Cache-Aware Scratchpad Allocation Algorithm

Manish Verma, Lars Wehmeyer, Peter Marwedel
Department of Computer Science XII
University of Dortmund
44225 Dortmund, Germany
{Manish.Verma, Lars.Wehmeyer, Peter.Marwedel}@uni-dortmund.de

Abstract

In the context of portable embedded systems, reducing energy is one of the prime objectives. Most high-end embedded microprocessors include onchip instruction and data caches, along with a small energy efficient scratchpad. Previous approaches for utilizing scratchpad did not consider caches and hence fail for the au courant architecture. In the presented work, we use the scratchpad for storing instructions and propose a generic Cache Aware Scratchpad Allocation (CASA) algorithm. We report an average reduction of 8-29% in instruction memory energy consumption compared to a previously published technique for benchmarks from the Mediabench suite.

The scratchpad in the presented architecture is similar to a preloaded loop cache. Comparing the energy consumption of our approach against preloaded loop caches, we report average energy savings of 20-44%.

1. Introduction

Contemporary Embedded Systems have to satisfy stringent constraints concerning power, performance and cost. For mobile embedded devices, reduced energy consumption translates to either increased battery life or reduced dimensions, weight and cost of the device or both. Consequently, the overall competitiveness of the product is improved.

Several researchers [4, 11] have identified the memory subsystem as the energy bottleneck of the entire system. Onchip instruction and data caches were introduced to improve the performance of computer systems by exploiting the available locality in the program. Caches allow easy integration and improve both performance and energy of programs without the need for program analysis and optimization. However, caches are not the most energy efficient option available for embedded systems. Scratchpad memories were proposed as an energy efficient alternative to caches. They also require less onchip area and allow tighter bounds on WCET prediction of the system. However unlike caches, scratchpads require explicit support from the compiler. To strike a balance between these two contrasting approaches, most of the high-end embedded microprocessors (e.g. ARM10E [1], ColdFire MCF5 [9]) include both onchip caches and a scratchpad.

We assume the memory hierarchy as shown in figure 1.(a) and utilize the scratchpad for storing instructions. The decision to store only instructions is motivated by the fact that the instruction memory is accessed on every instruction fetch and the size of programs for mobile embedded devices is smaller compared to their data size. This implies that smaller scratchpad memories allocated with instructions can achieve greater energy savings than with data. In this paper, we model the cache behavior as a conflict graph and allocate objects onto the scratchpad considering their effect on the I-cache. As shown later, the problem of finding the best set of objects to be allocated on the scratchpad can be formulated as a variant of the Maximum Independent Set problem. The problem is solved using an ILP approach. We compare our approach against a published technique [13] for the aforementioned architecture. Due to the presence of an I-cache in our architecture, the latter fails to produce optimal results and may even lead to the problem of *cache thrashing*.

We also compare the energy savings due to our approach for scratchpad to that achieved by preloaded loop caches [12], as the utilization of the scratchpad in the current setup (see figure 1) is similar to a loop cache. Preloaded loop caches are architecturally more complex than scratchpads, but are less flexible as they can be preloaded with only a limited number of loops. We demonstrate that with the aid of a sophisticated allocation algorithm, scratchpad memories can outperform their complex counterparts.

In the next section, we describe related work and detail the shortcomings of previous approaches. Section 3 describes the information regarding memory objects, cache behavior and the energy model. The proposed algorithm is presented in detail in section 4, followed by the description of the experimental setup. In section 6 we present the results for an ARM7T based system and end the paper with a conclusion and future work.

1530-1591/04 $20.00 (c) 2004 IEEE

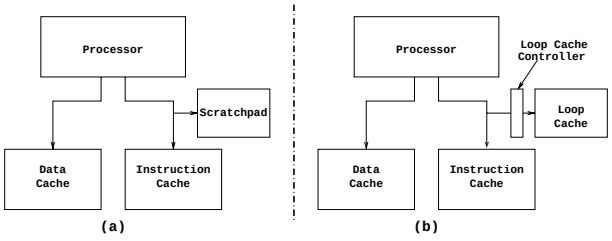

Figure 1. System architecture: (a) scratchpad (b) loop cache

2. Related Work

Analytical energy models for memories [7] have been found to be fairly accurate. We use *cacti* [15] to determine the energy per access for caches and preloaded loop caches. The energy per access for scratchpad memories was determined using the model presented in [3].

Application code placement techniques [10, 14] were developed to improve the CPI (cycles per instruction) by reducing the number of I-cache misses. During the first step, traces were generated by combining the frequently executed basic blocks, followed by the trace placement step. Authors in [10] placed traces within the function boundary, while [14] placed them across function boundaries, to reduce the I-cache misses.

Several researchers [2, 11] have utilized scratchpad memories for assigning global/local variables, whereas only Steinke et al. [13] considered both program and data parts (memory objects) to be allocated onto the scratchpad. They assumed a memory hierarchy composed of only scratchpad and main memory. Profit values were assigned to program and data parts according to their execution and access counts, respectively. They then formulated a knapsack problem to determine the best set of memory objects to be allocated on the scratchpad.

Although the aforementioned approach is sufficiently accurate for their memory hierarchy, it is fairly imprecise for the current setup. The first imprecision of the approach stems from the assumption that execution (access) counts are sufficient to represent energy consumption by a memory object. This assumption fails in the presence of a cache, where execution (access) counts can be decomposed into cache hits and misses. The energy consumption of a cache miss is significantly larger than that of a cache hit. Consequently, two memory objects can have the same execution (access) counts, yet have substantially different cache misses and hence the energy consumption. The above discussion stresses the need for a fine-grained energy model. The second imprecision is due to the fact that conflict relationship between memory objects is not modeled and hence they are moved instead of copying from main memory to the scratchpad. As a result, the layout of the entire program is changed, which may cause non-conflicting memory objects to conflict with each other and lead to erratic results.

Various instruction buffers have been proposed to improve the energy consumption of the system. Ross et al. [12] proposed a Preloaded Loop Cache which can be statically loaded with pre-identified memory objects. Start and end addresses of the memory objects are stored in the controller, which on every instruction fetch determines whether to access the loop cache or the L1 I-cache. Consequently, the loop cache can be preloaded with complex loops as well as functions. However, to keep the energy consumption of the controller low, only a small number of memory objects (typically 2-6) can be preloaded.

The first disadvantage of the aforementioned approach is due to the architectural feature of the loop cache that allows only a fixed number of memory objects to be preloaded. The problem will get prominent for large programs with several hot spots. The second disadvantage is similar to the one explained above for [13], memory objects are greedily selected on the basis of their execution time density (execution time per unit size). In the wake of the discussion we enumerate the following contributions of this paper.

- It for the first time studies the effect of a scratchpad and an I-cache on the system's energy consumption.
- It stresses the need for a sophisticated allocation algorithm by demonstrating the inefficiency of previous algorithms when applied to the present architecture.
- It presents a novel scratchpad allocation algorithm which can be easily applied to any memory hierarchy.
- It demonstrates that scratchpad together with an allocation algorithm can replace loop caches.

In the following section, we describe preliminary information required for our algorithm.

3. Preliminaries

We start with describing the assumed architecture for the current research work, followed by the description of the memory objects. The interaction of memory objects within the cache is represented using a conflict graph, which forms the basis of the proposed energy model and the algorithm.

3.1. Architecture

For the presented research work we assume a Harvard architecture (see figure 1(a)) with the scratchpad present at the same horizontal level as the L1 I-cache. The scratchpad is mapped to a region in the processor's address space and acts as an alternative location for fetching instructions. As shown in figure 1(b), the preloaded loop cache setup is similar to using a scratchpad.

3.2. Memory Objects

In the first step of our approach we generate traces and then distribute these traces between offchip main memory and non-cacheable scratchpad memory. A *trace* is a

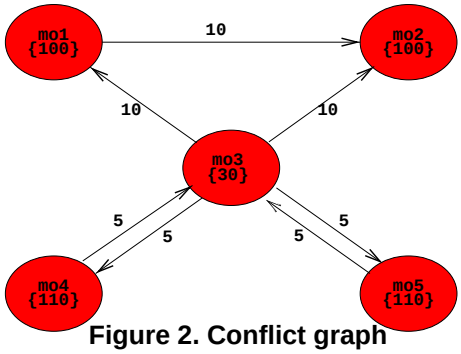

Figure 2. Conflict graph

frequently executed straight-line path, consisting of basic blocks connected by fall-through edges. Our traces are similar to the traces in [14] except they are smaller than the scratchpad size, as larger traces can not be placed on to the scratchpad as whole. The traces are appended with NOP instructions to align them to cache line boundaries. Consequently, a trace will start and end at a cache line, ensuring a *one-to-one relationship* between cache misses and corresponding traces. The rational behind using traces is threefold. Firstly, traces improve the performance of both the cache and the processor by enhancing the spatial locality in the program code. Secondly, due to the fact that traces always end with an unconditional jump [14], they form an atomic unit of instructions which can be placed anywhere in memory without modifying other traces. Finally, traces are accountable for every cache miss caused by them. In the rest of the paper, unless specified, traces will be referred to as memory objects (MO). In the following subsection, we represent the cache behavior at the granularity of memory objects by a conflict graph.

3.3. Cache behavior (conflict graph)

The cache maps an instruction to a cache line according to the following function:

$$Map(addr) = addr \,\mathbf{mod}\, \frac{CacheSize}{Associativity * WordsPerLine}$$

Similarly, a memory object is mapped to cache line(s) depending upon its start address and size. Two memory objects potentially conflict if they are mapped to at least one common cache line. Conflict cache misses can only be caused by conflicting memory objects and can be represented by a conflict graph. The Conflict Graph G (see figure 2) is defined as follows:

Definition: The *Conflict Graph* $G = (X, E)$ is a directed weighted graph with node set $X = \{x_1, \ldots, x_n\}$. Each vertex x_i in G corresponds to a memory object (MO) in the application code. The edge set E contains an edge e_{ij} from node x_i to x_j if a cache-line belonging to x_j is replaced by a cache-line belonging to x_i using the cache replacement policy. In other words, $e_{ij} \in E$ if there occurs a cache miss of x_i due to x_j. The weight m_{ij} of the edge e_{ij} is the number of cache misses of x_i that occur due to x_j. The weight f_i of a vertex x_i is the total number of instruction fetches within x_i.

Conflict graph as shown in figure. 2 is a directed graph because the conflict relationship determined by any of the cache replacement policies is *antisymmetric*. The vertices and the edges are marked with the corresponding weights determined by profiling the application. The conflict graph G and the energy values are utilized to compute the energy consumption of a memory object according to the energy model proposed in the following subsection.

3.4. Energy Model

The energy $E(x_i)$ consumed by an MO x_i is expressed as:

$$E(x_i) = \begin{cases} E_{SP}(x_i) & \text{if } x_i \text{ is present on scratchpad} \\ E_{Cache}(x_i) & \text{otherwise} \end{cases} \quad (1)$$

where E_{Cache} can be computed as follows:

$$\begin{aligned} E_{Cache}(x_i) &= Hit(x_i) * E_{Cache\_hit} \\ &+ Miss(x_i) * E_{Cache\_miss} \end{aligned} \quad (2)$$

where functions $Hit(x_i)$ and $Miss(x_i)$ return the number of hits and misses, respectively, while fetching the instructions of MO x_i. E_{Cache_hit} is the energy of a hit and E_{Cache_miss} is the energy of a miss in the I-cache.

$$\begin{aligned} Miss(x_i) &= \sum_{x_j \in N_i} Miss(x_i, x_j) \quad \text{with} \\ N_i &= \{x_j : e_{ij} \in E\} \end{aligned} \quad (3)$$

where $Miss(x_i, x_j)$ denotes the number of conflict cache misses of MO x_i caused due to conflicts with MO x_j. The sum of the number of hits and misses is equal to the number of instruction fetches f_i in an MO x_i:

$$f_i = Hit(x_i) + Miss(x_i) \quad (4)$$

For a given input data set, the number of instruction fetches f_i within an MO x_i is a constant and is independent of the memory hierarchy. Substituting the terms $Miss(x_i)$ from equation (3) and $Hit(x_i)$ from equation (4) in equation (2) and rearranging derives the following equation:

$$\begin{aligned} E_{Cache}(x_i) &= f_i * E_{Cache\_hit} + \\ &\sum_{x_j \in N_i} Miss(x_i, x_j) * (E_{Cache\_miss} - E_{Cache\_hit}) \end{aligned} \quad (5)$$

Observing the above equation, we find that the first term is a constant while the second term is variable which depends on the overall program code layout and the memory hierarchy. We would like to point out that the approach [12] only considered the constant term in its energy model. Consequently, could not optimize the overall memory energy consumption.

Since there are no misses when an MO x_i is present in the scratchpad, we can deduce the following energy equation:

$$E_{SP}(x_i) = f_i * E_{SP\_hit} \quad (6)$$

where E_{SP_hit} is the energy per access of the scratchpad.

4. Algorithm

Once we have created the conflict graph G annotated with vertex and edge weights, energy consumption of the memory objects can be computed. Now, the problem is to select a subset of memory objects which minimize the number of conflict edges and the overall energy consumption of the system. The subset is bounded in size by the scratchpad size. In the simplest form, when every node and edge has a unit weight, the problem is reduced to finding a Maximum Independent Set [6]. Unfortunately, even in the simplest form the problem is NP-complete [6]. We will present an *Integer Linear Programming* ILP based solution, as it can be easily extended to handle complex memory hierarchies and requires an acceptable computation time using a commercial ILP solver. Moreover, the problem can be elegantly represented using a set of inequations.

In order to explain the algorithm we need to define a number of variables. The binary variable $l(x_i)$ denotes the location of the memory object in the memory hierarchy:

$$l(x_i) = \begin{cases} 0, & \text{if } x_i \text{ is present on scratchpad} \\ 1, & \text{otherwise} \end{cases} \quad (7)$$

$Miss(x_i,x_j)$ is the number misses of MO x_i caused due to conflict with MO x_j. Since a memory object present in the scratchpad does not conflict with other memory objects, we can represent $Miss(x_i,x_j)$ as follows:

$$Miss(x_i,x_j) = \begin{cases} 0, & \text{if } x_j \text{ is present on scratchpad} \\ m_{ij}, & \text{otherwise} \end{cases} \quad (8)$$

where m_{ij} is the weight of the edge e_{ij} connecting vertex x_i to x_j. Function $Miss(x_i,x_j)$ can be reformulated using the location variable $l(x_j)$ and represented as:

$$Miss(x_i,x_j) \quad = \quad l(x_j) * m_{ij} \quad (9)$$

Similarly, the location variable $l(x_i)$ can be used to reformulate the energy equation (1) denoting the energy consumed by the memory object.

$$E(x_i) = [1 - l(x_i)] * E_{SP}(x_i) + l(x_i) * E_{Cache}(x_i) \quad (10)$$

We substitute the energy equations for E_{Cache} and E_{SP} from equations (5) and (6), respectively, into the above equation. By rearranging the terms we transform the equation (10) into the following form.

$$\begin{aligned} E(x_i) = \quad & f_i * E_{SP\_hit} + \\ & f_i * [E_{Cache\_hit} - E_{SP\_hit}] * l(x_i) + \\ & [E_{Cache\_miss} - E_{Cache\_hit}] * [\sum_{j \in N_i} l(x_j) * l(x_i) * m_{ij}] \end{aligned} \quad (11)$$

Observing the above equations, we find the last term is a quadratic degree term. This can be justified by the fact that the number of misses of a memory object x_i not only depends upon its location but also upon the location of the conflicting memory objects x_j. Prior to formulating an ILP problem, we need to linearize the above equation. This can be achieved by replacing the expression $l(x_i) * l(x_j)$ by an additional variable $L(x_i,x_j)$ in the following equation.

$$\begin{aligned} E(x_i) = \quad & f_i * E_{SP\_hit} + \\ & f_i * [E_{Cache\_hit} - E_{SP\_hit}] * l(x_i) + \\ & [E_{Cache\_miss} - E_{Cache\_hit}] * [\sum_{j \in N_i} L(x_i,x_j) * m_{ij}] \end{aligned} \quad (12)$$

In order to prevent the linearizing variable $L(x_i,x_j)$ from taking arbitrary values, the following linearization constraints are added to the set of constraints.

$$l(x_i) \quad - \quad L(x_i,x_j) \geq 0 \quad (13)$$

$$l(x_j) \quad - \quad L(x_i,x_j) \geq 0 \quad (14)$$

$$l(x_i) + l(x_j) \quad - \quad 2 * L(x_i,x_j) \leq 1 \quad (15)$$

The best set of memory objects which fits into the scratchpad and minimizes the total energy consumption now has to be identified. The objective function E_{Total} denotes the total energy consumed by the system.

$$E_{Total} \quad = \quad \sum_{x_i \in X} E(x_i) \quad (16)$$

The scratchpad size constraint can be modeled as follows:

$$\sum_{x_i \in X} [1 - l(x_i)] * S(x_i) \quad \leq \quad scratchpadsize \quad (17)$$

The size $S(x_i)$ of memory object x_i is computed without considering the appended NOP instructions. These NOP instructions are stripped away from the memory objects prior to allocating them to the scratchpad. A commercial ILP Solver [5] is used to obtain an optimal subset of memory objects which minimizes the objective function. The number of vertices $|V|$ of the conflict graph G is equal to the number of memory objects, which is bounded by the number of basic blocks in the program code. The number of linearizing variables is equal to the number of edges $|E|$ in the conflict graph G. Hence, the number of variables in the ILP problem is equal to $|V| + |E|$ and is bounded by $O(|V|^2)$. Nevertheless, the maximum runtime of the ILP solver for our set of real-life benchmarks (upto 19.5kBytes program size) was found to be less than a second.

Our ILP formulation can be easily extended to handle complex memory hierarchies. For example, if we had more than one scratchpad at the same horizontal level in the memory hierarchy, then we only need to repeat inequation (17) for every scratchpad. An additional constraint ensuring that a memory object is assigned to at most one scratchpad is also required. If we had I-caches at different levels (e.g. L1, L2) in the memory hierarchy, we need not do anything, as the algorithm tries to minimize the L1 I-cache misses. The L2 I-cache misses, being a subset of the L1 I-cache misses, are thus also minimized. Consequently, the energy consumption of the whole memory hierarchy is minimized.

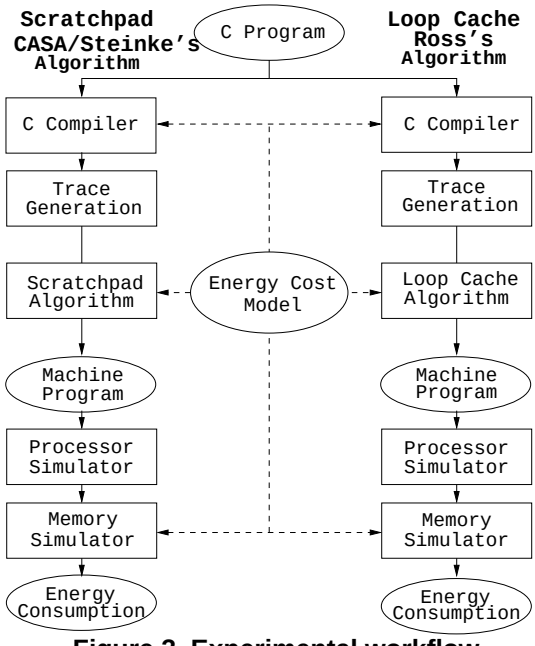

Figure 3. Experimental workflow

5. Experimental Setup

The experimental setup consists of an ARM7T processor core, an onchip cache, an onchip scratchpad and an offchip main memory. We want to compare the effect of allocation techniques for scratchpad on the energy consumption of the instruction memory subsystem. The *cacti* cache model was used to calculate the energy consumption per access for onchip 0.5μm technology cache, loop cache and scratchpad memories. The loop cache was assumed to contain a maximum of 4 loops. The energy consumption of the main memory was measured from the evaluation board.

Experiments were conducted according to the workflow presented in figure 3. In the first step, the benchmarks programs are compiled using an energy optimizing C compiler. Trace generation [14] is a well known I-cache performance optimization technique. Hence, for a fair comparison, traces are generated for both the allocation techniques. either CASA or the scratchpad allocation algorithm [13] allocates memory objects to the scratchpad memory. The generated machine code is then fed into ARMulator [1] to obtain the instruction trace. Our custom memory hierarchy simulator [8] based upon the instruction trace, memory hierarchy and the energy cost model, computes the aggregate energy consumed by the memory subsystem.

For the loop cache configuration, the workflow is similar to scratchpad workflow. The loop cache allocation algorithm [12] is utilized for assigning loops and functions to the loop cache. The energy consumed by the memory subsystem is computed similarly, using the appropriate memory hierarchy. The runtime overhead of CASA compared against the other two approaches was negligible.

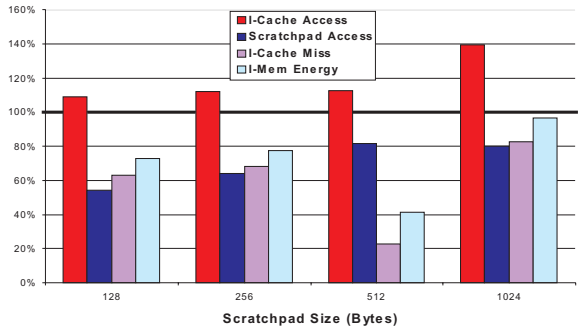

Figure 4. Comparison of CASA against Steinke's algorithm for MPEG benchmark.

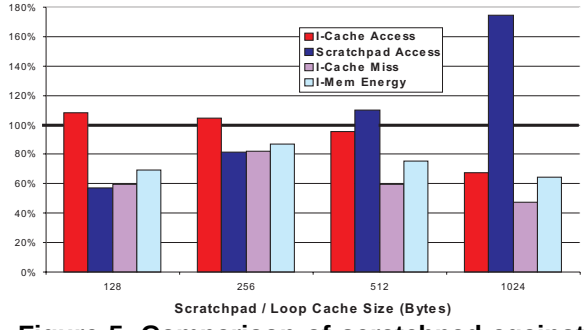

Figure 5. Comparison of scratchpad against loop cache for MPEG benchmark.

6. Results

A subset of benchmarks from the Mediabench suite were used to substantiate our claims on energy savings by the proposed algorithm. The size of the scratchpad/loop cache was varied while the rest of instruction memory subsystem was kept invariant and the number of hits and misses to the various levels of memory hierarchy were counted. Based upon the gathered information and the energy model (subsection 3.4), energy consumption was computed. Figure 4 displays the parameters of CASA algorithm for the MPEG benchmark against the published algorithm by Steinke et al. [13]. Both the algorithms assign memory objects to the scratchpad incorporated in the presented memory hierarchy. A direct mapped 2kB I-cache was chosen for these experiments. All results are shown as the percentage of the algorithm's [13] parameters, denoted as 100%. The first fact to observe is that the number of I-cache accesses are higher, while the number of scratchpad accesses are lower than for Steinke's algorithm. This might beguile the reader that CASA might cause an increase in energy consumption. However, this is incorrect since Steinke's algorithm tries to reduce energy consumption by increasing the number of accesses to the energy efficient scratchpad. In contrast, CASA

Benchmark (size)	Mem Size (Bytes)	Energy Consumption (μJ)			Improvement(%)	
		SP (CASA)	SP (Steinke)	LC (Ross)	CASA vs. Steinke	SP (CASA) vs. LC
adpcm	64	3398.37	3261.04	3779.80	-4.2	10.1
(1 kByte)	128	1694.71	2052.12	2702.20	17.4	37.3
	256	224.55	856.83	1480.59	73.8	84.8
					29.0	44.1
g721	128	7493.75	8011.68	8343.61	4.0	10.2
(4.7 kBytes)	256	6640.65	6510.00	6734.41	-2.0	1.4
	512	4941.53	4951.91	5616.16	0.2	12.0
	1024	2106.53	3033.11	4707.76	30.6	55.2
					8.2	19.7
mpeg	128	7554.88	10364.46	10918.01	27.1	30.8
(19.5 kBytes)	256	7521.28	9744.85	8624.61	22.8	12.8
	512	3904.27	9502.60	5189.06	58.9	24.8
	1024	3400.70	3518.72	5261.94	3.4	35.4
					28.0	26.0

Table 1. Overall energy savings

tries to reduce I-cache misses by assigning conflicting memory objects to the scratchpad. Since I-cache misses are the major source of energy consumption, CASA is able to conserve up to 60% energy against Steinke's algorithm.

Next we compare (see figure 5) scratchpad allocated with CASA against loop cache preloaded with Ross's [12] algorithm. Loop cache results are denoted as 100% in figure 5. For small sizes (128 and 256 bytes), the number of accesses to loop cache are higher than those to scratchpad. However, as we increase the size, loop cache's performance is restricted by the maximum number of preloadable memory objects. On the other hand, the scratchpad can be preloaded with any number of memory objects. Consequently, we observe a higher percentage of scratchpad accesses. Also using CASA, the number of I-cache misses for scratchpad are substantially lower than those for loop cache. Consequently, scratchpad is able to reduce energy consumption at an average of 26% against loop cache.

Finally, table 1 summarizes the energy consumption using CASA for scratchpad. Instruction cache of size 2kB, 1kB and 128 Bytes was assumed for the *mpeg*, *g721* and *adpcm* benchmarks, respectively.

7. Conclusion and Future Work

In this paper we presented a generic cache-behavior based scratchpad allocation technique. The technique reduced the energy consumption of the system against a published algorithm. We also demonstrated that the simple scratchpad memory allocated with the presented technique is better than a loop cache. The overall average energy savings of scratchpad allocated with our approach against scratchpad and loop cache allocated with their respective allocation algorithms are 21.1% and 28.6% respectively. We intend to extend the approach by considering preloading of data and dynamic copying (overlay) of memory objects on the scratchpad.

References

[1] ARM. *Advanced RISC Machines Ltd.* http://www.arm.com/armtech/ARM10_Thumb.

[2] O. Avissar, R. Barua and D. Stewart. An Optimal Memory Allocation Scheme for Scratch-Pad-Based Embedded Systems. In *IEEE Transactions on Embedded Computing Systems*, 1(1):6-26 Nov. 2002.

[3] R. Banakar, S. Steinke, B.-S. Lee et al. Scratchpad Memory: A Design Alternative for Cache On-chip Memory in Embedded Systems. In *Proc. of the 10th International Symposium on Hardware/Software Codesign*, Estes Park, CO, May. 2002.

[4] N. Bellas, I. Hajj, C. Polychronopoulos et al. Architectural and Compiler Support for Energy Reduction in the Memory Hierarchy of High Performance Microprocessors. In *Proc. of the ISLPED Monterey, CA, USA*. ACM, Aug. 1998.

[5] CPLEX. *CPLEX limited* www.cplex.com.

[6] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide To the Theory of NP-Completeness*. W. H. Freeman and Company, New York, 1979

[7] M. Kamble and K. Ghosh. Analytical Energy Dissipation Models for Low Power Caches. In *Proc. of the ISLPED Monterey, CA, USA*. ACM, Aug. 1997.

[8] MEMSIM. Dept. of Computer Science XII, Univ. of Dortmund. http://ls12.cs.uni-dortmund.de/research/memsim/

[9] Motorola. *Motorola ColdFire MCF5XXX processor family.* http://e-www.motorola.com/

[10] P. Pettis and C. Hansen. Profile Guided Code Positioning. In *Proc. of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation*. ACM, pages 16-27, Jun. 1990

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

[12] S. C. A. Gordon-Ross and F. Vahid. Exploiting Fixed Programs in Embedded Systems: A Loop Cache Example. In *IEEE Computer Architecture Letters*, Jan. 2002.

[13] S. Steinke, L. Wehmeyer, B. S. Lee et al. Assigning Program and Data Objects to Scratchpad for Energy Reduction. In *Proc. of the DATE Conference, Paris, France*. Mar. 2002.

[14] H. Tomiyama and H. Yasuura. Optimal Code Placement of Embedded software for Instruction Caches. In *Proc. of the 9th European Design and Test Conference ET&TC'96 Paris, France*. Mar. 1996.

[15] 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.