Improved Procedure Placement for Set Associative Caches

Yun Liang
Advanced Digital Sciences Center
Illinois at Singapore
eric.liang@adsc.com.sg

Tulika Mitra
School of Computing
National University of Singapore
tulika@comp.nus.edu.sg

ABSTRACT

The performance of most embedded systems is critically dependent on the memory hierarchy performance. In particular, higher cache hit rate can provide significant performance boost to an embedded application. Procedure placement is a popular technique that aims to improve instruction cache hit rate by reducing conflicts in the cache through compile/link time reordering of procedures. However, existing procedure placement techniques make reordering decisions based on imprecise conflict information. This imprecision leads to limited and sometimes negative performance gain, specially for set-associative caches. In this paper, we introduce intermediate blocks profile (IBP) to accurately but compactly model cost-benefit of procedure placement for both direct mapped and set associative caches. We propose an efficient algorithm that exploits IBP to place procedures in memory such that cache conflicts are minimized. Experimental results demonstrate that our approach provides substantial improvement in cache performance over existing procedure placement techniques. Furthermore, we observe that the code layout for a specific cache configuration is not portable across different cache configurations. To solve this problem, we propose an algorithm that exploits IBP to place procedures in memory such that the average cache miss rate across a set of cache configurations is minimized.

Categories and Subject Descriptors

B.3.2 [**Memory Structures**]: Design Styles—*Cache memories*; D.3.4 [**Programming Languages**]: Processors—*Compilers*

General Terms

Algorithms, Performance, Design

Keywords

Memory, Instruction Cache, Procedure Placement, Code Layout, Cache Miss, Intermediate Blocks Profile

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CASES'10, October 24–29, 2010, Scottsdale, Arizona, USA.
Copyright 2010 ACM 978-1-60558-903-9/10/10 ...$10.00.

1. INTRODUCTION

Cache memories are employed by most of modern processors to hide the high latency to memory. Instruction cache plays a critical role in terms of both performance and energy consumption as instructions are fetched almost every clock cycle. Thus, careful tuning and optimization of instruction cache memory can lead to significant performance gain. Profile based procedure placement is proposed as one of the well-known instruction cache optimization techniques, which aims to reorder the procedures in the compile/link time such that cache conflict misses are eliminated during run-time. State of the art of procedure placement techniques [8, 9] generate a specific code layout for a particular cache configuration. All these techniques require the cache design parameters such as cache line size and total cache size as inputs. This is because the solutions are created by reasoning about where the procedures should be placed in the cache, which inevitably requires the knowledge of line size and cache size. Being aware of the underlying cache parameters, these techniques are shown to be better than the earlier works that neglect them [12, 18].

However, state of the art procedure placement techniques suffer from some drawbacks. First, the cost and benefit of placing a procedure are modelled approximately. The conflict metric (the approximation of cache misses) is defined at granularity of procedures. However, inside a procedure, there might be more than one program path (i.e., a sequence of instructions). The conflict metric of different paths in the same procedure may not be the same. Thus, using the existing techniques, it is possible that the new code layout generated is worse than the original code layout due to this imprecise conflict information. Second, the techniques are mainly designed for direct mapped caches and do not model set associative caches accurately. Due to the above two reasons, the existing procedure placement techniques are not very effective for set associative caches. To solve these two problems, we introduce intermediate blocks profile (IBP) to precisely model the cost and benefit of procedure placement. IBP is significantly more compact compared to memory traces. So, the cache performance evaluation using IBP is much more efficient than cache simulation. Based on the precise cache model using IBP, our procedure placement algorithm starts from the original procedure ordering and selects the most beneficial procedures along with their placements iteratively.

Moreover, we observe that the code layout generated for a specific cache configuration by utilizing its parameters (cache size, associativity) may not be portable across platforms with varying cache configurations. This problem exists for all procedure placement techniques that rely on cache parameters [8, 9, 13, 10, 3]. Such portability issue is very important in situations where the underlying hardware platform (cache configuration) is unknown. This is common for embedded systems where the code becomes available during deployment through either download or portal media [16]. In such situations, compiler/linker may not know the underlying cache configurations and thus is unable to generate a code layout appropriate for the particular cache configuration. More importantly, the cache configurations across platforms could be different due to different versions of the processor or technology evolution. Thus, the same executable (code layout) may have to run on systems with different cache configurations.

For the portability problem, we concern ourselves with the scenario where an application can be run on platforms with same instruction set architecture but different cache configurations. To overcome the portability problems across platforms with varying cache configurations, we propose a procedure placement algorithm to generate a "neutral" code layout by using IBP and structural relations among different cache configurations. The neutral code layout performs well for a set of cache configurations on an average.

In summary, in this paper, we introduce the intermediate blocks profile (IBP) that allows us to model the cost and benefit of procedure placement accurately and compactly for both direct mapped and set associative caches. Based on IBP, we first propose a procedure placement algorithm that places procedures in memory such that cache conflicts are minimized for a specific cache configuration. Experiments demonstrate that our code layout achieves more cache miss reduction than the state of the art. To attack the portability problem, we propose another algorithm that generates a neutral code layout for a set of different cache configurations and the neutral code layout is shown to perform well across a set of cache configurations on an average.

2. RELATED WORK

For embedded systems design, it is important to minimize cache miss rate to improve performance as well as power consumption. Various methods targeting improvement of instruction cache performance through software solutions have been proposed such as efficient cache design space exploration [14, 7, 19, 11], instruction cache locking [1, 15] and code reorganization. In this paper, we focus on code reorganization through procedure placement.

Procedure placement techniques to improve instruction cache performance have been around for more than a decade. Earlier procedure placement techniques rely on procedure call graph to model the conflicts among procedures, where the vertices are the procedures and the edges are weighted by the number of calls between two procedures [12, 18]. These approaches put conflicting procedures next to each other to reduce cache conflicts. However, the underlying cache parameters are not considered.

By taking cache parameters (line size, cache size) into account, a better procedure placement is proposed in [10]. The algorithm maintains the set of unavailable cache locations (colors) for each procedure. The colors are used to guide procedure placement. Later on, the technique is extended to model indirect procedure calls [13]. Gloy et al. in [8] build temporal relationship graph (i.e., which procedures are referenced between two consecutive accesses to another procedure). Based on temporal relationship graph and also considering the cache parameters, they show better cache performance improvement than [12, 18]. For all the above techniques [10, 13, 8], their conflict metric is just an approximation of conflict misses and they are designed for direct mapped cache. Bartolini and Prete propose a precise procedure placement technique [3] using detailed trace driven simulation to evaluate the effect of procedure placements. In their work, the number of simulations required increases linearly with the number of procedure and cache size. However, detailed simulation could be extremely slow [21], even if the trace is slightly compressed. Thus, simulation based approach is not feasible for not so small applications, long trace, or large cache size. On the contrary, our technique is based on the compact intermediate block profile which models the cache accurately and efficiently.

Existing procedure placement techniques allow gaps among procedures to improve cache performance. This leads to code size expansion. Although various simple heuristics have been proposed to reduce the code size in [8, 13, 3], the code size still could expand significantly as shown in [9]. Thus, the cache performance is improved at the cost of code size expansion. Such huge code size expansion make these techniques unusable in the context of embedded systems. Guillon et al. extend the technique in [8] to deal with code size. They introduce a parameter to guide the tradeoff between performance and code size. They also develop a polynomial optimal algorithm to minimize code size. It is shown that good performance is still achieved but with a small code size expansion [9]. However, the technique developed in [9] is mainly for direct mapped cache and the cache miss is modelled using imprecise conflict information.

In this work, we introduce intermediate blocks profile to model cache behavior. Previously, reuse distance has been proposed for the same purpose [5, 6]. Reuse distance is defined as the number of distinct data accesses between two consecutive references to the same address and it accurately models cache behavior of a fully associative cache. However, to precisely model the effect of procedure placement, we need the content rather than the number (size) of distinct data accesses between two consecutive references. IBP records both the reuse content and their frequencies.

Code reordering can be done at granularity of basic block level too [17, 20] with additional instruction (i.e., jump instructions) insertion.

3. PROCEDURE PLACEMENT PROBLEM

In this section, we first introduce the cache terminology and then formally define procedure placement problem.

Cache Terminology. A cache memory is defined in terms of four major parameters: *block or line size* L, *number of sets* N, *associativity* A, and *replacement policy*. The block or line size determines the unit of transfer between the main memory and the cache. A cache is divided into N sets. Each cache set, in turn, is divided into A cache blocks, where A is the associativity of the cache. N, L and A are power of 2. For a direct-mapped cache $A = 1$, for a set-associative cache $A > 1$, and for a fully associative cache $N = 1$. In other words, a direct-mapped cache has only one cache block per set, whereas a fully-associative cache has only one cache set. Now the cache size is defined as $(N \times A \times L)$. For a set-associative or fully-associative cache, the replacement policy (e.g., LRU, FIFO, etc.) defines the block to be evicted when a cache set is full. In this work, we consider Least Recently Used (LRU) replacement policy where the block replaced is the one that has been unused for the longest time.

We use memory block to refer to block (line) sized block in the memory. Given a memory address m, its corresponding memory block and the cache set where the memory block is mapped to are

$$memory\_block(m) = \lfloor m/L \rfloor \tag{1}$$

$$cache\_set(m) = \lfloor m/L \rfloor \ modulo \ N \tag{2}$$

Thus, given a memory address or memory block, it is mapped to only one cache set. Figure 1 shows an example of memory address mapping. In the example, line size is considered to be two bytes

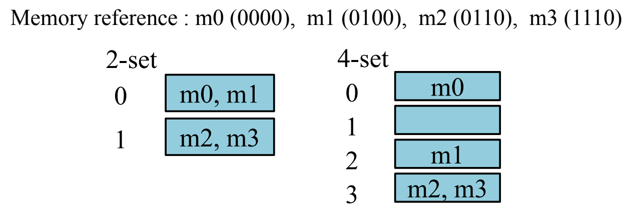

Memory reference : m0 (0000), m1 (0100), m2 (0110), m3 (1110)
2-set
0
1
m0, m1
m2, m3
4-set
0
1
2
3
m0
m1
m2, m3

Figure 1: Memory address mapping. The address is byte address and line size is assumed to be 2 bytes (last bit).

(last bit). Memory address $m1$ is mapped to set 2 in a 4-set cache, but mapped to set 0 in a 2-set cache.

Given a procedure p, its starting memory line can be defined as $K \times N + s$, where $0 \leq s < N$ and $K \geq 0$. For procedure p, s is its starting cache set number when mapped to cache and s affects the cache conflicts between p and other procedures; K determines its location in memory and K does not affect the cache conflicts but memory size. Thus, procedure placement technique involves two phases: cache placement and memory placement. Cache placement phase determines s for each procedure to minimize conflicts; memory placement determines K for each procedure and aims to minimize the code size. In this paper, we propose a new procedure placement algorithm using IBP for cache placement phase. As for memory placement, Guillon et al. [9] provides an optimal solution for memory placement problem and we employ their technique to minimize the code size.

4. INTERMEDIATE BLOCKS PROFILE

Let $\mathcal{P}$ be the set of procedures of the program. Given a procedure $p \in \mathcal{P}$, we use p_{start} to denote its starting address in the original code layout, p_{set} to denote its starting cache set number ($0 \leq p_{set} < N$) and p_{size} to denote its size in bytes. For a procedure $p \in \mathcal{P}$, p_{set} may be changed by procedure placement to improve the cache performance. However, procedure placement reorders instructions at the granularity of procedures. Thus, the instructions inside a procedure are still contiguous even if the procedure's location is changed. Thus, given an instruction, its relative offset to the staring address of the procedure is never changed during procedure placement.

DEFINITION 1 **(Procedure Block)**. *Given a memory address m, its procedure block is defined as a tuple $\langle p, l \rangle$ where m belongs to procedure p and $l = \lfloor \frac{m - p_{start}}{L} \rfloor$.*

Now, let $\mathcal{T}$ be the memory trace (sequence of memory references) generated by executing a program on the target architecture. This trace is generated using the original code layout. We transform the memory trace $\mathcal{T}$ to its corresponding procedure block trace $\mathcal{T}'$ by representing each memory reference in $\mathcal{T}$ using its corresponding procedure block. The trace $\mathcal{T}'$ remains the same during procedure placement while trace $\mathcal{T}$ is not. Furthermore, for caches with different size and associativity, but with same line size, their procedure block traces are the same.

Let $\mathcal{B}$ be the set of procedure blocks of the program. Given a procedure block $b \in \mathcal{B}$, let us define the j^{th} reference of b in the trace $\mathcal{T}'$ as $b[j]$.

DEFINITION 2 **(Conflict)**. *Given two procedure blocks b and b', let b be $\langle p, l \rangle$ and b' be $\langle p', l' \rangle$*

$$conflict(b, b') = \begin{cases} 1 & \text{if } \exists k \in Z \text{ s.t.} \\ & l + p_{set} - l' - p'_{set} = k \times N \\ 0 & \text{otherwise} \end{cases}$$

In other words, $conflict(b, b')$ returns 1 if b and b' are mapped to the same cache set; otherwise, it returns 0.

DEFINITION 3 **(Procedure Block Interval)**. *A procedure block interval is a tuple $\langle p, s, e \rangle$. It represents a sequence of contiguous procedure blocks which belong to the same procedure $\{\langle p, l \rangle : s \leq l \leq e\}$.*

Given two procedure block intervals which belong to the same procedure $\langle p, s_1, e_1 \rangle$ and $\langle p, s_2, e_2 \rangle$, they can be merged to a bigger procedure block interval $\langle p, s_1, e_2 \rangle$ if $s_2 = e_1 + 1$ or $\langle p, s_2, e_1 \rangle$ if $s_1 = e_2 + 1$.

DEFINITION 4 **(Intermediate Blocks Set (IBS))**. *Given a procedure block reference $b[j] (j > 1)$ in the trace where $b \in \mathcal{B}$, let $S_{between}$ be the set of unique procedure blocks referenced between $b[j-1]$ and $b[j]$ in $\mathcal{T}'$. If there is no such reference, then $S_{between} = \phi$. Let procedure block b be $\langle p, l \rangle$ and S_{other} be $\{\langle p', l' \rangle : \langle p', l' \rangle \in S_{between} \wedge p' \neq p\}$. Then, the intermediate blocks set (IBS) of procedure block reference $b[j]$, $IBS_{b[j]}$ is defined as a tuple $\langle S, C \rangle$, where*

- $S =$ *the smallest set of procedure block intervals representing S_{other}*
- $C = \sum_{b' \in S_{between} \setminus S_{other}} conflict(b', b)$

More clearly, $IBS_{b[j]}$ has two parts. The first part is the set of unique procedure blocks from other procedures referenced between $b[j-1]$ and $b[j] (S_{other})$ in procedure block interval format. The second part is the number of conflicts encountered from the procedure blocks in between which are from procedure p itself. Given two procedure blocks belonging to the same procedure, their conflict (Definition 2) is not affected by procedure placement because their relative offset are not changed by procedure placement.

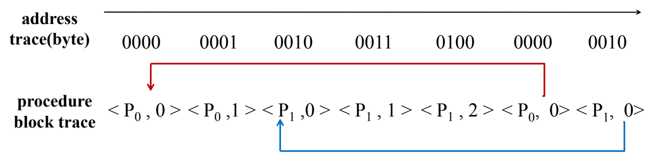
For different references of the same procedure block, they may have the same intermediate blocks set. More importantly, for intermediate blocks set $\langle S, C \rangle$ which does not interact with other procedures (i.e., $S = \phi$), it is not affected by procedure placement. Let

$$\mathcal{IBS}_b = \{\langle S, C \rangle : \exists j > 1 \text{ s.t. } \langle S, C \rangle = IBS_{b[j]} \wedge S \neq \phi\}$$

DEFINITION 5 **(Intermediate Blocks Profile)**. *The intermediate blocks profile IBP_b of a procedure block b is defined as a set of 2-tuples $\{\langle s, f(s) \rangle\}$ where $s \in \mathcal{IBS}_b$ and $f(s)$ denotes the frequency of the intermediate blocks set s of procedure block b in the trace.*

In Figure 2, we show an example of procedure block trace and its corresponding intermediate blocks profile. More concretely, for the second reference of procedure block $\langle P_0, 0 \rangle$, its intermediate blocks set is $\langle \{P_1, 0, 2\}, 0 \rangle$ because 3 procedure blocks $\langle P_1, 0 \rangle$, $\langle P_1, 1 \rangle$ and $\langle P_1, 2 \rangle$ from P_1 are accessed in between and there is no procedure blocks from P_0 which conflict with $\langle P_0, 0 \rangle$ are accessed in between (i.e., $\langle P_0, 1 \rangle$ does not conflict with $\langle P_0, 0 \rangle$). For the second reference of procedure block $\langle P_1, 0 \rangle$, its intermediate blocks set is $\langle \{P_0, 0, 0\}, 1 \rangle$ because one procedure block $\langle P_0, 0 \rangle$ from P_0 is accessed in between and there is one procedure block from $P_1 (\langle P_1, 2 \rangle)$ which conflicts with $\langle P_0, 0 \rangle$ is accessed in between. $\langle P_1, 0 \rangle$ conflicts with $\langle P_1, 2 \rangle$ because line gap between them is 2 and the number of cache sets is 2 in the example.

Now, given a procedure block b and one of its intermediate blocks set, its cache behavior under least recently used replacement policy can be determined as follows:



Procedure Attributes

Procedures	Start Address (byte)	Size (byte)
P_0	0000	2
P_1	0010	3

Procedure block	Intermediate Blocks Profile
$< P_0, 0 >$	$\{ < s, f(s) = 1 > \}$, where $s = < \{ < P_1, 0, 2 > \}, 0 >$
$< P_1, 0 >$	$\{ < s, f(s) = 1 > \}$, where $s = < \{ < P_0, 0, 0 > \}, 1 >$

Figure 2: Procedure block trace and intermediate blocks profile. Block (line) size is assumed to be 1 byte. The number of cache sets is assumed to be 2.

DEFINITION 6 (**Cache Hit**). *Given a procedure block b and intermediate blocks set $\langle S, C\rangle \in \mathcal{IBS}_b$.*

$$hit(b, \langle S, C\rangle) = \begin{cases} 1 & \text{if} \left(\sum_{\langle p,s,e\rangle \in S} \sum_{s \le l \le e} conflict(b, \langle p, l\rangle)\right) + C < A \\ 0 & otherwise \end{cases}$$

5. PROCEDURE PLACEMENT ALGORITHM

In this section, we present our procedure placement algorithm for a specific cache configuration by utilizing IBP. However, not all the procedures are invoked during execution or frequently called. We only consider the hot procedures for placement.

Hot Procedures. For a procedure p, we define its hot attribute according to its interaction with other procedures, not its execution time. For procedure p, its hot attribute is defined as

$$p_{hot} = \sum_{\forall b \in p, <s, f(s)> \in IBP_b} f(s)$$

We sort procedures in decreasing order according to their hot attribute p_{hot}. Let $total = \sum_{p \in \mathcal{P}} p_{hot}$. We use $HotP$ to denote the set of hot procedures we will consider for procedure placement. We keep adding the next hottest procedure among the rest of procedures to $HotP$ until $\sum_{p \in HotP} p_{hot} \ge total \times Thres$, where $0 < Thres \le 1$. Initially, $HotP = \emptyset$.

Note that the hot procedures we define may be different from traditional hot procedures (i.e., procedures which consume significant portion of a program's total execution time). This is because time consuming procedures may rarely switch control flow to other procedures or be frequently called by other procedures. On the other hand, procedure placement is a technique that aims to reduce conflict misses due to procedure switching. Thus, our procedure placement technique uses the interaction of a procedure with other procedures rather than execution time as the hot attribute.

Our procedure placement technique is presented in Algorithm 1. We start from the original procedure order (i.e., the procedure sequence in the original code layout). We place procedures one by one and place them at a cache line boundary (line 3-6). We use a two dimension array $hit[b][s]$ to record the hits of procedure block b for intermediate blocks set $s \in IBP_b$. Array $hit[][]$ is initialized based on the original procedure order (line 7-9). Then, in each iteration of the loop, we walk through all the hot procedures which have not been selected for placement so far and try out all the displacement values $dis \in \{0, \ldots, N-1\}$ for them. For each iteration, we select the procedure and its corresponding displacement value that results in maximum benefit. If there is no benefit, the iterative process is terminated.

Algorithm 1: Procedure Placement Algorithm

```
1  $set = 0$;
2  Let $List$ be the list of procedures in the original order;
3  for $i \leftarrow 1$ to $|List|$ do
4      $p = L[i]$; $p_{set} = set$;
5      $set = set + \lceil p_{size}/L \rceil$;
6  foreach $b \in \mathcal{B}$ do
7      foreach $\langle s, f(s)\rangle \in IBP_b$ do
8          $hit[b][s] = hit(b, s) \times f(s)$;
9
10 $flag = TRUE$;$Placed = \emptyset$;
11 while $flag$ do
12     $benefit = 0$;
13     foreach $p \in HotP \wedge p \notin Placed$ do
14         $old\_set = p_{set}$;
15         for $dis \leftarrow 0$ to $N - 1$ do
16             $p_{set} = dis$;
17             $new\_benefit = getBenefit(p)$;
18             if $new\_benefit > benefit$ then
19                 $benefit = new\_benefit$;
20                 $selected\_set = dis$;
21                 $p' = p$;
22
23         $p_{set} = old\_set$;
24
25     if $benefit > 0$ then
26         $p'_{set} = selected\_set$;
27         $Placed = Placed \cup p'$;
28         foreach $b \in \mathcal{B}$ do
29             foreach $\langle s, f(s)\rangle \in IBP_b$ do
30                 if $p' \in IPS_b^s$ then
31                     $hit[b][s] = hit(b, s) \times f(s)$;
32
33
34
35     else
36         $flag = FALSE$;
37
38 function($getBenefit(p)$)
39 $benefit = 0$;
40 foreach $b \in \mathcal{B}$ do
41     foreach $\langle s, f(s)\rangle \in IBP_b$ do
42         if $p \in IPS_b^s$ then
43             $benefit = benfit + hit(b, s) \times f(s) - hit[b][s]$;
44
45
46 return $benefit$;
```

Let us assume procedure p is selected for placement. Function *getBenifit(p)* returns the benefit of this placement compared to the code layout of previous iteration.

DEFINITION 7 (**Influential Procedure Set (IPS)**). *Given a procedure block b and one of its intermediate blocks set $s \in \mathcal{IBS}_b$, let b be $\langle p, l\rangle$ and s be $\langle S, C\rangle$. The influential procedure set IPS_b^s is $\{p\} \cup \{p_1 : \langle p_1, s_1, e_1\rangle \in S\}$.*

The influential procedure set (IPS) for the procedure block b and intermediate blocks set $s \in \mathcal{IBS}_b$, IPS_b^s, is just the set of procedures invoked between the two occurrences of b.

PROPERTY 1. *Given a procedure block b and intermediate blocks set $s \in \mathcal{IBS}_b$, $hit(b, s)$ is not affected by the placement of procedure p if $p \notin IPS_b^s$.*

Property 1 can be easily observed following Definition 2 and 6. In function *getBenifit(p)*, given a procedure block b and one of its intermediate blocks set s, we will consider them only if they are affected by the placement of procedure p (i.e., $p \in IPS_b^s$). If $p \notin IPS_b^s$, then $hit(b, s)$ is not affected. In other words, there is no performance gain or loss. In each iteration, once a procedure and its corresponding cache set number are selected, the affected entries in $hit[][]$ will be updated (line 30-34).

6. NEUTRAL PROCEDURE PLACEMENT

In the last section, we describe an algorithm to generate a new code layout for a specific configuration using IBP. We observe that the code layout generated for a specific cache configuration is so tied to the specific configuration that it is not portable across different cache configurations (see section 7.3). More importantly, the code layout portability is a problem for all the techniques that are aware of cache parameters [10, 8, 13, 9, 3]. In this section, we will present an algorithm to generate a "neutral" code layout for a set of cache configurations. The neutral code layout achieves better average performance (i.e., average number of cache misses across a set of cache configurations) than any specific code layout.

We are interested in the set of cache configurations with different cache size and associativity, but with constant line size. Given cache size (S), line size (L) and associativity (A), the number of cache sets is $\frac{S}{L \times A}$. We use C_A^S to denote the cache configuration with size S and associativity A. Then, the set of cache configurations we support is $Config = \{C_A^S | S_{min} \leq S \leq S_{max}; A_{min} \leq A \leq A_{max};\}$, where $A_{min}(A_{max})$ is the minimum (maximum) associativity and $S_{min}(S_{max})$ is the minimum (maximum) cache size. We use $N_{max}(N_{min})$ to represent the maximum (minimum) number of cache sets for the configurations in $Config$. Obviously, $N_{max} = \frac{S_{max}}{L \times A_{min}}$ and $N_{min} = \frac{S_{min}}{L \times A_{max}}$.

For a procedure block reference $b[j]$, let b be $\langle p, l \rangle$. Its intermediate blocks set $IBS_{b[j]}$ is defined as $\langle S, C \rangle$ in section 4, where C is the number of procedure blocks from p itself which conflict with b between the references $b[j-1]$ and $b[j]$. However, for the configurations in the set $Config$, they may have different number of cache sets. According to Definition 2, the conflict for two procedure blocks may be different for caches with different number of cache sets. Thus, we extend the definition of intermediate blocks set $IBS_{b[j]}$ to $\langle S, C[] \rangle$, where S is the same as before, but $C[]$ is an array and $C[i]$ returns the conflicts for the cache with i cache sets.

Our aim is to improve the average performance for the set of configurations $Config$. Our algorithm is similar to Algorithm 1 with a few changes. First, for a procedure p, we use p_{set} to represent its starting cache set number when mapped to cache $C_{A_{min}}^{S_{max}}$ (i.e., the cache with maximum number of cache sets, N_{max}). Thus, $0 \leq p_{set} < N_{max}$. When mapped to cache C_A^S, starting cache set number of procedure p is (p_{set} $modulo$ N'), where N' is the number of cache sets for cache $C_A^S(N' = \frac{S}{L \times A})$. In other words, for a procedure p, its starting cache set in $C_{A_{min}}^{S_{max}}$ uniquely determines its starting cache set when mapped to other cache configurations in $Config$ with smaller number of cache sets. When a hot procedure is selected for placement, we try all the displacements $dis \in \{0, \ldots, N_{max} - 1\}$ for it. Second, given a procedure block b and intermediate blocks set $s \in IBP_b$, we use $hit[b][s]$ to represent the total number cache hits of all the configurations in $Config$ rather than a specific configuration. Finally, $getBenefit(p)$ function in Algorithm 1 only returns the benefit of a specific cache configuration when p is selected for placement. In the following, we will define $getBenefit\_Set(p)$ function which returns the benefit for all the configurations in $Config$.

Algorithm 2: Benefits for a set of configurations

```
1  function($getBenefit\_Set(p)$)
2  Let conf[] be an array ;
3  $benefit = 0$ ;
4  foreach $b \in \mathcal{B}$ do
5      foreach $\langle s, f(s) \rangle \in IBP_b$ do
6          if $p \in IPS_b^s$ then
7              Initialize conf[] to 0;
8              Let $s$ be $\langle S', C'[] \rangle$ and $b$ be $\langle p', l' \rangle$ ;
9              foreach intermediate blocks set $\langle p'', s'', e'' \rangle \in S'$ do
10                 for $l'' \leftarrow s''$ to $e''$ do
11                     $set = Match(p''_{set} + l'', p'_{set} + l')$ ;
12                     if $set > N_{max}$ then
13                         $set = N_{max}$ ;
14                     $conf[set] = conf[set] + 1$;
15
16             for $set \leftarrow N_{max}/2$ to $N_{min}$ do
17                 $conf[set] = conf[set] + conf[set * 2]$;
18             for $set \leftarrow N_{max}$ to $N_{min}$ do
19                 $conf[set] = conf[set] + C'[set]$;
20             $total\_hit = 0$;
21             foreach $C_A^S \in Config$ do
22                 $set = \frac{S}{A \times L}$;
23                 if $conf[set] < A$ then
24                     $total\_hit = total\_hit + f(s)$;
25
26             $benefit = benefit + total\_hit - hit[b][s]$;
27
28
29 return $benefit$ ;
```

For a procedure block b, let b be $\langle p, l \rangle$. It will be mapped to cache set (($p_{set} + l$) $modulo$ N') in cache with N' cache sets. According to Definition 2, we have

PROPERTY 2. *Two procedure blocks that conflict in $C_1 \in Config$ with N_1 cache sets, will conflict in $C_2 \in Config$ with N_2 cache sets, if $N_1 > N_2$.*

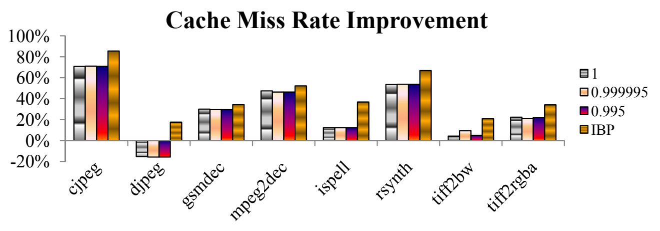
The new $getBenefit\_Set(p)$ function is detailed in Algorithm 2. Let us assume the selected procedure for placement is p. Then, for a procedure block b and its intermediate blocks set s pair, we compute its benefit only if it is affected ($p \in IPS_b^s$). If it is affected, then we compare procedure block b with every procedure block in intermediate blocks set s. We determine the maximum number of cache sets at which two procedure blocks conflict by using $Match$ function. $Match(a, b) = 2^k$, where k is the contiguous right matched bits of a and b.

According Property 2, the conflicts are propagated from cache with more cache sets to cache with less cache sets (line 16-17). The number of cache sets is always power of 2. The conflicts from the procedure blocks in the same procedure are added too (line 18-19). In the end, we walk through all the cache configurations and collect the benefits.

7. EXPERIMENTAL RESULTS

7.1 Experiments Setup

We select benchmarks from MiBench and MediaBench for evaluation purposes as shown in Table 1. We conduct our experiments using SimpleScalar framework [2]. Each benchmark is first run with a training input to generate an execution trace. Then, each



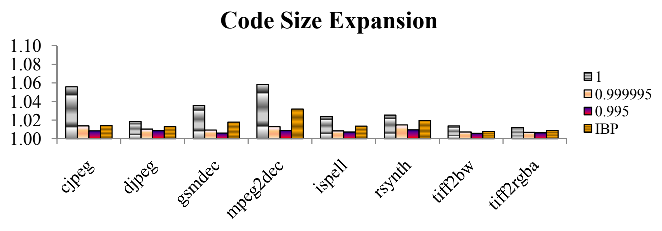
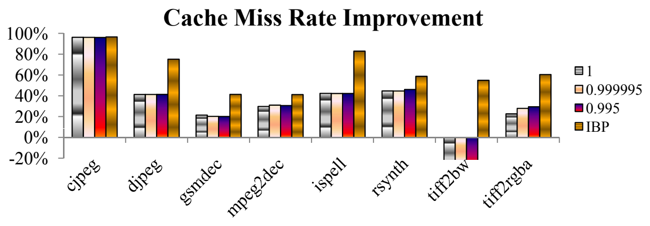


Figure 3: **Cache miss rate improvement and code size expansion compared to original code layout for 4K direct mapped cache.**



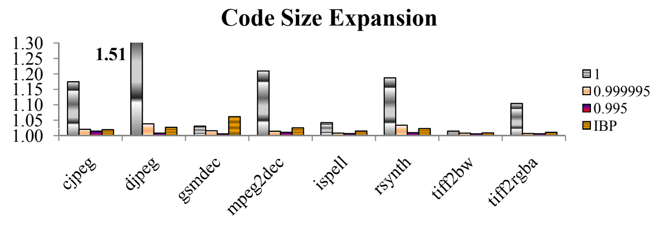


Figure 4: **Cache miss rate improvement and code size expansion compared to original code layout for 8K direct mapped cache.**

Benchmark	# Procedures	# Hot procedures	Trace (MB)	IBP (MB)
Cjpeg	324	27	235	5.1
Djpeg	351	29	63	5.2
Gsmdec	188	16	101	2.6
Mpeg2dec	216	23	222	27
Ispell	261	12	242	6.8
Rsynth	188	21	495	6.9
Tiff2bw	498	61	365	4.5
Tiff2rgba	493	68	359	3.2

Table 1: **Characteristics of Benchmarks.**

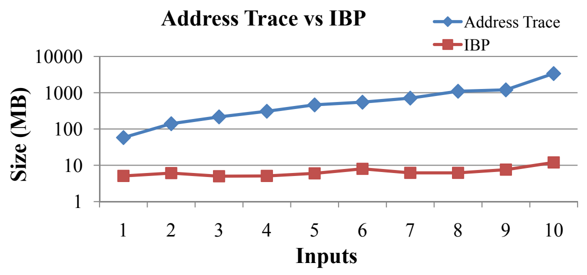


Figure 5: **Address trace vs IBP for various inputs with different sizes for *Cjpeg*. Inputs are sorted in ascending order according to their sizes.**

benchmark is recompiled with our analysis activated. We generate the instruction trace of each benchmark using *sim-profile*, a functional simulator. Given the address trace, our analysis transforms it to the procedure block trace and builds corresponding IBP. As discussed in section 5, we only select the hot procedures for procedure placement and in our experiment, we set the threshold for hot procedures as 0.99. Benchmarks characteristics such as number of hot procedures, size of address trace, and IBP are shown in Table 1.

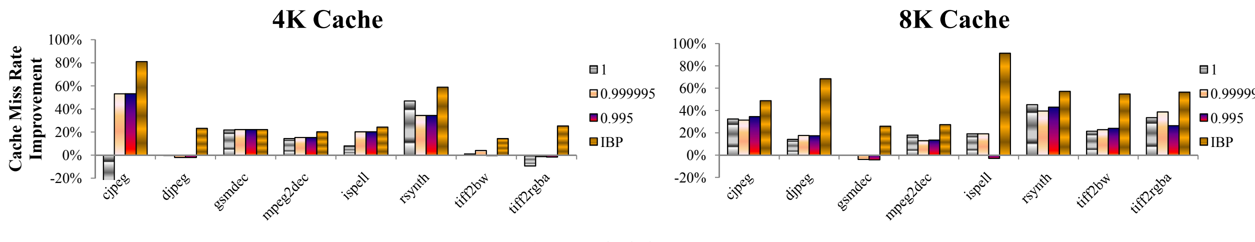
We evaluate the effectiveness of our algorithms with different cache parameters. We vary the cache size (4KB, 8KB), associativity (1, 2, 4, 8), and block size (32 bytes). We collect the cache misses and execution cycles using cache simulator and cycle accurate simulator in SimpleScalar. The performance numbers are collected by running the optimized program with a different input. In other words, we use different inputs for training and evaluation run. We perform all the experiments on a 3GHz Pentium 4 CPU with 2GB memory. In this paper, we focus on the cache placement (i.e., assign starting set number to procedures to minimize cache conflict misses). As for memory placement, Guillon et al. [16] present a polynomial optimal algorithm and we implement their technique to minimize code size.

IBP vs Trace. Both the address trace and IBP size are shown in Table 1. As shown, IBP is significantly more compact compared to the address trace. More importantly, when the input size increases, its address trace will increase while IBP size most likely will remain the same. This is because when the input size increases, the intermediate blocks set will most likely stay stable and only the frequency is increased.

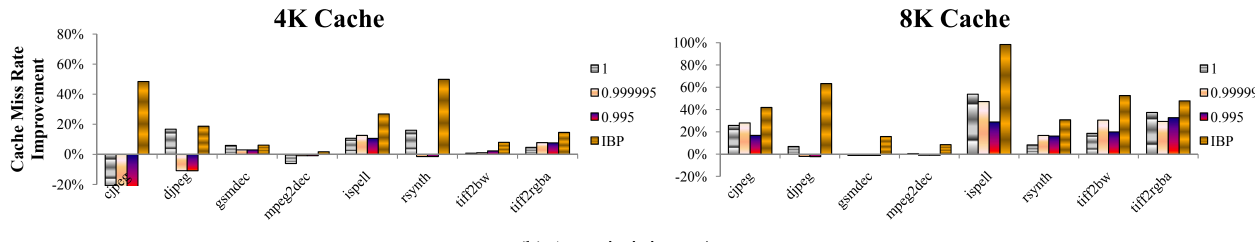
In Figure 5, we show how the address trace and IBP size change for various inputs for *Cjpeg* benchmark. We tried 10 different inputs with different sizes. The raw image sizes vary from $34K$ to $2.9M$. In Figure 5, the inputs are sorted according to their size. As shown, the trace size increases significantly when the input size increases. However, the IBP size stays stable when the input size increases. Different inputs may cover different paths of the program. Thus, a small input may have more intermediate blocks set than a big input. As a result, the size of the intermediate blocks profile for a small input may be larger than that of the big input. Hence, it is possible that the IBP size decreases while the input size increases as shown in Figure 5.

7.2 Layout for a Specific Cache Configuration

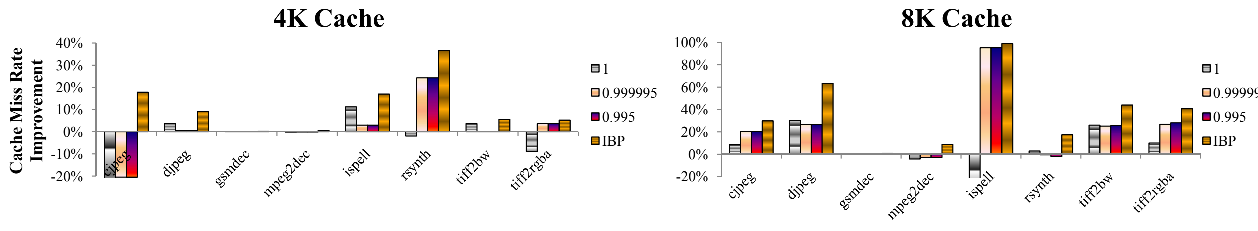
To evaluate our technique, we compare our work with the state of art [9] which extends [8] to deal with code size. In [9], parameter $\alpha(0 \leq \alpha \leq 1)$ is used to control the tradeoff between miss rate reduction and code size expansion. When $\alpha = 1$, the algorithm is biased toward miss rate reduction only, which is exactly the same as technique [8]. With $\alpha = 1$, cache miss rate is reduced but with significant code size expansion. Two other values of α, 0.995 and 0.999995 are used in [9] which gives the miss rate reduction nearly the same as $\alpha = 1$ but with smaller code size expansion. Though the techniques in [8, 9] are mainly for direct mapped cache, as shown in [8], they can be applied to set associative caches as well.



(a) Associativity = 2



(b) Associativity = 4



(c) Associativity = 8

Figure 6: **Cache miss rate improvement compared to original code layout for set associative cache.**

We use the original code layout as baseline. Original code layout is the code layout from the compiler without any procedure reordering. For each technique, we show its cache miss rate improvement and code size expansion compared to original code layout. We discuss the comparison between our technique (IBP) and Guillon method [9] for direct mapped and set associative caches, respectively. For Guillon method [9], we tried various values of α (1, 0.999995, 0,995).

Direct Mapped Cache. The results are shown in Figure 3 and 4 for $4K$ and $8K$ cache, respectively. Both techniques are effective for direct mapped cache as shown. As for Guillon method, our finding is that when α is set to $0.999995(0.995)$, it achieves similar miss rate improvement as $\alpha = 1$, but the code size expansion is much smaller than $\alpha = 1$ as shown in [9]. However, Guillon method may generate worse code layout compared to original code layout due to its imprecise conflict model (e.g., *djpeg* for $4K$ cache and *tiff2bw* for $8K$ cache).

For every benchmark and configuration pair, our IBP achieves more cache miss rate improvement than Guillon method. For $4K$ cache, IBP improves cache miss rate by 43.5% on an average; Guillon method improves performance by 30%—30.5% on an average depending on the value of α. For $8K$ cache, IBP improves cache miss rate by 64% on an average; Guillon method improves performance by 37.3%—38.3% on an average depending on the value of α. For Guillon method, if the miss rate improvement is negative, we consider it as 0 when computing average values.

There are two reasons for the gain of IBP over Guillon method. First, IBP is a precise model working at the granularity of procedure block, but Guillon method is based on imprecise conflict information. In addition to cache modeling, two techniques differ in the nature of the procedure placement algorithm. In IBP, we start from original procedure order and align the procedure at cache line boundaries. In each round, we try all the hot procedures not placed so far to determine the best procedure for placement and its corresponding placement. However, in Guillon method, the sequence to place procedures is pre-determined by conflict graph and only the different placement of procedures are attempted.

IBP based method achieves more miss rate reduction with a very small code size expansion. For $4K$ cache, IBP expands code size by 1.6% on an average; Guillon method expands code size by 0.8%—3% on an average depending on the values of α. For $8K$ cache, IBP expands code size by 2.4% on an average; Guillion method expands code size by 0.9%—16% on an average depending on the values of α.

Set Associative Caches. As shown in Figure 6(a), (b) and (c), Guillon method is not always effective for set associativity caches especially when associativity is high (4, 8). For some benchmarks, the code layout from Guillon method is much worse than the original code layout. For 2-way associative cache, 12 out of 48 code layouts are worse than the original code layout; for 4-way associative cache, 17 out of 48 code layouts are worse than the original code layout; for 8-way associative cache, 16 out of 48 code layouts are worse than the original code layout. On the other hand, IBP is always better than original code layout. This is because IBP allows us to precisely model the cache performance for set associative caches while Guillon method does not model set associative caches accurately.

For 2-way associative cache, IBP improves cache miss rate by 43.7% on an average; Guillon method improves cache miss rate by 17.3%—20.7% on an average depending on the value of α. For 4-

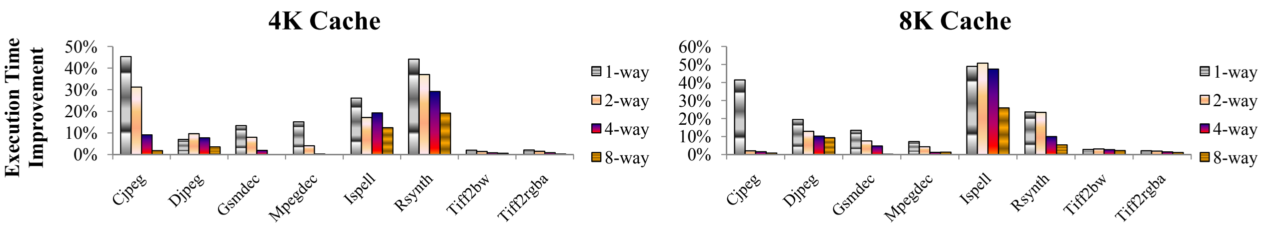


Figure 7: **Execution time improvement compared to original code layout.**

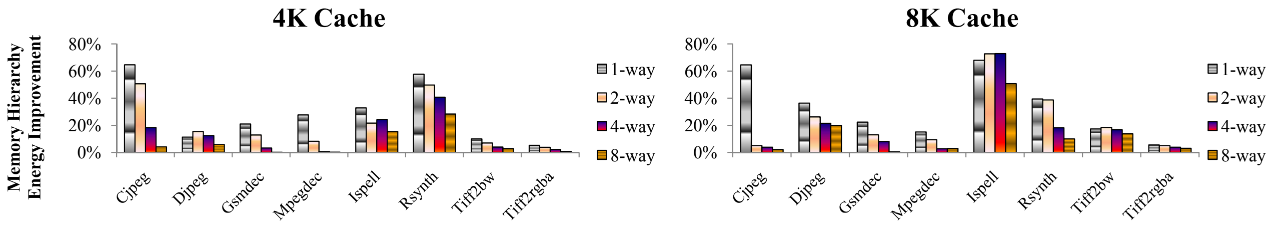


Figure 8: **Energy consumption improvement compared to original code layout.**

way associative cache, IBP improves cache miss rate by 33.4% on an average; Guillon method improves cache miss rate by 8.6%—12.9% on an average depending on the value of α. For 8-way associative cache, IBP improves cache miss rate by 24.8% on an average; Guillon method improves cache miss rate by 6.1%—14.2% on an average depending on the value of α. For Guillon method, if the miss rate improvement is negative, we consider it as 0 when computing average values.

Finally, for both techniques, the cache miss rate improvement decreases when the associativity increases. It is because higher associativity leads to fewer cache sets leaving little opportunity for procedure reordering.

IBP achieves high miss rate improvement with only small code expansion. As for code size, for 2-way associative cache, IBP expands code size by 1.3% on an average; for 4-way associative cache, IBP expands code size by 1.1% on an average; for 8-way associative cache, IBP expands code size by 0.9% on an average.

Execution Time Improvement. Figure 7 shows the execution time improvement of new code layout (IBP) compared to the original code layout. These experiments are conducted assuming single-issue in-order processor with 100 cycles cache miss latency and 1 cycle cache hit latency. Some benchmarks do not gain considerable execution time improvement even though cache miss rate is improved. This is because for these benchmarks the absolute cache miss number without procedure placement is very small. Thus, improvement in cache miss rate will not contribute much to the overall execution time reduction. IBP obtains 19.67% execution time improvement on an average for 1-way cache, 13.53% execution time improvement on an average for 2-way cache, 9.32% execution time improvement on an average for 4-way cache and 5.3% execution time improvement on an average for 8-way cache.

Energy Consumption Improvement. Figure 8 shows the memory hierarchy energy improvement of new code layout (IBP) compared to the original code layout. For different cache configurations, we model the energy consumption of the memory hierarchy using the CACTI [22] model for $0.13\mu m$ technology. In this paper, our focus is dynamic energy consumption. As for the energy consumption for one access to memory, it is assumed to be 200 times of energy consumption of one access to standard level one cache [23]. IBP obtains 31.2% energy consumption improvement on an average for 1-way cache, 22.4% energy consumption improvement on an average for 2-way cache, 15.8% energy consumption improvement on an average for 4-way cache and 10% energy consumption improvment on an average for 8-way cache.

Impact of Replacement Policy. IBP based cache modeling is developed under the assumption that the replacement policy is least recently used (LRU). However, Berg and Hagerston observed that different replacement policies may have little effect on the miss ratio for most of applications, but small differences exist [4]. We evaluate IBP layout under *FIFO* replacement policy. The code layout is generated based on LRU replacement policy, but miss rate is computed using *FIFO* replacement policy.

In Figure 9, we show the cache miss improvement compared to original code layout for *FIFO*. We observe that for most applications, IBP code layout is still quite effective. IBP obtains 43.3% miss rate improvement on an average for 2-way cache, 32.2% miss rate improvement on an average for 4-way cache, and 23.7% miss rate improvement on an average for 8-way cache. However, we notice that small differences do exist. For 4K, 8-way set associative cache, IBP code layout is a little bit worse than the original code layout for *Mpeg2dec* for *FIFO* replacement policy. This is probably because IBP code layout is not quite effective for this configuration for *Mpeg2dec* as shown in Figure 6.

Runtime. Our procedure placement algorithm is very efficient thanks to the compact format of IBP. It only takes a few minutes to complete our analysis for any considered settings.

7.3 Neutral Layout

We evaluate our neutral code layout using the set of configurations $Config = \{C_A^S | 4K \leq S \leq 8K; 1 \leq A \leq 8;\}$. For each configuration C_A^S, we generate a specific S-A-way code layout. So there are total 8 specific code layouts and 1 neutral code layout.

We first present the code layout portability problem in Table 2. For each code layout, we evaluate its portability using all the 8 configurations in $Config$. Table 2 shows the results of 2 benchmarks *Ispell* and *Rsynth*. The other benchmarks are not shown here due to space limitation.

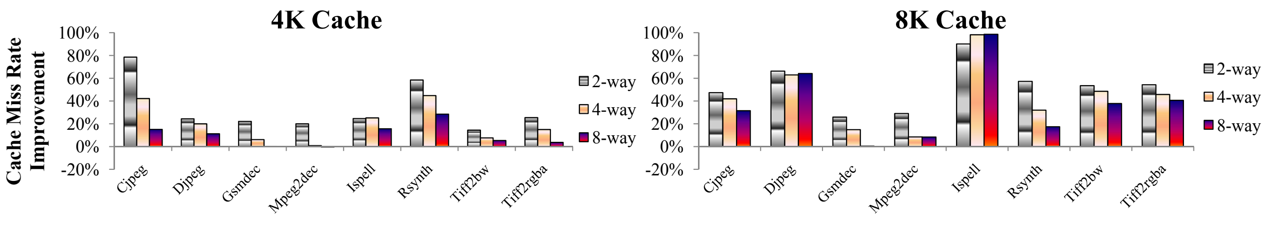


Figure 9: Cache miss rate improvement of IBP over original code layout for FIFO replacement policy.

Portability. As shown in the Table 2, for each configuration, the cache miss varies significantly across different code layouts. For example, for 4K size, 1-way cache, *Rsynth* incurs about 8 million cache misses using the code layout generated for 4K-1-way configuration while the number of cache miss goes up to more than 62 million using code layout generated for 4k-2-way cache configuration. As expected, we also observe that, in most cases, the code layout C generated for the cache configuration C is the best code layout for cache configuration C among all the code layouts in the set (diagonal line from top left to bottom right). This is because the underlying cache parameters match exactly with the assumed cache parameters during procedure placement. However, the code layout generated for C may perform badly for other configurations, though it is good for configuration C. For example, for benchmark *Ispell*, 4k-1-way code layout is better than 4k-2-way code layout for 4k-1-way cache configuration, but worse than 4k-2-way code layout for 4k-2-way cache configuration. In other words, there is no single code layout that performs better than other code layouts for all the configurations.

More importantly, the above portability problem exists for all procedure placement techniques [8, 9, 13, 10, 3] that take cache parameters into account. This has been observed for [8, 9], which is not shown here due to space limitation.

Performance. The comparison of 9 (8 specific + 1 neutral) code layouts in terms of average performance is shown in Figure 10. Y-axis in Figure 10 shows the average cache miss improvement over all cache configurations compared to original code layout. Only 6 benchmarks are shown due to space limitation. Similar trend has been observed for others.

First, the neutral code layout always performs better than any specific code layout in terms of average performance for all the benchmarks. For all the benchmarks, neutral code layout achieves positive performance improvement. Though the neural code layout is better than any other specific code layout, it does not win for all the configurations. For most of the cases, the S-A-way code layout is the best for C_A^S configuration. So, our neutral code layout is not the best code layout for a specific cache configuration, but the best code layout for the average performance across a set of configurations.

Second, we notice that the best specific code layout is different for different benchmarks. For example, for *Djpeg*, the best specific code layout is 8K-1-way; for *Mpeg2dec*, the best specific code layout is 4K-1-way. Moreover, some specific code layouts for highly associative caches (4-way and 8-way) degrade average performance compared to the original code layout (e.g., *Mpeg2dec*, *Rsynth* etc). Though the specific code layouts are better than the original code layout for their own configurations, they are worse than the original code layout for the rest of the configurations. As a result, they are worse than the original code layout on an average.

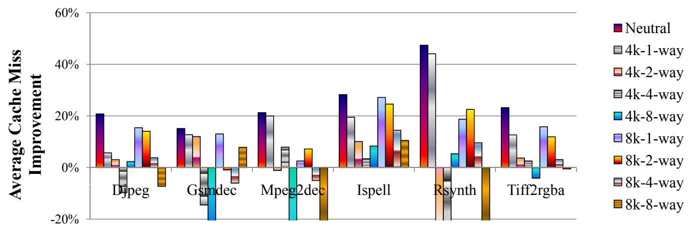


Figure 10: Average cache miss rate improvement comparison.

Code Size. Neutral code layout achieves better average miss rate improvement with small code size expansion. On an average, neutral code layout expands code size by 2.3%.

8. CONCLUSION

Procedure placement is a popular instruction cache optimization technique that aims to improve instruction cache performance by reducing conflicts in the cache through compile/link time reordering of procedures. However, state of the art of procedure placement techniques make reordering decisions based on imprecise conflict information and they mainly target direct mapped cache only. Therefore, the code layout generated by the state of the art may be worse than the original code layout for set associative caches. In this paper, we first introduce intermediate blocks profile (IBP) to precisely model the cost and benefit of procedure placement. Then, we propose a procedure placement algorithm using IBP. Our algorithm starts from the original procedure order and selects the most beneficial procedures and their placements iteratively. Experiments indicate that our technique improves both cache performance and energy consumption compared to the state of the art. Furthermore, we notice that the code layout generated for a specific cache configuration is not portable across different cache configurations. Thus, we propose another algorithm that uses IBP to generate a neutral layout for a set of cache configurations.

9. ACKNOWLEDGMENTS

This work was partially supported by research grants R-252-000-387-112 & MOE academic research fund Tier 2 MOE2009-T2-1-033.

10. REFERENCES

[1] K. Anand and R. Barua. Instruction cache locking inside a binary rewriter. In *CASES: Proceedings of the 2009 international conference on Compilers, architecture, and synthesis for embedded systems*, 2009.

[2] T. Austin et al. Simplescalar: An infrastructure for computer system modeling. *Computer*, 35(2):59–67, 2002.

[3] S. Bartolini and C. A. Prete. Optimizing instruction cache performance of embedded systems. *ACM Trans. Embed. Comput. Syst.*, 4(4), 2005.

	Cache Configuration							
	Ispell							
Layout	4K-1-way	4K-2-way	4K-4-way	4K-8-way	8K-1-way	8K-2-way	8K-4-way	8K-8-way
4K-1-way	1,102,555	1,452,167	1,786,182	1,879,787	863,642	498,419	325,743	9,758
4K-2-way	1,508,960	1,287,602	1,428,123	1,752,959	744,746	649,710	729,356	743,937
4K-4-way	1,559,625	1,666,711	1,327,332	1,779,965	858,854	800,705	717,038	791,925
4K-8-way	1,522,588	1,633,681	1,657,776	1,635,333	928,764	961,416	362,726	314,796
8K-1-way	1,487,002	1,558,181	1,811,224	1,946,053	163,051	115,887	68,588	12,269
8K-2-way	1,373,225	1,692,295	1,791,685	1,985,196	395,723	71,407	100,958	7,153
8K-4-way	1,598,770	1,543,161	1,750,547	1,946,337	1,039,925	518,231	9,577	9,018
8K-8-way	1,529,317	1,665,145	1,725,674	1,899,575	1,152,934	653,463	173,629	2,293
	Rsynth							
Layout	4K-1-way	4K-2-way	4K-4-way	4K-8-way	8K-1-way	8K-2-way	8K-4-way	8K-8-way
4K-1-way	8,672,681	9,648,096	10,411,082	10,905,595	6,436,063	4,640,705	5,102,060	5,337,687
4K-2-way	62,448,135	9,172,659	10,514,702	11,333,635	37,662,987	6,661,833	5,175,124	5,553,046
4K-4-way	43,399,761	46,574,878	9,158,260	9,508,933	12,728,470	11,786,572	565,3478	5,564,737
4K-8-way	27,808,251	24,049,320	17,140,999	8,922,127	7,768,132	6,808,949	5,327,826	5,674,078
8K-1-way	25,684,168	16,974,226	15,573,160	13,028,555	3,467,594	3,960,793	4,680,007	5,487,350
8K-2-way	20,305,855	13,342,605	12,224,826	12,393,127	12,837,206	3,702,064	4,440,956	5,434,249
8K-4-way	25,773,676	21,407,231	12,029,527	10,012,284	12,094,403	8,132,121	4,094,100	5,307,878
8K-8-way	76,932,309	24,046,861	13,506,907	12,147,314	28,077,477	15,401,963	4,743,588	4,570,997

Table 2: Cache misses of different code layouts running on different cache configurations.

[4] E. Berg and E. Hagersten. Statcache: A probabilistic approach to efficient and accurate data locality analysis. In *ISPASS: Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, 2004.

[5] K. Beyls and E. H. D'Hollander. Reuse distance as a metric for cache behavior. In *Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Systems*, 2001.

[6] C. Ding and Y. Zhong. Predicting whole-program locality through reuse distance analysis. *SIGPLAN Not.*, 38(5), 2003.

[7] A. Ghosh and T. Givargis. Cache optimization for embedded processor cores: An analytical approach. *ACM Trans. Des. Autom. Electron. Syst.*, 9(4), 2004.

[8] N. Gloy and M. D. Smith. Procedure placement using temporal-ordering information. *ACM Trans. Program. Lang. Syst.*, 21(5):977–1027, 1999.

[9] C. Guillon et al. Procedure placement using temporal-ordering information: dealing with code size expansion. In *CASES: Proceedings of the international conference on Compilers, architecture, and synthesis for embedded systems*, 2004.

[10] A. H. Hashemi et al. Efficient procedure mapping using cache line coloring. *SIGPLAN Not.*, 32(5):171–182, 1997.

[11] M. D. Hill and A. J. Smith. Evaluating associativity in cpu caches. *IEEE Transactions on Computers*, 38(12), 1989.

[12] W. W. Hwu and P. P. Chang. Achieving high instruction cache performance with an optimizing compiler. *SIGARCH Comput. Archit. News*, 17(3):242–251, 1989.

[13] J. Kalamatianos et al. Analysis of temporal-based program behavior for improved instruction cache performance. *IEEE Trans. Comput.*, 48(2):168–175, 1999.

[14] Y. Liang and T. Mitra. Static analysis for fast and accurate design space exploration of caches. In *CODES+ISSS: Proceedings of the 6th IEEE/ACM/IFIP international conference on Hardware/Software codesign and system synthesis*, 2008.

[15] Y. Liang and T. Mitra. Instruction cache locking using temporal reuse profile. In *DAC: Proceedings of the 47th Design Automation Conference*, 2010.

[16] N. Nguyen, A. Dominguez, and R. Barua. Memory allocation for embedded systems with a compile-time-unknown scratch-pad size. In *CASES: Proceedings of the international conference on Compilers, architectures and synthesis for embedded systems*, 2005.

[17] S. Parameswaran and J. Henkel. I-copes: fast instruction code placement for embedded systems to improve performance and energy efficiency. In *ICCAD : Proceedings of the IEEE/ACM international conference on Computer-aided design*, 2001.

[18] K. Pettis and R. C. Hansen. Profile guided code positioning. *SIGPLAN Not.*, 25(6), 1990.

[19] R. A. Sugumar and S. G. Abraham. Set-associative cache simulation using generalized binomial trees. *ACM Transactions on Computer Systems*, 13(1), 1995.

[20] H. Tomiyama and H. Yasuura. Code placement techniques for cache miss rate reduction. *ACM Trans. Des. Autom. Electron. Syst.*, 2(4):410–429, 1997.

[21] R. A. Uhlig and T. N. Mudge. Trace-driven memory simulation: a survey. *ACM Comput. Surv.*, 29(2):128–170, 1997.

[22] S. J. E. Wilton and N. P. Jouppi. Cacti: An enhanced cache access and cycle time model. *IEEE Journal of Solid-State Circuits*, 31:677–688, 1996.

[23] C. Zhang, F. Vahid, and W. Najjar. A highly configurable cache architecture for embedded systems. *SIGARCH Comput. Archit. News*, 31(2), 2003.