Fine-grain Dynamic Instruction Placement for L0 Scratch-Pad Memory

Jongsoo Park
Stanford University
Stanford, California, USA
jongsoo@cva.stanford.edu

James Balfour*
NVIDIA Research
Santa Clara, California, USA
jbalfour@cva.stanford.edu

William J. Dally
Stanford University
Stanford, California, USA
dally@cva.stanford.edu

ABSTRACT

We present a fine-grain dynamic instruction placement algorithm for small L0 scratch-pad memories (SPMs), whose unit of transfer can be an individual instruction. Our algorithm captures a large fraction of instruction reuse missed by coarse-grain placement algorithms whose unit of transfer is restricted to loops or functions within the capacity of SPMs. Evaluation of L0 SPMs with our fine-grain algorithm in 17 applications shows that the energy consumed by instruction storage hierarchy is reduced by 38% and 31% compared to that of L0 instruction caches and L0 SPMs with an ideal coarse-grain algorithm, respectively.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*Compilers, Code Generation, Optimization*

General Terms

Algorithms, Experimentation, Performance

Keywords

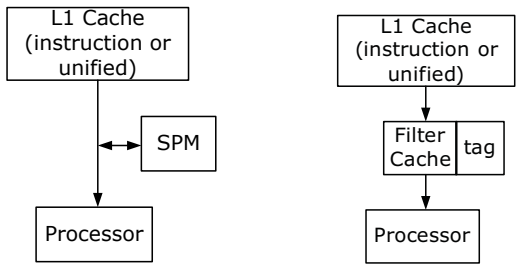
Code Placement, Compilers, Embedded Systems, Scratch-Pad Memory

1. INTRODUCTION

Instruction storage hierarchy accounts for a large fraction of the energy consumed by embedded processors. For example, instruction cache contributes to 27% of the total energy consumed in the StrongARM processor [30]. To reduce *instruction delivery energy* (which in this paper refers to the energy consumed in instruction storage hierarchy), researchers have proposed extending the hierarchy by adding small instruction stores (typically 1KB or smaller) between the L1 instruction cache and the processor [25, 28, 29, 14, 24, 19, 20, 21, 22, 7]; in this paper, we call these *L0 instruction stores*.

Scratch-pad memories [33, 5] (SPMs), shown in Figure 1(a), are compiler-managed stores in which no tags are used to associate locations in SPMs with memory addresses. Therefore, SPMs consume less energy per access than caches with the same capacity, lending themselves to being a natural choice for L0 instruction stores. There are primarily two types of instruction placement algorithms that target SPMs: static and dynamic instruction placement. In static instruction placement algorithms [5, 39, 14, 2, 3, 45, 43, 42, 31], the most frequently executed instructions are identified by the compiler and preloaded prior to starting an application. Throughout the application execution, the set of instructions that reside in the SPM does not change. Due to their static nature, static placement algorithms cannot efficiently utilize the SPM when an application has multiple hotspots that do not fit in the SPM altogether. In dynamic instruction placement algorithms [38, 37, 44, 41, 23, 40, 10, 11, 32], instructions are dynamically transferred into the SPM as needed, thereby utilizing SPM space more efficiently. For example, Udayakumaran *et al.* [40] show that their dynamic placement technique achieves an average of 31% energy reduction over static placement.

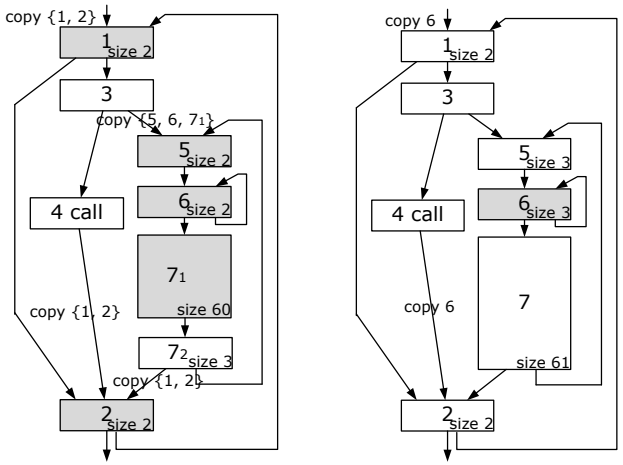
Alternatively, a tag-based design for L0 called *filter cache* [25] (FC), shown in Figure 1(b), can be used. FCs do not require any compiler modification and preserve instruction set compatibility. Given this relative simplicity of FCs, SPMs must have a sufficiently large energy efficiency advantage over FCs to be the preferred choice. Although energy efficiency has been the main motivation for using SPMs [5, 39, 38], they have yet to show a notably higher energy efficiency over FCs'. Section 2 details reasons behind this, one of which is



L0 Scratch-Pad Memory (SPM) (b) Filter cache (FC)

Figure 1: L0 instruction stores

*This work was done when the author was at Stanford University.



(a) fine (b) coarse

Figure 2: An example of (a) a fine-grain dynamic instruction placement and (b) its coarse-grain counterpart when the capacity of the SPM is 64. Blocks placed in the SPM are shaded. Section 4 describes the placement process in detail using the same example.

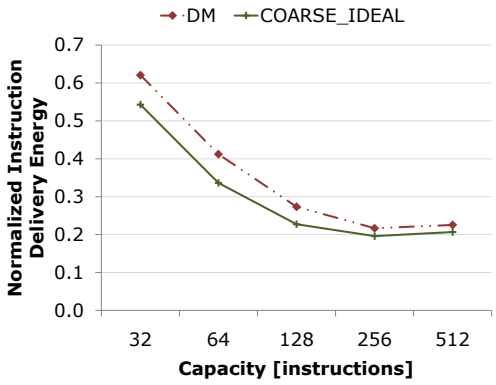


Figure 3: Instruction delivery energy of filter caches (FC) and SPMs with an ideal coarse-grain instruction placement (COARSE_IDEAL). The instruction delivery energy is normalized to the case when every instruction is fetched from the L1 instruction cache. Details on the evaluation setup are presented in Section 5.

the coarse-grain placement of instructions in SPMs — the smallest unit of transfer is a loop or function.

This paper presents a fine-grain dynamic instruction placement algorithm for SPMs that achieves an average of 38% instruction delivery energy savings over FCs. We evaluate 17 representative and non-trivial embedded applications from MiBench [15] and rigorously compare SPMs to FC configurations with the best energy efficiency. We also show that our fine-grain algorithm achieves 31% instruction delivery energy savings over even an ideal coarse-grain dynamic placement algorithm which achieves zero miss rate for instructions in loops or functions that fit the SPM. The fine granularity of our algorithm does not cause proliferation of copy instructions, therefore maintaining execution time and static code size similar to those of the coarse-grain algorithm.

The remainder of this paper is organized as follows. Section 2 reviews related work. Section 3 discusses the advantages and disadvantages of SPMs and FCs with respect to miss rates. Section 4 describes our fine-grain placement algorithm. Section 5 presents the results of our evaluation, and Section 6 concludes.

2. RELATED WORK

Here, we focus on differentiating our algorithm from other dynamic instruction placement algorithms for SPMs [38, 37, 44, 41, 23, 40, 10, 11, 32], instead of attempting to extensively review the whole body of SPM work.

Udayakumaran *et al.* [40], Egger *et al.* [10], and Pabalkar *et al.* [32] restrict the smallest unit of instruction transfer to a loop or function, which results in a large fraction of instruction reuse being missed. For example, Figure 2 shows that, while a fine-grain algorithm can place blocks 1, 2, 5, 6, and 7_1 in the SPM, a coarse-grain algorithm can only place block 6 in the SPM since the other blocks belong to loops that exceed the SPM size. Figure 3 shows that even an ideal SPM placement algorithm cannot achieve more than 10% energy reduction over FCs with coarse-grain instruction transfers. In Figure 3, we assume that the SPM placement algorithm achieves zero miss rate (including compulsory misses) for instructions in loops or functions that fit the SPM.

Steinke *et al.* [38], Verma *et al.* [44], and Egger et al. [10] use integer linear programming to select the best set of instructions to be placed in SPMs, which does not scale well for large applications.

Ravindran *et al.* [37] and Janapsatya *et al.* [23] use neither coarse-grain placement nor an exponential time algorithm. Ravindran *et al.* use traces and Janapsatya *et al.* use basic blocks as their unit of instruction transfer, both of which are less flexible than our algorithm where the length of transfer blocks can be adjusted in increments of one instruction. Ravindran *et al.* use *temporal relation* [12] to measure the cost of placing multiple traces in the same SPM location, and Janapsatya *et al.* use a similar metric called *concomitance*. Although using temporal relation or concomitance can minimize conflict misses of *caches*, we show in Section 3 that applying these metrics to *SPMs* overlooks the tagless and compiler-managed properties of SPMs. Using these metrics not only unnecessarily complicates profiling and compilation but can also misguide placement algorithms to make decisions that are beneficial only in the presence of tags.

We also point out that the literature has not rigorously compared SPMs against the best cache configurations. Several papers use 4-way [38, 41, 10, 32] or 2-way associative caches [37]. In Section 3 and 5.2, we show that higher associativity does not necessarily result in lower miss rates when the cache is small, which limits its instruction reuse mainly to loops. Egger *et al.* [11] confirm this by showing that direct-mapped caches outperform 4-way associative caches with respect to execution time and energy.

Verma *et al.* [41] compare energy consumption of SPMs and caches averaged over multiple capacities (128 to 1024 bytes). A more meaningful comparison would be between maximums rather than average energy reductions.

Janapsatya *et al.* [23] measure performance by accumulating access times of SPMs and caches (e.g., the execution time

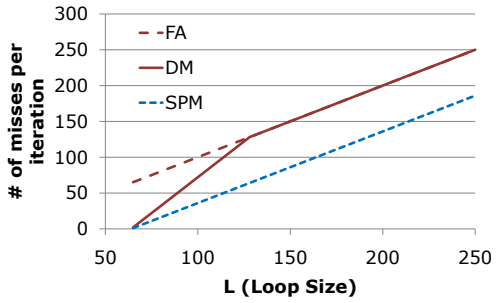


Figure 4: The number of misses per iteration for straight-line loops where C (the capacity of stores) is 64.

is 1μs if a cache with access time 1ns is accessed 1000 times). However, since the access times of all SPMs and caches they evaluated are faster than 1.7ns in a 0.18μm process, SPMs' faster access time will not convert into fewer cycle counts in most contemporary embedded processors.

A large body of SPM-related work [38, 37, 44, 41, 23, 40, 10, 11] focuses on *substituting* L1 instruction caches. We instead focus on *extending* the memory hierarchy by adding another level (L0) with SPMs and comparing this to FCs. This is because in L0 the majority of instruction reuse comes from loops, for which the compiler has a proven ability to optimize [1, 8, 26, 36].

As an alternative to software-managed SPMs, loop caches [28, 29] can be used as L0 instruction stores. Loop caches identify loops by observing backward jumps and store the identified loops to filter out costly access to larger stores. Loop caches serve well for applications in which straight-line loops dominate the performance. However, loop caches cannot store loops with if-else branches, and Gordon-Ross et al. [14] demonstrate that this is too inflexible in dealing with diverse embedded applications. Gordon-Ross et al. [14] address this inflexibility by pre-loading performance critical loops with arbitrary shapes. However, the pre-loaded loop caches [14] cannot overlay loops in different program phases, and thus cannot efficiently use the loop cache capacity as Ravindran et al. [37] show.

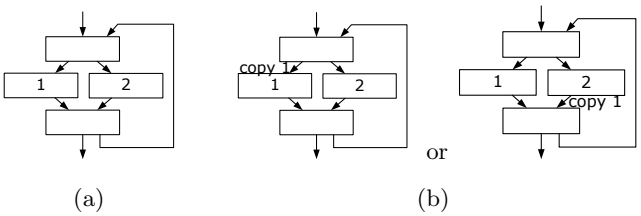
3. MISS RATES OF SPMS AND FILTER CACHES

This section discusses advantages and disadvantages of SPMs compared to FCs with respect to miss rates.

For a straight-line loop of length L, the following equations compute the number of misses (except compulsory misses) per iteration for a fully associative FC with least recently used replacement policy (y_{FA}), a direct-mapped FC (y_{DM}), and an SPM (y_{SPM}), all with capacity C. We assume that C instructions of the loop are placed in the SPM and the other L - C instructions are directly fetched from L1.

$$y_{\text{FA}} = \begin{cases} 0 & \text{if } L \leq C, \\ L & \text{if } L > C \end{cases} \quad (1)$$

$$y_{\text{DM}} = \begin{cases} 0 & \text{if } L \leq C, \\ 2(L-C) & \text{if } C < L \leq 2C, \\ L & \text{if } L > 2C \end{cases} \quad (2)$$



(a) (b)

Figure 5: (a) A simple loop and (b) the result of placing blocks 1 and 2 of the loop in the same SPM location.

$$y_{\text{SPM}} = \begin{cases} 0 & \text{if } L \leq C, \\ L - C & \text{if } L > C \end{cases} \quad (3)$$

Figure 4 plots these equations when C is 64 and shows that SPMs suffer from fewer conflict misses in straight-line loops. The fewer conflict misses are achieved by placing only C instructions in the SPM, while the other L - C instructions bypass it through the path from L1 to the processor shown in Figure 1(a).

While we can easily minimize miss rates of SPMs for straight-line loops, the same optimization is not trivial for codes whose control flow is less regular. In fact, it is quite a challenge for SPMs to achieve fewer misses than FCs for less regular code.

Consider a loop shown in Figure 5(a) that typically executes 1 during its first half of iterations and 2 for the second half. For FC, placing 1 and 2 in memory addresses mapped to the same FC location incurs few conflict misses. In other words, instructions 1 and 2 have a weak *temporal relation* [12]. Gloy *et al.* [12] minimize instruction cache conflict misses by finding a layout in which instructions mapped to the same cache location have weak temporal relations.

We cannot, however, apply the same optimization scheme to SPMs due to their lack of tags. In fact, we show that it is always disadvantageous to place multiple instructions in a single-level loop (a loop without inner loops) at the same SPM location. In this light, an optimal set of instructions to be placed in SPMs with capacity C is the C most frequently executed instructions for single-level loops (proof shown in Appendix A), which motivates our algorithm (described in Section 4).

In contrast to Ravindran *et al.* [37] and Janapsatya et al. [23], we show why temporal relation is not a relevant metric for SPMs as follows: Suppose that we place 1 and 2 in Figure 5(a) in the same SPM location. The compiler targeting the SPM must conservatively assume that 2 may be executed between consecutive executions of 1. Consequently, the compiler must copy 1 from L1 to the SPM either immediately before every execution of 1 or immediately after every execution of 2, as shown in Figure 5(b). Although many of these copies will unnecessarily transfer instructions that already reside in the SPM, this will not be noticed by the SPM due to the absence of tags.

4. ALGORITHM

This section describes how our algorithm finds a set of instructions to be placed in SPMs in fine granularity to minimize the number of L1 accesses. We first give an overview of our algorithm, then describe the details of each step.

We process each control flow graph by traversing the call

```
for each cfg of call graph in a post-order {
  T = construct a loop tree of cfg
  re-layout(cfg, T)
  for each loop L of T in a post-order {
    B_L = find the longest B_L
      subject to the constraints shown in Section 4.2
    insert copies and jumps for B_L
    remove redundant copies and jumps in inner loops
  }
}
```

Figure 6: Pseudo code of our algorithm

graph in a post order. The call graph handles function pointers by adding edges from a function pointer call site to all callees that may be referenced by the pointer. For each control flow graph, we construct a loop tree and re-layout the code. Then, we traverse the loop tree in a post order. For each loop visited, we select instructions to be placed in the SPM, and then insert copy and jump instructions. Figure 6 shows a pseudo-code of our algorithm.

4.1 Loop Tree Construction and Re-layout

We construct a loop tree as shown in Figure 7(a) by Havlak's algorithm [16] which can be used for both reducible and irreducible control flow graphs [17]. In the loop tree, the root represents the entire control flow graph, other non-leaf nodes correspond to loops, and leaf nodes are basic blocks.

We estimate the execution frequency of each loop tree node as follows: For each loop L, we estimate the average iteration count of L, n_L. We build a *sub-region graph* from the subgraph of the control flow graph induced by L by removing back-edges of L and contracting inner loops to single nodes, as shown in Figure 7(b). For each child x of L, we compute $p_L(x)$, the probability of executing x per execution of the sub-region graph of L. We estimate $p_L(x)$s either by profiling or static analysis. In our static analysis, we propagate $p_L(x)$s starting from the loop header, assuming that each branch direction is independently taken with 50% probability. The execution probabilities annotated in Figure 7(b) are estimated by this static analysis. In our static analysis, we set $n_L = \infty$. Section 5.3 shows that the energy consumption difference between profiling and the static analysis is less than 5%.

After estimating $p_L(x)$s, we re-layout the code so that the C most frequently executed children of a loop tree node are contiguous in memory and can be copied to the SPM as a single group, where C is the SPM capacity. In Figure 7, blocks are numbered according to the ordering after re-layout. For example, in Figure 7(c), blocks 1 and 2 are assigned smaller numbers than block 3 since blocks 1 and 2 are more frequently executed, assuming the execution probability shown in Figure 7(b). To keep the overhead of inserting jump instructions from the re-layout to a minimum, instead of sorting every child, we partition the children into a primary and a secondary partition so that the primary partition contains the C most frequently executed instructions. This partitioning is similar to Pettis and Hansen's *function splitting* [35] used for instruction cache miss optimization. Among the children with the same execution probability, we prioritize for inner loops.

4.2 Instruction Placement

For each loop L, we select a block of instructions to be placed in the SPM, denoted as B_L. We find the longest B_L subject to the following three constraints:

1. $|B_L| \leq C$, where C is the SPM capacity.
2. B_L is a contiguous block of instructions starting from the first instruction in L.
3. Let S_{Lx} be the set of inner loops and function calls of L that can overwrite the SPM location mapped to instruction x. $\forall$ instruction $x \in B_L$, $c_1 \cdot p_L(x) > c_2 \cdot (\frac{1}{n_L} + p_L(S_{Lx}))$, where c_1 is the access energy difference between the SPM and L1 cache, and c_2 is the energy for copying an instruction from the L1 cache to SPM; $p_L(S)$ is the probability of executing any instruction of S and $p_L(\emptyset) = 0$.

The first constraint is trivial[1]. The second constraint and our re-layout method ensure that instructions copied into the SPM at incoming edges of L are the $|B_L|$ most frequently executed ones. Note that this constraint is based on the claim proven in Appendix A — placing the C most frequently executed instructions of L in the SPM minimizes the number of L1 accesses for single-level loops. We assume that the SPM supports wrapping around the control from its last entry to the first entry, which is important for reducing fragmentation. The third constraint ensures that energy saved by fetching x from the SPM during iterations of L outweighs the cost of copying x at incoming edges to L and at outgoing edges from S_{Lx} (inner loops or function calls that conflict with x). Figure 7(e) shows an example of applying the third constraint: 3 is not placed in the SPM because its execution probability does not exceed the probability of executing function call in 4 or executing inner loop {5, 6, 7}, both of which use the entire SPM (i.e., $c_1 \cdot p_{\{1,2,3,4,5,6,7\}}(3) = c_1 \cdot 0.5 < c_2 \cdot p_{\{1,2,3,4,5,6,7\}}(\{4, \{5, 6, 7\}\}) = c_2 \cdot 0.5$, because $c_1 < c_2$). Since we visit the call graph and loop trees in a post order, we can determine S_{Lx}s, except for recursive functions. For recursive functions, we conservatively assume that the callee uses the entire SPM.

Note that we keep our algorithm simple by imposing the following two restrictions: First, the location from which an instruction is fetched is constant regardless of which call graph path or which control flow edge was taken. In other words, the placement of an instruction is neither context-sensitive nor flow-sensitive [1]. We can therefore denote instructions placed in SPMs as *SPM instruction*s and the others as *non-SPM instruction*s. Second, at incoming edges of loop L, we copy only one block of instructions, B_L, into the SPM. The only other places where instructions are copied are at outgoing edges from those inner loops or function calls that

[1]However, we need to consider the code size increase from inserting copy and jump instructions when we compute $|B_L|$. Since we traverse the call graph and loop trees in post orders, $|B_L|$ can decrease as we eliminate redundant copies and jumps when we visit an outer loop or callee as described in Section 4.3. In other words, we conservatively compute $|B_L|$ larger than its eventual value. To reduce the gap between the conservative $|B_L|$ and its eventual value, we apply simple rules such as the following: when the outer loop fits in the SPM, we do not count copy or jump insertions between the current loop and the outer loop since they will be eliminated when we visit the outer loop.

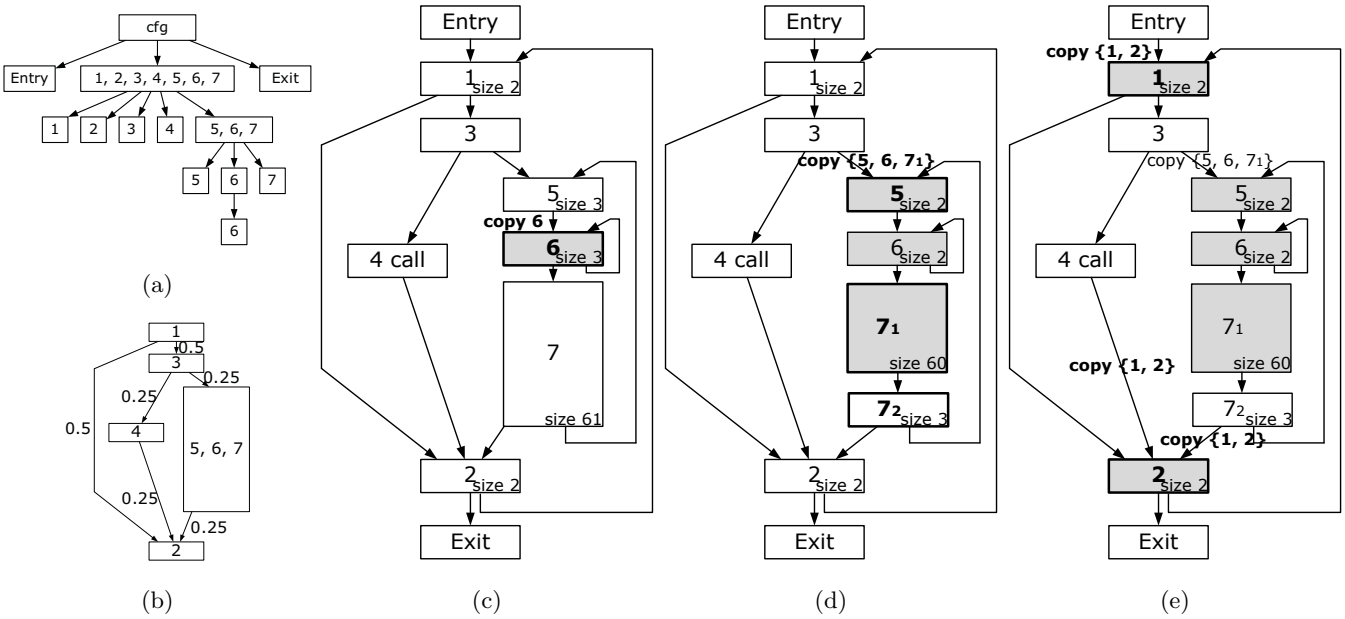


Figure 7: Example of placing instructions of loops in a 64-entry SPM. In (c)-(e), instructions placed in the SPM are shaded. Basic blocks are numbered in the ordering after re-layout; we generate the code following this order in our algorithm's assembly output. Block 4 has a function call whose callee uses the entire SPM.
(a) The loop tree of the control flow graph shown in (c)-(e).
(b) The sub-region graph of the outer-most loop, in which edges are annotated with the execution probability per iteration.
(c) A schedule after processing loop {6}. Since this loop fits within the SPM, the entire loop is placed in the SPM.
(d) A schedule after processing loop {5, 6, 7}. Blocks 5 and 6 are placed in the SPM, and the redundant copy of 6 at its incoming edge is eliminated. The first 59 instructions of block 7 are placed in the SPM as well and extracted as a separate block 7_1.
(e) A schedule after processing the outer-most loop. This loop has an inner loop bigger than the SPM and a function call whose callee uses the entire SPM. According to (b), the probability of executing the inner loop or the function call per iteration is 0.5. Since 1 and 2 are the only ones with execution probability higher than 0.5, we place 1 and 2 in the SPM.

can overwrite a portion of B_L. As a result, the fine granularity of our algorithm does not cause proliferation of copy instructions and therefore maintains execution time and static code size similar to those of a coarse-grain algorithm, as will be shown in Section 5.

4.3 Copy and Jump Insertion

After selecting *SPM instructions* (instructions to be placed in the SPM), we adjust the control flow by inserting copy and jump instructions and then eliminate redundant copies and jumps of inner loops.

We first insert copy instructions at incoming edges to the current loop and outgoing edges from conflicting inner loops and function calls. For example, in Figure 7(c), we insert "copy 6" at the incoming edge of {6}. Since jumping to the first instruction right after copying a block of instructions is a common case, in addition to `copy` instructions, we support `jcopy` instructions that transfer instructions from the L1 cache to the SPM and jump to the first transferred instruction. To avoid an unnecessary copy at a basic block with outgoing edges with different copy targets, we modify jumps as shown in Figure 8. In Figure 8, symbolic addresses that start with `@` are mapped to main memory, while the first arguments of `jcopy` instructions denote absolute addresses

```
@bb_i: ...
...
jump.lt @taken
@nontaken: ...
```

(a)

```
@bb_i: ...
...
jump.lt @bb_i_t
jcopy 17 @nontaken 15
@bb_i_t: jcopy 32 @taken 7
@nontaken: ...
```

(b)

Figure 8: Modifying a jump to avoid an unnecessary copy when outgoing edges of `bb_i` have different copy targets. (a) Before and (b) after the modification.

mapped to the SPM. The third arguments of `jcopy` instructions denote the number of instructions that are transferred by the `jcopy` instructions. When we return from a function to an SPM location, we use an indirect copy instruction whose source memory address and target SPM location are stored in registers.

If either the source or target of a fall-through control flow edge becomes an SPM instruction, we insert jump instructions at the edge. For example, in Figure 7(c), we insert an

Table 1: Experimental Setup

Baseline	No L0 instruction store, 4-way 16KB L1 I-cache with 8-instruction cache lines
LC	Loop cache with flexible loop size scheme [29]
FA	Fully associative cache with LRU replacement policy and 8-instruction (32-byte) cache lines
DM	Direct-mapped FC with 8-instruction cache lines
COARSE	SPM with an ideal coarse-grain placement
FINE_P	SPM with fine-grain placement
FINE	FINE_P without profiling
OPT	Fully-associative FC with optimal replacement policy [6]

instruction at the end of 5 that jumps to 6. We also insert an instruction that jumps from 6 to 7.

A copy for an inner loop can be redundant after copies for the current loop are inserted. For example, in Figure 7(d), "copy 6" at the incoming edge of loop {6} becomes redundant after "copy {5, 6, 7_1}" is inserted at the incoming edge of loop {5, 6, 7}.

Placing instructions of an outer loop in the SPM can render certain jump instructions in its inner loops unnecessary. For example, the jumps from 5 to 6 and from 6 to 7 that are added in Figure 7(c) become unnecessary in Figure 7(d) because the edges from 5 to 6 and from 6 to 7 are no longer the ones between an SPM instruction and a non-SPM instruction. Therefore we eliminate the jump instructions as shown in Figure 7(d).

5. EVALUATION

This section describes the experimental setup for our algorithm evaluation and analyzes the results.

5.1 Experimental Setup

For our evaluation, we use ELM [4], a multi-core architecture with 32-bit instructions, an in-order dual-issue pipeline with 4 stages, software-managed memories, and a mesh on-chip interconnection network. To make our evaluation less sensitive from ELM-specific features, we modify the architecture model to a single-core one with a single-issue pipeline and an L1 instruction cache, and change the compiler and the simulator accordingly. Our algorithm is implemented in `elmcc`, a compiler back-end for ELM that reads fully-optimized LLVM intermediate representation [27].

We use all integer and fixed-point applications of MiBench [15]. We also use `fft` in MiBench after converting its floating point operations to fixed-point ones. Since our processor does not support floating point operations, we exclude the other applications.

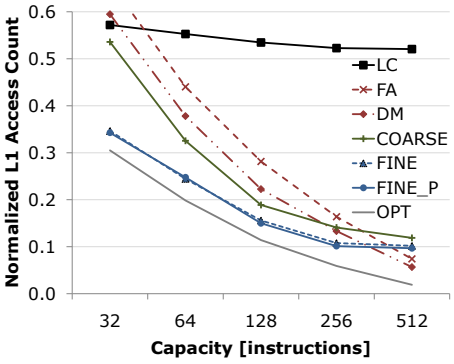
Table 1 summarizes the configurations used in the evaluation. We compare SPMs with 32 - 512 instructions to fully associative filter caches (FA), direct-mapped filter caches (DM), and loop caches (LC) [28, 29]. For FA and DM, we use 8-instruction (32-byte) cache lines, which achieve the best energy-delay product [13] (under the assumption that the instruction cache consumes 27% of the total energy as in the StrongARM processor [30]). To control for improvements due to code re-layout, we apply the same re-layout method to DMs when it is beneficial. In the COARSE configuration, we evaluate the maximum energy savings that can be achieved by a coarse-grain instruction placement: we assume that SPMs achieve zero miss rates for instructions in loops or functions that fit the SPM. We have two configurations for our fine-grain dynamic instruction placement algorithm: the FINE_P configuration uses profiling information, while FINE uses a static method for computing execution frequency as described in Section 4.1. To provide a lower bound for the number of L1 cache accesses, we include fully associative caches with an optimal replacement policy [6] (OPT). Note that this optimal replacement policy requires an oracle that predicts the future, thus cannot be implemented. In our evaluation, we are able to evaluate the performance of OPT as a theoretical bound by off-line trace-based simulations. We use a 16KB L1 instruction cache with 8-instruction cache lines and 4-way set associativity. The L1 instruction cache with no L0 instruction store is the baseline of our comparison.

Table 2: Energy per operation in pJ. "Refill" is the per cache line size energy for FCs and the per word energy for SPMs.

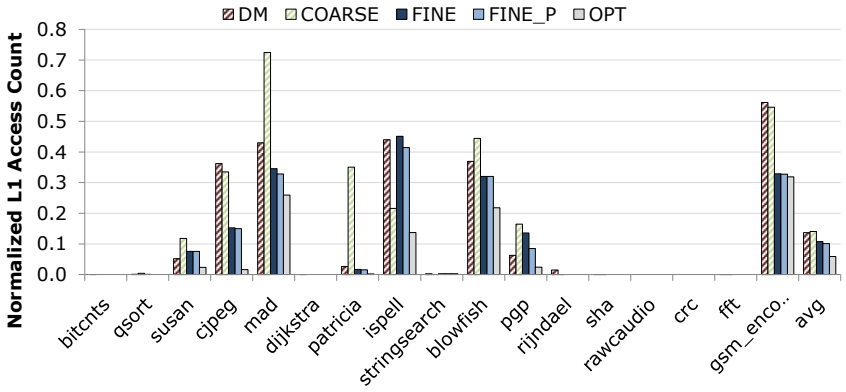
	Hit [pJ]	Miss [pJ]	Refill [pJ]
32-instruction FA	0.28	0.09	3.76
64-instruction FA	0.50	0.17	6.04
128-instruction FA	0.92	0.33	10.60
256-instruction FA	1.74	0.62	19.70
512-instruction FA	3.37	1.18	37.93
32-instruction DM	0.23	0.23	3.73
64-instruction DM	0.39	0.39	5.99
128-instruction DM	0.72	0.72	10.50
256-instruction DM	1.35	1.35	19.55
512-instruction DM	2.64	2.64	37.64
32-instruction SPM	0.11	—	0.33
64-instruction SPM	0.18	—	0.61
128-instruction SPM	0.33	—	1.16
256-instruction SPM	0.63	—	2.26
512-instruction SPM	1.22	—	4.47
16KB L1	20.35	2.68	37.01

We measure the performance of FA and DM using the Dinero IV trace-driven cache simulator [9]. We have implemented trace-driven simulators for LC and OPT.

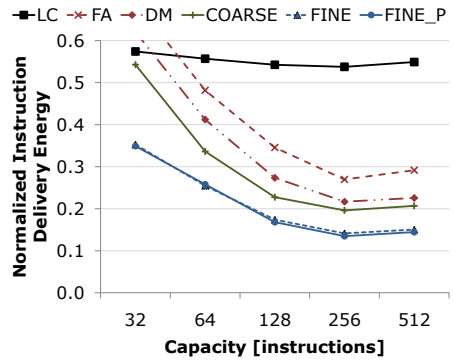
Table 2 lists the energy of each operation estimated from detailed circuit models of caches and memories realized in a commercial 45 nm low-leakage CMOS process. The models are validated against HSPICE simulations, with device and interconnect capacitances extracted after layout. Leakage current contributes a negligibly small component of the energy consumption due to the use of low-leakage devices. DMs use SRAMs to store tags and instructions; the tag array and data array are accessed in parallel, and the tag check is performed after both arrays are accessed. FAs use CAMs to store the tags and SRAMs to store the instructions. FAs are designed so that the SRAMs is only read when there is a hit in the tag CAM; consequently, a miss consumes less energy, as only the tag array is accessed. When transferring instruc-



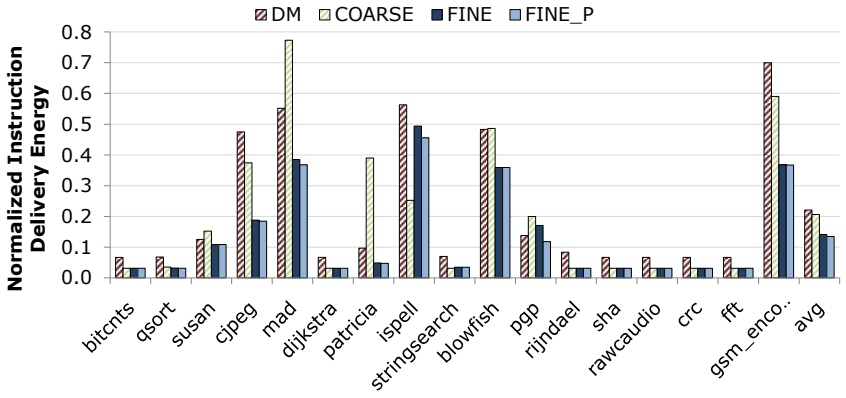
(a) The number of L1 accesses normalized to the baseline.



(b) The number of L1 accesses for 256-instruction configurations normalized to the baseline where no L1 access is filtered.



(c) Instruction delivery energy normalized to the baseline.



(d) Normalized instruction delivery energy for 256-instruction configurations.

Figure 9: L1 access and energy consumption results. The averages are obtained by computing arithmetic means over per-instruction-value of each benchmark, then normalizing each mean to the baseline processor configuration.

tions from the L1 cache, the L1 tag is checked once and the instructions are transferred over multiple cycles.

5.2 L1 Cache Access

Figure 9(a) compares the number of L1 cache accesses for each configuration. The number of L1 cache accesses is normalized to that of the baseline (no L0 store) and accounts for the additional copy instructions in SPM configurations. At smaller capacities, SPMs result in fewer L1 accesses than filter caches since there are many loops whose size is slightly larger than the capacities of SPMs, for which the advantage of SPMs is maximized as illustrated in Figure 4. When the capacity is as large as 512, FAs and DMs have fewer L1 accesses because the proportion of instruction reuse not analyzable at compile-time increases as the working set size increases. Despite of its higher associativity, FAs result in more L1 accesses than DMs because the LRU replacement policy works poorly for (almost) straight-line loops that are slightly larger than the cache capacity as described in Section 3. LC's L1 access decreases by only 9% as we increase its capacity from 32 to 512 since there are not many straight-line loops larger than 32, which is consistent with [14].

Figure 9(b) shows the number of L1 cache accesses for each benchmark for the 256-instruction configurations. Note that we execute whole programs, not just loops. To avoid clutter, we omit LC and FA, which do not show advantages over DM. Our fine-grain instruction placement algorithm (FINE_P and FINE) outperforms DMs on most applications. COARSE suffers from more L1 accesses than DMs on several applications, especially for `susan`, `mad`, `patricia`, and `pgp`. These applications have performance-critical loops bigger than SPMs, where frequently executed portion of the loops can only be captured by a fine-grain placement algorithm or by DMs (e.g., the outer-most loop in Figure 7).

5.3 Energy Consumption

Figure 9(c) presents instruction delivery energy for each configuration. Figure 9(d) shows the energy consumed in each benchmark for the 256-instruction configurations, where the maximum energy reduction is achieved. FINE_P achieves an 87% reduction, while FA, DM and COARSE achieve 73%, 78% and 80% reduction, respectively. Although COARSE achieves more energy reduction than DMs, their difference, 2%, is small considering the fact that we assume an ideal SPM

placement algorithm that achieves zero miss rate (including compulsory misses) in COARSE configuration. Our algorithm without profiling (FINE) does not consume more than 5% additional energy compared to FINE_P, which demonstrates that our fine-grain instruction placement algorithm achieves most of its benefit without profiling. Therefore, our algorithm can be used without profiling by default, allowing programmers to avoid complications from profiling and selecting a representative input data set.

To provide context, an 87% reduction in the energy consumed by the instruction storage hierarchy would result in a 23% reduction in the total dynamic energy consumed in processors such as the StrongARM [30], in which 27% of the total dynamic energy is consumed by the instruction cache.

While energy consumption is meaningful as the final metric, L1 access count is the most important variable that the compiler can directly optimize. The relative energy efficiency of an SPM placement algorithm compared to that of FCs varies as the memory hierarchy or circuit design changes. For example, if an SPM placement algorithm achieves smaller energy consumption than FCs despite of more L1 accesses, the same SPM placement algorithm may result in worse energy efficiency when we have an L1 cache with a larger capacity or higher associativity, where reducing L1 access is more important that reducing the unit L0 access energy. In addition, in [34], it is shown that CACTI [46] tends to overestimate energy consumption in small L0 stores since the cache architecture assumed by CACTI mainly targets caches that are bigger than or equal to typical L1 cache capacities. Conversely, if an SPM placement algorithm achieves smaller energy consumption *with* fewer L1 accesses, its relative energy efficiency compared to that of FCs is less dependent on a specific memory hierarchy or circuit design. Therefore, energy reduction of SPMs should not be reported without the number of L1 accesses in order to measure the energy efficiency of an SPM placement algorithm in a less architecture and circuit dependent manner.

5.4 Performance and Code Size

To quantify FC's and SPM's impact on execution time, we assume a penalty of 1 cycle for each FC miss as in [22, 25] and a load-use penalty of 1 cycle for SPMs. Gordon-Ross et al. [14] assume a penalty of 4 cycles for each FC miss, but the 1 cycle penalty can be achieved by critical word first technique [18]. For a copy whose target SPM location is stored in a register (e.g., a copy of function return target), we assume a penalty of 2 cycles. The processor allows one outstanding copy and stalls when a second one is attempted before the first completes. To focus on the aspect of instruction delivery, we disregard L1 data cache miss and branch miss prediction penalty. Within this setup, the 256-instruction FINE_P incurs an average of 1.0% execution time overhead, while 256-instruction DM incurs 1.7% overhead[2]. We optimize the cache line size of DMs for the best energy-delay product [13]. By increasing the cache line size, we capture more spatial locality and miss fewer instructions, resulting in a lower performance overhead. However, at the same time, this leads to the transfer of more unnecessary instructions from the L1 cache. We find that 8-instruction cache lines balance this trade-off and achieve the best energy-delay product. For example, by increasing the cache line size from 2 to 8, the performance overhead of the 256-instruction DM filter cache decreases from 5.8% to 1.7%, while the reduction of energy consumed by the instruction hierarchy changes minimally (from 78.4% to 78.3%).

Copy instructions increase the code size by, on average, 2.9% with 256-instruction FINE_P and 2.1% with the same capacity COARSE. This demonstrates that the fine granularity of our algorithm does not cause proliferation of copy instructions. Note that, although FINE_P increases the code size, it achieves smaller performance overhead than DM. This is because FINE_P pre-fetches instructions hiding the L1 access latency.

[2]This is an upper bound of FINE_P's performance overhead since its baseline is an ideal case without L1 cache and branch miss prediction penalty; e.g., if we assume an L1 cache miss penalty of 32 cycles, a 128-instruction bimodal branch predictor, and a branch miss penalty of 2 cycles, the 256-instruction FINE_P's performance overhead is reduced to 0.7%.

6. CONCLUSION

This paper presents a dynamic instruction placement algorithm for L0 SPMs that shows a notable instruction delivery energy savings (38%) over FCs. This is achieved by 1) fine-grain instruction placement where the length of transfer blocks can be adjusted in increments of one instruction and 2) careful consideration of the tagless and compiler-managed properties of SPMs. Since our fine-grain algorithm achieves 31% instruction delivery energy reduction over even an ideal coarse-grain algorithm, SPMs now have a better chance to become the preferred choice over FCs by providing energy saving that justifies the cost of compiler and instruction set modifications. In addition, processor designers will be able to make well-informed decisions on L0 instruction stores based on our rigorous comparison against the best FC configurations in 17 representative applications and detailed energy model.

7. ACKNOWLEDGMENTS

We thank Jooseong Kim for his help on our compiler backend implementation, and David Black-Schaffer and Clinton Buie for discussion on our algorithm in its initial stage. We thank Claude Reichard, Evelin Sullivan, and Hyejun Ra for their feedback on writing, and the anonymous reviewers who provided valuable comments. This work is supported in part by the Semiconductor Research Corporation under Grant 2007-HJ-1591 and in part by the National Science Foundation under Grant CNS-0719844. Jongsoo Park is supported in part by a Samsung Scholarship, and James Balfour is supported in part by the Cadence Design Systems Stanford Graduate Fellowship.

8. REFERENCES

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 2007.

[2] Federico Angiolini, Luca Benini, and Alberto Caprara. Polynomial-Time Algorithm for On-chip Scratchpad Memory Partitioning. In *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, pages 318–326, 2003.

[3] Federico Angiolini, Francesco Menichelli, Alberto Ferrero, Luca Benini, and Mauro Olivieri. A Post-Compiler Approach to Scratchpad Mapping of Code. In *International Conference on Compilers,*

Architecture, and Synthesis for Embedded Systems (CASES), pages 259–267, 2004.

[4] James Balfour, William J. Dally, David Black-Schaffer, Vishal Parikh, and Jongsoo Park. An Energy-Efficient Processor Architecture for Embedded Systems. *Computer Architecture Letters*, 7(1), 2008.

[5] Rajeshwari Banakar, Stefan Steinke, Bo-Sik Lee, M. Balakrishnan, and Peter Marwedel. Scratchpad Memory: A Design Alternative for Cache On-chip Memory in Embedded Systems. In *International Conference on Hardware Software Codesign*, pages 73–78, 2002.

[6] L. A. Belady. A Study of Replacement Algorithms for Virtual-storage Computer. *IBM Systems Journal*, 5(2):78–101, 1966.

[7] David Black-Schaffer, James Balfour, William J. Dally, Vishal Parikh, and Jongsoo Park. Hierarchical Instruction Register Organization. *Computer Architecture Letters*, 7(2):41–44, 2008.

[8] John Cocke and Ken Kennedy. An Algorithm for Reduction of Operator Strength. *Communications of the ACM*, 20(11):850–856, 1977.

[9] Jan Edler and Mark D. Hill. Dinero IV Trace-Driven Uniprocessor Cache Simulator. *Technical Report, University of Wisconsin, 1999. http://www.cs.wisc.edu/ markhill/DineroIV.*

[10] Bernhard Egger, Chihun Kim, Choonki Jang, Yoonsung Nam, Jaejin Lee, and Sang Lyul Min. A Dynamic Code Placement Technique for Scratchpad Memory Using Postpass Optimization. In *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, pages 223–233, 2006.

[11] Bernhard Egger, Jaejin Lee, and Heonshik Shin. Dynamic Scratchpad Memory Management for Code in Portable System with an MMU. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(2):1–38, 2008.

[12] Nikolas Gloy and Michael D. Smith. Procedure Placement using Temporal-ordering Information. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21(5):977–1027, 1999.

[13] Ricardo Gonzalez and Mark Horowitz. Energy Dissipation in General Purpose Microprocesors. *IEEE Journal of Solid-State Circuits*, 31(9):1277–1284, 1996.

[14] Ann Gordon-Ross, Susan Cotterell, and Frank Vahid. Tiny Instruction Caches for Low Power Embedded Systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 2(4):449–481, 2003.

[15] Matthew R. Guthaus, Jeffrey S. Ringenberg, Dan Ernst, Todd M. Austin, Trevor Mudge, and Richard B. Brown. MiBench: A Free, Commercially Representative Embedded Benchmark Suite. In *IEEE 4th Annual Workshop on Workload Characterization*, pages 83–94, 2001.

[16] Paul Havlak. Nesting of Reducible and Irreducible Loops. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 19(4):557–567, 1997.

[17] M. S. Hecht and Jeffrey D. Ullman. Characterizations of Reducible Flow Graphs. *Journal of the ACM (JACM)*, 21(3):367–375, 1974.

[18] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. 2003.

[19] Stephen Hines, Joshua Green, Gary Tyson, and David Whalley. Improving Program Efficiency by Packing Instructions into Registers. In *International Symposium on Computer Architecture (ISCA)*, pages 260–271, 2005.

[20] Stephen Hines, Gary Tyson, and David Whalley. Reducing Instruction Fetch Cost by Packing Instructions into Register Windows. In *International Symposium on Microarchitecture (MICRO)*, pages 19–29, 2005.

[21] Stephen Hines, Gary Tyson, and David Whalley. Addressing Instruction Fetch Bottlenecks by Using an Instruction Register File. In *Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 165–174, 2007.

[22] Stephen Hines, David Whalley, and Gary Tyson. Guaranteeing Hits to Improve the Efficiency of a Small Instruction Cache. In *International Symposium on Microarchitecture (MICRO)*, pages 433–444, 2007.

[23] Andhi Janapsatya, Aleksandar Ignjatović, and Sri Parameswaran. A Novel Instruction Scratchpad Memory Optimization Method based on Concomitance Metric. In *Asia and South Pacific Design Automation Conference*, pages 612–617, 2006.

[24] Murali Jayapala, Francisco Barat, Tom Vander Aa, Francky Catthoor, Henk Corporaal, and Geert Deconinck. Clustered Loop Buffer Organization for Low Energy VLIW Embedded Processors. *IEEE Transactions on Computers*, 54(6):672–683, 2005.

[25] Johnson Kin, Munish Gupta, and William H. Mangione-Smith. The Filter Cache: An Energy Efficient Memory Structure. In *International Symposium on Microarchitecture (MICRO)*, pages 184–193, 1997.

[26] Monica Lam. Software Pipelining: An Effective Scheduling Technique on VLIW Machines. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 318–328, 1988.

[27] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Aanalysis & Transformation. In *International Symposiumon Code Generation and Optimization (CGO)*, pages 75–86, 2004.

[28] Lea Hwang Lee, Bill Moyer, and John Arends. Instruction Fetch Energy Reduction Using Loop Caches for Embedded Applications with Small Tight Loops. In *International Symposium on Low Power Electronics and Design (ISLPED)*, pages 267–269, 1999.

[29] Lea Hwang Lee, Bill Moyer, and John Arends. Low-Cost Embedded Program Loop Caching - Revisited. *Technical Report CSE-TR-411-99, University of Michigan*, 1999.

[30] James Montanaro, Richard T. Witek, Krishna Anne, Andrew J. Black, Elizabeth M. Cooper, Daniel W. Dobberpuhl, Paul M. Donahue, Jim Eno, Gregory W. Hoeppner, David Kruckemyer, Thomas H. Lee, Peter C. M. Lin, Liam Madden, Daniel Murray, Mark H. Pearce, Sribalan Santhanam, Kathryn J. Snyder, Ray Stephany, and Stephen C. Thierauf. A 160-MHz, 32-b, 0.5-W CMOS RISC Microprocessor. *IEEE Journal of Solid-State Circuits*, 31(11):1703–1714, 1996.

[31] Nghi Nguyen, Angel Dominguez, and Rajeev Barua. Memory Allocation for Embedded Systems with a Compile-Time-Unknown Scratch-Pad Size. *ACM Transactions on Embedded Computing Systems (TECS)*, 8(3):1–32, 2009.

[32] Amit Pabalkar, Aviral Shrivastava, Arun Kannan, and Jongeun Lee. SDRM: Simultaneous Determination of Regions and Function-to-Region Mapping for Scratchpad Memories. *High Performance Computing*, pages 569–582, 2008.

[33] Preeti Ranjan Panda, Nikil D. Dutt, and Alexandru Nicolau. Efficient Utilization of Scratch-Pad Memory in Embedded Processor Applications. In *European Design and Test Conference*, pages 7–11, 1997.

[34] Jongsoo Park, James Balfour, and William J. Dally. Maximizing the Filter Rate of L0 Compiler-Managed Instruction Stores by Pinning. *Technical Report 126, Concurrent VLSI Architecture Group, Stanford University*, 2009.

[35] Karl Pettis and Robert C. Hansen. Profile Guided Code Positioning. In *ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, pages 16–27, 1990.

[36] Bob Ramakrishna Rau. Iterative Modulo Scheduling: An Algorithm for Software Pipelining Loops. In *International Symposium on Microarchitecture (MICRO)*, pages 63–74, 1994.

[37] Rajiv A. Ravindran, Pracheeti D. Nagarkar, Ganesh S. Dasika, Eric D. Marsman, Robert M. Senger, Scott A. Mahlke, and Richard B. Brown. Compiler Managed Dynamic Instruction Placement in a Low-Power Code Cache. In *International Symposium on Code Generation and Optimization (CGO)*, pages 179–190, 2005.

[38] Stefan Steinke, Nils Grunwald, Lars Wehmeyer, Rajeshwari Banakar, M. Balakrishnan, and Peter Marwedel. Reducing Energy Consumption by Dynamic Copying of Instructions onto Onchip Memory. In *International Symposium on Systems Synthesis*, pages 213–218, 2002.

[39] Stefan Steinke, Lars Wehmeyer, Bo-Sik Lee, and Peter Marwedel. Assigning Program and Data Objects to Scratchpad for Energy Reduction. In *Conference on Design, Automation and Test in Europe (DATE)*, pages 409–415, 2002.

[40] Sumesh Udayakumaran, Angel Dominguez, and Rajeev Barua. Dynamic Allocation for Scratch-Pad Memory Using Compile-Time Decisions. *ACM Transactions on Embedded Computing Systems (TECS)*, 5(2):472–511, 2006.

[41] Manish Verma and Pter Marwedel. Overlay Techniques for Scratchpad Memories in Low Power Embedded Processors. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 14(8):802–815, 2006.

[42] Manish Verma, Klaus Petzold, Lars Wehmeyer, Heiko Falk, and Peter Marwedel. Scratchpad Sharing Strategies for Multiprocess Embedded Systems: A First Approach. In *Workshop on Embedded Systems for Real-Time Multimedia*, pages 115–120, 2005.

[43] Manish Verma, Lars Wehmeyer, and Peter Marwedel. Cache-Aware Scratchpad Allocation Algorithm. In *Conference on Design, Automation and Test in Europe (DATE)*, 2004.

[44] Manish Verma, Lars Wehmeyer, and Peter Marwedel. Dynamic Overlay of Scratchpad Memory for Energy Minimization. In *International Conference on Hardware/software Codesign and System Synthesis*, pages 104–109, 2004.

[45] Lars Wehmeyer, Urs Helmig, and Peter Marwedel. Compiler-optimized Usage of Partitioned Memories. In *Workshop on Memory Performance Issues*, pages 114–120, 2004.

[46] Steven J.E. Wilton and Norman P. Jouppi. CACTI: An Enhanced Cache Access and Cycle Time Model. *IEEE Journal of Solid-State Circuits*, 31(5):677–688, 1996.

APPENDIX

A. APPENDIX

Let G be the subgraph of the control flow graph induced by a single-level loop (a loop without any inner loops), L. We follow Havlak's definition of loop header and back-edges [16]. The header of L is the first visited node of L when we depth-first search the control flow graph to construct a loop tree. The back-edges of L are the edges whose source is in L and whose target is the header of L. When L is a natural loop [17], the header is uniquely defined regardless of the particular depth-first search order used, and it has exactly one back-edge. Let L's header be G's entry. When L is a natural loop, let the source of L's unique back-edge be G's exit. When L is not a natural loop, we add an exit node in G and connect sources of L's back-edges to the exit node.

A set S dominates a node x, denoted by S *dom* x, if every path in G from the entry to x must go through at least one element in S. S post-dominates a set T, denoted by S *pdom* T, if every path in G from an element in T to the exit must go through at least one element in S. Let A_x be the set of program locations in L where a "`copy` x" resides. Let X_i be the set of instructions in L that are placed at the ith SPM location.

LEMMA A.1. *For a correct copy schedule, $\forall x \in X_i$,*
$(A_x$ dom $x) \vee$
$((A_x$ pdom $X_i - \{x\}) \wedge$
(x resides in the ith SPM location at incoming edges of L whose target is L's header)).

PROOF. Lemma A.1 We prove the contrapositive of Lemma A.1. Assume $(A_x \neg dom\ x) \wedge (A_x \neg pdom\ X_i - \{x\})$. By the definition of *dom* and *pdom*, this assumption implies that the control can follow a path from an element of $X_i - \{x\}$ to x through the loop header without executing any "`copy` x". In this case, the ith SPM location does not hold x when the processor tries to fetch it from the SPM to execute. Assume (A_x $\neg dom\ x) \wedge$ (x does not reside in the ith SPM location at incoming edges of L whose target is L's header). This implies that the control can flow from the outside of L to x through the loop header without executing any "`copy` x". □

Let $p(x)$ be the execution frequency of x and $p(S) = \sum_{x \in S} p(x)$. Let the baseline be a schedule such that $\forall x \in L$, $p(A_x) = p(x)$; e.g., copy x right before executing it. If x satisfies the first clause of Lemma A.1 (i.e. A_x *dom* x), then $p(A_x) \geq p(x)$. Therefore, the only way of reducing $p(A_x)$ from the baseline is through the second clause; but at most one instruction in X_i can satisfy the second clause since only one can reside in the ith SPM location at L's incoming edges. Hence, the implication of Lemma A.1 is that, among the instructions placed in the same SPM location, at most one can have fewer L1 accesses than the baseline. Based on this, we can easily prove the following claim.

CLAIM A.1. *Let C be the capacity of the SPM and L be a single-level loop. Placing the C most frequently executed instructions of L in the SPM achieves the minimum number of L1 accesses*[3].

[3]It is minimum under the assumption that loop fission and code duplication are not allowed. However, loop fission can be implemented as a separate compilation phase, while code duplication incurs an exponential code size increase in the worst case.