Efficient Scratchpad Allocation Algorithms for Energy Constrained Embedded Systems

Manish Verma, Lars Wehmeyer, Peter Marwedel

Department of Computer Science XII,
University of Dortmund, 44221 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. Memories are responsible for a significant percentage of a system's aggregate energy consumption. Consequently, novel memories as well as novel memory hierarchies are being designed to reduce the energy consumption. Caches and scratchpads are two contrasting variants of memory architectures. The former relies completely on hardware logic while the latter requires software for its utilization. Most high-end embedded microprocessors today include onchip instruction and data caches along with a scratchpad.
Previous software approaches for utilizing scratchpad did not consider caches and hence fail for the prevalent high-end system architectures. In this work, we use the scratchpad for storing instructions. We solve the allocation problem using a greedy heuristic and also solve it optimally using an ILP formulation. We report an average reduction of 20.7% in instruction memory energy consumption compared to a previously published technique. Larger reductions are also reported when the problem is solved optimally.
The scratchpad in the presented architecture is similar to a preloaded loop cache. Comparing the energy consumption of our approach against that of preloaded loop caches, we report average energy savings of 28.9% using the heuristic.

Key words: [Memory architectures, Memory allocation, Energy aware compilation, Integer Linear Programming, Memory energy modeling]

1 Introduction

Over the past decade, the popularity of mobile embedded devices such as mobile phones, digital cameras etc. has been one of the major driving forces in technology. The computing power of early desktop computers is now available in a handheld device. Unfortunately, battery technology could not keep pace with the advances made in silicon technology. As a result, contemporary mobile embedded systems suffer from limited battery capacity. Reduced energy consumption translates to reduced dimensions, weight and cost of the device. In such a competitive market, these reductions might be sufficient to provide an edge over competing products.

Several researchers [4, 16] have identified the memory subsystem as the energy bottleneck of the entire system. In fact, fetches from the instruction memory typically account for much of a system's power consumption [10]. Memory hierarchies are being introduced to reduce the memory system's energy dissipation. Caches and scratchpad memories represent two contrasting memory architectures. Caches improve performance by exploiting the available locality in the program. As a consequence, energy consumption is also reduced. However, they are not an optimal choice for energy constrained embedded systems. Caches, apart from the actual memory, consist of two additional components [22]. The first component is the tag memory required for storing information regarding valid addresses. The second component is the hardware comparison logic to determine cache hits and cache misses. These additional components consume a significant amount of energy per access to the cache irrespective of whether the access translates to a hit or a miss. Also, caches are notorious for their unpredictable behavior [14].

On the other end of the spectrum are the scratchpad memories, consisting of just data memory and address decoding circuitry. Due to the absence of tag memory and comparators, scratchpad memories require considerably less energy per access than a cache. In addition, they require less onchip area and allow tighter bounds on WCET prediction of the system. However unlike caches, scratchpads require complex program analysis and explicit support from the compiler. In order to strike a balance between these contrasting approaches, most of the high-end embedded microprocessors (e.g. ARM10E [1], ColdFire MCF5 [15]) include both onchip caches and a scratchpad.

We assume a 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 requirements. This implies that small scratchpad memories can achieve greater energy savings when they are filled with instructions rather 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 non-linear optimization problem. Under simplifying conditions, it can be reduced to either a Weighted Vertex Cover [9] problem or a Knapsack [9] problem, both of which are known to be NP-complete problems. A greedy heuristic is used to solve the scratchpad allocation problem. An optimal solution is also obtained by formulating the scratchpad allocation problem as an ILP problem. We compare our approach against a published technique [19]. Due to the presence of an I-cache in our architecture, the previous technique fails to produce optimal results and may even lead to the problem of *cache thrashing* [11].

We also compare our approach to that of preloaded loop caches [10], 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,

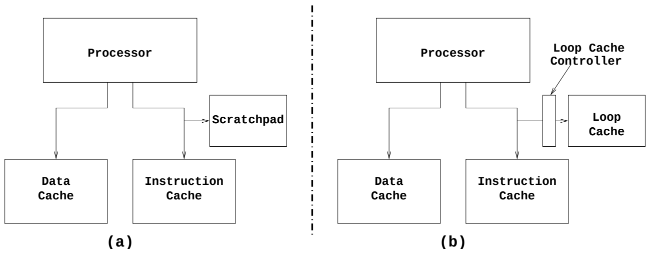


Fig. 1. System Architecture: (a) Scratchpad (b) Loop Cache

but are less flexible as they can be preloaded with only a limited number of loops. We demonstrate that using our allocation algorithm, scratchpad memories can outperform their complex counterparts.

In the next section, we describe related work and detail the shortcomings of the previous approaches. Section 3 describes the information regarding memory objects, cache behavior and the energy model. Section 4 presents the scratchpad allocation problem in detail, followed by the description of the proposed heuristic and the ILP formulation. The experimental setup is explained in section 5. In section 6 we present the results for an ARM based system and end the paper with a conclusion and future work.

2 Related Work

Analytical energy models for memories [12] have been found to be fairly accurate. We use *cacti* [22] 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 [17, 21] were developed to improve the CPI (cycles per instruction) by reducing the number of I-cache misses. Those basic blocks that are frequently executed in a contiguous way are combined to form so-called traces [17]. Authors in [17] placed traces within functions, while [21] placed them across function boundaries to reduce the I-cache misses.

Several researchers [2, 16] have utilized scratchpad memories for assigning global/local variables, whereas only Steinke et al. [19] 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 to the scratchpad memory.

Though this approach is sufficiently accurate for the used memory hierarchy, it is not suitable for the current setup. The assumption that execution (access) counts are sufficient to represent energy consumption by a memory object fails in the presence of a cache, where execution (access) counts have to 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 hit/miss ratio and hence energy consumption. This discussion stresses the need for a more detailed energy model taking these effects into account. In addition, maintaining the conflict relationships between memory objects is not considered during code placement using the previous approach. The memory objects are moved instead of copying them from main memory to the scratchpad. As a result, the layout of the entire program is changed, which may cause completely different cache access patterns and thus lead to erratic results.

Authors in [13] proposed an instruction buffer to act as an alternative location for fetching instructions in order to improve the energy consumption of a system. Loops identified by the short backward branch at the end of the first iteration are copied to the instruction buffer during the second iteration. From the third iteration onwards, instructions are fetched from the instruction buffer instead of the L1 I-cache, given that no *change-of-flow* (e.g. branch) statements are contained within the loop. Ross et al. [10] 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 problem of being able to store only a fixed number of memory objects in the loop cache will lead to problems for large programs with several hot spots. As in [19], memory objects are greedily selected only on the basis of their execution time density (execution time per unit size). In the wake of the discussion we enumerate the contributions of this paper.

- It for the first time studies the combined effect of a scratchpad and an I-cache on the memory 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 a host of complex memory hierarchies.
- It demonstrates that scratchpad memories together with an allocation algorithm can replace preloaded loop caches.

Please note that in the rest of this paper, energy consumption refers to the energy consumption of the instruction memory subsystem. In the following section, we describe preliminary information required for understanding our approach.

3 Preliminaries

We start by 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 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 non-cacheable 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, memory objects within the program code are identified. The memory objects are then distributed between offchip main memory and non-cacheable scratchpad memory to minimize energy consumption. The well known compiler optimization *trace generation* is used to identify the memory objects. A *trace* is a frequently executed straight-line path, consisting of basic blocks connected by fall-through edges [21]. Dynamic profiling is required to determine traces in the program. Our traces are kept smaller than the scratchpad size, as larger traces can not be placed onto the scratchpad as a whole. The traces are appended with NOP instructions to align them to cache line boundaries. This ensures 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 [21], 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:

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

Similarly, a memory object is mapped to cache line(s) depending upon its start address and size. Two memory objects potentially cause a conflict in the cache if they are mapped to at least one common cache line. This relationship can be represented by a conflict graph G (see figure 2), which is defined as follows:

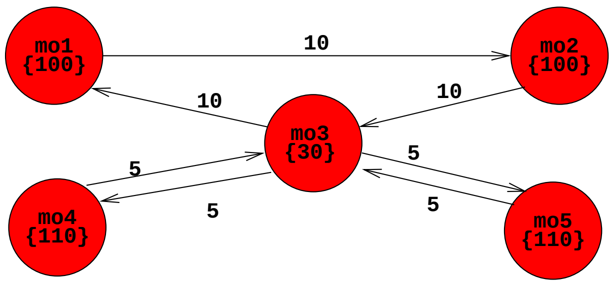


Fig. 2. Conflict Graph

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 lines that need to be fetched if there is a miss of x_i that occurs due to x_j. The weight f_i of a vertex x_i is the total number of instruction fetches within x_i.

In order to build up the conflict graph for a program, we first need to identify the memory objects to be considered by our algorithm. We use profiling to determine traces. In order to mark the vertices with the total number of instruction fetches and to determine the number of conflict misses among the memory objects, dynamic profiling is also required. The determined values are then attributed to vertices and conflict edges, respectively. In order to minimize the influence of the chosen input data set on the results, average values generated by using several distinct input vectors can be used.

The conflict graph as shown in figure 2 is a directed graph because the conflict relationship is *antisymmetric*. 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

As mentioned before, all energy values refer to the energy consumption of the instruction memory subsystem. The energy $E(x_i)$ consumed by an MO x_i is expressed as:

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

where E_{Cache} can be computed as follows:

$$E_{Cache}(x_i) = Hit(x_i) * E_{Cache\_hit} + Miss(x_i) * E_{Cache\_miss} \tag{2}$$

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 one line of the I-cache.

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

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:

$$f_i = Hit(x_i) + Miss(x_i) \tag{4}$$

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:

$$E_{Cache}(x_i) = f_i * E_{Cache\_hit} + \tag{5}$$
$$\textstyle\sum_{x_j \in N_i} Miss(x_i, x_j) * (E_{Cache\_miss} - E_{Cache\_hit})$$

The first term in equation (5) is a constant while the second term, which is variable, depends on the overall program code layout and the memory hierarchy. We would like to point out that the approach [10] only considered the constant term in its energy model. Consequently, the authors 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:

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

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

4 Problem Description

Once we have created the conflict graph G annotated with vertex and edge weights, the energy consumption of memory objects can be computed. Now, the problem is to select a subset of memory objects which minimizes the number of conflict edges and the overall energy consumption of the system. The subset is bounded in size by the scratchpad size.

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

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

Since a memory object allocated to the scratchpad does not conflict with other memory objects, we can represent $Miss(x_i, x_j)$ (see above) as follows:

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

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:

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

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

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

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 equation (10) into the following form.

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

We find the last term is a quadratic degree term, since 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.

The objective function E_{Total} denoting the total energy consumed by the system needs to be minimized.

$$E_{Total} = \sum_{x_i \in X} E(x_i) \tag{12}$$

Minimization of the objective function is to be performed while conforming to the scratchpad size constraint.

$$\sum_{x_i \in X} [1 - l(x_i)] * S(x_i) \leq ScratchpadSize \tag{13}$$

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. The non-linear optimization problem can be solved to obtain a scratchpad allocation optimized with respect to energy.

Our problem 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 (13) for every scratchpad. An additional constraint ensuring that a memory object is assigned to at most one scratchpad is also required.

```
Greedy-Heuristic(G(X,E), ScratchpadSize)
1  Rem_SPSize = ScratchpadSize
2  L = NIL
3  while ( $\exists x \in X : S(x) \leq$ Rem_SPSize )
4     do select $x_i \in X : S(x_i) \leq$ Rem_SPSize $\wedge$
           $E(x_i) > E(x_k)\ \forall x_k \in X : S(x_k) \leq$ Rem_SPSize
5          $X = X - \{x_i\}$
6          $E = E - \{e_{ij} | \forall j : j \in N_i\} - \{e_{ji} | \forall j : i \in N_j\}$
7          Rem_SPSize = Rem_SPSize - $S(x_i)$
8          $L = L \cup \{x_i\}$
9  return L
```

Fig. 3. Greedy Heuristic for Scratchpad Allocation Problem

The above optimization problem is related to two NP-complete problems viz. Weighted Vertex Cover [9] and Knapsack problem [9]: Under the simplifying assumption that the cache present in the system is large enough to hold all the memory objects without causing a single conflict miss, the energy consumption of a memory object becomes independent of other memory objects. Under this assumption, the problem is reduced to a Knapsack problem with each node having constant weights. On the other hand, if we assume that the energy of an access to the scratchpad E_{SP} is equal to the energy of a cache hit E_{Cache_hit}, equation (11) transforms to the following form and the problem is reduced to the Weighted Vertex Cover problem:

$$E(x_i) = f_i * E_{SP} + [E_{Cache\_miss} - E_{Cache\_hit}] * [\sum_{j \in N_i} l(x_j) * l(x_i) * m_{ij}] \quad (14)$$

Fortunately, approximation algorithms can be employed to obtain near-optimum solutions in polynomial time. In the following section, we will present a greedy heuristic which solves the scratchpad allocation problem near-optimally in most cases. We will also solve the problem optimally using an *Integer Linear Programming* (ILP) based approach.

4.1 Greedy Heuristic

The proposed greedy heuristic tries to put maximum weighted nodes on the scratchpad. It takes as input the conflict graph and the scratchpad size and returns the list of memory object to be allocated onto the scratchpad. The heuristic is formally presented in figure 3.

The heuristic iteratively computes the energy consumption of each memory object which can be placed on the scratchpad memory, considering not only execution counts but also the number of conflict cache misses caused by other memory objects. The maximum energy vertex to be allocated to the scratchpad is then greedily selected. This vertex is removed from the conflict graph G and appended to the list L and the unallocated scratchpad size (Rem_SPSize) is reduced appropriately.

A memory object present in the scratchpad does not conflict with the memory objects present in the cache. The energy of the conflicting memory objects is thus reduced by removing the vertex and the adjacent edges from the conflict graph. The energy consumption of a memory object x_i is computed according to the energy model proposed in subsection 3.4. The heuristic iterates as long as there exists a memory object which can be placed on the scratchpad without violating the scratchpad size constraint. On termination, a list of memory objects to be allocated onto the scratchpad is returned. The time complexity of the heuristic is $O(ScratchpadSize * (|X| + |E|))$ if we precompute and store the energy consumption of each memory object x_i at the end of each "while loop" iteration.

4.2 Integer Linear Programming

In order to formulate an *Integer Linear Programming* problem, we need to linearize the scratchpad allocation problem. This can be achieved by replacing the non-linear term $l(x_i) * l(x_j)$ of equation (11) by an additional variable $L(x_i, x_j)$:

$$\begin{aligned} E(x_i) = {} & f_i * E_{SP} + \\ & f_i * [E_{Cache\_hit} - E_{SP}] * l(x_i) + \\ & [E_{Cache\_miss} - E_{Cache\_hit}] * [\sum_{j \in N_i} L(x_i, x_j) * m_{ij}] \end{aligned} \tag{15}$$

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

$$l(x_i) - L(x_i, x_j) \geq 0 \tag{16}$$

$$l(x_j) - L(x_i, x_j) \geq 0 \tag{17}$$

$$l(x_i) + l(x_j) - 2 * L(x_i, x_j) \leq 1 \tag{18}$$

The objective function E_{Total} and the scratchpad size constraint remain unchanged (cf. equations (12) and (13)).

A commercial ILP Solver [6] is used to obtain an optimal subset of memory objects which minimizes the objective function. The number of vertices $|X|$ 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 $|X| + |E|$ and is bounded by $O(|X|^2)$. The actual runtime of the used ILP solver [6] was less than one second on a Sun-Blade 100 running at 500 MHz for a conflict graph containing 455 vertices. The computation times may be expected to increase if non-commercial tools (e.g. lp_solve [5]) are used. In the next section we describe the experimental setup used for conducting experiments.

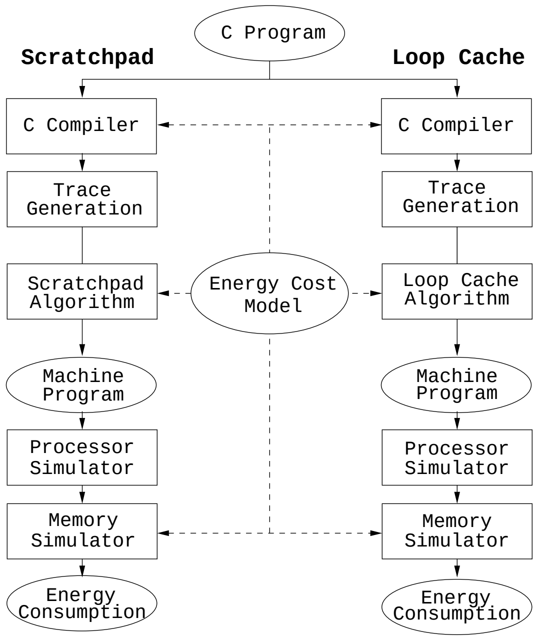
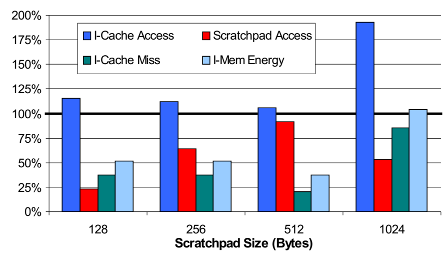


Fig. 4. Experimental Workflow

5 Experimental Setup

The experimental setup consists of an ARM7T processor core, onchip instruction and data caches, an onchip scratchpad and an off-chip main memory. The used instruction cache has a direct-mapped organization since this architecture has been found to be most suitable for low-power instruction caches [20]. The capacity of the instruction cache was selected according to the size of the corresponding benchmark. We determine the effect of allocation techniques for scratchpad on the energy consumption of the instruction memory subsystem. The *cacti* cache model [22] was used to calculate the energy consumption per access to a cache, loop cache and scratchpad memory, all assumed to be onchip and in 0.5μm technology. The loop cache was assumed to be able to hold a maximum of 4 loops. The energy consumption of the main memory was measured from our evaluation board [18].

Experiments were conducted according to the workflow presented in figure 4. In the first step, the benchmarks programs are compiled using ENCC [7], an energy aware C compiler. Trace generation [21] is a well known I-cache performance optimization technique. For a fair comparison, traces are generated for all the allocation techniques. In the following step, the scratchpad allocation algorithm can either be the greedy heuristic (cf. subsection 4.1), the ILP based allocation algorithm (cf. subsection 4.2) or Steinke's scratchpad allocation algorithm [19]. The generated machine code is then fed into ARMulator [1] to obtain



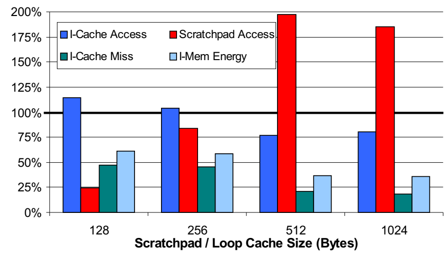
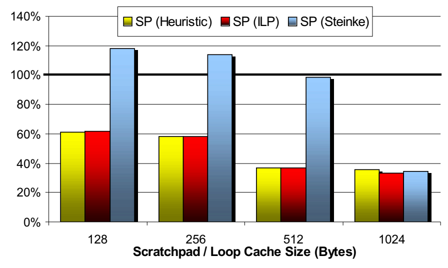


Fig. 5. Comparison of Scratchpad (Heuristic) against Scratchpad (Steinke) for MPEG

Fig. 6. Comparison of Scratchpad (Heuristic) against Loop Cache (Ross) for MPEG



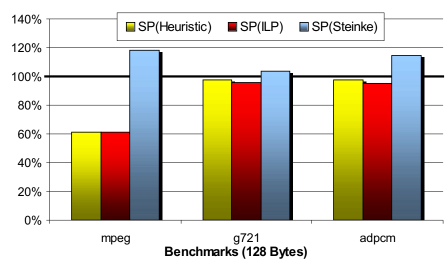


Fig. 7. Comparison of Heuristic, ILP, Steinke's and Ross's Algorithm for MPEG

Fig. 8. Comparison of Heuristic, ILP, Steinke's and Ross's Algorithm for all benchmarks

the instruction trace. Our custom memory hierarchy simulator [8], based on 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 loop cache is preloaded with the loops and functions selected by the allocation algorithm presented in [10]. The energy consumed by the memory subsystem is computed in a similar way, using the appropriate memory hierarchy and energy cost model.

6 Results

A subset of benchmarks from the Mediabench suite were used to substantiate our claims concerning energy savings using the proposed algorithm. The size of the scratchpad/loop cache was varied while keeping the rest of the instruction memory subsystem invariant. The number of accesses, hits and misses to every memory in the hierarchy were counted. Based on this information and the energy model (subsection 3.4), energy consumption was computed.

Figure 5 displays the energy consumption along with all its respective parameters (i.e. scratchpad accesses, cache accesses and cache misses) of the proposed heuristic for the MPEG benchmark. The instruction cache size was set to 2k for

Table 1. Overall Energy Savings

Benchmark	MemSize	Energy Consumption (μJ)				Improvement(%)		
(size)	(bytes)	SP	SP	SP	LC	SP(Heu)	SP(Heu)	SP (Heu)
		(Heu)	(ILP)	(Steinke)	(Ross)	vs.	vs.	vs.
						SP(ILP)	SP(Steinke)	LC (Ross)
adpcm	128	3567	3397	2763	2998	-5.0	-29.1	-19.0
(1 KB)	256	1744	1695	2040	1784	-2.8	14.6	2.3
	512	225	—	1400	1140	—	84.0	80.3
g721	128	7565	7393	8012	7739	-2.3	5.6	2.2
(4.7 KB)	256	6412	5984	6321	6446	-7.1	-1.4	0.5
	512	5249	4478	4469	6131	-17.2	-17.4	14.4
	1024	2566	2107	3033	6207	-21.8	15.4	58.7
mpeg	128	6318	6324	12161	10293	0.1	48.0	38.6
(21.4KB)	256	5983	5989	11697	10266	0.1	48.9	41.7
	512	3779	3755	10157	10291	-0.6	62.8	63.3
	1024	3709	3419	3579	10336	-8.5	-3.6	64.1
						-6.5	20.7	28.9

these experiments. All the results are shown as percentages of Steinke's algorithm [19], with the parameters of that algorithm being denoted as 100%. It is interesting to note that in spite of higher I-cache accesses and lower scratchpad accesses, the heuristic reduces energy consumption against Steinke's algorithm. The substantially lower I-cache misses are able to over-compensate for higher I-cache accesses and result in reduced energy consumption. The justification for this is that Steinke's algorithm tries to reduce energy consumption by increasing the number of accesses to the energy efficient scratchpad. In contrast, our heuristic tries to reduce I-cache misses by assigning conflicting memory objects to the scratchpad. Since I-cache misses account for a significant portion of energy consumption, the heuristic is able to conserve up to 63% energy against Steinke's algorithm. In one case (1024 bytes scratchpad), Steinke's algorithm performs marginally better than our approach. For this setup, moving (instead of copying) the memory objects seems to completely change the program's cache conflict behavior. However, there is no way of foreseeing this kind of effect when applying Steinke's algorithm and it might happen that instead of reducing the cache misses, cache performance and energy consumption are deteriorated since the algorithm doesn't account for cache behavior.

In fig. 6, we compare a scratchpad allocated with our heuristic against a loop cache preloaded with Ross's algorithm [10]. Similar to figure 5, all results are shown as percentages of the corresponding parameters of Ross's algorithm [10]. For small scratchpad/loop cache sizes (128 and 256 bytes), the number of accesses to loop cache are higher than those to scratchpad. However, as we increase the size, the loop cache's performance is restricted by the maximum number of only 4 preloadable memory objects. The scratchpad, on the other hand, can be preloaded with any number of memory objects as long as their aggregate size is less than the scratchpad size. Moreover, the number of I-cache misses is substantially lower if a scratchpad is used instead of a loop cache. Consequently, a scratchpad is able to reduce energy consumption at an average of 52% against a loop cache for the MPEG benchmark.

In figure 7, we compare the energy consumption of different scratchpad allocation algorithms (viz. Heuristic, ILP and Steinke's) for scratchpad based systems and that of Ross's algorithm [10] for loop cache based systems. As earlier, the energy consumption due to Ross's algorithm is denoted as 100% while the energy consumption of the scratchpad allocation algorithms are denoted as percentages of Ross's algorithm. A couple of interesting points can be noted for the figure. Firstly, the heuristic performs fairly close to the optimal solution obtained by the ILP based algorithm. Secondly, for the smaller sizes (128 and 256 bytes), loop cache performs better than the scratchpad allocated with Steinke's algorithm [19], while the opposite is true for larger sizes. Figure 8 depicts the comparison of scratchpad allocation algorithms and Ross's algorithm for all benchmarks. A scratchpad and a loop cache of 128 bytes was assumed to be present in the memory hierarchy. The instruction cache size was set to 1k and 128 bytes for *g721* and *adpcm*, respectively. Observations similar to the previous figure can be noted.

Finally, table 1 summarizes the energy consumption for scratchpad and loop cache allocated with the corresponding allocation algorithms.

7 Conclusion and Future Work

In this paper, we model the cache-behavior based scratchpad allocation problem as a generic non-linear optimization problem. The problem is solved near-optimally using a heuristic and also optimally using an ILP based approach. The energy consumption of the heuristic is on average a meagre 6.5% away from that of the optimal solution. The presented techniques reduce the energy consumption of the system against a published algorithm. An average reduction of 20.7% in energy consumption due the heuristic is observed. In addition, we also demonstrate that the simple scratchpad memory allocated with the presented techniques outperforms a preloaded loop cache. Average energy savings of 28.9% are observed for the proposed heuristic and even higher values can be reported for ILP based allocation algorithm. The presented techniques can be easily extended to handle a variety of complex memory hierarchies.

References

1. ARM. *Advanced RISC Machines Ltd.* www.arm.com.
2. O. Avissar, R. Barua, and D. Stewart. An Optimal Memory Allocation Scheme for Scratch-Pad-Based Embedded Systems. *IEEE Transactions on Embedded Computing Systems*, 1(1):6–26, November 2002.
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 10th International Symposium on Hardware/Software Codesign*, Colorado, USA, May 2002.

4. N. Bellas, I. Haji, C. Polychronopoulos, and G. Stamoulis. Architectural and Compiler Support for Energy Reduction in Memory Hierarchy of High Performance Microprocessors. In *Proceedings of the International Symposium on Low Power Electronics and Design ISPLED*, Monterey, CA, USA, August 1999.
5. M. Berkelaar. *lp_solve: a Mixed Integer Linear Program solver.* available from: ftp://ftp.es.ele.tue.nl/pub/lp_solve.
6. CPLEX. *CPLEX Ltd.* www.cplex.com.
7. Department of Computer Science XII, University of Dortmund. *ENCC.* http://ls12-www.cs.uni-dortmund.de/research/encc.
8. Department of Computer Science XII, University of Dortmund. *MEMSIM.* http://ls12.cs.uni-dortmund.de/∼wehmeyer/LOW_POWER/memsim_doc.
9. M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide To the Theory of NP-Completeness.* Freeman, New York, USA, 1979.
10. S.C.A Gordon-Ross and F. Vahid. Exploiting Fixed Programs in Embedded Systems: A Loop Cache Example. *Computer Architecture Letters*, January 2002.
11. J.L. Hennessy and D.A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, 3. edition, 2003.
12. M. Kamble and K. Ghosh. Analytical Energy Dissipation Models for Low Power Caches. In *Proceedings of the International Symposium on Low Power Electronics and Design ISPLED*, Monterey, CA, USA, August 1997.
13. L.H. Lee, B. Moyer, and J. Arends. Instruction Fetch Energy Reduction Using Loop Caches For Embedded Applications with small Tight Loops. In *Proceedings of the International Symposium on Low Power Electronics and Design ISPLED*, San Diego, CA, USA, August 1999.
14. P. Marwedel, L. Wehmeyer, M. Verma, S. Steinke, and U. Helmig. Fast, predictable and low energy memory references through architecture-aware compilation. In *Proceedings of the Asia and South Pacific Design Automation Conference ASPDAC 2004 (to appear)*, 2004.
15. MOTOROLA. *Motorola Inc.* http://e-www.motorola.com/files/shared/doc/ selector_guide/SG1001.pdf.
16. P.R. Panda, N.D. Dutt, and A. Nicolau. *Memory Issues in Embedded Systems-On-Chip.* Kluwer Academic Publishers, Norwell, MA, 1999.
17. P. Pettis and C. Hansen. Profile guided code positioning. In *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation.* ACM SIGPLAN, June 1990.
18. S. Steinke, M. Knauer, L. Wehmeyer, and P. Marwedel. An Accurate and Fine Grain Instruction-Level Energy Model Supporting Software Optimizations. In *Proceedings of International Workshop on Power And Timing Modeling, Optimization and Simulation PATMOS*, Yverdon-Les-Bains, Switzerland, Sep. 2001.
19. S. Steinke, L. Wehmeyer, B.-S. Lee, and P. Marwedel. Assigning Program and Data Objects to Scratchpad for Energy Reduction. In *Proceedings of Design Automation and Test in Europe DATE*, Paris France, March 2002.
20. C.-L. Su, , and A.M. Despain. Cache Design Trade-Offs and Performance Optimization: A Case Study. In *Proceedings of the International Symposium on Low Power Design ISLPD*, pages 63–68, 1995.
21. H. Tomiyama and H. Yasuura. Optimal Code Placement of Embedded Software for Instruction Caches. In *Proceedings of the 9th European Design and Test Conference ET&TC*, Paris, France, March 1996.
22. 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), May 1996.