Cache-Aware Scratchpad-Allocation Algorithms for Energy-Constrained Embedded Systems

Manish Verma, *Student Member, IEEE*, Lars Wehmeyer, and Peter Marwedel, *Senior Member, IEEE*

***Abstract*—In the context of mobile embedded devices, 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 architectures are being designed to reduce the energy consumption. Caches and scratchpads are two contrasting memory architectures. The former relies on hardware logic while the latter relies on software for its utilization. To meet different requirements, most contemporary high-end embedded microprocessors include on-chip instruction and data caches along with a scratchpad. Previous approaches for utilizing scratchpad did not consider caches and hence fail for the contemporary high-end systems. Instructions are allocated onto the scratchpad, while taking into account the behavior of the instruction cache present in the system. The problem of scratchpad allocation is solved using a heuristic and also optimally using an integer linear programming formulation. An average reduction of 7% and 23% in processor cycles and instruction-memory energy, respectively, is reported when compared against a previously published technique. The average deviation between optimal and nonoptimal solutions was found to be less than 6% both in terms of processor cycles and energy. The scratchpad in the presented architecture is similar to a preloaded loop cache. Comparing the energy consumption of the presented approach against that of a preloaded loop cache, an average reduction of 9% and 29% in processor cycles and instruction-memory energy, respectively, is reported.**

***Index Terms*—Memory hierarchy, memory management, optimizing compilers, SRAM chips.**

I. Introduction

OVER THE PAST decade, the popularity of mobile embedded devices such as mobile phones, digital cameras, MP3 players, etc., has been the one of the major forces driving the technology. The computing power of early computers is now available in a handheld device, exemplifying the progress made by silicon technology. Unfortunately, in comparison, the advances made in battery technology are diminutive. As a result, contemporary mobile embedded systems suffer from limited battery capacity. Rather than increasing the battery capacity, it is preferable to reduce the energy consumption, since it translates to reduced dimensions, weight, and cost of the device. In such a competitive market, these reductions might be sufficient to provide the product an edge over competing products.

Manuscript received March 1, 2004; revised July 26, 2004 and December 23, 2004. This work was supported by Deutsch Forschungsgemeinschaft (DFG) Grant Ma 943/8-3. This paper was recommended by Associate Editor G. Gielen.

The authors are with the Department of Computer Science, University of Dortmund, Dortmund D-44221, Germany (e-mail: Manish.Verma@cs.uni-dortmund.de; Lars.Wehmeyer@cs.uni-dortmund.de; Peter.Marwedel@cs.uni-dortmund.de).

Digital Object Identifier 10.1109/TCAD.2005.859523

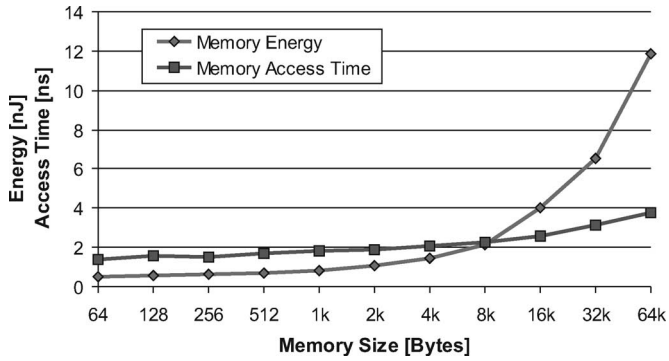

Fig. 1. Energy consumption and access time as function of memory size.

Several researchers [4], [17] have identified the memory subsystem as the energy bottleneck of the entire system. In fact, fetches to the instruction memory typically account for much of a system's power consumption [10]. The larger the memory, the larger the access times and the energy consumed per access. Fig. 1 displays the energy consumption and the access times of a static RAM (SRAM) memory as a function of its size. Note that the energy consumption per access shown in Fig. 1 differs by a factor of up to $\alpha = 24$, whereas the access times differ by a factor of up to $\beta = 2.7$. Consequently, memory hierarchies are being constructed to reduce the memory subsystem's energy dissipation. Caches and scratchpad memories (SPMs) represent two contrasting memory architectures. Caches improve performance by exploiting temporal and spatial locality present in the program. As a consequence, energy consumption is also reduced. However, caches are not an optimal choice for energy-constrained embedded systems, since they consist of two additional components besides the data memory [23]. The first component is the tag memory required for storing information regarding the valid addresses. The second component is the hardware comparison logic for determining 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. Moreover, caches are known for their notoriously unpredictable behavior [11].

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 the tag memory and the comparison hardware, scratchpad memories require considerably less energy than a cache [3]. Fig. 2 compares the energy consumption per access of scratchpads and caches of varying associativity. It is obvious that the scratchpad consumes the least energy per access. Moreover, a 2-kB scratchpad consumes

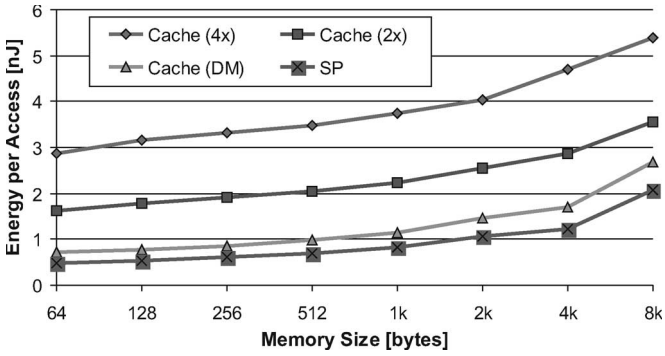

Fig. 2. Energy consumption of caches and scratchpad memories (SPMs).

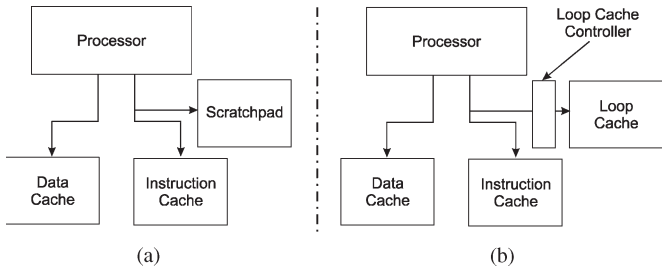

Fig. 3. System architecture: (a) scratchpad and (b) loop cache.

about one quarter (1/4) energy per access than a four-way set-associative cache of the same size. In addition, scratchpad memories require less on-chip area and allow tighter bounds on worst case execution time (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., ARM11 [1], ColdFire MCF5 [16]) include both on-chip caches and a scratchpad.

Many contemporary embedded systems are also real-time systems. For such systems, it is necessary to guarantee meeting real-time constraints. As a consequence, WCET information must be derived to prove that the real-time system possesses the required properties. In some cases, it is possible to prove that caches improve the WCET. However, formal methods employed to determine whether the current access to the cache results in a hit or a miss have been found to be fairly restrictive and imprecise. To make matters worse, caches with random replacement policy are being used in embedded systems, making it necessary to assume a cache miss on every access. Modeling conflict relationships between instruction and data in a unified cache is also difficult. On the other hand, scratchpads, unlike caches, are completely predictable. Due to fact that the contents of the scratchpad are controlled by the software, it is possible to compute tighter bounds on the WCET.

We assume a memory hierarchy as shown in Fig. 3(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. Moreover, contemporary high-end microprocessors (e.g., ARM11 [1]) feature separate dedicated scratchpad memories for data and instructions. 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 nonlinear optimization problem. Under simplifying conditions, it can be reduced to either a weighted vertex cover [9] or a knapsack [9] problem, both of which are known to be NP-complete problems. An optimal solution is obtained by formulating the scratchpad-allocation problem as an integer linear programming (ILP) problem, while a near-optimum solution is also obtained by using a heuristic. We compare our approach against a published technique [20]. Due to the presence of an I-cache in our architecture, the previous technique fails to produce optimal results. In the following section, we demonstrate, with the aid of an example, that the previous technique may even lead to the problem of cache thrashing [11]. This argument is strengthened by observing (cf., Section VII) a similar behavior for a real-life benchmark.

We also compare our approach to that of preloaded loop caches [10], as the utilization of the scratchpad in the current setup (see Fig. 3) 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/functions. We demonstrate that using our allocation algorithms, scratchpad memories can outperform their complex counterparts.

In the next section, we mention related work, and in Section III, with the help of an example, describe the shortcomings of previous approaches. Section IV provides the information regarding memory objects (MOs), cache behavior, and the energy model. Section V presents the scratchpad-allocation problem in detail, followed by a description of the proposed heuristic and the ILP formulation. The experimental setup is explained in Section VI. In Section VII, we present the results for an ARM-based system and end the paper with a conclusion and future work.

II. Related Work

Analytical energy models for memories [12] have been found to be fairly accurate. We use cacti [24] to determine the energy per access for caches and preloaded loop caches. A recent paper [25] found that cacti accurately models the energy consumption of an 0.18-μm cache. The energy per access for scratchpad memories was determined using the model presented in [3].

Application-code-placement techniques [18], [21] were developed to improve the cycles per instruction (CPI) 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. The authors in [18] placed traces within functions, whereas those in [21] placed them across function boundaries to reduce the number of I-cache misses.

Most of the research on scratchpad utilization [2], [6], [13], [17] has focused on allocating data elements onto the scratchpad. The authors in [2] and [17] statically allocated global/local

variables on the scratchpad, whereas the authors in [6] and [13] looked at the possibility of dynamically copying the data elements from the main memory onto the scratchpad.

Steinke *et al.* [20] demonstrated the obvious benefits that can be achieved by allocating both program and data elements memory objects to the scratchpad memory. They assumed a memory hierarchy composed of only scratchpad and main memory. Profit values were assigned to program and data elements 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 MO 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 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. 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 a completely different cache access pattern, and thus lead to erratic results.

The authors in [14] 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. To overcome this limitation, Gordon-Ross and Vahid [10] proposed a preloaded loop cache that can be statically loaded with preidentified 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 preloaded loop cache can be allocated 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 property of being able to store only a fixed number of memory objects in the loop cache leads to problems for large programs with several hot spots. As in [20], memory objects are greedily selected only on the basis of their execution-time density (execution time per unit size).

In the wake of the above discussion, we enumerate the contributions of this paper.

1) It studies for the first time the combined effect of a scratchpad and an I-cache on the memory system's energy consumption.
2) It stresses the need for a sophisticated allocation algorithm by demonstrating the inefficiency of previous algorithms when applied to the present architecture.
3) It presents a novel scratchpad-allocation algorithm that can be easily applied to a host of complex memory hierarchies.
4) It demonstrates that scratchpads along with an allocation algorithm can replace 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 some preliminary information required for understanding our approach.

III. Motivation

We assume a system consisting of an I-cache and a main memory as the only memories comprising of the instruction-memory subsystem. Given a weighted control flow graph (CFG), a layout of CFG nodes in the main memory, and an execution trace, we analyze the behavior of the I-cache and compute the energy consumed by the instruction-memory subsystem. Later, we add a scratchpad into the instruction-memory subsystem and statically allocate the scratchpad using a previous approach [20]. Due to the fact that this approach does not model an I-cache present in the system, it leads to erratic results. This demonstrates the negative implications of using an allocation that does not model caches present in the system and serves as a motivation for the techniques presented in this paper.

Before we start, we make a few assumptions. Firstly, for the sake of simplicity, the size of each basic block, as well as of each cache line, is assumed to be one word. However, the algorithm is general enough to handle basic blocks and cache lines of all sizes. Secondly, moving a basic block within the memory hierarchy does not modify the size of any other basic block. Lastly, we assume that a cache hit and a cache miss consumes 1 and 10 units of energy, respectively. Moreover, both the accesses to the loop cache and to the scratchpad cost 0.5 energy units. We would like to state that these assumptions are added for the sake of the clarity and that they apply only to the motivating example.

A. Motivating Example

Fig. 4 shows a weighted CFG, main memory layout, an instruction cache, and an execution trace for the program. The nodes of the CFG represent the corresponding basic blocks present in the application code. The edges connecting nodes represent possible flow of control during the execution of the program. The nodes and edges of the CFG are weighted according to the corresponding execution frequencies during a typical execution of the program. The main memory layout shows the relative location as well as the absolute addresses of each basic block in the main memory. The instruction cache is a direct-mapped cache of size four words. Hypothetical energy values for a cache hit ($E_{\mathrm{Cache\_Hit}}$) and a cache miss ($E_{\mathrm{Cache\_Miss}}$) are also given. These energy values are representative of a typical miss–hit energy ratio ($E_{\mathrm{Cache\_Miss}}/E_{\mathrm{Cache\_Hit}}$) found in embedded systems. The execution trace represents an example

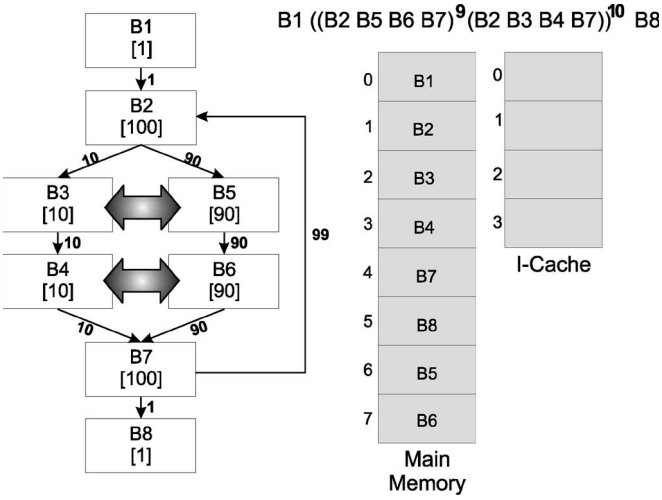

Fig. 4. Motivating example.

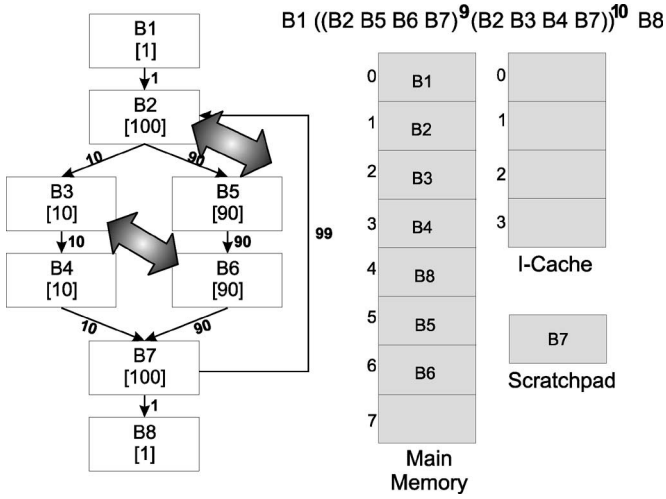

Fig. 5. Motivating example (Steinke's approach).

TABLE I
ENERGY VALUES

BB	I-Cache Accesses	I-Cache Misses	Energy
BB2	100	0	100
BB3	10	10	100
BB4	10	10	100
BB5	90	10	180
BB6	90	10	180
BB7	100	0	100
Total	400	40	760

TABLE II
ENERGY VALUES FOR SCRATCHPAD (ONE WORD)

BB	I-Cache Accesses	I-Cache Misses	SPM Accesses	Energy
BB2	100	90	0	910
BB3	10	10	0	100
BB4	10	0	0	10
BB5	90	90	0	900
BB6	90	10	0	180
BB7	0	0	100	50
Total	300	200	100	2150

TABLE III
ENERGY VALUES FOR SCRATCHPAD (TWO WORDS)

BB	I-Cache Accesses	I-Cache Misses	SPM Accesses	Energy
BB2	0	0	100	50
BB3	10	10	0	100
BB4	10	10	0	100
BB5	90	10	0	180
BB6	90	10	0	180
BB7	0	0	100	50
Total	200	40	200	660

execution of the program at the granularity of the basic blocks. The execution trace reveals that the right arm (i.e., B2, B5, B6, B7) of the loop is executed nine times before the left arm (i.e., B2, B3, B4, B7) is executed once. The execution of the right arm followed by that of the left arm is repeated for ten times before the end of the execution. According to the main memory layout and the modulo addressing ($2 \bmod 4 \equiv 6 \bmod 4 \equiv 2$) employed by caches, we observe that nodes B3 and B5 will share the same cache line. The same holds true for nodes B4 and B6. During the execution of the program, nodes B3 and B4 will constantly replace nodes B5 and B6 in the cache leading to an aggregate of 40 conflict cache misses. Table I shows the number of accesses, the number of misses, and the energy consumption of each memory object. The total energy consumption of the program amounts to 760 units.

B. Example (Steinke's Approach)

As represented in Fig. 5, we have introduced a scratchpad into the system. The scratchpad will be statically loaded using the approach [20] presented by Steinke *et al.*, which assumes energy consumption to depend solely upon execution frequencies. Consequently, in order to maximize the energy savings, the approach selects the nodes with the highest execution frequencies to be moved onto the scratchpad. If we assume that the size of the scratchpad is one cache line, we find that the approach can choose between node B2 or B7. Both nodes have equal execution frequency, higher than all the other nodes. Assuming that node B7 is moved and not copied to the scratchpad, the modified memory layout is shown in Fig. 5. It can be observed from the figure that in the modified memory layout, nodes B2 and B5 will share the same cache line. Upon execution, nodes B2 and B3 will constantly replace nodes B5 and B6 in the cache, respectively, leading to a total of 200 conflict cache misses. Consequently, the aggregate energy consumption (refer to Table II) rises steeply to 2150 units, which is about three times larger than that of the system without a scratchpad. The present situation in which frequently executed nodes (B2 and B5) recurrently replace each other in the cache is known as cache thrashing.

Let us assume that we have a scratchpad of size 2 cache lines present in the system. Then, the approach [20] will move

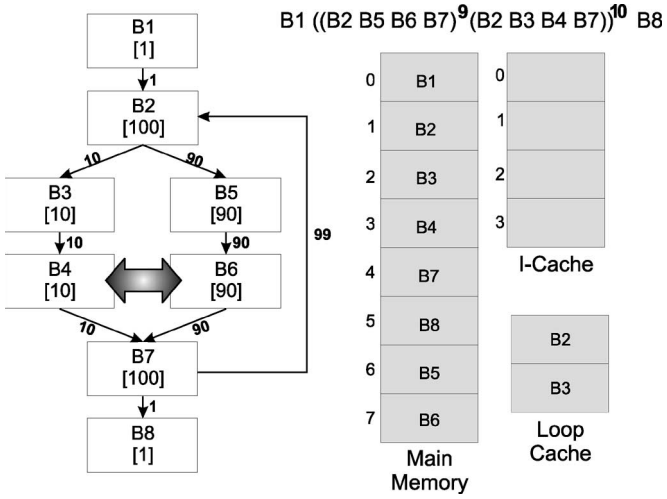

Fig. 6. Motivating example (loop-cache approach).

TABLE IV
ENERGY VALUES FOR LOOP CACHE (ONE WORD)

BB	I-Cache Accesses	I-Cache Misses	Loop Cache Accesses	Energy
BB2	0	0	100	50
BB3	10	10	0	100
BB4	10	10	0	100
BB5	90	10	0	180
BB6	90	10	0	180
BB7	100	0	0	100
Total	300	40	100	710

nodes B2 and B7 onto the scratchpad. While the number of accesses to the scratchpad doubles, the number of cache misses remain constant at 40. The aggregate energy consumption for the present setup, as shown in Table III, is 660 units.

C. Example (Loop-Cache Approach)

Let us assume a system with a preloaded loop cache (refer to Fig. 6) with a capacity of one cache line is present. Also, the preloaded loop cache is assumed to be able to accommodate only one loop. According to the allocation strategy of preloaded loop caches [10], only loops and functions can be fully or partially allocated to the loop cache. Moreover, allocation of loops or functions can only begin from the starting basic block and can only be extended to the contiguous basic block in memory. Finally, loops and functions that are preloaded onto the loop cache are copied instead of moved. As a consequence, the program memory layout remains invariant. In the presented system, the loop cache will be preloaded with node B2. As shown in Table IV, the number of cache misses remains pegged at 40 and the aggregate energy consumption of the system is 710 units.

Let us assume that we have a loop cache of size 2 cache lines present in the system. According to the allocation strategy, nodes B2 and B3 will be allocated to the loop cache. In the present setup, the number of cache misses (cf. Table V) is reduced to 20 and the total energy consumption is 525 units.

TABLE V
ENERGY VALUES FOR LOOP CACHE (TWO WORDS)

BB	I-Cache Accesses	I-Cache Misses	Loop Cache Accesses	Energy
BB2	0	0	100	50
BB3	0	0	10	5
BB4	10	10	0	100
BB5	90	0	0	90
BB6	90	10	0	180
BB7	100	0	0	100
Total	290	20	110	525

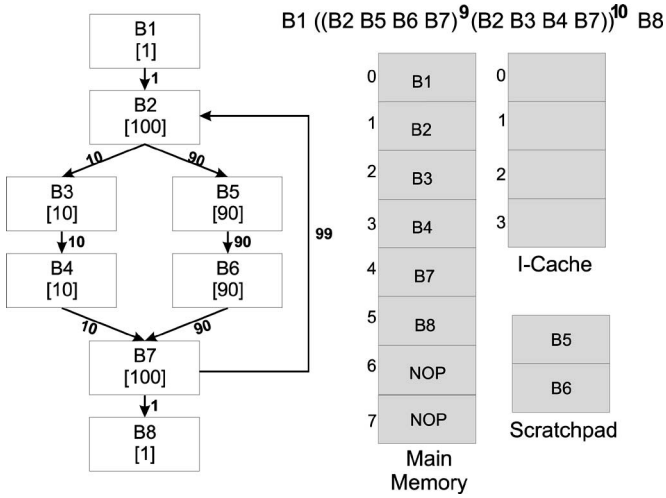

Fig. 7. Motivating example (proposed approach).

TABLE VI
ENERGY VALUES FOR SCRATCHPAD (ONE WORD)

BB	I-Cache Accesses	I-Cache Misses	Scratchpad Accesses	Energy
BB2	100	0	0	100
BB3	10	0	0	10
BB4	10	10	0	100
BB5	0	0	90	45
BB6	90	10	0	180
BB7	100	0	0	100
Total	310	20	90	535

D. Example (Proposed Approach)

We will now demonstrate the effectiveness of our approach for a system containing a scratchpad along with an I-cache. Our approach uses a precise energy model based on cache hits and cache misses. Consequently, our approach chooses the most energy-consuming basic blocks for allocation to the scratchpad. Our approach also keeps the program memory layout invariant (cf. Fig. 7) by copying basic blocks on the scratchpad and replacing the basic blocks in the main memory by No-Operation (NOP) instructions. Assuming that a scratchpad of size 1 cache line is present, then our approach would choose the most energy-consuming node to be allocated to the scratchpad. From Table I, we observe that both nodes B5 and B6 consume the maximum amount of energy. Consequently, our approach can choose between one of the two nodes. As shown in Table VI,

TABLE VII
ENERGY VALUES FOR SCRATCHPAD (TWO WORDS)

BB	I-Cache Accesses	I-Cache Misses	Scratchpad Accesses	Energy
BB2	100	0	0	100
BB3	10	0	0	10
BB4	10	0	0	10
BB5	0	0	90	45
BB6	0	0	90	45
BB7	100	0	0	100
Total	220	0	180	310

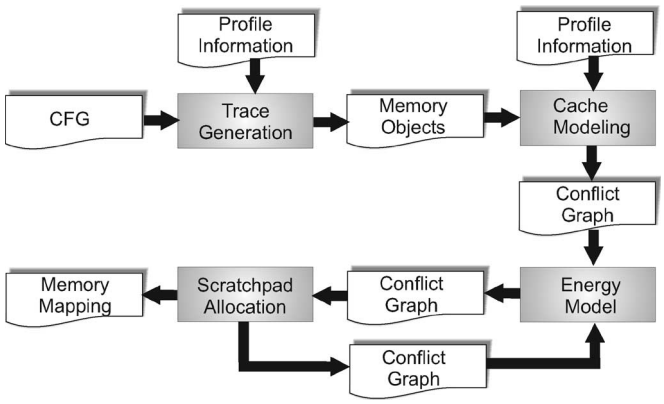

Fig. 8. Workflow of scratchpad-allocation algorithms.

this reduces the number of cache misses by half, and the aggregate energy consumption is also reduced to 535 units.

Next, we assume that we have scratchpad of size 2 cache lines present in the system. In this case, both nodes B5 and B6 are copied to the scratchpad by our approach. Consequently, our approach eliminates all cache misses, and thus minimizes the energy consumption. The energy consumption of the system (cf. Table VII) is 310 units, the minimum compared against all the previously presented approaches.

IV. PRELIMINARIES

We start by describing the assumed architecture for the current research work. Before explaining the technique in detail, an overview is provided in Section IV-B. The generation of MOs is described in the following section. The interaction of MOs within the cache is represented using a conflict graph, which forms the basis of the proposed energy model and the algorithm.

A. Architecture

In this paper, we assume a Harvard architecture [see Fig. 3(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 noncacheable location for fetching instructions. Instruction fetches that do not access the scratchpad address space are accessed through the cache. The data cache shown in Fig. 3(a) is irrelevant since in the present paper, only instructions are being utilized for preloading the scratchpad. However, we would like to extend the present approach to allocate data along with instructions to the scratchpad. As shown in Fig. 3(b), the preloaded-loop-cache setup is similar to using a scratchpad.

B. Workflow

The scratchpad-allocation algorithms presented in the following sections are based upon the workflow shown in Fig. 8. The first step of the proposed approaches is trace generation. This step works upon the input CFG and the profiling information to generate traces [21]. A trace possesses the property of being independent of other traces. In other words, placement of a trace at any location in the memory hierarchy does not affect any other trace. Consequently, a trace is assumed to be an atomic unit of the approach and is referred to as a memory object. The second step analyzes the interaction of MOs within the cache. The behavior of the cache is modeled as a conflict graph with nodes denoting the MOs and edges between two nodes denoting the conflict relationship between two corresponding MOs. In the third step, an accurate energy model is used to compute the energy-consumption function for each MO. This information is attributed to each node in the conflict graph. Finally, based upon the conflict graph, the scratchpad-allocation algorithm determines a mapping of MOs. MOs are mapped to noncacheable scratchpad memory or to cacheable main memory in order to minimize the energy consumption of the system. The ILP-based allocation algorithm selects the MOs to be mapped to the scratchpad in a single step. In contrast, the greedy heuristic iterates over the conflict graph, every time removing the most energy-consuming MO from the conflict graph and mapping it on to the scratchpad. In the following section, we provide a detailed description of the trace-generation step.

C. Trace Generation (MOs)

In the first step of our approach, memory objects within the program code are identified. The well-known compiler optimization trace generation is utilized to identify the MOs. A trace is a frequently executed straight-line path consisting of basic blocks connected by fall-through edges [21]. Dynamic profiling of the application is required to determine the execution count of each basic block as well as information regarding the control flow. The authors in [21] proved that the problem of trace generation is NP-complete. Consequently, we use a greedy heuristic similar to the one used in Trimaran's C compiler [22], with the only difference being that our traces are kept smaller than the scratchpad size, as larger traces cannot be placed on to the scratchpad. A one-to-one relationship between cache misses and corresponding traces is ensured by appending the traces with NOP instructions to align them to cache-line boundaries. Due to the fact that traces always end with an unconditional jump, these appended NOP instructions are never executed. However, the addition of NOP instructions causes an increase in the program size, which can be neglected under the assumption that the main memory is usually large enough to hold the program and is available in predetermined sizes (e.g., 512, 1024 kB, etc.). Traces result in substantial performance improvements due to the following couple of reasons.

```
Trace-Generation(CFG(X,E), MaxTraceSize)
1    TraceList = NIL
2    while (X ≠ {} ) do
3       Let N be the maximum weight node of X
4       CurrTrace = NIL
5       CanBeExtended = TRUE
6       CurrTrace = Append(CurrTrace,N)
7       while (CanBeExtended = TRUE) do
8          Let P be the maximum weight node of Parent(Head(CurrTrace))
9          Let C be the maximum weight node of Children(Tail(CurrTrace))
10         choice = ((P ≠ NIL) and (Size(CurrTrace) + Size(P) ≤ MaxTraceSize)) ? 0 : 1
11         choice = choice ≪ 1
12         choice = ((C ≠ NIL) and (Size(CurrTrace) + Size(C) ≤ MaxTraceSize)) ? 0 : 1
13         switch (choice) do
14            case 0:
15               if (Weight(C) > Weight(P)) then
16                  CurrTrace = Append(CurrTrace,C)
17                  X = X - {C}
18               else
19                  CurrTrace = Prepend(CurrTrace,P)
20                  X = X - {P}
21               end-if
22            break
23            case 1:
24               CurrTrace = Prepend(CurrTrace,P)
25               X = X - {P}
26            break
27            case 2:
28               CurrTrace = Append(CurrTrace,C)
29               X = X - {C}
30            break
31            case 3:
32               CanBeExtended = FALSE
33            break
34         end-switch
35      end-while
36      TraceList = Append(TraceList,CurrTrace)
37   end-while
38   return TraceList
```

Fig. 9. Greedy heuristic for trace-generation problem.

First, the number of cache misses is reduced as frequently executed basic blocks are coalesced to form traces. This reduces the probability of sharing the same cache line by frequently executed basic blocks. Second, due to the fact that jump edges between frequently executed basic blocks are changed to fall-through edges, traces reduce the number of pipeline stalls caused by jump instructions. The improved performance of the processor's pipeline results in improved performance of the processor. The impact of increased program size is minimal when compared against performance improvements achieved by trace generation.

The heuristic used to generate traces is given in Fig. 9. It takes as input a weighted CFG and the maximum trace size and returns a list of traces (MOs) identified from the application. The heuristic starts with the highest weight node and creates a new trace consisting of the current node. A trace can be extended upwards by coalescing a parent or downwards by coalescing a child into the trace. The heuristic makes a greedy choice by considering only the highest weight parent and the highest weight child as candidates for extending the trace. However, extending the trace is feasible only if the size of the trace after coalescing remains less than the maximum trace size. The variable `choice` encodes the feasibility of extending the trace in the upward or downward direction. The process of extending the trace continues until no valid candidate exists or the trace cannot be extended without violating the maximum trace size constraint. After this, the trace is appended to the list of traces. New traces are generated and extended as long as there exists a node that is not coalesced in any trace.

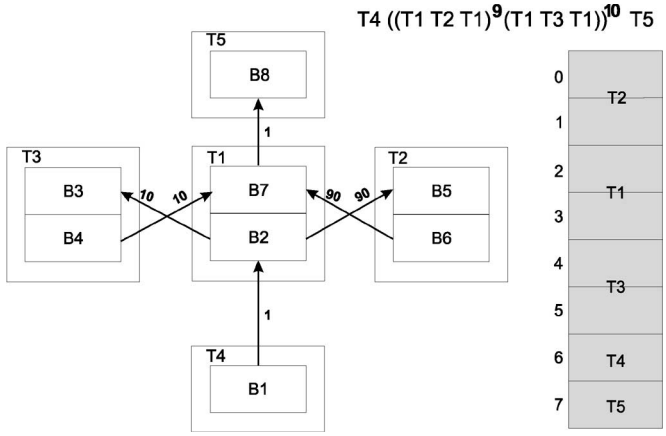

Fig. 10. Motivating example (after trace generation).

Applying the trace-generation step to the motivating example results in the modified CFG shown in Fig. 10. The trace-generation heuristic was applied to the weighted CFG of Fig. 4, and the maximum trace size was assumed to be two words. Fig. 10 represents the CFG, the main memory layout, and the execution trace at the granularity of traces or MOs. Let us explain the heuristic for trace generation with the help of the CFG given in Fig. 4. In the first iteration, node B2 is chosen as the maximum weighted node and a new trace called T1 incorporating B2 is created. The maximum weight parent and child (cf., line numbers 8 and 9 of Fig. 9) of node B2 are B7 and B5, respectively. The trace T1 is extended by prepending node B7. The size of the trace is now equal to two words, and hence cannot be extended any further. In the second iteration of the while loop (cf., line 2 of Fig. 9), we choose node B5 and create a new trace T2, which is then extended by appending node B6. The CFG generated by the trace-generation step is shown in Fig. 10.

Finally, we summarize the justification for using traces in our approach. First, traces improve the performance of both the cache and the processor by enhancing the spatial locality in the program code. Second, due to the fact that traces always end with an unconditional jump [21], they form an atomic unit of instructions that 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 MOs. In the following section, we represent the cache behavior at the granularity of MOs by a conflict graph.

D. Cache Model (Conflict Graph)

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

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

Similarly, an MO is mapped to cache line(s) depending upon its start address and size. Two MOs 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 Fig. 11), which is defined as follows.

Definition: The conflict graph $G = (X, E)$ is an edge and node weighted directed graph with node set $X = \{x_1, \ldots, x_n\}$. Each vertex x_i in G corresponds to an 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, which occurs due to x_j. The weight f_i of a vertex x_i is the total number of instruction fetches within x_i.

A conflict graph is built using both static and dynamic analyses of a program. A node is created for every MO identified in the program code. The address range of every MO is analyzed. For all pairs of MOs that share a common cache line, two directed edges are created, connecting the corresponding nodes in the conflict graph. The weight of each node and each edge in the conflict graph is initialized to zero. Dynamic profiling of the application is required to compute the real weight of each node and each edge. Since the trace-generation step modifies the program memory layout, the profiling information that was used for trace generation cannot be annotated to the conflict graph. Consequently, we again profile the application after the trace-generation step. The total number of instruction fetches of an MO is attributed as the weight of the corresponding node. The number of conflict misses of MO m_i caused due to MO m_j is attributed as the weight of the directed edge e_{ij} from node x_i to node x_j. Finally, the graph is pruned to remove nodes and edges with zero weights. In order to minimize the influence of the chosen input data set on the conflict graph, average values generated by using several distinct input vectors can be used.

The conflict graph, as shown in Fig. 11, is a directed graph because the conflict relationship is antisymmetric. The conflict graph has the advantage that it can precisely model a wide range of cache memories. Any cache with a fixed set of parameters (e.g., associativity, size, replacement policy, etc.) can be represented using a conflict graph. The conflict graph G and the energy values are utilized to compute the energy consumption of an MO according to the energy model proposed in the following section.

E. 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

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

where E_{Cache} can be computed as

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

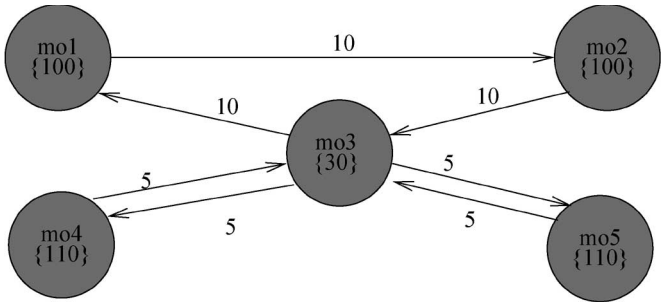

Fig. 11. Cache model (conflict graph).

where functions $\mathrm{Hit}(x_i)$ and $\mathrm{Miss}(x_i)$ return the number of hits and misses, respectively, while fetching the instructions of MO x_i. $E_{\mathrm{Cache\_hit}}$ is the energy of a hit and $E_{\mathrm{Cache\_miss}}$ is the energy of a miss in one line of the I-cache

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

where $\mathrm{Miss}(x_i, x_j)$ denotes the number of conflict cache misses of MO x_i caused due to conflicts with MO x_j and corresponds to the weights of the edge e_{ij} in the conflict graph. The neighbor set of MO x_i is represented by N_i in the above equation. The sum of the number of hits and misses is equal to the number of instruction fetches f_i within an MO x_i

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

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 $\mathrm{Miss}(x_i)$ from (3) and $\mathrm{Hit}(x_i)$ from (4) in (2) and rearranging derives

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

The first term in (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 just the constant term in its energy model, and thus 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:

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

where $E_{\mathrm{SP\_hit}}$ is the energy per access to the scratchpad.

V. Problem Formulation

Once we have created the conflict graph G annotated with vertex and edge weights, the energy consumption of MOs can be computed. Now, the problem is to select a subset of MOs that

minimizes the number of conflict edges and the overall energy consumption of the system. The subset should be 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 MO x_i in the memory hierarchy

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

Since an MO allocated to the scratchpad does not conflict with other MOs, we can represent $\text{Miss}(x_i, x_j)$ as follows

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

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

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

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

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

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

$$E(x_i) = f_i * E_{\text{SP\_hit}} + f_i * [E_{\text{Cache\_hit}} - E_{\text{SP\_hit}}] * l(x_i) + [E_{\text{Cache\_miss}} - E_{\text{Cache\_hit}}] * \left[\sum_{j \in N_i} l(x_j) * l(x_i) * m_{ij} \right]. \quad (11)$$

We find that the last term is a quadratic-degree term, since the number of misses of an MO x_i not only depends upon its location but also upon the location of the conflicting MOs x_j. The objective function E_{Total} denoting the total energy consumed by the system needs to be minimized.

$$E_{\text{Total}} = \sum_{x_i \in X} E(x_i). \quad (12)$$

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

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

The size $S(x_i)$ of MO x_i is computed without considering the appended NOP instructions. The NOP instructions were only appended to align MOs to cache-line boundaries. Since scratchpad is a noncacheable memory, these NOP instructions are stripped away from the MOs prior to allocating them to the scratchpad. The underlying assumption in the problem formulation is that no new edges in the conflict graph or no new conflict relationships are created when an MO is mapped to the noncacheable scratchpad. This assumption can be partially fulfilled by keeping the program memory layout invariant. However, the assumption may fail for certain graphs with circular edge dependences. Solving the nonlinear optimization problem results in 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 (13) for every scratchpad. An additional constraint ensuring that an MO is assigned to at most one scratchpad is also required.

The above optimization problem is related to two NP-complete problems, viz., weighted vertex cover [9] and knapsack [9] problem. Let us make the simplifying assumption that the cache present in the system is large or high-associative enough to hold all the MOs such that not a single conflict cache miss occurs. The energy consumption of an MO under the above assumption becomes independent of other MOs, and 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 $E_{\text{SP\_hit}}$ is equal to the energy of a cache hit $E_{\text{Cache\_hit}}$, (11) transforms to the following form and the problem is reduced to the weighted vertex cover problem

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

Fortunately, approximation algorithms can be employed to obtain near-optimum solutions in polynomial time. We will propose two techniques to solve the problem optimally as well as near optimally. In the following section, we will present an ILP-based technique to solve the problem optimally. After that, we will present a greedy heuristic that solves the scratchpad-allocation problem near optimally with a better runtime complexity.

A. Integer Linear Programming (ILP)

In order to formulate a 0–1 ILP problem, we need to linearize the scratchpad-allocation problem. This can be achieved by replacing the nonlinear term $l(x_i) * l(x_j)$ of (11) by an additional binary variable $L(x_i, x_j)$

$$E(x_i) = f_i * E_{\text{SP\_hit}} + f_i * [E_{\text{Cache\_hit}} - E_{\text{SP\_hit}}] * l(x_i) + [E_{\text{Cache\_miss}} - E_{\text{Cache\_hit}}] * \left[\sum_{j \in N_i} L(x_i, x_j) * m_{ij} \right]. \quad (15)$$

Greedy-Heuristic(G(X,E), ScratchpadSize)
1 Rem_SPSize = ScratchpadSize
2 L = NIL
3 **while** ($\exists x \in X : S(x) \leq$ Rem_SPSize) **do**
4 select $x_i \in X$ such that $S(x_i) \leq$ Rem_SPSize and $E(x_i) > E(x_k)\ \forall x_k \in X : S(x_k) \leq$ 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 $L = Append(L, \{x_i\})$
9 **end-while**
10 **return** L

Fig. 12. Greedy heuristic for scratchpad-allocation problem.

In order to prevent the linearizing variable $L(x_i, x_j)$ from assuming 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 in the current formulation

$$E_{\text{Total}} = \sum_{x_i \in X} E(x_i) \tag{19}$$

$$\sum_{x_i \in X} [1 - l(x_i)] * S(x_i) \leq \text{ScratchpadSize}. \tag{20}$$

A commercial ILP Solver [7] is used to obtain an optimal subset of MOs that minimizes the objective function. The number of vertices $|X|$ of the conflict graph G is equal to the number of MOs, 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 [7] was found to be less than a 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 noncommercial tools (e.g., lp_solve [5]) are used. In the following section, we describe the greedy heuristic utilized for obtaining near-optimum solutions.

B. 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 MOs to be allocated onto the scratchpad. The heuristic is presented in Fig. 12.

The heuristic iteratively computes the energy consumption of each MO that can be placed on the scratchpad memory, considering not only execution counts but also the number of conflict cache misses caused by other MOs. 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.

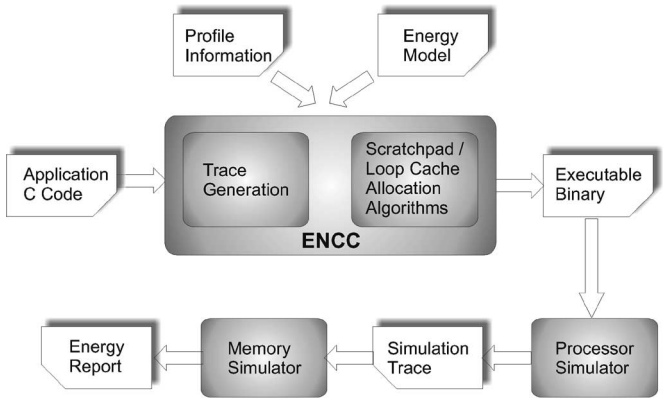

Fig. 13. Experimental workflow.

An MO present in the scratchpad does not conflict with the MOs present in the cache. The energy of the conflicting MOs is thus reduced by removing the vertex and the adjacent edges from the conflict graph. The energy consumption of an MO x_i is computed according to the energy model proposed in Section IV-E. The heuristic iterates as long as there exists an MO that can be placed on the scratchpad without violating the scratchpad-size constraint. On termination, a list of MOs to be allocated onto the scratchpad is returned. The time complexity of the heuristic is $O(\text{ScratchpadSize} * (|X| + |E|))$ if we precompute and store the energy consumption of each MO x_i at the end of each "while loop" iteration. In the next section, we describe the experimental setup used for conducting experiments.

VI. Experimental Setup

The experimental setup consists of an ARM7T processor core, on-chip instruction and data caches, an on-chip scratchpad, and an off-chip main memory. We perform extensive experiments on direct-mapped as well as on two- and four-way set-associative instruction caches of varying sizes. We determine the effect of scratchpad-allocation techniques on the energy consumption of the instruction-memory subsystem. The cacti cache model [24] was used to calculate the energy consumption per access to a cache, loop cache, and scratchpad memory, and all are assumed to be on chip and use 0.5-μm technology. The loop cache was assumed to hold a maximum of four loops. The energy consumption of the main memory was measured from our evaluation board [19]. The time to access off-chip and on-chip memories (I-cache, D-cache, scratchpad, and loop cache) was assumed to be ten CPU cycles and one CPU cycle, respectively.

Experiments were conducted according to the workflow presented in Fig. 13. In the first step, the benchmark programs are compiled using an energy-aware C compiler (ENCC) [8]. Trace generation [21] is a well-known I-cache performance-optimization technique, and hence, for a fair comparison, traces are generated for all the allocation techniques. In the following step, the scratchpad-allocation algorithm is applied, which can either be the greedy heuristic (cf., Section V-B), the

ILP-based allocation algorithm (cf., Section V-A), or Steinke's scratchpad-allocation algorithm [20]. The generated machine code is then fed into the instruction set simulator from advanced RISC machines (ARM) Ltd. [1] to obtain the instruction trace. The instruction trace is fed into a cycle accurate memory hierarchy simulator (MEMSIM) [15]. On the basis of the instruction trace, the memory hierarchy, and the energy-cost model, it computes the number of CPU cycles and 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 Ross's allocation algorithm [10]. The memory subsystem consists of an I-cache, a preloaded loop cache, and the main memory. The energy consumption of the memory subsystem is again computed by using our memory-hierarchy simulator [15].

VII. Results

A subset of benchmarks from the Mediabench suite was used to substantiate our claims concerning energy savings using the proposed algorithm. The benchmarks along with their corresponding program size are mentioned in Table VIII. Experiments were conducted by varying the size of the scratchpad/loop cache as well as the size and the associativity of the I-cache present in the system, while the I-cache line size was kept constant at 16 B. The number and the type of accesses to every memory in the hierarchy was counted by the memory simulator. Based on this information and the energy model presented in Section IV-E, the energy consumption of the instruction-memory subsystem was computed. The number of CPU cycles spent during the execution of the benchmarks was also determined.

In the following section, we first determine the most energy-efficient I-cache architecture for the mpeg benchmark. Next, we demonstrate that by the addition of a scratchpad, far better energy savings can be achieved even with a smaller instruction-memory hierarchy. In Section VII-C, we demonstrate that in order to achieve energy savings along with the addition of a scratchpad, an efficient allocation algorithm is also required. A comparison between the proposed allocation algorithm (cf., Section V-A) and a known allocation algorithm [20] under varying I-cache parameters is presented in Section VII-D. In Section VII-E, experimental results are presented to determine the best scratchpad size for a given I-cache configuration. A comparison between the scratchpad allocated with the proposed optimal algorithm and the loop cache is presented in Section VII-F. We end the experimental section by an overall comparison of the scratchpad allocated with all the mentioned allocation algorithms and the loop cache in Section VII-G.

A. Energy-Efficient Instruction-Cache Architecture

In the current section, we assume that the instruction-memory hierarchy consists of an I-cache and the main memory. Experiments are conducted to determine the most energy-efficient I-cache for the mpeg benchmark by varying the size and the associativity of the I-cache. The energy consumption of the instruction-memory subsystem is computed using the memory-hierarchy simulator [15]. Fig. 14 presents the energy consumption as a function of the size for three different set-associative I-caches. From Fig. 14, it is easy to observe that for each associativity, the energy consumption of the instruction-memory subsystem monotonically decreases down to the minimum point with the increase in the I-cache size. Thereafter, the energy consumption increases with further increase in size. This happens because a large cache would cause a reduced number of cache misses but would also require a large energy per access. The minimum energy points represent the most energy-efficient I-cache size for each associativity. It is easy to observe that I-caches of size 2, 4, and 8 kB are the minimum energy sizes for four- and two-way set-associative and direct-mapped I-caches, respectively. Moreover, from Fig. 14, we observe that the 8-kB direct-mapped I-cache is the most energy-efficient I-cache configuration for the mpeg benchmark. Next, we justify the addition of a scratchpad in the memory hierarchy.

TABLE VIII
MEDIABENCH BENCHMARKS

Benchmark	Program Size	I-Cache Size
adpcm	1kB	128B DM
g721	4.7kB	1kB DM
epic	12kB	1kB DM
mpeg	21.4kB	2kB DM

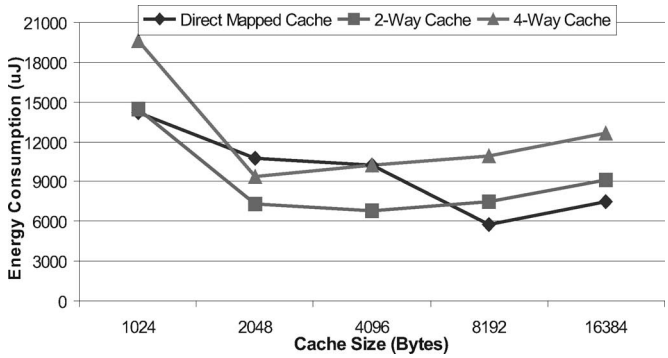

Fig. 14. Energy consumption of instruction memory for mpeg.

B. Benefits of Instruction-Cache- and Scratchpad-Based Systems

Experiments for the present section were conducted assuming that the instruction-memory subsystem comprises a scratchpad along with an I-cache and the main memory. The scratchpad is allocated with the ILP-based optimal algorithm presented in Section V-A. Fig. 15 represents the relative energy consumption of the instruction-memory subsystem consisting of a 512- or 1024-B scratchpad. The energy consumption of the most energy-efficient memory hierarchy found in the previous section is represented as the 100% baseline. There are a couple of important observations to be made from Fig. 15. First, except for the 1-kB four-way I-cache, the energy consumption of the system with a 1024-B scratchpad is always better than that of the system with the most energy-efficient I-cache. Second, the energy consumption of the system with 1-kB I-cache and 1 kB of scratchpad is about 60% of the energy-efficient 8-kB I-cache-based system. Consequently, the effectiveness of the

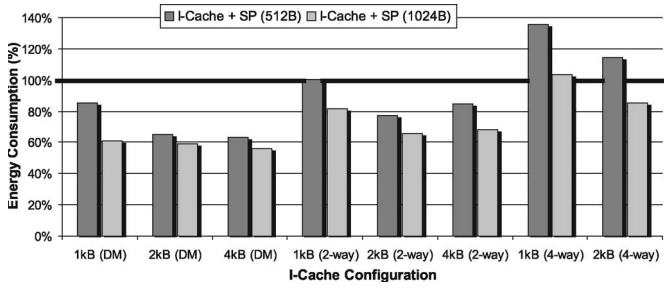

Fig. 15. Comparison of energy consumption of I-cache + scratchpad versus 8-kB I-cache for mpeg.

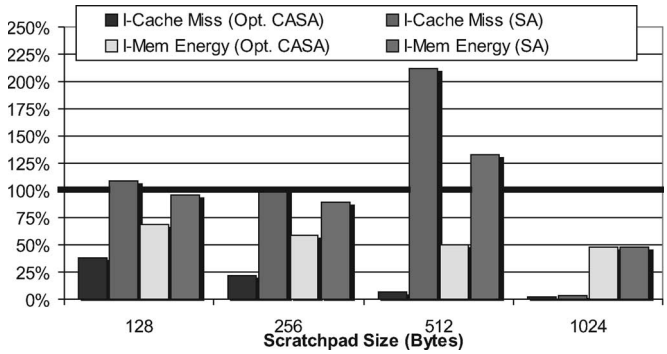

Fig. 16. Cache behavior for epic benchmark.

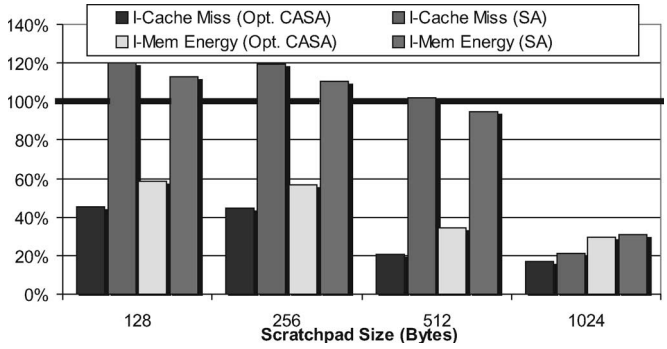

Fig. 17. Cache behavior for mpeg benchmark.

scratchpad-based system is demonstrated by the fact that the energy consumption of the system with the most energy-efficient I-cache can be reduced to 60% by a memory hierarchy whose aggregate on-chip size (2 kB) is merely 25% of the on-chip size of the original memory hierarchy. In the following section, we stress the need for an effective allocation algorithm for a scratchpad-based instruction-memory subsystem.

C. Comparison of Scratchpad-Allocation Algorithms

We will start by demonstrating the negative implications of not modeling a cache by Steinke's approach. Fig. 16 displays the number of I-cache misses and the energy consumption of the epic benchmark using Steinke's algorithm and the proposed ILP-based algorithm. In order to enable a better appreciation of the results, the number of cache misses and energy-consumption values are presented as percentages of a 100% baseline. The baseline in Fig. 16 represents the number of cache misses and the energy consumption of the system without a

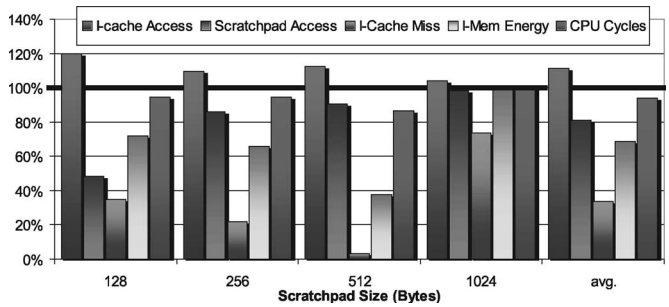

Fig. 18. Comparison of scratchpad (ILP) against scratchpad (Steinke) for epic.

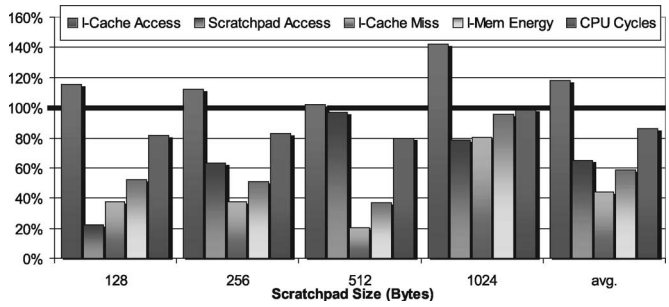

Fig. 19. Comparison of scratchpad (ILP) against scratchpad (Steinke) for mpeg.

scratchpad. The 100% baseline is independent of the scratchpad size and is pegged at a constant value for all the experiments. From Fig. 16, we observe that the number of cache misses for Steinke's approach demonstrate unpredictable behavior. At 128 B of scratchpad, the number of cache misses are slightly higher than the baseline, but are lower for 256 B. However, at 512 B, a steep rise in the number of cache misses is observed. This phenomenon of excessive increase in cache misses is known as cache thrashing, and also causes an increase in the energy consumption of the system. On the other hand, the number of cache misses and the energy-consumption values monotonically decrease with the increase in scratchpad size for our proposed method. This characteristic behavior of our approach originates from the precise cache and energy models. A similar behavior of Steinke's algorithm for the mpeg benchmark can be observed from Fig. 17. Again, the baseline represents the number of cache misses and the energy consumption of the mpeg benchmark on a system without a scratchpad. For Steinke's algorithm, the energy consumption is higher than the baseline for scratchpad sizes of 128 and 256 B. The high number of cache misses for these scratchpad sizes nullify any gains achieved by the use of an energy-efficient scratchpad.

A detailed comparison of all the parameters of the instruction-memory subsystem is presented to enable a better appreciation of the results. Figs. 18 and 19 display the energy consumption of the instruction-memory subsystem with all the respective parameters (i.e., scratchpad accesses, cache accesses, and cache misses) of the ILP-based proposed algorithm for the epic and the mpeg benchmarks, respectively. The last column of the figures displays the execution time of the two benchmarks in terms of CPU cycles. A direct-mapped I-cache of size 1 and 2 kB is assumed to be present in the system for the

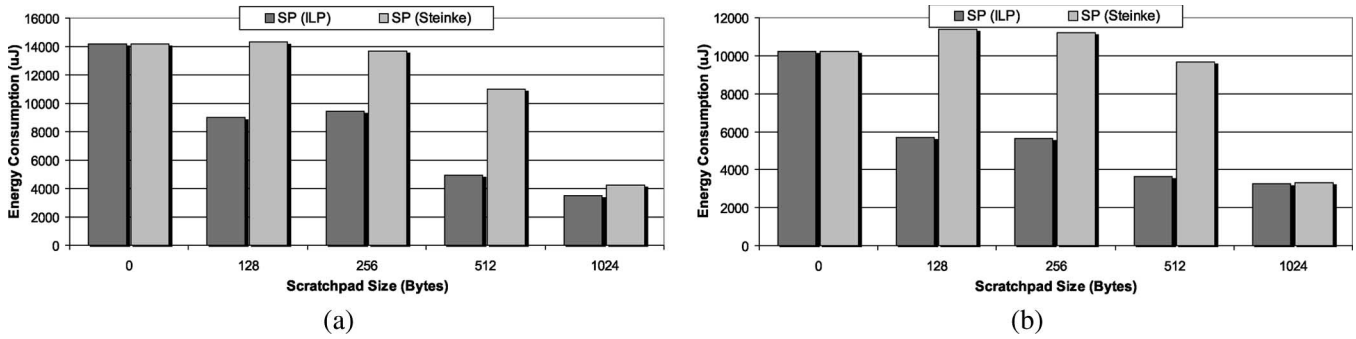

Fig. 20. Energy consumption of ILP and Steinke's algorithm for mpeg. (a) 1-kB direct mapped (DM) I-cache. (b) 4-kB DM I-cache.

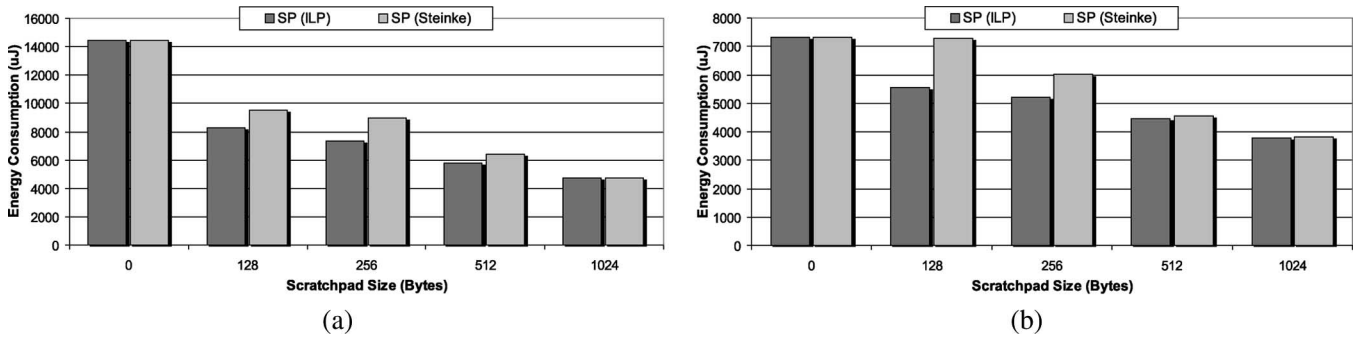

Fig. 21. Energy consumption of ILP and Steinke's algorithm for mpeg. (a) 1-kB two-way I-cache. (b) 2-kB two-way I-cache.

epic and the mpeg benchmarks, respectively. The results of the proposed algorithm are compared against the corresponding results of Steinke's algorithm [20]. Unlike the baseline of Fig. 16, the 100% baseline in Fig. 18 represent the varying experimental values achieved by Steinke's algorithm for each scratchpad size. However, it enables a direct comparison of the proposed approach against Steinke's approach at each scratchpad size. It should be noted that the I-cache misses and the energy consumption of the system (see Fig. 16) for the proposed approach decreases monotonically with the increase in the scratchpad size.

From Fig. 18, it is interesting to note that in spite of the higher I-cache accesses and lower scratchpad accesses, the pr oposed algorithm reduces energy consumption against Steinke's algorithm. The substantially lower I-cache misses are able to overcompensate for higher I-cache accesses and result in reduced energy consumption. The justification for this behavior is that Steinke's algorithm tries to reduce energy consumption by increasing the number of accesses to the energy-efficient scratchpad. In contrast, our algorithm reduces energy-consuming I-cache misses by assigning conflicting MOs to the scratchpad. Since on every cache miss, the slow and power-hungry main memory is accessed, avoiding cache misses is beneficial both in terms of energy and performance. For the mpeg benchmark, our algorithm achieves up to 80% reduction in I-cache misses and, as a consequence, results in substantial savings in energy and execution time. The algorithm, on the average, conserves 26% and 42% energy against Steinke's algorithm for the epic and the mpeg benchmarks, respectively. A reduction of 14% in CPU cycles is also reported for the mpeg benchmark. However, at 1024 B, the reduction in energy consumption and the execution time using our algorithm is minimal. The reason for this behavior is that the scratchpad of size 1024 B is large enough to hold all important MOs and, as a consequence, the solution sets of the previous approach as well as our approach are fairly similar. Next, we compare the two allocation algorithms by varying the size and the associativity of the I-cache.

D. Modifying Instruction-Cache Parameters

Up to this point, we compared the scratchpad-allocation algorithms for epic and mpeg benchmarks for systems containing direct-mapped I-caches of 1 and 2 kB, respectively. Now, we compare the proposed ILP-based allocation algorithm against Steinke's algorithm for the mpeg benchmark for the systems with different I-cache configurations. The considered I-cache configurations include 1- and 4-kB direct-mapped I-caches and two- and four-way set-associative I-caches of size 1 and 2 kB.

Fig. 20(a) and (b) presents the energy consumption of the system whose scratchpads are allocated using the two allocation algorithms (viz., ILP and Steinke's) for the mpeg benchmark. Fig. 20(a) presents the energy-consumption values of the system containing 1-kB direct-mapped I-cache, while Fig. 20(b) presents those of the system with 4-kB direct-mapped I-cache. The proposed allocation algorithm always leads to a more energy-efficient allocation than that obtained using Steinke's algorithm. On average, the optimal solution leads to an energy reduction of 35% over Steinke's algorithm for a system with 1-kB direct-mapped I-cache. An even higher average energy reduction of 41% is reported for the system with 4-kB direct-mapped I-cache.

Fig. 21(a) and (b) presents the comparison of the proposed allocation algorithm and Steinke's algorithm for the systems with 1 and 2 kB of two-way set-associative I-cache,

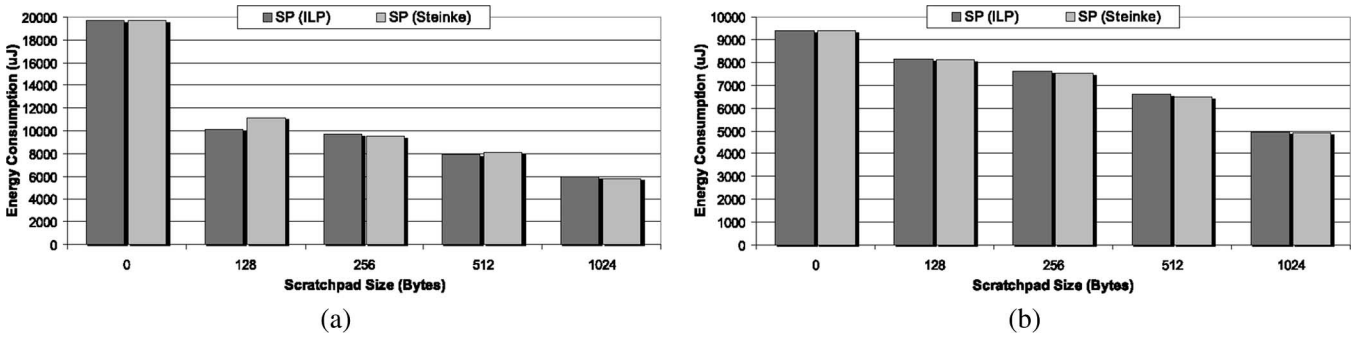

Fig. 22. Energy consumption of ILP and Steinke's algorithm for mpeg. (a) 1-kB four-way I-cache. (b) 2-kB four-way I-cache.

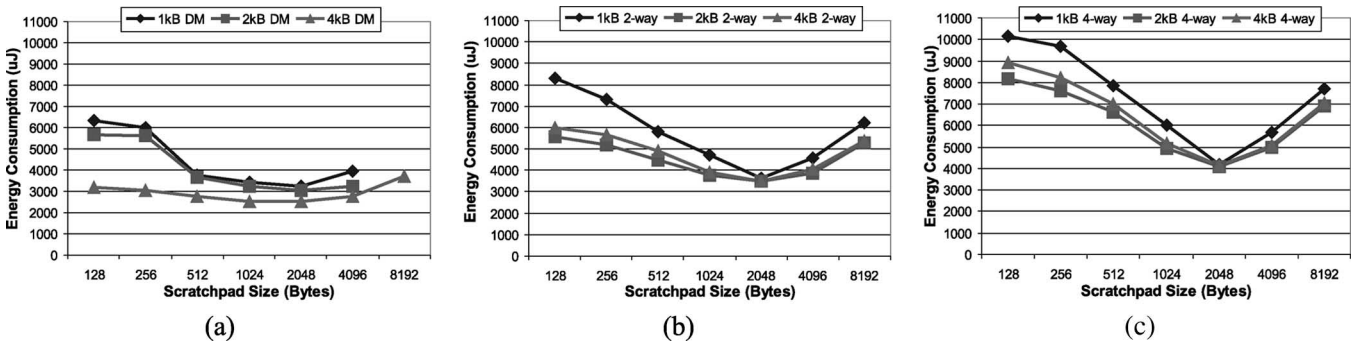

Fig. 23. Determining optimal scratchpad size for mpeg. (a) DM I-cache. (b) Two-way I-cache. (c) Four-way I-cache.

respectively. In Fig. 21(b), Steinke's algorithm again displays unpredictable behavior, as it leads to an increase in cache misses for the 128-B scratchpad. The proposed algorithm performs better than Steinke's algorithm, although the reduction in energy consumption is less than that achieved for direct-mapped I-caches. This behavior is justified as the two-way set-associative I-caches result in a hit ratio of more than 99% for the mpeg benchmark. Nevertheless, the algorithm achieves the maximum energy savings of 19% and 24% against Steinke's algorithm for 1- and 2-kB I-caches, respectively.

The last set of Fig. 22(a) and (b) presents the comparison of the allocation algorithms for 1- and 2-kB four-way set-associative I-cache-based systems, respectively. For high associative I-caches, the number of conflict cache misses are substantially less than those for low associative I-caches. Consequently, very few conflict edges are present in the conflict graph, which are used to model the behavior of MOs present in the I-cache. As discussed in Section V, our allocation problem reduces to the knapsack problem, which also forms the basis of Steinke's approach. Under these conditions, the allocation of the MOs achieved by the proposed approach is similar to that achieved by Steinke's approach. This is corroborated by energy-consumption values presented in Fig. 22.

E. Determining Optimal Scratchpad Size

In the experiments presented so far, MOs were efficiently allocated onto the given memory hierarchy, consisting of a scratchpad and an I-cache memory. In the current section, we present the results of the experiments conducted to determine the optimal scratchpad for the mpeg benchmark. The experimental results also determine the set of pareto-optimal scratchpads for the mpeg benchmark. The experiments were conducted by increasing the scratchpad size from 128 B to 8 kB, and the energy consumption of the system was computed. The scratchpad present in the system was allocated using the algorithm presented in Section V-A. Fig. 23(a) presents the energy consumption of the system with a direct-mapped I-cache and a varying scratchpad. Additional experiments were conducted by also varying the size and the associativity of the I-cache memory. Fig. 23(b) and (c) presents the energy consumption of the systems consisting of two- and four-way set-associative I-caches, respectively.

From Fig. 23, we observe that the energy consumption of the system decreases as we increase the scratchpad size till it reaches the minimum point. Any further increase in the scratchpad size also increases the energy consumption of the system. As shown in Fig. 23(a), the minimum energy point occurs at 1 kB of scratchpad memory for a 4-kB direct-mapped I-cache-based system. However, for systems with 1- and 2-kB direct-mapped I-caches, 2 kB of scratchpad memory leads to the minimum energy consumption. Similarly, for two- and four-way set-associative I-cache-based systems [cf., Fig. 23(b) and (c)], the minimum energy consumption occurs when 2 kB of scratchpad memory is present in the system. Scratchpad memories larger than the minimum energy configuration scratchpad memory are an unattractive option as compared to an I-cache. The high energy per access to the large scratchpad memory offsets the gains that can be achieved by allocating more MOs to the scratchpad. Consequently, the algorithm allocates more MOs to the I-cache and less to the large scratchpad memory.

The benefits of performing the above set of experiments are threefold. Firstly, we are able to study the variation in the energy consumption of the system with the increase in scratchpad size.

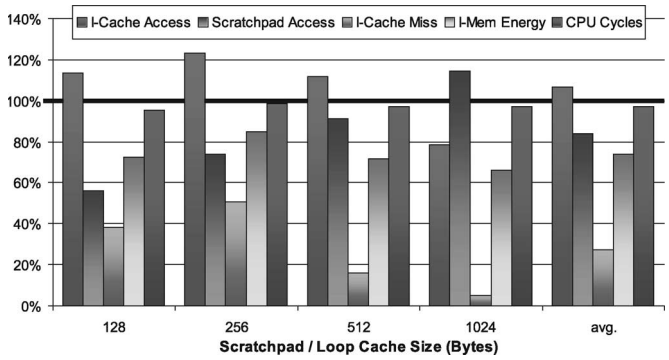

Fig. 24. Comparison of scratchpad (ILP) against loop cache (Ross) for epic.

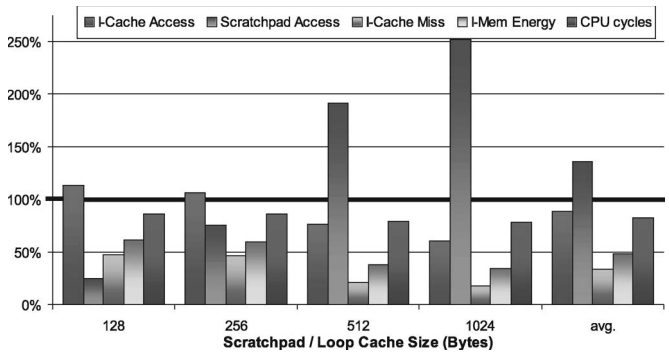

Fig. 25. Comparison of scratchpad (ILP) against loop cache (Ross) for mpeg.

It is interesting to observe that 2 kB of scratchpad memory forms the minimum energy configuration in all but one I-cache configurations. However, the energy consumption of the remaining configuration (1-kB scratchpad and 4-kB I-cache) is the global minimum energy-consumption value. Secondly, we are able to determine the range of scratchpad sizes that would be interesting for a system designer to perform design-space exploration. For the mpeg benchmark, the minimum energy-consumption scratchpad size is 1 or 2 kB. Consequently, scratchpad memories between 128 B and 1 or 2 kB form the set of energy-efficient (pareto-optimal) scratchpad sizes that allow a tradeoff between the on-chip area and the energy consumption of the system. Scratchpads larger than 2 kB consume more on-chip area and also result in increased energy consumption of the system. Hence, they are not part of the energy-efficient range of scratchpad sizes. Finally, an iterative or a binary search-based algorithm can be employed for determining the optimal scratchpad size for a given I-cache-based architecture. In the following section, scratchpad-based systems are compared against loop-cache-based systems.

F. Comparison of Scratchpad and Loop-Cache-Based Systems

In the present section, we compare the energy savings achieved by a scratchpad-based system against those achieved by a preloaded-loop-cache-based system. The scratchpad is allocated using the ILP-based algorithm while the preloaded loop is allocated using Ross's algorithm [10]. Figs. 24 and 25 display the energy consumption of the instruction-memory subsystem and CPU cycles for the epic and the mpeg benchmarks, respectively. The number of scratchpad accesses, I-cache accesses, as well as the I-cache misses are also displayed in the figures. All results are shown as percentages of the corresponding parameters of Ross's algorithm. Similar to Fig. 18, the values of 100% baseline vary for each loop-cache size. The size of the loop cache was assumed to be equal to the size of the scratchpad, even though a loop cache requires more area than a scratchpad due to the presence of a controller. For small scratchpad/loop-cache sizes (128 and 256 B), the number of accesses to the loop cache are higher than those to the scratchpad. However, as we increase the size, the loop cache's performance is restricted by the maximum number of preloadable MOs. The scratchpad, on the other hand, can be preloaded with any number of MOs as long as their aggregate size is less than the scratchpad size. Moreover, the number of I-cache misses is substantially lower for all sizes if a scratchpad allocated with our technique is used instead of a loop cache. Consequently, a scratchpad is able to reduce energy consumption at an average of 24% and 52% against a loop cache for the epic and the mpeg benchmarks, respectively. Average reductions of 6% and 18% in the execution time is also reported for the epic and the mpeg benchmarks, respectively. In the following section, we present the overall comparison of the allocation approaches for scratchpad- and loop-cache-based approaches.

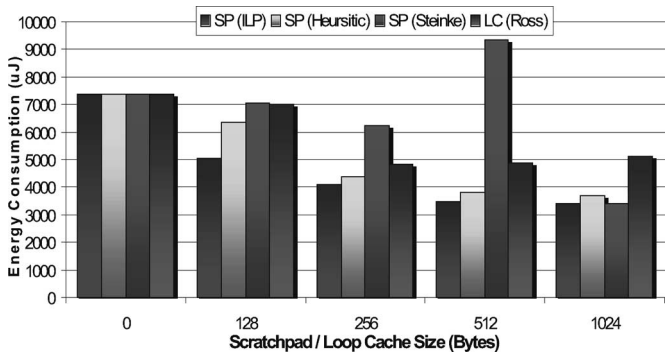

Fig. 26. Energy comparison of ILP, heuristic, Steinke's, and Ross's algorithms for epic.

G. Overall Comparison

In Figs. 26 and 27, we compare the energy consumption of the epic and the mpeg benchmarks for the scratchpad- and loop-cache-based systems. The comparison of the execution time of the epic and the mpeg benchmarks is presented in Figs. 28 and 29, respectively. The scratchpad is allocated using scratchpad-allocation algorithms (viz., ILP, Heuristic, and Steinke's), while Ross's algorithm is used for allocating the loop cache. A few interesting points can be noted from the figures. Firstly, the proposed optimal and near-optimal scratchpad-allocation algorithms result in a monotonically decreasing energy consumption and execution-time behavior of the benchmark. However, the reductions in energy consumption are larger than those in execution time. Secondly, the energy consumption of the benchmarks due to the heuristic is fairly

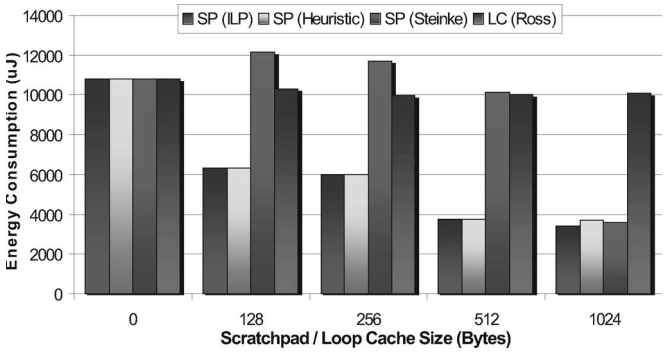

Fig. 27. Energy comparison of ILP, heuristic, Steinke's, and Ross's algorithms for mpeg.

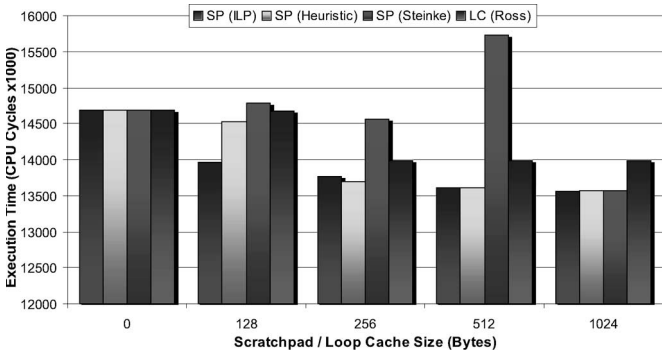

Fig. 28. Execution-time comparison of ILP, heuristic, Steinke's, and Ross's algorithms for epic.

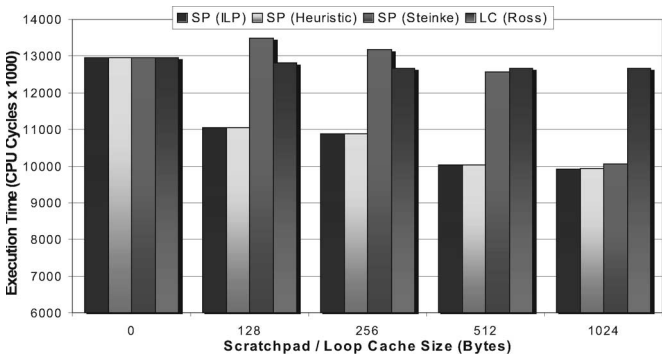

Fig. 29. Execution-time comparison of ILP, heuristic, Steinke's, and Ross's algorithms for epic.

close to the optimal energy consumption achieved by the ILP-based algorithm. Finally, for smaller sizes (128 and 256 B), loop cache performs better than the scratchpad allocated with Steinke's algorithm, while the opposite is true for large sizes.

VIII. Conclusion and Future Work

The essence of this paper is that equal emphasis should be given to both the novel memories and also to the allocation algorithms that lead to their proper utilization. In this paper, we demonstrated that the addition of a scratchpad to an instruction cache leads to substantial savings in the energy consumption and the execution time of the application, as well as reduces the aggregate on-chip area required by the instruction-memory hierarchy. We report energy and on-chip area reductions of 40% and 75% over the most energy-efficient I-cache configuration found for one benchmark. We also presented a generic scratchpad-allocation technique that can handle a variety of complex memory hierarchies. We modeled the allocation problem as a generic nonlinear optimization problem. The problem was solved optimally using an ILP-based approach and also near optimally using a heuristic. The near-optimal solutions obtained by the heuristic were on an average a meager 6.0% and 4.0% worse than the optimal solutions, in terms of energy consumption and performance, respectively. The presented techniques reduce both energy consumption of the system and the execution time of the applications against a published technique. Average reductions of 23.4% and 7.0% in energy consumption and execution time are reported for the ILP-based approach. We also determined the pareto-optimal scratchpad sizes for one benchmark. In addition, we also demonstrated that the simple scratchpad memory allocated with the presented techniques outperforms a preloaded loop cache. Average reductions of 29.4% and 8.7% in energy consumption and execution time, respectively, are also reported. The allocation algorithms were also tested by varying the size and the associativity of the instruction cache present in the system. As a part of future work, we would like to extend our approach by considering allocation of data along with instructions onto the scratchpad and also consider dynamic allocation of both data and instructions at execution time.

References

[1] Advanced RISC Machines Ltd. [Online]. Available: http://www.arm.com/products/CPUs/ARM1156T2-S.html

[2] O. Avissar, R. Barua, and D. Stewart, "An optimal memory allocation scheme for scratch-pad-based embedded systems," *IEEE Trans. Embed. Comput. Syst.*, vol. 1, no. 1, pp. 6–26, Nov. 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. 10th Int. Symp. Hardware/Software Codesign*, Estes Park, CO, May 2002, pp. 73–78.

[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 *Proc. ISLPED*, Monterey, CA, Aug. 1998, pp. 70–75.

[5] M. Berkelaar, *lp_solve: A Mixed Integer Linear Program Solver*. [Online]. Available: ftp://ftp.es.ele.tue.nl/pub/lp_solve

[6] E. Brockmeyer, M. Miranda, H. Corporaal, and F. Cathoor, "Layer assignment techniques for low energy in multi-layered memory organization," in *Proc. DATE*, Munich, Germany, Mar. 2003, pp. 1070–1075.

[7] CPLEX Ltd. [Online]. Available: http://www.cplex.com

[8] ENCC, Univ. Dortmund, Dept. Comput. Sci. XII. [Online]. Available: http://ls12-www.cs.uni-dortmund.de/research/encc

[9] M. Garey and D. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. New York: Freeman, 1979.

[10] S. Gordon-Ross and F. Vahid, "Exploiting fixed programs in embedded systems: A loop cache example," *Comput. Archit. Lett.*, vol. 1, no. 1, p. 2, Jan. 2002.

[11] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*, 3rd ed. San Mateo, CA: Morgan Kaufmann, 2003.

[12] M. Kamble and K. Ghosh, "Analytical energy dissipation models for low power caches," in *Proc. ISLPED*, Monterey, CA, Aug. 1997, pp. 143–148.

[13] M. Kandemir, J. Ramanujam, M. J. Irwin, N. Vijaykrishnan, I. Kadayaif, and A. Parikh, "A compiler-based approach for dynamically managing scratchpad memories in embedded systems," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 23, no. 2, pp. 243–260, Feb. 2004.

[14] L. Lee, B. Moyer, and J. Arends, "Instruction fetch energy reduction using loop caches for embedded applications with small tight loops," in *Proc. ISLPED*, San Diego, CA, Aug. 1999, pp. 267–269.

[15] MEMSIM, Univ. Dortmund, Dept. Comput. Sci. XII. [Online]. Available: http://ls12-www.cs.uni-dortmund.de/~wehmeyer/LOW_POWER/memsim_doc
[16] Motorola Inc. [Online]. Available: http://e-www.motorola.com/files/shared/doc/selector_guide/SG1001.pdf
[17] P. Panda, N. Dutt, and A. Nicolau, *Memory Issues in Embedded Systems-on-Chip*. Norwell, MA: Kluwer, 1999.
[18] P. Pettis and C. Hansen, "Profile guided code positioning," in *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation*, White Plains, NY, Jun. 1990, pp. 16–27.
[19] S. Steinke, M. Knauer, L. Wehmeyer, and P. Marwedel, "An accurate and fine grain instruction-level energy model supporting software optimizations," in *Proc. Int. Workshop PATMOS*, Yverdon-Les-Bains, Switzerland, Sep. 2001, pp. 3.2.1–3.2.10.
[20] S. Steinke, L. Wehmeyer, B.-S. Lee, and P. Marwedel, "Assigning program and data objects to scratchpad for energy reduction," in *Proc. DATE*, Paris, France, Mar. 2002, pp. 409–415.
[21] H. Tomiyama and H. Yasuura, "Optimal code placement of embedded software for instruction caches," in *Proc. 9th ED&TC*, Paris, France, Mar. 1996, pp. 96–101.
[22] Trimaran, University of Illinois at Urbana-Champaign, Georgia Institute of Technology, Hewlett Packard Laboratories. [Online]. Available: http://www.trimaran.org
[23] S. Wilton and N. Jouppi, "An enhanced access and cycle time model for on-chip caches," Western Research Lab., Palo Alto, CA, Tech. Rep. 93/5, Jul. 1994.
[24] ——, "CACTI: An enhanced cache access and cycle time model," *IEEE J. Solid-State Circuits*, vol. 31, no. 5, pp. 677–688, May 1996.
[25] C. Zhang, F. Vahid, and R. Lysecky, "A self-tuning cache architecture for embedded systems," in *Proc. DATE*, Paris, France, Feb. 2004, pp. 142–147.

Manish Verma (S'06) received the B.Tech. degree in computer science and engineering from the Indian Institute of Technology, Delhi, India, in 2001. He is currently working toward the Ph.D. degree at the University of Dortmund, Dortmund, Germany.

His research interests include optimizing compilers and memory-hierarchy optimization of uni- and multiprocessor embedded systems.

Lars Wehmeyer received the diploma degree in computer science from the Technical University of Darmstadt, Darmstadt, Germany.

In August 1999, he joined the Embedded Systems Research Group at the University of Dortmund, Dortmund, Germany. His research interests include the areas of energy-aware code generation and timing predictability, both considering the underlying memory hierarchies.

Peter Marwedel (M'00–SM'05) received the Ph.D. degree in physics and the Dr.habil. degree in computer science from the University of Kiel, Kiel, Germany, in 1974 and 1987, respectively.

He has been a Professor at the University of Dortmund, Dortmund, Germany, since 1989 and is also the head of the local technology transfer centre Informatik Centrum Dortmund. His research interests include high-level synthesis and embedded systems, in particular code generation for embedded systems. His early work includes the design of Machine Independent Microprogramming Language (MIMOLA), a high-level synthesis system. Currently, his focus is on generation techniques for efficient embedded code.

Dr. Marwedel is a Senior Member of the IEEE and a Member of the Association for Computing Machinery and the Gesellschaft für Informatik. He is also a Member of the ARTIST European Network of Excellence on Embedded and Real-Time Systems.