to the interference of data and instructions.

For architecture 3, we are using a 64 byte SPM with no wait cycles and the static memory allocation technique described above. The corresponding number of cycles is shown in table 3. We assume all instructions to be allocated to the SPM, whereas the array remains in the main memory.

The resulting WCET is 2,688 cycles, whereas the actual execution time in simulations is 2,292 cycles. Fig. 5 shows the resulting number of cycles in context. A decrease in the computed WCET of 48% compared to the system with only main memory can be established. Using a cache, the computed WCET goes up significantly. This loose estimate can only be improved by further cache-related analyses, an effort that is not required if SPMs are used.

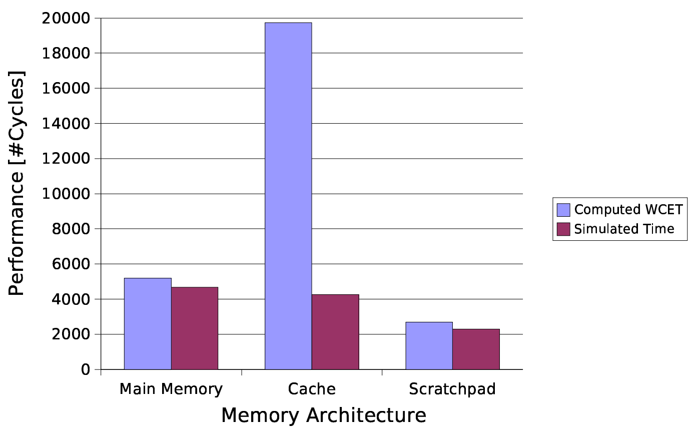

Fig. 5. Results for WCET versus Simulation

It can be expected that the benefits of the SPM become larger as the speed gap between processor and memory widens.

3.4 Combining scratch pads and caches

In the previously discussed approaches, SPMs are used to replace the more popular caches. In many modern processors, both caches and SPMs are available. If this is the case, then the algorithm presented above may not yield acceptable results since it does not consider the behavior of the cache. This is solved by the "Cache Aware Scratch pad Algorithm" CASA [32]. It assumes a Harvard architecture with the SPM at the same level as the instruction cache. The instruction cache behavior is modeled using a conflict graph, where two nodes (corresponding to memory objects) are connected by an edge whenever they might be placed in the same cache line, i.e. if the two memory objects might result in a conflict in the cache. Nodes and edges are weighted with the number of instruction fetches and the number of cache misses, respectively. These values are obtained using profiling. The energy model considers the cache hit energy to be much lower than the cache miss energy. By modeling the dependencies in the conflict graph as an integer programming (IP) problem, the compiler can decide which objects actually contribute most to the application's energy consumption, including the effect of cache misses caused by conflicting memory objects. After solving the IP problem, the selected memory objects are copied (not moved!) to the SPM, leaving holes in the main memory address space. This is necessary in order to avoid the main memory addresses to be changed by the linker, which would invalidate the cache behavior analysis results. Using this new algorithm, improvements of 8 – 29% in instruction memory energy compared to the previously mentioned techniques were obtained. Comparing the approach to the similar loop cache architecture [16, 9], average savings of 20 – 44% were achieved with respect to the energy consumption of the instruction memory subsystem.

3.5 Compiler-controlled block copying

In addition to considering only a static placement of memory objects onto the SPM, it is also possible to dynamically copy parts of the program from the main memory to the SPM [28]. The compiler has to insert program code to do the actual copying at runtime. The advantage of this approach compared to the static methods previously explained is that for a large application, the SPM may not be large enough to hold all the hotspots. The disadvantage is that the processor-driven copying from main memory to the SPM is slower and more energy consuming than a cache line fill. This is compensated by the better choice of copied memory objects: Only beneficial memory objects are ever copied to the SPM. This is in clear contrast to a cache which stores all objects, even those that are only accessed once.

The compiler's task consists of finding the objects to be copied to the SPM (in our example, we are only considering instructions) as well as the position of the copy functions within the program. A copy function can only lead to an energy saving if the copied instructions are executed more frequently than the copy function itself. Copy functions are therefore only considered at loop entries in the program. In this way, the instructions within the loop can be copied to the SPM once, and they will be executed many times within the loop. After having determined the possible copy function locations in the program for each basic block, the energy for copying memory objects using each possible copy function location is calculated. This copy cost has to be subtracted from the energy gain that is achieved by moving memory objects to the SPM. The resulting equations are again formulated as an IP problem which is solved using a commercial IP solver [11].

Fig. 6 compares the results of the dynamic allocation technique with a 4 way set-associative cache commonly found in the processors of the ARM7 family.

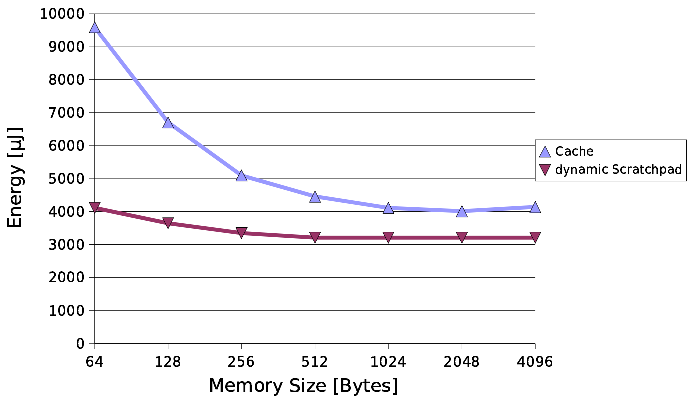

Fig. 6. Cache vs. dynamic scratch pad allocation [28]

Average energy savings of 29.9% and performance improvements of 25.2% were determined for this approach compared to a cache of the same capacity as the SPM. Compared to the static approach described above, energy was reduced by up to 38% for one benchmark [28].

3.6 Hardware-support for block copying

Inserting copy functions into the code at compile time as described in the previous paragraph leads to a substantial overhead, especially concerning the code size. For the benchmarks considered in [28], the dynamic approach increased the code size by at least 50% compared to the static approach. In order to prevent this, a functional unit can be used that is capable of copying instructions from main memory to the SPM, similar to a DMA unit. The compiler only needs to insert code to activate this unit by writing its memory mapped registers with the source and target addresses as well as the number of instructions to be copied. Once triggered, the unit will copy the instructions, whereas the processor can be put in a low power mode to preserve energy.

The DMA unit was modeled in VHDL, simulated and synthesized. Results indicate that the size of the additional unit only makes up 4% of the area of an ATM7TDMI processor using the same feature size. In addition, copy functions are usually executed infrequently and the unit can be put to sleep when it is unused. Code size reductions of up to 23% for a 256 byte SPM were determined using the DMA unit instead of the dynamic approach that uses processor instructions for copying.

3.7 Multiple scratch pads

Due to the characteristics of memories, energy and access time savings can also be expected from using multiple SPMs, as shown in fig. 7.

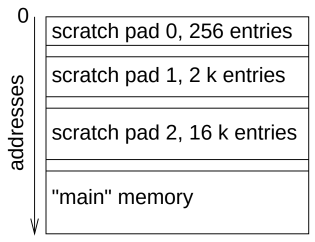
0
addresses
scratch pad 0, 256 entries
scratch pad 1, 2 k entries
scratch pad 2, 16 k entries
"main" memory

Fig. 7. Using multiple scratch pads

Let E_j be the energy per access to memory j, let K_j be its size and let n_i be the number of accesses to segment i. Let $x_{j,i}$ denote the mapping of memory segments to memories, with $x_{j,i} = 1$ if segment i is mapped to memory j, and $x_{j,i} = 0$ otherwise. Note that memory segments are mapped to only one memory. The corresponding optimization problem for the minimization of the energy has the following form:

Minimize

$$C = \sum_j E_j * \sum_i x_{j,i} * n_i \tag{3}$$

Subject to the constraints

$$\forall\ j : \sum_i x_{j,i} * s_i \leq K_j \tag{4}$$

$$\forall\ i : \sum_j x_{j,i} = 1 \tag{5}$$

Note that the problem is formulated as a minimization problem of the total energy now since there is no “reference memory” that could be used to express the “gain” achieved by moving an object to a different memory. Also, we are no longer using the knapsack formulation. In order to model leakage currents, additional cost factors representing idle memories can be added. This would enable the compiler to select a subset of the initial number of memories that yield optimal results with respect to energy consumption.

Fig. 8 shows a setup where partitioning one 256 byte SPM into two 128 byte scratch pads is beneficial, whereas further memory splitting leads to an increase in overall energy. This may either be due to the required additional jumps, or the 64 byte SPMs are too small to hold the most promising memory objects.

4 Future work

Future work will also consider the use of SPMs in a multi-process context. Space in the SPMs may be shared among multiple processes. Furthermore, the case of multiple SPMs may be extended to also include dynamic mapping of all kinds of memory segments to multiple SPMs. Also, more advanced methods for design space exploration of scratch pad architectures are needed. WCET computation needs to be done for larger

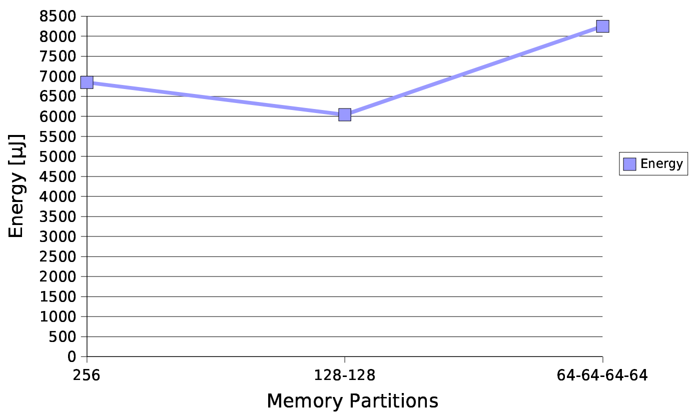

Fig. 8. Energy for using multiple SPMs

examples, and parts of the analysis should be automated and integrated into the compiler.

5 Conclusion

Achieving high-speed, low-energy memory accesses with predictable access times is one of the important problems in the design of embedded systems. This problem can be expected to become more severe as the speed gap between processors and memories widens and the memory sizes of applications increase. Scratch pad memories can potentially ease the problems. This paper gives a comprehensive overview over a set of approaches for exploiting the presence of scratch pad memories in compilers. In the case of an ATMEL evaluation board, energy savings of up to about 80% can be achieved. Run-times can be improved by about 50%. The computed WCET can be reduced by 48% when considering a scratch pad memory instead of main memory.

References

1. ARM Ltd. ARM946E-S: Embedded core with flexible cached memory system & DSP instruction set extensions. http://www.arm.com/armtech/ARM946E_S?OpenDocument.
2. Atmel. Atmel Corporation Homepage. http://www.atmel.com, 2003.
3. O. Avissar, R. Barua, and D. Stewart. An Optimal Memory Allocation Scheme for Scratch-Pad-Based Embedded Systems. *ACM Transactions on Embedded Computing Systems*, 1(1):6–26, November 2002.
4. R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, and P. Marwedel. Scratchpad Memory: A Design Alternative for Cache On-chip Memory in Embedded Systems. In *10th Int. Symp. on Hardware/Software Codesign (CODES)*, May 2002.
5. M. Chen. A Timing Analysis Language - (TAL) - Programmer's Manual. Technical report, Dept. of Computer Sciences, University of Texas, Ausin, TX, USA, 1987.
6. K.D. Cooper and T.J. Harvey. Compiler-Controlled Memory. In *Architectural Support for Programming Languages and Operating Systems*, pages 2–11, 1998.

7. H. De Man. Keynote session at DATE'02. *http://www.date-conference.com/conference/keynotes/index.htm*, 2002.
8. L. Eggermont. Embedded Systems Roadmap. Technical report, STW, http://www.stw.nl/progress/ESroadmap/index.html, 2002.
9. S.C.A. Gordon-Ross and F. Vahid. Exploiting Fixed Programs in Embedded Systems: A Loop Cache Example. *Computer Architecture Letters*, January 2002.
10. Y. Hur, Y.H. Bea, S. Kin, B. Rhee, W.L. Min, C.Y. Park, M. Lee, H. Shin, and C.S. Kim. Worst Case Timing Analysis of RISC Processors: R3000/R3010. In *Proceedings of the 16th Real-Time Systems Symposium*, pages 308–319, 1995.
11. ILOG. CPLEX. http://www.ilog.com/products/cplex.
12. M. Kandemir, I. Kadayif, and U. Sezer. Exploiting Scratch-Pad Memory Using Presburger Formulas. In *Proceedings of the 14th Internation Symposium on System Synthesis ISSS*, page 7ff, 2001.
13. R. Kirner and P. Puschner. Consideration of Optimizing Compilers in the Context of WCET Analysis. In *Proc. Deutsche Informatiktage 2000, Bad Schussenried*, pages 123–126. GI Gesellschaft für Informatik e.V., Oct. 2000.
14. R. Kirner and P. Puschner. International Workshop on WCET Analysis - Summary. Research Report 12/2002, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-1, 1040 Vienna, Austria, 2002.
15. H. Kopetz. *Real-Time Systems – Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, 1997.
16. L. H. Lee, B. Moyer, and J. Arends. Instruction Fetch Energy Reduction Using Loop Caches For Embedded Applications with small Tight Loops. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED)*, San Diego, CA, USA, August 1999.
17. Y.-T.S. Li, S. Malik, and A. Wolfe. Effi cient Microarchitecture Modeling and Path Analysis for Real-Time Software. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 298–307, December 1995.
18. Y.-T.S. Li, S. Malik, and A. Wolfe. Performance Estimation of Embedded Software with Instruction Cache Modeling. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, pages 380–387, November 1995.
19. Y.-T.S. Li, S. Malik, and A. Wolfe. Cache Modeling for Real-Time Software: Beyond Direct Mapped Instruction Caches. In *Proceedings of the IEEE Real-Time Systems Symposium*, December 1996.
20. S. Lim, Y.H. Bea, G.T. Jang, B. Rhee, S.L. Min, C.Y. Park, H. Shin, and C.S. Kim. An Accurate Worst Case Timing Analysis for RISC Processors. In *Proceedings of the 15th Real-Time Systems Symposium*, pages 97–108, 1994.
21. P. Machanik. Approaches to Addressing the Memory Wall. *Technical Report, November, Univ. Brisbane*, 2002.
22. A.K. Mok, P. Amerasinghe, M. Chen, and K. Tantisirivat. Evaluating Tight Execution Time Bounds of Programs by Annotations. In *Proc. of the 6th IEEE Workshop on Real-Time Operating Systems and Software*, pages 74–80, 1989.
23. S.S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, San Francisco, California, 1997.
24. P. R. Panda, N. D. Dutt, and A. Nicolau. *Memory Issues in Embedded Systems-On-Chip*. Kluwer Academic Publishers, 1999.
25. P. Puschner and A. Burns. A review of Worst-Case Execution-Time Analysis (Editorial). *Journal of Real-Time Systems*, 18:115–128, 1999.
26. R. Sedgewick. *Algorithms*. Addison Wesley, Massachusetts, 1988.

27. S. Steinke. *Investigation of the Potential for Energy Savings in Embedded Systems enabled by Energy Optimizing Compilers*. PhD thesis, (in German), Embedded Systems Group, CS Dept., University of Dortmund, Dortmund, Germany, 2003.
28. S. Steinke, N. Grunwald, L. Wehmeyer, R. Banakar, M. Balakrishnan, and P. Marwedel. Reducing Energy Consumption by Dynamic Copying of Instructions onto Onchip Memory. *Int. Symp. on System Synthesis (ISSS)*, pages 213–218, 2002.
29. S. Steinke, L.Wehmeyer, B.-S. Lee, and P. Marwedel. Assigning Program and Data Objects to Scratchpad for Energy Reduction. *Design, Automation and Test in Europe (DATE)*, pages 409–417, 2002.
30. M. Theokharidis. Energiemessung von ARM7TDMI Prozessor-Instruktionen. Master's thesis, (in German), Embedded Systems Group, CS Dept., University of Dortmund, Dortmund, Germany, 2000.
31. M. Verma, S. Steinke, and P. Marwedel. Data Partitioning for Maximal Scratchpad Usage. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASPDAC)*, page 77, January 2003.
32. M. Verma, L. Wehmeyer, and P. Marwedel. Cache-aware Scratchpad Allocation Algorihm. In *Proceedings of Design, Automation and Test in Europe (DATE 2004) (to be published)*, February 2004.
33. L. Wang, W. Tembe, and S. Pande. A Framework for Loop distribution on Limited On-Chip Memory Processors. In *Proceedings of the 9th International Conference on Compiler Construction, CC/ETAPS'00, volume 1781 of LNCS*, pages 141–156, 2000.
34. S. Wilton and N. Jouppi. CACTI: An enhanced access and cycle time model. *Int. Journal on Solid State Circuits*, 31(5):677–688, 1996.
35. W. A. Wulf and S. A. McKee. Hitting the Memory Wall: Implications of the Obvious. *Computer Architecture News*, 1995.
36. J. Xu and D. L. Parnas. On Satisfying Timing Constraints in Hard-Real-Time Systems. *ACM SIGSOFT Software Engineering Notes, Proceedings of the Conference on Software for Critical Systems*, 16(5):132–146, September 1991.
37. N. Zhand, N.A. Burns, and M. Nicholson. Pipelined Processors and Worst Case Execution Times. *Real-Time Systems*, 4(5):319–343, 1993.