Assigning Program and Data Objects to Scratchpad for Energy Reduction

Stefan Steinke, Lars Wehmeyer, Bo-Sik Lee, Peter Marwedel
University of Dortmund
Computer Science 12
44221 Dortmund, Germany
{steinke, wehmeyer, lee, marwedel}@ls12.cs.uni-dortmund.de

Abstract

The number of embedded systems is increasing and a remarkable percentage is designed as mobile applications. For the latter, the energy consumption is a limiting factor because of today's battery capacities. Besides the processor, memory accesses consume a high amount of energy. The use of additional less power hungry memories like caches or scratchpads is thus common.

Caches incorporate the hardware control logic for moving data in and out automatically. On the other hand, this logic requires chip area and energy. A scratchpad memory is much more energy efficient, but there is a need for software control of its content.

In this paper, an algorithm integrated into a compiler is presented which analyses the application and selects program and data parts which are placed into the scratchpad. Comparisons against a cache solution show remarkable advantages between 12% and 43% in energy consumption for designs of the same memory size.[1]

1. Introduction

The number of mobile applications is steadily increasing. Many of them include processors which control their functionality, as well as memories for data storage, e.g. audio information. The limiting factors for mobile systems are their size, weight and their battery capacity. Improvements concerning battery capacity have been made, but the rate is low (for example, only a factor of two in the last 30 years for rechargeable Ni-Cd batteries [17]). Compared to this, the increase of energy consumption of common processors and memories is very much higher. Additionally, the heat which stems from the consumed energy has to be removed from the system. The size of some laptops cannot be reduced further because of the surface necessary for heat dissipation [5]. Due to this increasing gap, computer architects have the task to reduce the energy consumption to overcome this limitation and permit new applications or reduced size and weight of embedded systems.

Besides shrinking technology sizes, new hardware techniques are being developed. Today, the CMOS technology is mostly used, where the switching activity of a transistor is responsible for 70 to 90 % of the total energy consumption [16]. This energy consumption depends on the number of switching operations and the load capacity of the attached nets. Therefore, it is most promising to focus on the reduction of the switching activity.

Another way of energy reduction is to build up a memory hierarchy. The off-chip main memory is the slowest and the most energy consuming memory type. Energy E is the product of time t, current I and voltage V_{dd}. The time t is the number of cycles n multiplied with the cycle time T.

$$E = V_{dd} * I * t = V_{dd} * I * n * T$$

The low speed which leads to a high number of cycles n and the high amount of main memory accesses are the main reasons for the relatively high energy cost of the main memory. Additional memories like caches or scratchpads - smaller memories - are able to reduce the number of main memory accesses for frequently used instructions or variables. Caches are well known and included in many processor designs. Besides the data memory itself, they consist of two additional components. Firstly, a tag memory is required for the storage of valid addresses. Secondly, logic components are necessary for a fast comparison of addresses with the contents of the tag memory, so that cache hits and misses can be detected. These memory parts are all energy consuming since accesses to the tag array and the comparisons are performed during each memory access.

The advantage of caches is the easy integration with the software of the system. The detection of a cache hit or miss is done automatically. If the accessed data is currently not available in the cache, the hardware control automatically copies the data into the cache. This mechanism allows the

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

use of software without any adaption to the changed memory hierarchy. A disadvantage of caches occurs in realtime embedded systems where a certain response time has to be guaranteed. For the worst case execution time (WCET), a cache miss has to be assumed meaning that the WCET does not benefit from the presence of a cache. Scratchpad using static assignment may be considered to improve WCET. However, because the tag array and comparators are not necessary, the software has to be adapted in order to control the filling of the scratchpad depending on the executed software.

In this work, a compiler extension is presented which analyses the most frequently executed instructions and accessed variables. The best set of instructions and variables is then identified using integer linear programming [9] and the selected objects are placed in the scratchpad. Possible program parts are functions or basic blocks (sequentially executed instructions without a jump).

In the next section we describe related research work. In section 3 a memory model of cache and scratchpad memories is presented together with their energy costs. The algorithm for identification and selection of program parts and variables is presented in section 4.

The experimental environment used for our simulations is described in section 5 and the presentation of the results of the comparison for different memory sizes in section 6.

The conclusion of this work is given in section 7.

2. Related Work

The optimization of energy consumption by changing the software has been a research topic for nearly ten years. One of the first energy models was published by Tiwari et al. [18][19]. For each processor instruction, a certain energy amount called *base cost* was determined and a change from the execution of one instruction to another was named *interinstruction cost*. With several measurement series a database was built up and was used for different optimizations which showed an energy reduction of up to 40%. This energy model can be used especially for the use in compilers, because the compiler can rate every selected instruction and can thus choose the most energy efficient one.

The limitation of this model is its lack of taking other system components into account. Especially for low power processors, the energy consumption of the memory accesses must not be neglected. Explicitly modeling memory accesses is useful since the compiler can then take the costs of memory accesses into account. If this is not done, the generated code can be optimal for the processor energy but not for the whole system including memory.

Simunic et al. [13] presented a different approach where the processor and memory energy consumption is based on the manufacturer's data sheet. Simunic distinguishes between the energy consumption in the active and the idle state. This model is used for simulation and estimation of the energy consumption of complete systems. The potential of optimization was shown by Kandemir et al. [8], who studied several compiler optimizations and caches with different organizations and their impact on the energy consumption.

To overcome the limitation of the energy model of Tiwari, Steinke et al. [15] presented an energy model which was developed especially for optimizing the bus encoding and therefore takes the state of each bus line as well as the consumption of the different memories into account.

The efficient use of the memory hierarchy was presented by Panda et al. [10][11] who analyzed the accesses to variables and chose a set of variables to be placed within the scratchpad memory. A further approach by Sjödin et al. [14] places some variables into the scratchpad, based on a static analysis, showing that this is sufficiently precise and no dynamic analysis is needed. A further power reduction technique by Ishihara et al. [6] merges frequently executed sequences of object codes.

Whereas these approaches are based solely on a software solution, Benini et al. [3] generated application-specific memories which are scratchpad memories with an additional decoder for distinguishing between a hit and a miss. These memories show a significant improvement concerning energy consumption of 12% to 68% compared to caches.

For a comparison of scratchpad and cache memory, detailed values of the energy consumption of both architectures are necessary. This work was done by Wilton et al. [20][21] who presented a cache model named CACTI. This model was also used for our evaluation of the cache. To compare it to a scratchpad of the same technology, we used the values for the data memory array of the cache, ignoring the energy consumption of the tags and comparators. The detailed model was described by Banakar et al. [2]. The following section describes these memory models in depth.

3. Memory Models

For this work, two systems were compared, one with a cache and the second with a scratchpad memory. The latter was used together with the algorithm presented here for the allocation of program segments and of variables.

3.1 Cache model

Both cache and scratchpad comprise a data memory array, the data column multiplexers, the data sense amplifiers and the data output driver. Additionally, the cache requires a decoder, tag memory array, tag column multiplexers, tag sense amplifiers and tag output drivers. The processor we used to analyse the memory interface is the ARM7T processor [1], which is an ultra low power processor with a 16 bit Instruction Set for low power applications. The number of accesses for the different access types are presented in table 1. For a read hit, one cache read is executed. If the requested data is not available in the cache (read miss), a read of the main memory consisting of L words (=blocksize) and a corresponding write into the cache have to be executed before the cache read. The architecture uses write-through and therefore executes a write into the cache as well as the main memory following a write hit. A write miss is recognized by a cache read and followed by a main memory write.

For comparison with the available ARM7T processor, a 4-way set associative organization was selected with a write-through architecture and a block size of 8 bytes.

Table 1. *number of accesses for cache system*

Access Type	cache read	cache write	main memory read	main memory write
read hit	1	0	0	0
read miss	1	L	L	0
write hit	0	1	0	1
write miss	1	0	0	1

3.2 Scratchpad model

The scratchpad memory uses software to control the location assignment of data. As a design comparable to a scratchpad memory, we can use the cache architecture presented in the previous subsection without the tag memory array, tag column multiplexer, tag sense amplifier and tag output drivers. The energy consumption of this subset of the cache can be calculated using the CACTI cache model. This results in a fair comparison because both memories are designed using the same technology and design.

The kind of memory access from the processor depends on the selected address. Accesses to the scratchpad address space require only 1 cycle and no wait state. Accesses to the main memory depend on the data width and cause 1 or 3 wait states (c.f. table 2).

Table 2. *processor cycles for scratchpad system*

Access Type	number of cycles
scratchpad	1 cycle
main memory 16 bit	1 cycle + 1 wait state
main memory 32 bit	1 cycle + 3 wait states

3.3 Energy values

For the calculation of the energy consumption of cache and scratchpad accesses, the CACTI model was used. This model determined the values shown in table 3 for a 0.5μm technology memory which is the technology used for the ARM7T. The memory sizes vary from 64 bytes to 2048 bytes. For 64 bytes and 128 bytes, the values were obtained by approximation because CACTI does not support these small memory sizes. The compared scratchpad always shows lower energy values as expected because it consists of a subset of the cache. In the last column, the ratio between cache and scratchpad energy consumption is calculated. For example, 1 single cache access consumes nearly the same amount of energy as 4 scratchpad accesses for a memory size of 4 KByte.

Table 3. *energy consumption of memories*

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
4096 bytes	4.71 nJ	1.21 nJ	3.9
8192 bytes	5.39 nJ	2.07 nJ	2.6

4. Algorithm

In this section we explain how the algorithm within the compiler assigns a set of memory objects (functions, basic blocks and variables) to the scratchpad on a static basis. First we describe the identification and evaluation of instructions. Then, the identification and evaluation of variables and finally the selection of the best set of memory objects is explained.

4.1 Program memory objects

The execution of each function is started at its beginning and is terminated by a return statement. There are no further jumps into a function. Thus, each function can be handled as one memory object which can possibly be moved into the scratchpad and which requires neither changing any of the included instructions nor the corresponding function call.

To compute the energy consumption of a function, i, we sum up the product of the number of executions m_k of each instruction k within function i with the energy consumption of a single instruction fetch E_{instr_fetch}:

$$E(F_i) = \sum_k m_k * E_{instr\_fetch}$$

A function can be decomposed into basic blocks which can also be treated as program memory objects.

Moving basic blocks instead of complete functions requires the addition of jump instructions to jump from blocks mapped to regular memory to blocks stored in the scratchpad and back. The jump instructions are an overhead especially if there is a number of small basic blocks. In order to minimize jump instructions, moving consecutive basic blocks is preferred.

For each basic block j, we can compute the energy consumption by multiplying the number of instructions m, the number of executions of this basic block n_j and the energy consumption of a single instruction fetch E_{instr_fetch}. Additionally, we have to add the energy costs for l jumps from main memory to scratchpad or vice versa.

$$E(BB_j) = m * n_j * E_{instr\_fetch} + l * E(jump)$$

Note that the program is statically allocated to the scratchpad memory. There is no dynamic reloading of memory blocks even though this could be useful for long programs having more hot spots than the scratchpad can accomodate. The extension to dynamic reloading (a kind of program-controlled overlay) is part of our future work.

4.2 Data memory objects

Apart from the program, variables can also be allocated to scratchpad. Each variable is viewed as one data memory object. This is limited to global scalar and non scalar variables because local variables may exist as multiple instances in recursive functions. The number of accesses to a global variable $acc(v)$ is the sum of the number of accesses $acc_i(v)$ in each of the blocks i. $acc_i(v)$ is computed as the number of static references $stat_i(v)$ to variable v in block i multiplied by the number of executions n_i of block i:

$$acc(v) = \sum_i acc_i(v) = \sum_i stat_i(v) * n_i$$

For the energy $E(v)$ consumed by all accesses to the variable, this number of accesses has to be multiplied by the energy cost E_{data} of a single memory access with a load or store instruction:

$$E(v) = acc(v) * E_{data}$$

4.3 Selection of memory objects

The best set of memory objects which fits into the scratchpad and saves the highest amount of energy now has to be identified. Moving a certain memory object to the scratchpad will result in a certain gain in terms of saved energy. Moving has to be done such that the combined size of the memory objects does not exceed the size of the scratchpad. The size of each memory object is independent of the other objects. Maximizing the total gain can be formulated as a knapsack problem [12].

Our formulation of the problem uses the following definitions for moving functions F, basic blocks BB and variables var with $x \in F \cup BB \cup var$:

$$\begin{aligned} E(x) &= \text{saved energy consumption for } x \\ S(x) &= \text{size of } x \end{aligned}$$

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

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

$$\begin{aligned} sav = & \sum_{i \in I} m(F_i) * E(F_i) + \\ & \sum_{j \in J} m(BB_j) * E(BB_j) + \\ & \sum_{k \in K} m(var_k) * E(var_k) \end{aligned}$$

Index sets I, J, and K correspond to index values for functions, all basic blocks and all variables, respectively.

The size constraint can be modeled as follows:

$$\begin{aligned} \sum_{i \in I} m(F_i) * S(F_i) & \quad + \\ \sum_{j \in J} m(BB_j) * S(BB_j) & \quad + \\ \sum_{k \in K} m(var_k) * S(var_k) & \quad \leq \quad scratchpadsize \end{aligned}$$

Up to this point, two consecutive basic blocks moved to the scratchpad are both counted with a jump to the scratchpad and a further jump back. The jumps can be omitted between these two basic blocks. To model this, memory objects - so called multi basic blocks - are generated for all possible combinations of consecutive basic blocks.

To prevent a basic block x from being selected twice, e.g. as a single basic block x and also as part of a multi memory block i or a function j, equations of the following type have to be added:

$$m(BB_x) + m(F_i) + \sum_{\substack{j \in J \\ j \neq x}} m(BB_j) \leq 1$$

Index set J corresponds to index values for all multi memory blocks which include basic block BB_x.

Based on the above equations, an IP solver [9] can find the optimal solution for the given cost function for the use of a scratch pad memory. The chosen memory objects can then be placed in the scratch pad memory.

5. Experimental Setup

For the evaluation of the two memory configurations, the ARM7T processor mentioned above with an onchip cache was compared to a version with a scratchpad configuration. For the two memories, the energy data was estimated using the CACTI tool. For the processor and main memory current the data is not available from the manufacturer and therefore a series of measurements of a real ARM board was taken based on the energy model presented by Steinke et al. [15].

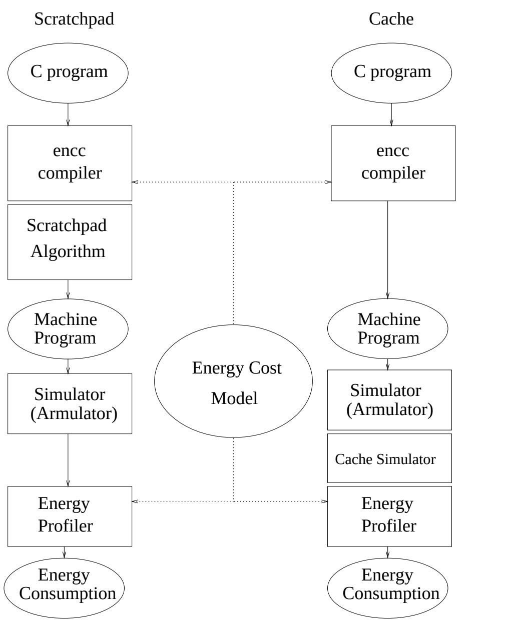

Figure 1. Flow of memory comparison

The diagram in figure 1 shows the work flow starting with programs compiled using the energy aware C compiler encc [4]. For the scratchpad configuration, the algorithm presented in the previous section is executed. The generated machine code is simulated by the simulator from ARM Ltd. which is extended for the cache configuration by the ARM cache simulator. Based on the instruction trace, the energy profiler calculates the total amount of energy consumed for the different processor instructions and memory accesses.

6. Results

The work flow was used to compare different benchmarks such as sorting algorithms, two filter applications and one media application.

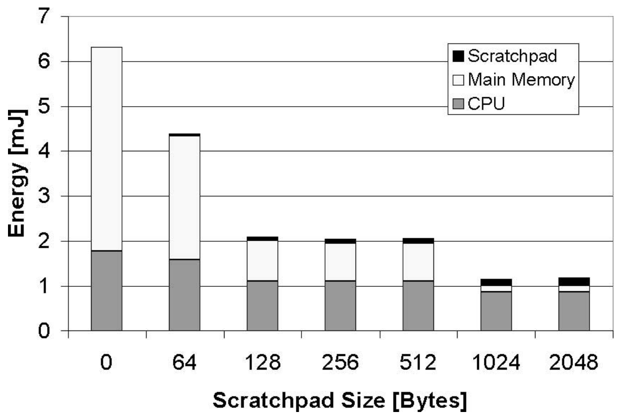

Figure 2. Bubble_sort: CPU, Main Memory, Scratchpad Energy

The results in figure 2 show the effect of the use of the scratchpad memory for different memory sizes. It can be seen that the main memory energy decreases and the scratchpad energy increases. The observed benchmark $bubblesort$ uses 196 bytes program memory and 436 bytes data memory without scratchpad memory or cache.

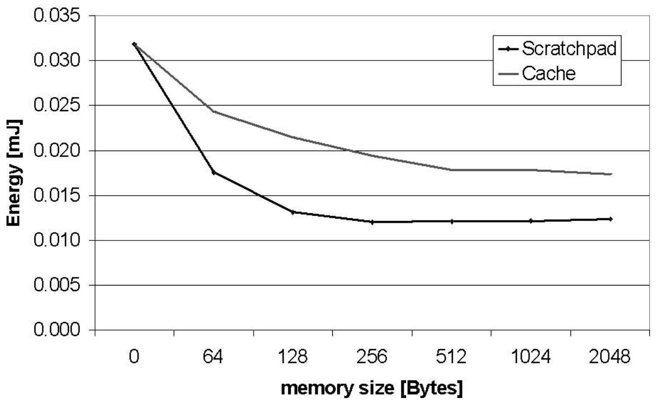

Figure 3. biquad: Cache vs. Scratchpad

For the comparison with a common cache system the curves for cache vs. scratchpad are presented in figures 3, 4 and 5. There is a clear trend in favor of the scratchpad which consumes less energy than a cache memory of the same size. This is not really fair because the area required for a cache is much bigger than the area for a scratchpad. Area is the important factor for the production cost. Therefore it would be more realistic to compare a scratchpad with a cache of the same area which means with less capacity. This leads to even higher advantages for the scratchpad. Here we present only the data for the same memory size. The comparison for different benchmarks and 2 KByte memory

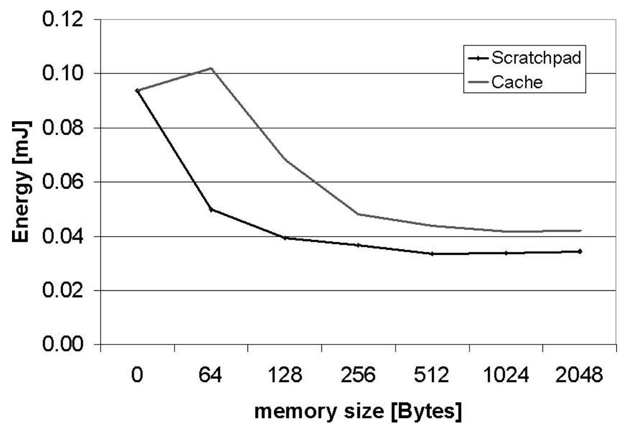

Figure 4. matrixmult: Cache vs. Scratchpad

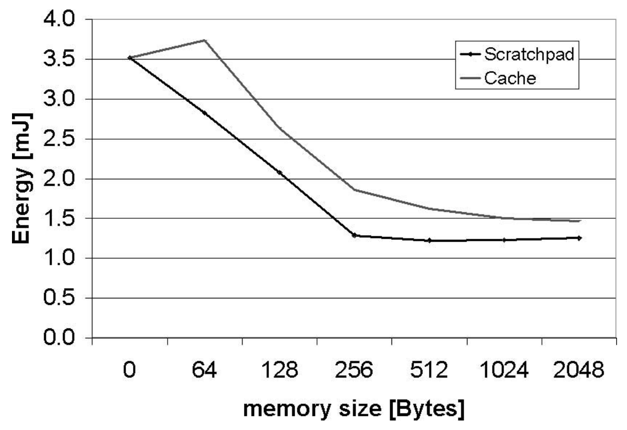

Figure 5. lattice: Cache vs. Scratchpad

Table 4. *cache vs. scratchpad energy [nJ]*

benchmark	cache	scratchpad	improv.
biquad_N_sections	17,361	12,361	18%
bubblesort	1,913,242	1,191,574	38%
heapsort	598,191	491,897	18%
insertionsort	965,170	661,809	31%
lattice	1,467,450	1,252,753	15%
matrixmult	41,981	34,375	18%
me_ivlin	4,558,811	2,610,799	43%
quicksort	75,153	66,054	12%
selectionsort	1,090,276	911,720	16%
average			23%

Table 5. *cache vs. scratchpad performance [cycles]*

benchmark	cache	scratchpad	improv.
biquad_N_sections	1,656	1,268	23%
bubblesort	241,458	191,953	21%
heapsort	74,343	64,918	13%
insertionsort	119,191	95,783	20%
lattice	165,886	141,402	15%
matrixmult	4,487	3,687	18%
me_ivlin	646,024	497,314	23%
quicksort	8,240	7,652	7%
selectionsort	153,514	142,498	7%
average			16%

shows an energy reduction between 12% and 43% with an average of 23% (c.f. table 4). The increase for the cache configuration at 64 bytes in figures 4 and 5 is caused by a high rate of cache misses.

Even for performance (c.f. table 5), there is an improvement between 7% and 23% with an average of 16%. This improvement is less than the energy results because the energy consumption of main memory accesses is much higher than scratchpad accesses. This effect is additional to the reduction of the number of cycles and concerns only the energy values.

The IP solver as part of the scratchpad algorithm determines the optimal set of memory objects. In our benchmarks the average runtime of the solver is less than 0.1s. To overcome inacceptable runtimes, a timelimit can be chosen which ensures finding a solution sufficiently close to the optimum.

7. Conclusion

The presented algorithm as part of a compiler selects program parts and variables and places them into a scratchpad memory. The ILP model delivers an optimal solution and saves about 22% of the electrical energy compared to a cache. Future work will improve these results by also considering the stack and by dynamically moving memory objects in and out of the scratchpad. Furthermore, research can be done for the extension of this approach to multitasking systems.

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] 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.

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

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

[6] 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, France, Mar. 2000.

[7] 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.

[8] 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.

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

[10] P. R. Panda, N. D. Dutt, and A. Nicolau. Efficient utilization of scratch-pad memory in embedded processor applications. In *Proc. of European Design and Test Conference*, Paris, France, Mar. 1997.

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

[12] R. Sedgewick. *Algorithms*. Addison Wesley, Massachusetts, 1988.

[13] 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.

[14] 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.

[15] 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.

[16] Synopsys. *Power Products Reference Manual V3.5*. synopsys, 1996.

[17] V. Tiwari. *Logic and System Design for Low Power Consumption*. PhD Thesis, Princeton University, Princeton, NJ, 1996.

[18] 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.

[19] V. Tiwari, S. Malik, and A. Wolfe. Instruction level power analysis and optimization of software. *Journal of VLSI Signal Processing Systems*, Aug. 1996.

[20] S. J. E. Wilton and N. P. Jouppi. An Enhanced Access and Cycle Time Model for On-Chip Caches. Technical Report 93/5, Western Research Laboratory, Jul. 1994.

[21] 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.