Compiler-optimized Usage of Partitioned Memories

Lars Wehmeyer, Urs Helmig, Peter Marwedel
{Lars.Wehmeyer, Urs.Helmig, Peter.Marwedel}@udo.edu

Embedded Systems Group, CS Dept., University of Dortmund, Germany

Abstract. *In order to meet the requirements concerning both performance and energy consumption in embedded systems, new memory architectures are being introduced. Beside the well-known use of caches in the memory hierarchy, processor cores today also include small onchip memories called scratchpad memories whose usage is not controlled by hardware, but rather by the programmer or the compiler. Techniques for utilization of these scratchpads have been known for some time. Some new processors provide more than one scratchpad, making it necessary to enhance the workflow such that this complex memory architecture can be efficiently utilized. In this work, we present an energy model and an ILP formulation to optimally assign memory objects to different partitions of scratchpad memories at compile time, achieving energy savings of up to 22% compared to previous approaches.*

1 Introduction

Since the design of embedded systems is very much driven by applications, it can be expected that future devices will require significantly more processing power, due to audio and video applications as well as high computational demands for channel coding. 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 [1]. A significant amount of research on low-power design techniques has been performed, but the 100 to 1000 fold improvement demanded by De Man [2] 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 [3].

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 times and the energy consumed per access. The impact of memory size on energy and performance can be seen in figure 1, which shows the increase in energy and cycle times as the memory size increases. The values in the figure were obtained using a subset of the CACTI model [4, 5], assuming a feature size of 0.5μm.

In general, due to the increasing sizes of applications and the corresponding memory sizes, access times and energy consumption will become even larger in the future.

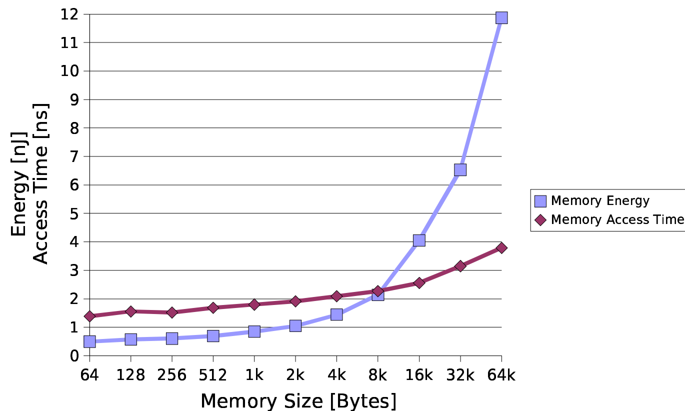

Fig. 1. Relation of energy and memory size

Therefore, it does make sense to map hot spots in applications to smaller memories instead of using just one large, homogeneous memory.

Caches have been established as the key solution to ease the problem for PCs and mainframes. They are not ideal solutions to the aforementioned problem, however. Small onchip memories, called scratchpad memories, show faster access times, reduced energy consumption and better predictability [5]. The idea of dividing the large main memory into a small scratchpad memory and the "remaining" main memory can be extended to form several small scratchpad memory partitions that are close to the processor and therefore fast and energy efficient. In this work, we show how the available compiler-based allocation algorithms [6, 7] can be extended to be able to allocate memory objects to several memory partitions.

The remainder of this paper is structured as follows: a description of related work is provided in section 2. Section 3 presents the toolchain required to exploit the presence of partitioned memories. Section 4 and 5 introduce the used energy model and the optimization problem formulation, respectively. We conclude the paper with results in section 6 and a summary and an outlook on future work.

2 Related Work

Scratchpad memories are being used in small, efficient embedded systems as an alternative to caches due to their performance and their reduced energy consumption [5]. Scratchpad memories do not have a hardware to control their contents at runtime. The scratchpad can either retain the assigned memory objects throughout the running time of the application (static case), or the contents of the scratchpad may change at runtime (dynamic case). Allocation techniques to statically allocate data to the scratchpad were introduced e.g. in [8], whereas [9] and [6] presented a dynamic approach for data and instructions, respectively. Further work concerning the utilization of scratchpad memories was conducted by [10, 11]. Both static and dynamic scratchpad usage are under

full control of the compiler or the programmer. In this paper, we will concentrate on the static allocation technique.

The goal of the static allocation of both instructions and data to the scratchpad memory in [7] is saving energy. In order to evaluate the energy consumption of a given memory allocation, an instruction level energy model for the used processor, an ARM7TDMI [12], was developed [13] and used in the *encc* compiler [14]. The compiler determines the execution counts of functions and basic blocks and the number of accesses to variables in order to compute the most promising objects to be assigned to the limited scratchpad space. The actual optimization problem, being similar to the well-known knapsack problem, is solved using an ILP solver. Then, the chosen memory objects are placed on the scratchpad, making control flow and address corrections where necessary. Comparisons against a cache solution showed advantages between 12% and 43% in energy consumption.

In this work, we consider not only one single scratchpad memory, but assume a set of scratchpad memory partitions. A trend towards small partitioned memories can also be found in industry, making our approach relevant e.g. for the latest available ARM processor cores. The ARM9E features two so-called tightly coupled memories (TCMs) that may either be used as SmartCaches or as plain memory regions [15]. In the latter case, TCMs can be used like scratchpad memories. Implementations of this core architecture are available from several hardware vendors [16–18]. ARM's architecture model implies that one of the TCMs be used to hold only data while the other is used for instructions. In the current work, we present a more general approach: a multi-purpose scratchpad memory is partitioned into several smaller contiguous regions, each of which is allowed to hold instructions as well as data. In order to optimally exploit these memory partitions, we have integrated an algorithm into our energy aware *encc* compiler to map the most promising memory objects to the different scratchpad memory partitions.

3 Workflow

The workflow used to generate memory layouts and to determine energy consumption values is depicted in figure 2.

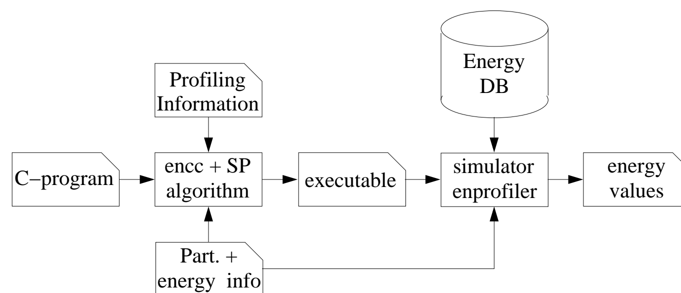

Fig. 2. Workflow

The scratchpad allocation algorithm integrated into the *encc* compiler takes the application program written in C as one of its inputs. The architectural description of the currently used memory partitioning also includes the “energy per access” values of the considered memory partitions. Finally, *encc* requires some profiling information. The profiling can either be performed on a static basis, i.e. by only considering the source code of the application, or by compiling the application once, simulating it and back-annotating the information onto the internal representation of the program. As will be shown in the results section, even the simple static analysis is capable of characterizing the program behavior in a sufficiently precise way. The information obtained from either static analysis or from dynamic profiling and the energy model presented in the following section 4 are then used to determine

a) the set of most promising objects to allocate to the scratchpad memory and
b) the exact memory layout, i.e. which object to allocate to which memory partition.

This optimization problem is formulated as an integer programming problem as detailled in section 5. We use a commercial ILP solver [19] to solve this problem and thus to find a mapping of memory objects to memory partitions which yields maximum energy savings. The resulting executable, with instructions and data distributed among the different memory partitions, is then simulated using ARM’s instruction set simulator ARMulator and analyzed using our profiling tool *enprofiler*. In this step, *enprofiler* incorporates values from an energy database which contains information derived from the instruction level energy model described in [13]. These energy values are used to determine the overall energy consumption of the executable at runtime. The results presented in this work only consider the energy consumption in the memory subsystem. These values can be used to compare different memory architectures and allocation techniques.

4 Energy Model

The energy model used in this work is based on the instruction level energy model presented in [13], which assumes the overall energy to be the sum of processor and memory energy:

$$E_{total} = E_{CPU} + E_{IMem} + E_{DMem} \quad (1)$$

where E_{IMem} is the energy required to fetch an instruction from memory and E_{DMem} is the energy required to access (read and write) data. Since the proposed optimization only modifies the memory layout, but not the individual instructions of the original program, the processor energy E_{CPU} is considered to have no influence on the results and is therefore ommitted in the following considerations. As part of the future work, we will also consider different access times for the memory partitions which will have a positive impact on the energy consumption of the processor due to the reduced number of memory cycles. The impact of the used memory partition size on the instruction and data access costs can be seen in figure 1. Depending on the memory partition that a particular object is assigned to, the corresponding memory access costs have to be assumed when accessing this object. This is done as follows for global variables and basic blocks:

4.1 Global Variables

Local variables within a function are usually kept in registers or on the stack, and there may be several instances of local variables in different contexts. For these reasons, only global data elements are considered in the current implementation of the optimization problem. We only consider complete objects such as arrays at the moment. An array splitting approach as presented in [20] might be added to our method later. The simple methodology for moving the stack to a different memory partition as described in [7] can be integrated into the proposed model in a straight-forward manner, thus including some of the local variables in the consideration.

The energy assigned to a global variable in the model corresponds to the sum of all accesses to this variable, taking into account the per-access energy of the memory partition mp used to store the global variable:

$$E_{var} = E_{Daccess,mp} \cdot \#accesses \tag{2}$$

The number of accesses is determined using either static analysis or dynamic profiling, as described above in section 3. The energy contribution of all variables is then summed up to form E_{DMem} in equation (1).

4.2 Basic Blocks

Since the processor's contribution to energy dissipation is not considered in this work, the energy consumption of a basic block is only caused by fetching the instructions from the corresponding memory partition mp. It can thus be expressed using the following equation:

$$E_{bb} = E_{Ifetch,mp} \cdot (\#executions) \cdot (\#instructions + 2) \tag{3}$$

The number of instructions within the basic block are multiplied with the number of executions of this particular basic block. The latter value is again determined using static analysis or dynamic profiling. Finally, we multiply the energy required for a single instruction fetch from the assumed memory partition mp to determine the total energy consumption of a basic block in our energy model. The contribution of all basic blocks is added up to from E_{IMem} in equation (1).

The general addition of the value two to the number of instructions will be refined later in the extended optimization problem. It is due to the fact that a basic block, in contrast to e.g. a function, is not always a self-contained object. Consider the example given in figure 3: in the left part of the figure, all basic blocks are stored contiguously in one memory region. This means that the edge from `BB1` to `BB2` is an implicit control flow edge, i.e. if the conditional branch `beq` is not taken, then control passes on to the next consecutive address, which is the first instruction of `BB2`.

If `BB2` is moved to a different memory partition, however, the implicit control flow edge would lead to the subsequent basic block `BB3` being executed after `BB1` independently of the result of the conditional branch. In order to preserve the original control flow, an additional unconditional branch has to be inserted following the conditional

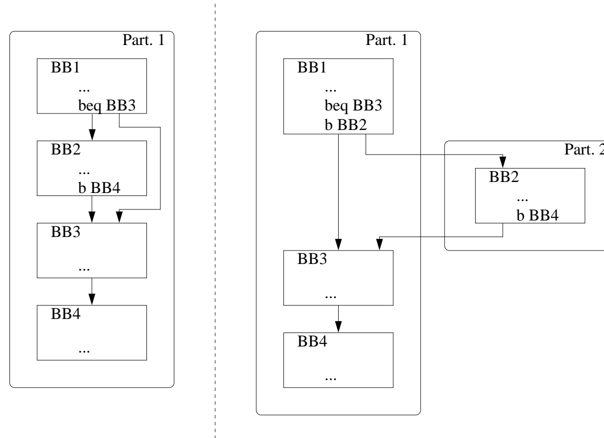

Fig. 3. An additional jumps is required in BB1 when moving BB2 to a different memory partition

branch in `BB1`. In this way, if the conditional branch is not taken, control passes on to `BB2` in memory partition 2.

In order to model the penalty of moving a single basic block to a different memory partition, two (potential) instructions are added to the instruction count of each basic block. In the next section we will see how the relationship between basic blocks is modelled in such a way that contiguous basic blocks that reside in the same memory partition are handled correctly, i.e. without considering any additional jump penalty.

5 Optimization Problem Formulation

In order to formulate the optimization problem of assigning memory objects (instructions and data) to a set of available memory partitions, a couple of preliminary definitions are necessary.

5.1 Preliminaries

We start with the set of memory partitions, which is denoted as

$$MP := \{mp_1, \ldots, mp_m\} \tag{4}$$

Each of the m memory partitions has a certain size in bytes that can be queried using the function $Size(mp_i)$.

The sets of objects considered in the optimization process are defined as follows:

– set of global variables:
$$G := \{g_1, \ldots, g_q\} \tag{5}$$
– set of basic blocks:
$$BB := \{bb_1, \ldots, bb_r\} \tag{6}$$

These sets are then combined into one set of memory objects O:

$$O \subseteq G \cup BB = \{o_1, \ldots, o_n\} \quad (7)$$

Like for the memory partitions, function $Size(o)$ returns the size in bytes of memory object $o \in O$.

In order to determine the energy consumption of a memory object o in memory partition mp, we define the function $Energy(o, mp)$ which returns the amount of energy consumbed by object o if it is allocated to memory partition mp. The value returned by this function corresponds to E_{var} or E_{bb} from the previous section, depending on the type of memory object. This formulation leads to a minimization problem, since the overall energy consumption of the program should be minimized.

5.2 Basic ILP model

Using the sets and functions defined in the previous section, we can now proceed to model the optimization problem of allocating memory objects to the different memory partitions as an ILP problem, which can subsequently be solved using an ILP solver [19].

The optimization problem uses the matrix $\tilde{O}$ containing binary decision variables $\tilde{o}_{i,j}$ which assume the value 1 if memory object o_i is allocated to memory partition mp_j. The objective function can then be written as

$$Minimize \sum_{i=1}^{n} \sum_{j=1}^{m} [Energy(o_i, mp_j) \cdot \tilde{o}_{i,j}] \quad (8)$$

The model has to ensure that each memory object is assigned to only one memory partition. This is achieved by the following constraint:

$$\forall i : 1 \leq i \leq n : \sum_{j=1}^{m} \tilde{o}_{i,j} = 1 \quad (9)$$

The solver also has to take care not to allocate too many objects to a particular memory partition: the size of the objects allocated to a certain memory partition must not exceed this partition's capacity:

$$\forall j : 1 \leq j \leq m : \sum_{i=1}^{n} [Size(o_i) \cdot \tilde{o}_{i,j}] \leq Size(mp_j) \quad (10)$$

This concludes the basic formulation of the ILP optimization problem. However, this does not yet include the dependencies between basic blocks. If two contiguous basic blocks stay in the same memory, there is no need to take any additional jumps between memory partitions into account. Therefore, the next section covers the extension of the model to correctly handle contiguous basic blocks.

5.3 ILP model extension: contiguous basic blocks

In order to correctly consider basic blocks that are mapped to contiguous memory addresses in one partition and thus do not need to be connected by an added jump instruction, the model considers the edges of the program's control flow graph. Additional constraints are used to describe whether a particular edge between two subsequent basic blocks leaves a memory partition or stays within this partition. The advantage of this approach is that any number of basic blocks can be considered as a unit and stored in a memory partition without having to enumerate all possible combinations of basic blocks (as was done e.g. in [7]). This leads to good results and due to the reduced number of memory objects, also to acceptable running times of the used ILP solver [19] for our benchmarks. As part of the future work we will work on finding approximation algorithms that have polynomial running time.

In order to model the edges of the control flow graph, we first introduce a corresponding set

$$C := \{c_1, \cdots, c_p\} \tag{11}$$

which represents the control flow edges that connect basic blocks. In order to determine whether an edge connects two specific basic blocks, we additionally define the function $V(x,y)$ which returns the index number k if $o_x \in BB$, $o_y \in BB$ and if o_x and o_y are connected by edge $c_k \in C$. If the two nodes are not connected by an edge, the function returns -1.

These definitons are used to add another matrix $\tilde{C}$ containing a binary decision variable $\tilde{c}_{k,j}$ for each edge k and every memory partition j. Variable $\tilde{c}_{k,j}$ is assigned the value 1 if memory partition j is not left when control flows along edge k. In other words: if two connected basic blocks are within the same memory partition, the corresponding decision variable is set to 1.

With this additional matrix, a modified objective function can be formulated. By default, the potentially required additional jumps are always considered. Therefore, if the corresponding memory partition is not left and the jumps are not required, their contribution is subtracted from the cost function:

$$Minimize \sum_{i=1}^{n} \sum_{j=1}^{m} [Energy(o_i, mp_j) \cdot \tilde{o}_{i,j}] - \sum_{k=1}^{p} \sum_{j=1}^{m} [StmtEnergy_{jump} \cdot \tilde{c}_{k,j}] \tag{12}$$

$StmtEnergy_{jump}$ corresponds to the energy consumed by one jump instruction. Since we know which jump instructions are necessary, we can also adjust the size constraint to reflect the fact that not all initially assumed jumps are required:

$$\forall j : 1 \le j \le m : \sum_{i=1}^{n} [Size(o_i) \cdot \tilde{o}_{i,j}] - \sum_{k=1}^{p} [StmtSize_{jump} \cdot \tilde{c}_{k,j}] \le Size(mp_j) \tag{13}$$

Up to this point, we have assumed the decision variables $\tilde{c}_{k,j}$ to be set correctly. We do require additional equations to ensure that they actually reflect the relationship of basic blocks, edges and memory partitions. This can be achieved by specifying additional constraints of the form

$$(k = V(i,x), \forall k \neq -1, 1 \leq i \leq n, 1 \leq x \leq n), (\forall j, 1 \leq j \leq m):$$

$$\tilde{o}_{i,j} + \tilde{o}_{x,j} - 2\tilde{c}_{k,j} \geq 0 \quad (14)$$

In this way, a decision variable $\tilde{c}_{k,j}$ has to be set to 0 if the two basic blocks connected by edge k do not reside in the same memory partition. Please note that the specified constraint will only prevent the solver from seting variables to 1 if the edge does not stay within one memory partition. All remaining variables will be set to 1 automatically by the solver since this will help to maximize the objective function.

The complete ILP model consists of the objective function (12), the memory partition size constraint (13), a constraint to ensure that each object is selected only once (9) and, finally, the edge constraints (14). Passing this model to the ILP solver, it is possible to determine the optimal allocation of objects to a given set of memory partitions.

6 Results

In order to keep the search space for the different memory partitions at an acceptable level, not all possible combinations of partitioning memory regions were considered. Rather, we have decided to consider as the base configuration one scratchpad providing the entire capacity. This scratchpad memory is then partitioned into two memories of half the capacity. One of these two scratchpads is then left untouched, whereas the second scratchpad is iteratively partitioned in two halfs. The resulting memory partitions for a total capacity of 4096 bytes are shown in table 1.

# of partitions	number of partitions of size:						
	4K	2K	1K	512	256	128	64
7	0	1	1	1	1	1	2
6	0	1	1	1	1	2	0
5	0	1	1	1	2	0	0
4	0	1	1	2	0	0	0
3	0	1	2	0	0	0	0
2	0	2	0	0	0	0	0
1	1	0	0	0	0	0	0

Table 1. Example of all considered memory partitions for a total capacity of 4096 bytes

The benchmarks used to evaluate our approach of assigning objects to partitioned scratchpad memories are shown in table 2.

First, we present the overall energy savings achievable using the model presented in this work. Since only memory energy is modified, all results are given as energy savings in the memory subsystem. For only one single scratchpad memory partition,

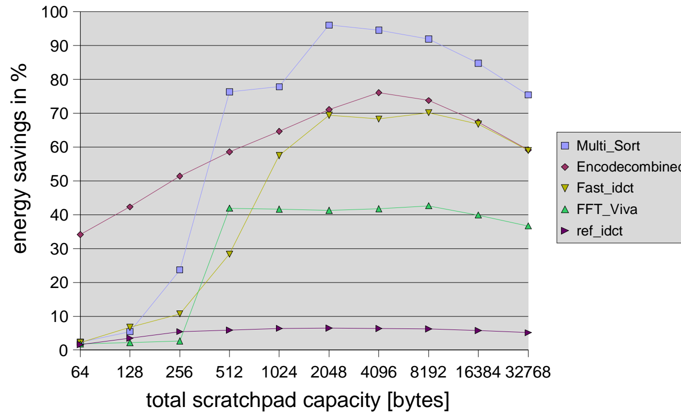

Fig. 4. Energy savings with only one scratchpad partition using static analysis

the results as given in figures 4 and 5 can be achieved. Comparing the two figures, it becomes obvious that even the simple static analysis used for figure 4 yields good results. Especially for the “Multi_Sort” benchmark, however, figure 5 shows a much more steady increase in energy savings since the execution/access counts were determined more precisely using dynamic profiling. For the remaining figures, we have decided to present results obtained using dynamic profiling in order to show the potential of the proposed approach. In general, we observe that the energy savings increase with the available amount of scratchpad space. If the entire program fits in the scratchpad, further increasing the scratchpad capacity leads to an increase in the energy consumption due to the higher per-access costs of larger partitions. These results correspond to the results presented in [7], where only one scratchpad and a simpler allocation algorithm was used.

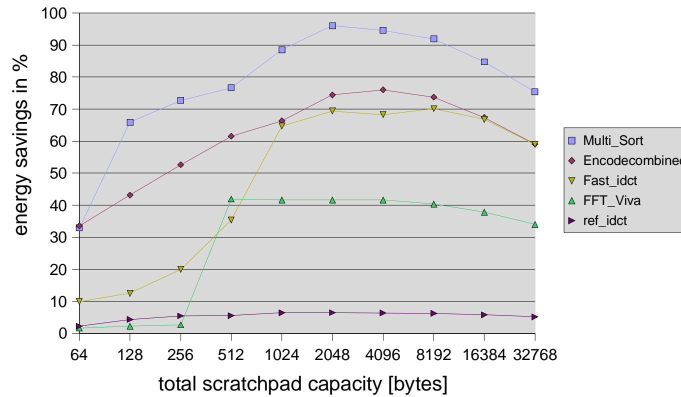

Fig. 5. Energy savings with only one scratchpad partition using dynamic profi ling

benchmark	description
multi_sort	combination of different sorting algorithms
encodecombined	part of the GSM kernel
fft_viva	fast fourier transform
fast_idct	inverse discrete cosine transform
ref_idct	idct reference implementation

Table 2. Selected Benchmarks

For the “encodecombined” benchmark, figure 6 shows the possible energy savings obtainable for different scratchpad capacities when the number of partitions is varied. The partition sizes are chosen according to table 1. Up to a size of 4096 bytes, the energy savings can be seen to increase even when only a single scratchpad partition is used. After that, the energy gain goes down for a single scratchpad partition. If several partitions are used, however, the maximum gain obtained can be sustained even for larger scratchpad capacities. This shows that if a processor with a given amount of scratchpad is being used, it can be profitable to provide the total capacity not as one big scratchpad, but rather as several smaller partitions, assuming that the used compiler is able to exploit this kind of memory architecture.

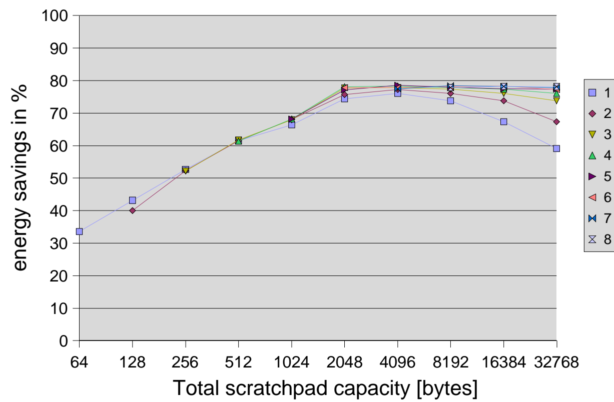

Fig. 6. Results for encodecombined with increasing scratchpad capacity and varying number of scratchpad partitions

Based on these values, figure 7 shows the maximum achievable benefit obtained from partitioning the single scratchpad into smaller parts for all benchmarks using the described model. The single scratchpad case serves as the baseline in this figure. For each benchmark, total scratchpad capacity and number of partitions, the combination with maximum benefit was chosen. Improvements of up to 22% are possible by using partitioned scratchpad memories instead of just one memory.

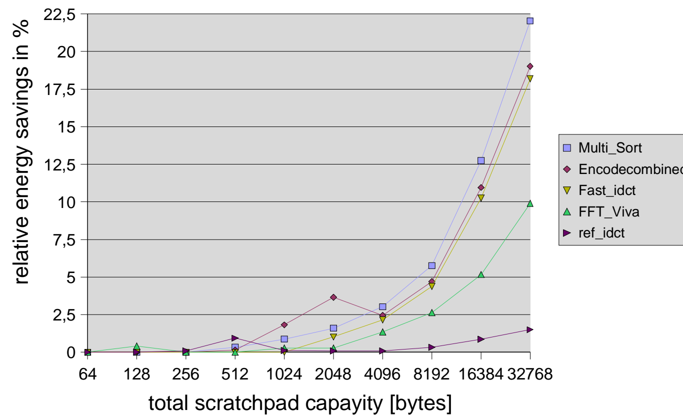

Fig. 7. Relative energy savings with multiple scratchpad partitions

7 Summary and Future Work

In this work, we have described a model incorporated into a compiler that is capable of allocating program objects (instructions as well as data) to a partitioned onchip scratchpad memory. The optimization problem is formulated in the form of ILP equations. This allows the use of standard solvers and makes it easy to integrate extensions into the model. We have shown that by using the proposed model and partitioned scratchpad memories (instead of just one single larger scratchpad), improvements of up to 22% in the memory subsystem can be obtained. The algorithm presented in [7] can be completely replaced by the new algorithm presented in this paper since using a single scratchpad is included in the presented model as a special case.

As part of the future work, reduced access times for smaller memory partitions will be considered. Also, we plan to integrate into the model the notion of a standby energy that is always consumed by an active memory partition. In this way, the compiler could actually influence the decision of whether an additional partition is required or not. Finally, we are working on ways to avoid the possible worst-case complexity of the ILP approach by looking for methods that approximate the optimal solutions using algorithms with polynomial running time.

References

1. L. Eggermont. Embedded Systems Roadmap. Technical report, STW, http://www.stw.nl/progress/ESroadmap/index.html, 2002.
2. H. De Man. Keynote session at DATE'02. *http://www.date-conference.com/conference/keynotes/index.htm*, 2002.
3. P. Machanik. Approaches to Addressing the Memory Wall. *Technical Report, November, Univ. Brisbane*, 2002.
4. S. Wilton and N. Jouppi. CACTI: An enhanced access and cycle time model. *Int. Journal on Solid State Circuits*, 31(5):677–688, 1996.

5. 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.
6. 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.
7. 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.
8. P. R. Panda, N. D. Dutt, and A. Nicolau. *Memory Issues in Embedded Systems-On-Chip*. Kluwer Academic Publishers, 1999.
9. M.Kandemir, J.Ramanujam, M.J.Irwin, N.Vijaykrishnanand I.Kadayif, and A.Parikh. Dynamic Management of Scratch-Pad Memory Space. In *Proceedings of the 2001 ACM Design Automation Conference*. DAC, June 2001.
10. J.Ph. Diguet, S. Wuytack, F. Catthoor, and H. De Man. Formalized Methodology for Data Reuse Exploration in Hierarchical Memory Mappings. In *ISLPED 1997 Monterey CA*. ACM, August 1997.
11. P.R.Panda, F.Catthoor, N.D.Dutt, K.Danckaert, E.Brockmeyer, C.Kulkarni, A.Vandercapelle, and P.G.Kjeldsberg. Data and memory optimization techniques for embedded systems. *Trans. on Design Automation of Electronic Systems (TODAES)*, pages 149–206, April 2001.
12. ARM Ltd. ARM7TDMI Technical Reference Manual, Ref: DDI0029G. Available from http://www.arm.com, 2001.
13. S. Steinke, M. Knauer, L. Wehmeyer, , and P. Marwedel. An Accurate and Fine Grain Instruction-Level Energy Model Supporting Optimizations. In *Proceedings of the International Workshop - Power and Timing Modeling, Optimization and Simulation, Yverdon-les-bains, Switzerland*, September 2001.
14. Lars Wehmeyer, Stefan Steinke. encc Compiler. Information available from: http://ls12-www.cs.uni-dortmund.de/research/encc, 2004.
15. ARM Inc. ARM Documentation - ARM Processor Cores. Available from: http://www.arm.com/documentation/ARMProcessor_Cores/index.html, 2004.
16. Oki. Oki Global Homepage. http://www.oki.com, 2004.
17. Samsung. Samsung's Digital World. http://www.samsung.com, 2004.
18. Atmel Corporation. Atmel Corporation Homepage. http://www.atmel.com, 2004.
19. ILOG. CPLEX. http://www.ilog.com/products/cplex.
20. Manish Verma, Stefan Steinke, and Peter Marwedel. Data Partitioning for Maximal Scratchpad Usage. In *Proceedings of the Asia and South Pacific Design Automation Conference*, page 77, January 2003.