Reducing Energy Consumption by Dynamic Copying of Instructions onto Onchip Memory

Stefan Steinke*, Nils Grunwald*, Lars Wehmeyer*, Rajeshwari Banakar+,
M. Balakrishnan+, Peter Marwedel*
*University of Dortmund, Computer Science 12, 44221 Dortmund, Germany
+Indian Institute of Technology, Delhi, India

ABSTRACT

The number of mobile embedded systems is increasing and all of them are limited in their uptime by their battery capacity. Several hardware changes have been introduced during the last years, but the steadily growing functionality still requires further energy reductions, e.g. through software optimizations. A significant amount of energy can be saved in the memory hierarchy where most of the energy is consumed.

In this paper, a new software technique is presented which supports the use of an onchip scratchpad memory by dynamically copying program parts into it. The set of selected program parts are determined with an optimal algorithm using integer linear programming.

Experimental results show a reduction of the energy consumption by nearly 30%, a performance increase by 25% against a common cache system and energy improvements against a static approach of up to 38%. [1]

Categories and Subject Descriptors

C.3 [**Computer Systems Organization**]: Special-Purpose and Application-based Systems

General Terms

Algorithms

Keywords

Energy optimization, Onchip memory

1. INTRODUCTION

Since the introduction of mobile systems, the energy supply has been a permanent problem and limiting factor. The improvement of the battery capacity is only very slow whereas the increase in number of transistors, coupled with increasing complexity of applications (reflected in software complexity) causes a strong demand for reducing the energy consumption. But the increasing energy consumption also leads to other limitations, e.g. size constraints of notebooks because of the heat dissipation [6] or the lifetime of electronic circuits which depends on the temperature. Finally, the capacity of the power supply depends on the maximum current required by the system. Reductions in the maximum currents might lead to smaller power supplies with less weight and less cost. Overall, there is a strong demand for reducing the energy consumption of electronic circuits.

To reduce energy consumption, hardware modifications have been introduced at several levels. The feature sizes are permanently shrinking and the supply voltages are being reduced. Furthermore, modern processor designs incorporate several power down modes to shut down parts of the system or even the whole system if it is not used for a longer time frame.

In addition, energy can be saved by optimizing the software application itself. The analysis of systems based on RISC processors which are well known for their reduced circuitry and energy consumption reports the memory system as the main cause of energy consumption. Therefore, the number of accesses to the memory is a key factor that needs to be minimized. Furthermore, the overall energy consumption can be improved by adding additional onchip memories to the memory system.

A memory system consists of different memory types at different locations in the system. In contrast to a regular main memory which is located offchip, onchip memories are of small size and located on the same chip as the processor. This saves energy because of reduced access times and due to less active bus drivers. Besides the location, the memory organization can be varied betweeen RAMs or e.g. Caches. RAMs of a limited size which are located onchip are commonly called scratchpad.

First, we observe the well known caches. They can perform automatic copying of blocks into the fast cache memory using hardware control. If the memory content is valid in the cache, an access to the next higher level of the memory hierarchy can be avoided. The main drawback is the high amount of energy which is spent for the additional tag memory containing the cache addresses and for the comparators which do a lot of comparisons to determine hits and misses.

To overcome this drawback, we introduce in this paper an algorithm for an onchip scratchpad memory which is filled with program parts and accessed by software control. At

[1]This work has been supported by Agilent Technologies, USA. and DAAD

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ISSS'02, October 2–4, 2002, Kyoto, Japan.
Copyright 2002 ACM 1-58113-576-9/02/0010 ...$5.00.

certain predetermined points in the program, parts of the program are copied into the scratchpad and are subsequently executed from the just loaded scratchpad. Energy hungry comparisons of the cache address with the current valid address can be saved. No cache misses are possible since the software control always "knows" the content of the scratchpad. On the other hand, copying program parts is more energy consuming than with a cache solution.

The rest of the paper is structured as follows: After the presentation of related work, the algorithm for the scratchpad control is described in detail in Section 3. This is followed by a presentation of the experimental setup. The results of these experiments which were performed on an ARM7T RISC processor [1] are shown in Section 4. The paper ends with a conclusion and future work.

2. RELATED WORK

The energy optimization of application programs generated by compilers was studied by Tiwari et al. [16] who developed an instruction level power model by physical measurements. This model consists of energy costs of a single instruction ("basic costs") plus the overhead for switching to the next instruction ("interinstruction costs"). The sum corresponds to the total amount of energy consumed by the observed application. The main drawback is the missing consideration of memories. The instruction energy model by Simunic [12] takes memories into account, but because all energy values are only taken from the vendor's datasheet, there is no distinction between different processor instructions. The combination of these two approaches and a further extension for treating the bit patterns on busses was presented by Steinke et al. [14]. This model was also choosen for this research work.

For a fair comparison of the two onchip memory alternatives cache and scratchpad, the energy consumption has to be examined for the same technology. For this purpose, the cache energy model of Wilton et al. [17] was used together with a subset of the cache not including the tag memories and the comparators. This subset was described by Banakar et al. [2] and used to determine energy values for the scratchpad memory.

Based on instruction level power models, different energy optimizations have been integrated into compiler environments. The influence of compiler optimizations was studied in the above mentioned research work by Tiwari et al. [16] and Simunic et al. [12]. In addition, Kandemir et al. [9] observed the behavior of energy consumption with different cache sizes.

Further work of software improvements was combined with the development of specific hardware. The more efficient treatment of object code was developed by Ishihara et al. [7] and the design of application specific memories by Benini et al. [4]. Especially these memory related works have a high potential for improving power and performance optimizations.

Besides these application specific hardware approaches, even the usage of standard memory types during the system design phase is a potential choice. Common is the addition of a cache which will be treated here for comparison purposes as a reference. The replacement of caches by a scratchpad for the most often accessed variables was studied by Panda et al. [11]. The same approach was taken by Sjödin et al. [13], but with a more simplistic algorithm for choosing the variables. The dynamic copying of array parts was studied in [8].

In addition to the mapping of variables, Steinke et al. [15] presented an approach for the selection of program parts and variables on a static base. The program parts are never exchanged during the application run. In contrast, this paper only treats program parts, but performs a dynamic copy during program execution. This is especially advantageous for applications with more than one hotspot which do not fit in the scratchpad together.

3. ALGORITHM

The execution of instructions from the scratchpad is faster and more energy efficient than from the cache. The drawback of the scratchpad is that copying the instructions into it with the help of load and store statements is slower and therefore more energy consuming than a hardware controlled cache fill. To fill the scratchpad, copy functions are inserted into the application at compile time to copy a set of basic blocks which can later be executed from the scratchpad.

For each copied basic block, the best point in the control flow has to be chosen. It is advantageous to choose a point which is executed less frequently than the block that is to be copied and therefore causes less copy costs, but on the other hand a copy function close to the copied basic block is advantageous because the memory is not occupied that long.

The algorithm is structured as follows:

1. analysis of the application program to identify program parts
2. determination of possible program parts and locations for the copy functions in the control flow of the application
3. a choice of the best set of program parts to be copied together with associated copy functions

We will also prove the optimality of this algorithm.

3.1 Analysis of the Application

For the software control of the scratchpad, a detailed analysis has to be undertaken. Therefore, the structure of the program is analyzed. We consider functions and basic blocks, the latter being sequences of instructions with a single entry point and not including any branches, except at the very end.

For functions and basic blocks which are called program objects, the characteristics *size* and *number of executions* have to be determined with highest possible precision. The *number of executions* can be identified by a static analysis or by profiling during a simulation run. In this algorithm we chose the profiling approach because it is more precise.

In combination with the control flow among these program parts all necessary inputs are now available for dynamic mapping of program parts.

3.2 Search Space Reduction

There is a potentially large search space for the optimal combination of program parts and the corresponding copy functions. Each program part can potentially be copied by each copy function positioned beforehand. For this reason it is desirable to prune the search space without excluding the

optimal solution. In the following we introduce two assertions, the first for limiting the number of possible program parts to be copied and the second for reducing the number and position of copy function candidates:

Assertion 1: only those program parts are worthwhile to be copied which are executed more often than their corresponding copy function.

The copy function has to execute a load statement from the offchip and a store statement to the onchip memory for copying a single instruction. Even when using the ARM 16 bit instruction set's "load/store multiple", this consumes more energy than a single offchip instruction fetch. To copy an instruction into the scratchpad and execute it from there is thus not advantageous if the copy function is executed the same or a larger number of times than the instruction in the treated program part. It is obvious that only program parts which are executed in loops are worthwhile to be copied.

Assertion 2: positions of copy function candidates can be limited to loop entries

As shown in assertion 1, each copied program part has to be executed more frequently than its corresponding copy function in order to reduce the energy consumption. This occurs only in loops where the loop body is performed more often than the instructions directly before the loop. Due to this, the search space for the optimal solution can be limited to copy functions which are positioned directly before loops.

```
// -- copy point 1 --
for (i = 0; i < 100; i++) {

  .. // basic block 1

  // -- copy point 2 --
  for (j = 0; j < 20; j++) {

    ..// hotspot 1
  }
  // -- copy point 3 --
  for (k = 0; k < 30; k++) {

    ..// hotspot 2a

    if (..) {

      ..// hotspot 2b
    }
  }
}
```

Figure 1: program example

As an example, let us consider the program in figure 1. There are three loops, two of which are inside the first loop. If we observe the control flow of this program in figure 2[2], we can identify four basic blocks. The first block *basic block 1* represents the first instructions inside this loop up to and including the initialization of the second loop. The inner part of the second loop is represented by the basic block *hotspot 1*. The third loop is shown as basic block *hotspot 2a* followed by the basic block *hotspot 2b* generated by the "if" statement. The jump back from the end of the outer loop to its beginning goes back to *basic block 1*.

We can limit the possible insertion points of copy functions to the beginning of loops as shown before. Thus, there are three possible copy points (CP) in the control flow. They

[2]For simplification, we do not insert an additional basic block for the initialization of the third loop.

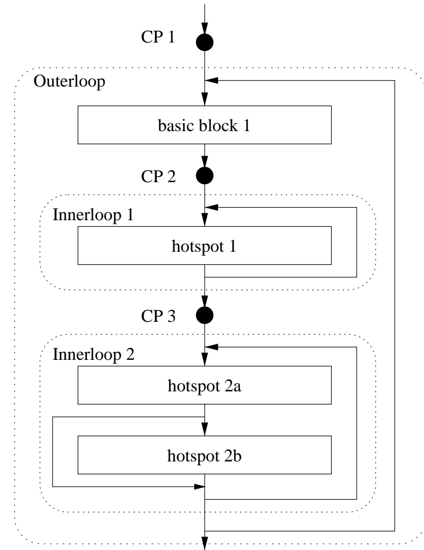

Figure 2: control flow of example program

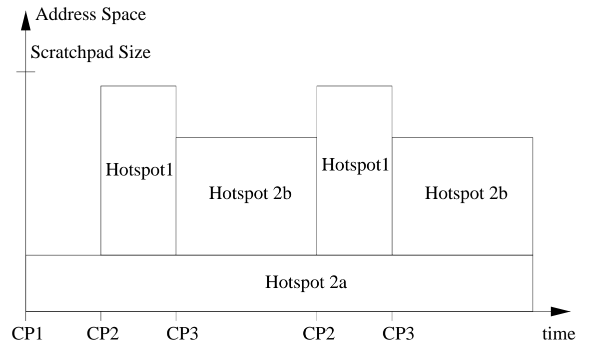

Figure 3: memory map of scratchpad

are *CP1*, *CP2*, and *CP3* where we can potentially insert a copy function. It has to be decided where e.g. *hotspot 1* should be copied. This can be done either at *CP1* or at *CP2*. The advantage of *CP1* is that it is executed less times compared to *CP2* which saves energy copy costs. On the other hand, the scratchpad memory which is allocated for *hotspot 1* can be reused by other program parts, e.g. *hotspot 2a* or *hotspot 2b*, if it is copied at *CP2*.

Depending on the loop count and the size of the basic blocks, a possible assignment of program objects to copy functions is shown in figure 3. Copy function *CP1* copies *hotspot 2a* of the *innerloop 2* into the scratchpad. Afterwards, *basic block 1* is executed from the offchip memory. The following copy function *CP2* assigns *hotspot 1* to the scratchpad. This memory space is assigned alternately to *hotspot 1* and *hotspot 2b* which is copied by *CP3*.

3.3 Selection of Program parts

After determining the possible copy function points for each basic block (1 for each loop level), we calculate the energy costs and select the optimal set of program objects.

First of all, we determine the energy costs for copying the program object PO at copy function CP_x into the scratchpad. These costs can be computed by multiplying the num-

ber of executions of the copy function $n(CP_x)$, the energy consumption for copying a single instruction E_{single_cp} and the number of instructions k:

$$E_{cp}(PO) = n(CP_x) * E_{single_cp} * k$$

This copy costs have to be subtracted from the benefit of executing a function or basic block from the scratchpad instead of the main memory.

The energy saving for the execution of a single instruction is defined as E_{if} which is the difference of executing an instruction fetch from the offchip against from the onchip memory:

$$E_{if} = E_{offchip,if} - E_{onchip,if}$$

In addition to basic blocks, a whole function F_i can be treated as a program object too. This object with k instructions with each instruction executed n_k times moved into the scratchpad saves the following amount of energy, taking the copy cost $E_{cp}(F_i)$ of the function into account:

$$E(F_i) = \sum_k n_k * E_{if} - E_{cp}(F_i)$$

In contrast to functions where the jump to the function and the return from it can be executed over a far distance of the address space, the basic blocks have a closer link with their neighboring blocks. Therefore we have to insert additional jumps with an energy cost of E_{jmp} from the main memory to the scratchpad and back. This has to be considered during size and energy calculation of these objects. The energy saving $E(B_j)$ for a basic block B_j with k instructions which is executed n times with l additional jumps is computed as follows:

$$E(B_j) = k * n * E_{if} - l * E_{jmp} - E_{cp}(B_j)$$

For each combination of a basic block with a corresponding copy function, a program object PO has to be created.

For identifying the best set of program objects and the corresponding copy functions, we use an integer linear programming approach [10]. The equations are constructed using the following definitions:

$$S(PO) = \text{size of } PO$$

$$m(x) = \begin{cases} 1, & \text{if x is moved to the scratchpad} \\ 0, & \text{otherwise} \end{cases}$$

To optimize the energy saving sav, the following cost function needs to be maximized:

$$sav = \sum_{i \in I} m(F_i) * E(F_i) + \sum_{j \in J} m(B_j) * E(B_j)$$

Index sets I and J correspond to index values for functions and basic blocks which are active at the same time in the scratchpad:

The size constraint for a fixed $scratchpadsize$ can be modeled as follows:

$$\sum_{i \in I} m(F_i) * S(F_i) + \sum_{j \in J} m(B_j) * S(B_j) \leq scratchpadsize$$

To prevent a basic block x from being selected twice, e.g. as a single basic block x and also as part of a function i, constraints of the following type have to be added with B_x being included within F_i:

$$m(B_x) + m(F_i) \leq 1$$

Based on this model, an IP solver can find the optimal solution for all memory objects and for all copy points in one single run for the use of a scratchpad memory. All additional costs for copying objects during program execution into the scratchpad or jumping to the scratchpad are incorporated in the ILP model.

4. EXPERIMENTAL SETUP

4.1 Models

To compare the scratchpad and the cache systems we need to define a model of the system consisting of processor and memories. We chose the ARM7 processor [1] which is a widespread ultra low power RISC processor. A detailed description of the energy model is given in [14].

Table 1: *energy per access (32 bit)*

memory size	cache	scratchpad	ratio
64 bytes	2.87 nJ	0.49 nJ	5.9
128 bytes	3.15 nJ	0.53 nJ	5.9
256 bytes	3.32 nJ	0.61 nJ	5.4
512 bytes	3.48 nJ	0.69 nJ	5.0
1024 bytes	3.75 nJ	0.82 nJ	4.6
2048 bytes	4.04 nJ	1.07 nJ	3.8

For a fair comparison of values for the two onchip memories the same technology and production process has to be used. Since it is difficult to obtain cache and scratchpad memories fulfilling these requirements, the cache values are based on the CACTI model [17]. For the scratchpad a subset of the cache was used [2, 3]. The gained values are shown in Table 1 for a 4 way set associative cache which is a commonly used architecture for ARM7 systems. The advantage of the scratchpad is mainly caused by the missing tag comparators and tag memory. Even compared with the simplest cache architecture, a direct mapped cache, the energy ratio compared to the scratchpad is still high [2].

The values for the fixed onchip memory size of the observed system are taken from Table 1 for the use in the ILP system.

4.2 Workflow

The diagram in figure 4 shows the workflow starting with programs compiled using the energy aware C compiler *encc* [5]. For the scratchpad configuration, the algorithm presented in the previous section is executed. The generated machine code is simulated using the simulator from ARM Ltd. which is extended for the cache configuration by the ARM cache simulator. This cache simulator also computes the exact number of cache misses.

Based on the instruction trace, the energy profiler calculates the total amount of energy consumed for the different processor instructions and memory accesses.

5. RESULTS

For the experiments, different benchmarks were selected. First, the sorting algorithms *bubble sort*, *heap sort* and *quick*

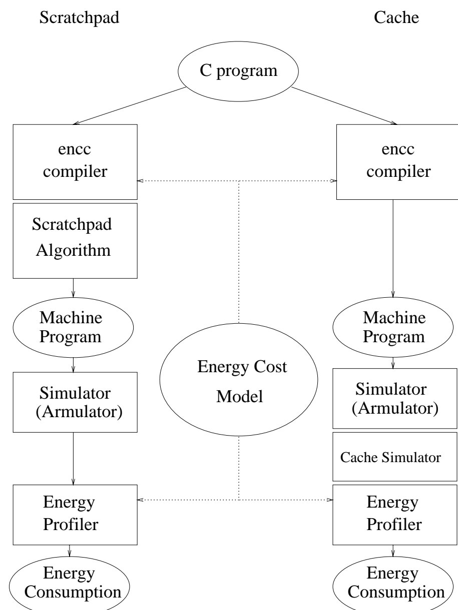

Figure 4: experimental workflow

sort are studied. Secondly, an application from the DSP-stone benchmark suite, *biquad_N_sections*, and a filter application, *lattice*, were chosen and finally, a matrix multiplication program, *matrix mult*. To show the behavior for a larger application with more than one hotspot, a combination of different sorting algorithms *multi sort* was used.

The results are presented in Table 2 for realistic ratios of approximately up to 1:10 between the onchip size and the total program size. We can state a clear advantage for the scratchpad system against the cache system. For the smallest onchip sizes the difference is higher because the cache miss rate is also higher.

Only the *biquad_N_sections* shows a better value for a cache with a size of 256 Bytes. This occurs due to the small number of executions of the inner loops of the benchmark. The dynamic copying of blocks by executing load and store instructions consumes much more energy than filling a cache line. Therefore, the advantage of the scratchpad is less compared to the cache in such cases.

The improvements are up to 55% with an average of 29.9%. The values for the scratchpad can be further improved by also copying data. This will be part of the future work.

Another series of experiments has been performed for performance evaluation (Table 3). Even here, a clear advantage for the scratchpad system can be stated. The performance improves up to 40% with an average of 25.2%.

Additionally, in figure 5 the results for the larger benchmark *multisort* are shown for varying onchip sizes. Furthermore, we present the comparison against the static approach [15]. It can be seen that the limitation of the static approach for different hotspots is solved by the dynamic exchange of program objects with an improvement of up to 38%. Depending on the benchmark, it is expected that the advantage of the dynamic approach will be lower in the average case.

The running time of the ILP solver for the results presented above is less than 50 ms. The complexity of the problem and the number of program objects depends on the number of nested loops and the number of included basic blocks. For larger programs with the same loop depth and size of the individual loops, the runtime increases in a linear way.

Table 2: *energy of scratchpad against cache*

Benchmark	onchip Bytes	cache (μJ)	scratch-pad (μJ)	saving (%)
bubble sort	64	4130	2064	50
bubble sort	128	2242	1727	23
heap sort	64	1677	1053	37
heap sort	128	1223	786	36
heap sort	256	673	649	3.5
quick sort	64	307	183	40
quick sort	128	259	168	35
quick sort	256	161	135	16
biquad	64	24.7	24.1	2.4
biquad	128	22.3	23.0	-3.1
lattice	64	3129	2210	29
lattice	128	2188	983	55
lattice	256	1662	942	43
matrixmult	64	93.6	43.8	53
matrixmult	128	59.2	42.1	29
average				29.9

Table 3: *performance of scratchpad against cache*

Benchmark	onchip Bytes	cache (cycles)	scratch-pad (cycles)	saving (%)
bubble sort	64	367,049	247,048	33
bubble sort	128	248,493	216,141	13
heap sort	64	135,134	86,227	36
heap sort	128	106,287	72,794	32
heap sort	256	72,315	68,897	4.7
quick sort	64	23,610	14,707	38
quick sort	128	20,517	14,958	27
quick sort	256	14,385	12,910	10
biquad	64	2,085	1,885	9.6
biquad	128	1,926	1,947	-1.1
lattice	64	264,282	164,432	38
lattice	128	203,516	120,594	40
lattice	256	169,602	118,885	30
matrixmult	64	7,507	4,180	44
matrixmult	128	5,337	4,057	24
average				25.2

6. CONCLUSION

In this paper an algorithm was presented for copying instructions into a scratchpad memory during program execution. The possible exchange of instructions improves the energy consumption against a cache system by an average of 29.9%. Furthermore, the performance is improved by an average of 25.2%. Compared to static approaches the energy is reduced by up to 38% for a sample benchmark.

We intend to evaluate our work with more complex benchmarks. Furthermore, the handling of data objects will be integrated into the algorithm.

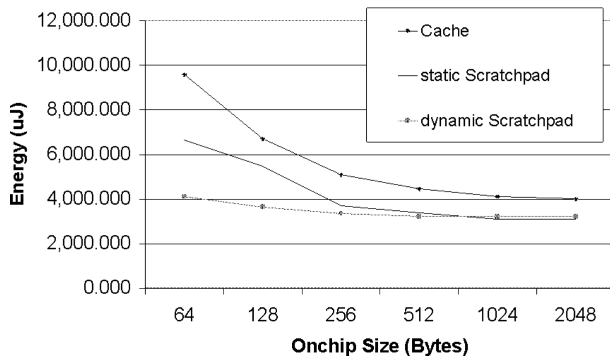

Figure 5: multisort program

7. REFERENCES

[1] ARM. *Advanced RISC Machines Ltd.* www.arm.com.

[2] R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, and P. Marwedel. Comparison of cache- and scratch-pad-based memory systems with respect to performance, area and energy consumption. Technical Report 762, University of Dortmund, Sep. 2001.

[3] 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 *Proc. of the 10th International Workshop on Hardware/Software Codesign*, CODES, Estes Park (Colorado), May 2002.

[4] L. Benini, A. Macii, E. Macii, and M. Poncino. Synthesis of application-specific memories for power optimization in embedded systems. In *Proc. of the 37th Design Automation Conference*, pages 300–303, Los Angeles, CA, Jun. 2000.

[5] encc. University of Dortmund, Computer Science Dep., ls12-www.cs.uni-dortmund.de/research/encc.

[6] Intel. Mobile Power Guidelines 2000. Technical Report 1.0, Intel Corporation, Dec. 1998.

[7] T. Ishihara and H. Yasuura. A power reduction technique with object code merging for application specific embedded processors. In *Proc. of the Design, Automation and Test in Europe Conference*, pages 617–623, Paris, Mar. 2000.

[8] M. Kandemir, J. Ramanujam, M. J. Irwin, N. Vijaykrishnan, I. Kadayif, and A. Parikh. Dynamic Management of Scratch-Pad Memory Space. In *Proc. of 38th Design Automation Conference*, pages 690–695, Las Vegas, NV, Jun. 2001.

[9] M. Kandemir, I. M. Vijakrishnan N., and W. Ye. Influence of compiler optimizations on system power. In *Proc. of the 37th Design Automation Conference*, pages 304–307, Los Angeles, CA, Jun. 2000.

[10] G. L. Nehmhauser and L. A. Wolsey. *Integer and Combinatorial Optimization*. John Wilsey and Sons, New York, NY, 1988.

[11] P. R. Panda, N. D. Dutt, and A. Nicolau. *Memory Issues in Embedded Systems-On-Chip*. Kluwer Academic Publishers, Norwell, MA, 1999.

[12] T. Simunic, L. Benini, and G. De Micheli. Cycle-accurate simulation of energy consumption in embedded systems. In *Proc. of the 36th Design Automation Conference*, pages 867–872, New Orleans, LA, Jun. 1999.

[13] J. Sjödin, B. Fröderberg, and T. Lindgren. Allocation of Global Data Objects in On-Chip RAM. In *Proc. Workshop on Compiler and Architectural Support for Embedded Computer Systems*, Washington DC, Dec. 1998. ACM.

[14] S. Steinke, M. Knauer, L. Wehmeyer, and P. Marwedel. An accurate and fine grain instruction-level energy model supporting software optimizations. In *PATMOS 01*, Yverdon-Les-Bains, Switzerland, Sep. 2001.

[15] S. Steinke, L. Wehmeyer, B. Lee, and P. Marwedel. Assigning program and data objects to scratchpad for energy reduction. In *Proc. of DATE*, Paris, France, Mar. 2002.

[16] V. Tiwari, S. Malik, and A. Wolfe. Compilation techniques for low energy: An overview. In *Proceedings of the IEEE Symposium on Low Power Electronics*, San Diego, CA, Oct. 1994.

[17] S. J. E. Wilton and N. P. Jouppi. CACTI: An enhanced cache access and cycle time model. *IEEE Journal of Solid-State Circuits*, 31(5):677–688, May 1996.