Procedure Placement Using Temporal-Ordering Information

NIKOLAS GLOY
Appliant, Inc.
and
MICHAEL D. SMITH
Harvard University

Instruction cache performance is important to instruction fetch efficiency and overall processor performance. The layout of an executable has a substantial effect on the cache miss rate and the instruction working set size during execution. This means that the performance of an executable can be improved by applying a code-placement algorithm that minimizes instruction cache conflicts and improves spatial locality. We describe an algorithm for procedure placement, one type of code placement, that significantly differs from previous approaches in the type of information used to drive the placement algorithm. In particular, we gather temporal-ordering information that summarizes the interleaving of procedures in a program trace. Our algorithm uses this information along with cache configuration and procedure size information to better estimate the conflict cost of a potential procedure ordering. It optimizes the procedure placement for single level and multilevel caches. In addition to reducing instruction cache conflicts, the algorithm simultaneously minimizes the instruction working set size of the program. We compare the performance of our algorithm with a particularly successful procedure-placement algorithm and show noticeable improvements in the instruction cache behavior, while maintaining the same instruction working set size.

Categories and Subject Descriptors: B.3.3 [**Memory Structures**]: Performance Analysis and Design Aids—*simulation*; D.3.4 [**Programming Languages**]: Processors—*compilers*; *optimization*

General Terms: Algorithms, Performance

Additional Key Words and Phrases: Code placement, conflict misses, temporal profiling, working-set optimization

Michael D. Smith is funded in part by a NSF Young Investigator award (grant no. CCR-9457779), NSF grant no. CDA-94-01024, DARPA grant no. NDA904-97-C-0225, and research grants from AMD, Compaq, Digital Equipment, HP, IBM, and Intel.
The research reported in this article was conducted while the first author was a graduate student in the Division of Engineering and Applied Sciences at Harvard University.
Authors' addresses: N. Gloy, Appliant Inc., 3513 NE 45th St. Suite 3, Seattle, WA 98105; email: gloy@appliant.com; M. Smith, Division of Engineering and Applied Sciences, Harvard University, 29 Oxford St., Cambridge, MA 02138; email: smith@eecs.harvard.edu.

1. INTRODUCTION

The average rate at which a machine can supply its processor core with instructions is an important component of program performance. The importance of this component will continue to grow as processor clock speeds increase faster than memory cycle times and as architects improve the ability of processor cores to exploit instruction-level parallelism. For example, on-chip growth in the number of parallel functional units will not improve overall processor performance if the memory is unable to supply instructions at a correspondingly higher rate. While the use of multilevel caches and other architectural improvements in the processor and memory subsystem work toward meeting instruction fetch requirements, research over the past 10 years in code-placement techniques has shown that compile-time reorganization of the program text segment can also contribute toward this goal. This article presents and evaluates a family of profile-driven, code-placement techniques that achieve noticeable improvements in the performance of modern instruction memory hierarchies.

The memory hierarchy works best if the requested instructions are most often found in the primary (first level or L1) instruction cache, if misses in the primary instruction cache are most often satisfied by the second-level cache (if any), and so on. Similarly, we can maximize program performance by minimizing the number of pages required during any interval in the program execution and over the entire program execution. The compile-time placement of program code blocks within an executable determines the cache mapping of its instructions and influences the size of its working set, and therefore this placement has a substantial impact on the performance of the instruction memory.

Because the layout of instructions affects program performance, code-placement techniques are also an extremely important tool in the evaluation of compile-time optimizations. In particular, the number of instruction cache conflict misses incurred during a program run are completely determined by the placement of the program's code. Compiler optimizations therefore that change the size or relative placement of the generated code often (unintentionally) change the number of instruction cache conflicts. Even small changes in the cache mapping of a program can have surprisingly large effects on the number of conflict misses during execution. The performance impact of these unintended changes can obscure or cancel out the intended performance improvement of an optimization. By applying a code-placement technique that reduces cache conflict misses, we can avoid unpredictable variations in conflict misses and more accurately evaluate an optimization.

Code placement was an active field of research from the mid-1960s to the mid-1970s, when there was much interest in improving the working set size of program code to reduce paging [Cytron and Loewner 1986; Ferrari 1974; 1975; Hatfield and Gerald 1971; Ryder 1974]. More recently, the introduction of instruction caches into microprocessors in the late 1980s led to renewed interest in code placement, this time with the goal of more efficiently using the instruction cache. These later techniques [Bershad et al. 1994; Hashemi et al. 1997; McFarling 1989; Pettis and Hansen 1990; Torellas et al. 1995] use heuristics and profile information to reduce the number of conflict misses in the primary instruction cache by reordering the program code, and most of this work uses cache parameters (such as cache size and

line size) as well as procedure sizes to accurately model the cache mapping of the code.

The best of these published techniques use some simple profile information to direct the code placement. Interestingly, all of the profile-driven techniques use essentially the same kind of profile information: the profiles summarize how often each program code block transferred control to each other code block. For example, Hatfield and Gerald [1971] describe a technique for working-set optimization that uses a matrix C of profile information, where $C[i, j]$ is a count of the transfers from code sector i to code sector j. Pettis and Hansen [1990] (and all subsequent modifications of their procedure-placement technique by other authors) use profile data in the form of a *weighted call graph* (WCG). In a WCG, there is a node for each program procedure. An undirected edge connects two nodes P and Q if P calls Q or Q calls P. Each edge $e_{P,Q}$ is annotated with a value that equals number of procedure calls that occurred between P and Q.

Though the previously published techniques for code placement work well, we can do better. The key to doing better is to use profile information that better summarizes the important temporal characteristics of the program execution. Figure 1 illustrates this point. It contains a simple program in which the conditional *cond* is taken 50% of the time. Figure 1(b) presents the corresponding WCG.[1] This WCG is independent of the actual branch trace that occurred during program execution. If the branch trace *interleaved* calls to procedures X and Y (call trace 1), the code layout in Figure 1(c) yields the lowest miss rate. If the branch trace *phased* the calls so that all calls to procedure X occurred before any calls to Y (call trace 2), the code layout in Figure 1(d) yields the lowest miss rate.

In this article, we define a type of profile, called *temporal-ordering* information, that is essential to predicting cache conflict misses, and we introduce a practical profiling technique that can extract this information during program execution. We also present a technique for optimizing the placement of a program's procedures based on this temporal-ordering information.

Though we focus on the placement of procedures, our approach is general enough to be applied at any granularity of code block.[2] Many previously published techniques place procedures, but there are also techniques working at a lower level (e.g., basic blocks) or higher level (e.g., object files). A smaller granularity has the potential for better results, but placing code blocks that are smaller than procedures involves more intrusive code transformation and requires more compiler support. Procedure placement has the advantage that it can be implemented by the linker or other postcompilation tools, and that it is orthogonal to other techniques to arrange code within procedures.

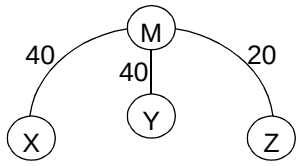
Our procedure-placement technique improves the instruction cache mapping to reduce the number of instruction cache misses in one or multiple levels of instruction cache. In addition, our approach considers the effect of the code placement on other levels of the memory hierarchy in order to optimize the spatial locality of the program's code, given the constraints of the cache mapping. The overall effect of

[1]We use WCGs in all of our following examples. We could have equally well used the profile matrix of Hatfield and Gerald [1971], since it also suffers from the same shortcoming.

[2]We use the term *code block* to refer to pieces of the program code at the chosen level of granularity.

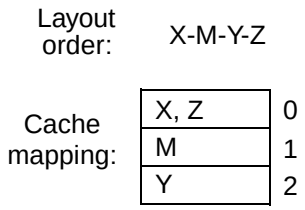
```
Proc M()
  loop 20 times
    loop 4 times
      if (cond)
        call X;
      else
        call Y;
    endloop
    call Z;
  endloop
endproc
```

(a) Example program.

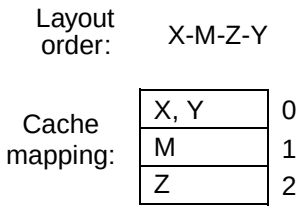


Call Trace #1: $M((XYXY)Z)^{20}$
Call Trace #2: $M((X)^4Z)^{10}((Y)^4Z)^{10}$

(b) The WCG and two possible call traces.



(c) Layout produced by PH given the WCG in (b).



(d) An alternative layout that works better for Trace #2.

Fig. 1. Example of a program with multiple possible temporal orderings. Assume that the condition *cond* is true 50% of the time. The same WCG is obtained regardless of the temporal ordering of the calls to procedures X and Y, as represented by the call traces 1 and 2. Parts 1(c) and 1(d) present two different cache layouts. Call trace 1 incurs fewer conflicts (39) for the layout in part 1(c) than the layout in part 1(d) (79). Call trace 2, however, incurs fewer conflicts (0 vs. 19) for the layout in part 1(d). For these calculations, we have assumed that each procedure occupies a single cache line, and that the target instruction cache is direct mapped and three lines large. Existing approaches would use the WCG to produce the layout in part 1(c). Our technique gathers profile information differentiating call trace 1 from call trace 2 and uses this information to produce the best layout for each execution.

these optimizations is to allow the memory system to provide instructions to the processor more efficiently. As a result, the processor spends less time waiting for instructions to be fetched, and it can thus execute the program faster.

We begin in Section 2 with a brief examination of how code placement impacts the performance of a memory system, and in particular, what aspects of the dynamic behavior of a program cause cache conflict misses. We use this discussion to justify the content of our temporal profiles. We continue in Section 3 with an overview of our methodology for the performance evaluation experiments in the following sections. In Section 4, we introduce the basic aspects of our code-placement algorithm by restricting our analysis to a single level of cache, and we present our method of capturing temporal-ordering information. We present some experimental

results comparing the performance of our algorithm to our implementation of the procedure-placement algorithm by Pettis and Hansen [1990]. Section 5 shows how to extend our algorithm to perform code placement in multilevel cache hierarchies. The distribution of procedures across virtual-memory pages determines the size of the instruction working set of the application. Changes in the working set size can affect the performance of the program by changing the number of translation lookaside buffer (TLB) misses and the number of pages fetched from disk. We address this aspect of code placement in Section 6 by extending the code-placement technique in Section 5 so that it additionally optimizes for the instruction working set size. Section 7 reviews some related work, and Section 8 presents our conclusions.

2. FUNDAMENTALS OF CODE PLACEMENT

This section is intended to provide a basis for the detailed discussion of our work in Sections 4, 5, and 6. We review aspects of memory systems that are relevant to code placement, and show how instruction addresses influence the cache and paging behavior of a program. Finally, we introduce our methodology for evaluating and comparing code-placement techniques.

2.1 Code-Placement Effects on Instruction Caches

The following summary of basic cache operation is a brief reminder of the connection between instruction addresses and the instruction cache. It will allow the reader to understand how code placement changes instruction cache conflicts, and which aspects of the temporal behavior of a program are of interest to a code-placement algorithm. For a comprehensive treatment of cache techniques, see Smith [1982].

2.1.1 *Basic Cache Operation.* Consider a memory system consisting of only an instruction cache and the main memory. The cache consists of an array of cache lines of equal size, which store instructions that have previously been fetched from main memory. As an example, the on-chip instruction cache of the DEC Alpha 21064 chip consists of 256 lines of 32 bytes each. Each cache line stores the contents of an aligned block of memory from the next level of the memory hierarchy. A memory block is "aligned" in the sense that its starting address is a multiple of the cache line size. When the processor fetches an instruction at a given address, it will try to fetch it from the cache, if it is present there, and otherwise fetch it from main memory. The processor computes the memory address of the aligned memory block containing the requested instruction. Then it checks if one of the cache lines currently contains the memory block with this address. If this is the case, the instruction is retrieved from the cache line that contains it. Otherwise, the processor fetches the entire memory block into a cache line, and thus replaces the previous contents of that cache line.

2.1.2 *Mapping Memory Addresses to Cache Addresses.* The discussion in the previous subsection leaves open the question of which cache lines can be used to store a memory block of a particular address. Under a fully associative scheme, any cache line can be used to store any memory block. Since this requires an expensive search through all cache lines for every instruction fetch, fully associative caches are used only when the number of cache lines is small, such as in translation lookaside buffers (TLBs). Set-associative schemes are less costly in terms of time and space,

but more restrictive in the possible placement of a memory block. In an N-way associative cache, for each memory block address, there is a set of N cache lines where it may be stored. When N equals 1, the cache is direct mapped. In this simple caching scheme, each memory block can be stored in only one cache line.

A direct-mapped cache is commonly used in microprocessors with a very high clock speed, since its lookup process is faster than that of a set-associative cache [Hill 1988]. In this article, we focus mostly on direct-mapped caches, though our techniques apply to set-associative caches. Direct-mapped caches suffer more from conflict misses than set-associative caches, because the latter can avoid some conflict misses by offering several alternative cache locations for each memory block. Therefore, our techniques will produce larger improvements for direct-mapped caches. For some programs, a set-associative cache with the typical set size of 2 or 4 will be sufficient to avoid many conflict misses, and therefore there will not be much of a difference between our code-placement technique and simpler methods. But for programs whose temporal behavior and memory layout is such that even a set-associative cache suffers from many conflict misses on some cache lines, our technique will generally be better able to avoid this behavior.

In a direct-mapped cache, the relationship between the memory address and the cache line number is simply

$$cache\_line = (memory\_address/line\_size) \bmod num\_cache\_lines \tag{1}$$

Therefore, any two code fragments whose addresses differ by a multiple of the cache size are mapped to the same cache line, and they cannot both be present in a direct-mapped cache simultaneously. This situation is called a *cache mapping conflict*. If both fragments are needed at the same time (e.g., referenced inside the same inner loop), the program will suffer from repeated misses on that cache line, even if there are other available cache lines that are not used at that moment. Such a cache miss that is caused by the restrictive cache mapping is known as a *conflict miss*.

2.1.3 *Identifying a Metric Related to Conflict Misses.* By moving pieces of the code within the executable, we can change which pieces experience cache mapping conflicts. The task of a procedure-placement technique is to find a placement of a program's procedures that avoids frequently occurring conflict misses. To perform this task, procedure-placement algorithms rely on some sort of metric that (hopefully) indicates how many conflict misses would result from a particular cache mapping.

Using the following equation, we can calculate exactly how many conflict misses will result from a particular cache mapping. Given a cache line n of a direct-mapped cache, a set S of program code blocks mapping to n, and an execution trace of program code blocks,

$$conflict\_misses(n) = \left(occ(S) - |S| - \sum_{b \in S} reuse(b, S) \right) \tag{2}$$

where $occ(S)$ is the number of occurrences of the blocks in S in the execution trace. We define $reuse(b, S)$ below.

Equation (2) works as follows. Clearly, the number of conflict misses must be less than the number of references to the code blocks in S. This number is too large for

two reasons. First, the initial reference to a code block is defined to be a cold start miss [Hatfield and Gerald 1971], and thus we subtract the size of S.[3] Second, we do not incur a cache miss if a reference is to a code block in the cache at the point of the reference. We refer to this as a *reuse* reference. For a direct-mapped cache, a reuse reference occurs if (1) there was an earlier reference to the same code block and (2) none of the code blocks referenced since then has replaced the contents of the cache line containing that code block. The term $reuse(b, S)$ represents the number of reuse references for b in the execution trace when it is mapped to the same cache line as the other blocks in S.

Looking at this another way, we notice that a conflict miss for a code block b occurs whenever two successive references to b are separated by a reference to at least one other code block c that maps to the same cache line as b. Therefore, we are not interested in the entire trace of code blocks, but simply in a measure of the interleaving of the code blocks. For example, two code blocks whose execution is not interleaved could occupy the same cache line without penalty. Thus, the frequency of interleaving is a crucial piece of information about the temporal behavior of the program needed for procedure placement. In Section 4, we describe our method of extracting this information from an execution trace and show that it is superior to the metric used by approaches employing a WCG.

2.2 Code-Placement Effects on Other Levels of the Memory Hierarchy

In addition to the instruction caches, code placement also affects the (code) working set of the program. We use Denning's [1970] definition of a program's working set:

$$W(k,h) = \{j : \text{page } j \text{ is referenced among } r_{k-h-1} \dots r_k\} \text{ where } (h \geq 1) \qquad (3)$$

At the kth reference r_k, the working set $W(k,h)$ is the "contents" of a "window" of size h looking backward at the trace of instruction references from reference r_k.

Under code placement, the trace of instructions remains the same; what changes is the set of pages in which the instructions are stored. Depending on how code blocks are distributed across pages, the same set of instructions may require a different number of pages to be referenced in order to access the instructions. Thus, it is the spatial locality at the page granularity level with which we are concerned.

For the purpose of evaluating a code-placement technique, one important metric is the average working set size over the program's execution. But it is not obvious which value should be used for h, the window size. For very large values, the working set becomes identical to the total number of pages touched. This number is relevant because this is the number of pages that will be paged in during the program's execution.

However, the working set size for smaller values of h also has an impact on performance. If the working set size, in pages, for a particular value of h is greater than the number of TLB entries, then we expect the program to suffer many TLB misses for each interval of h references. If a code placement somehow destroys spatial locality and drives the working set size above the number of TLB entries, we expect that the program's performance will be decreased due to an increase in TLB misses.

[3]To be precise, we want to subtract the size of the set of referenced members of S.

Table I. Summary of Our Benchmark Programs. The size of the executed code is from the testing data input. All sizes are given in kilobytes.

Name	Description	Static text size	Size of executed code
gcc	GNU C compiler, -O2 (Spec95)	1432	556
go	Go strategy game (Spec95)	384	282
ghostscript	Postscript interpreter	528	161
latex	LaTeX typesetting application	312	134
perl	Perl language interpreter (Spec95)	352	73
porky	SUIF compiler code transformation pass	1984	369
vortex	Object-oriented database (Spec95)	664	284

3. EXPERIMENTAL METHODOLOGY

To evaluate the success of a procedure-placement technique, we first run a benchmark program on a training data set to collect the profile information required by the technique. We then apply the procedure-placement technique to the benchmark program, producing new memory addresses for all instructions of the program. Next, we simulate the execution of the program on a testing data set (different from the training data set). We use a trace-driven memory system simulator that measures the behavior of the program in the various levels of the memory hierarchy. These measurements include all the aspects of the program behavior that are affected by code placement, allowing us to accurately evaluate the total effect of the placement.

In particular, we make measurements of the following components of overall program performance:

(1) the number of L1 instruction cache misses;

(2) the number of L2 instruction cache misses;

(3) the code working set sizes measured for intervals of several lengths.

The working set sizes capture both the impact on TLB misses (for short and medium-length intervals) and the amount of memory occupied by the code (for long intervals). We believe that these components are sufficient because there are no other sources of memory system penalties. It is possible that the delay for reading pages from a disk varies depending on the disk layout, but we consider this to be a file system issue and thus beyond the scope of this research.

We perform our experiments on programs compiled for a workstation based on a Alpha 21164 processor running Digital Unix 4.0. We use Digital's Atom tool [Srivastava and Eustace 1994] to generate the traces needed for profiling and simulation of the memory system. We have verified the correctness of our procedure-placement techniques by implementing them in the Machine-SUIF compiler back end [Smith 1996].

Table I lists the benchmarks we use for our experiments in Sections 4, 5, and 6. We choose these benchmarks because they exhibit interesting (i.e., nontrivial) instruction memory behavior. If the size of the code touched by a benchmark does not exceed the cache size, then any sensible code-placement technique will be able to avoid all conflict misses. Table II describes the training and testing traces we use.

Table II. Traces for Training and Testing. The "length" of a trace is measured in dynamic instructions. The "avg. miss rate" is the average miss rate for a 4KB direct-mapped instruction cache with a line size of 32 bytes, averaged over 20 random procedure orderings.

	Training			Testing			
Benchmark	name	description	length	name	description	length	avg. miss rate
gcc	g28	Spec 95 go source code	241 M	regcomp	Spec95 perl source code	307 M	7.2 %
go	915	9 handicap, level 4, 15x15 board	379 M	1511	level 15, 11x11 board	609 M	5.8 %
ghostscript	a94	3-page paper	93 M	m97	11-page paper	156 M	6.2 %
latex	aprl	14-page paper	153 M	gr	12-page paper	158 M	6.1 %
perl	foo	strings, patterns, arithmetic	398 M	scrabbl	Spec95 input, shortened	363 M	8.2 %
porky	g28	Spec95 go source code	869 M	regcomp	Spec95 perl source code	965 M	7.2 %
vortex	train	Spec95 training input	265 M	test	Spec95 testing input, shortened	516 M	9.0 %

3.1 An Effective Approach Based on the WCG

To help evaluate the effectiveness of our approach, we also implemented the WCG-based procedure placement-technique by Pettis and Hansen [1990]. While these authors also describe techniques for basic-block placement and branch alignment in their paper, we implemented only their procedure-ordering algorithm. We refer to our implementation of their algorithm as PH. Their procedure placement technique is considered the "reference standard" in this area of research because of its effectiveness and simplicity.

Though we only compare our approach to PH in this article, Gloy et al. [1997] have previously shown that our procedure-placement algorithm for a single-level instruction cache (described in Section 4) is superior to the algorithm proposed by Hashemi et al. [1997]. Hashemi et al. describe a WCG-based approach that is claimed to be one of the best of the recently published procedure placement techniques. Unlike Pettis and Hansen's approach, their approach uses information about the cache configuration and procedure sizes during placement, and it allows gaps between procedure placements in an attempt to reduce conflict misses. Gloy et al. shows that our approach is better at reducing the L1 cache miss rate than Hashemi et al.'s approach simply because our approach uses more detailed profile information. Both approaches precisely model the cache during placement and allow gaps between procedures. Since Hashemi et al. do not describe how one would extend their technique to optimize the performance of a multilevel cache hierarchy or to reduce a program's working set size, we do not include the approach by Hashemi et al. in our evaluations.

The rest of this subsection describes our implementation of Pettis and Hansen's procedure-placement algorithm. As we explain in the following sections, our new

algorithm retains much of the structure and many of the important heuristics found in the Pettis and Hansen approach, and thus it is helpful to review their algorithm here.

The PH algorithm is based on a simple heuristic: if two procedures are placed adjacent in memory (or close to each other), they are less likely to incur conflict misses, unless the sum of their sizes is greater than the cache size. It is a greedy algorithm that iterates over the edges of the WCG in order of decreasing weight. For each edge, it tries to place the procedures of the edge as close to each other as possible so as to reduce the probability of conflict. Each step is subject to the constraints imposed by the placement decisions of earlier steps; the algorithm does not backtrack or change earlier placement decisions.

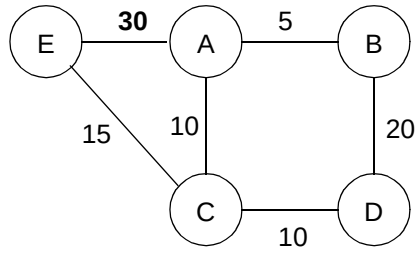
PH uses the WCG both to select the next procedure to place and to determine where to place that procedure in relationship to the already placed procedures. The algorithm begins by making a copy of this initial graph; we refer to this copy as the *working* graph. PH searches this working graph for the edge with the largest weight. Call this edge $e_{u,v}$. Once this edge is found, the algorithm merges the two nodes u and v into a single node u' in the working graph (more details in a moment). The remaining edges from the original nodes u and v to other nodes become edges of the new node u'. To maintain the invariant of a single edge between any pair of nodes, PH combines each pair of edges $e_{u,r}$ and $e_{v,r}$ into a single edge $e_{u',r}$ with weight $W(e_{u,r}) + W(e_{v,r})$. The algorithm then repeats the process, again searching for the edge with the largest weight in the working graph, until all edges have been removed from the working graph. Each step reduces the number of nodes by one, and the algorithm terminates if there is a single node left or if there are no more edges left.

PH attempts to reduce the chance of a conflict miss between procedures by placing procedures connected by a heavy weight edge in close proximity in the address space. The procedures within a node are organized as a linear list called a *chain*. When PH merges two nodes, their chains can be combined into a single chain in four ways. Let A and B represent the chains, and A^R and B^R the reverse of each chain. The four possibilities are AB, AB^R, A^RB, and A^RB^R. We want to choose one of these based on the "closest-is-best" heuristic. The paper by Pettis and Hansen [1990] does not specify this aspect in detail, but we believe that the following algorithm reflects their intentions. Our implementation of PH queries the original graph to determine the edge e with the largest weight between a procedure p in the first chain and a procedure q in the second chain and chooses the merged chain that minimizes the distance (in bytes) between p and q. Figure 2 shows each step of the PH algorithm for simple example WCG.

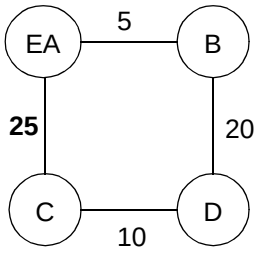
3.2 Using Randomization to Compare Placement Techniques

We normally expect code optimizations to behave similarly to a continuous function: small changes in the behavior of the optimization cause small changes in the performance of the resulting executable. With code-placement optimizations, this is often not the case: small changes in the layout of a program can cause dramatic changes in the cache miss rate.

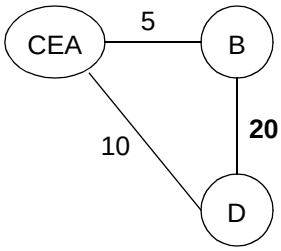
As an example, we simulated the instruction cache behavior of the SPECint95 *perl* program for two slightly different layouts. The first layout is the output of our



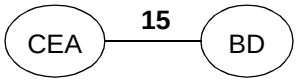
(a) Combine E and A; merge edges (E,C) and (A,C) into (EA,C).



(b) Combine EA and C; place C next to E because the original edge (E,C) is more important than (A,C).



(c) Combine B and D; merge edges (B,CEA) and (D,CEA).



(d) Combine CEA and BD; place C next to D because (C,D) is the most important original edge between the nodes.

(e) The result is the procedure order AECDB.

Fig. 2. This example shows how the PH algorithm processes a WCG to produce a procedure ordering. The original WCG is shown in part 2(a). The edge with the heaviest weight at each step is shown in bold.

own code layout algorithm, and the second layout is identical to the first except that each procedure is padded by an additional 32 bytes (one cache line) of empty space at its end. The instruction cache miss rate changed from 3.8% for the first layout to 5.4% for the second layout; this is a remarkable change for such a trivial difference between the layouts. In fact, it is possible to introduce a large number of misses by moving one code block by only a single cache line.

The relationship between code placement and working set size is usually not as chaotic. But when a small increase in the working set size pushes it beyond the number of TLB entries, it can cause a large increase in TLB misses. This is because a working set that is smaller than the number of TLB entries causes very few TLB misses, but a working set that is larger than the number of TLB entries suffers from thrashing.

For greedy code-layout algorithms, we have the additional problem that different layouts, in fact substantially different layouts, often result from small changes in the input profile data. At each step, the PH algorithm greedily chooses (as does our own algorithm) the highest-weight edge in the working graph. If there are two edges, say with weight 1,000,000 and 1,000,001, the (barely) larger edge will always be chosen first, even though such a small difference is unlikely to represent a statistically significant basis for preferring one edge over the other. Worse, ties resulting from identical edge weights are decided arbitrarily. Decisions between two equally good alternatives, which must be made one way or the other, affect not only the current step of the algorithm, but all future steps.

As a result, we find it difficult to draw conclusions about the relative performance of different code-layout algorithms from a small number of program traces. Ideally, we would like to have a large enough set of different inputs for each benchmark to get an accurate impression of the distribution of results. Unfortunately, this is very hard to do in practice, since common benchmark suites are typically distributed with only a few input sets for each benchmark application.

To overcome this obstacle, we use a randomization technique proposed by Blackwell [1998]. We simulate the effect of many slightly different application input sets by first running an application with a single input, and then applying random perturbations to the resulting profile data to produce a large set of slightly different profiles.

For the algorithms in our comparison, we perturb a weighted graph by multiplying each edge weight by a value close to one. Specifically, the initial weight w is replaced by the perturbed weight $\overline{w}$ according to the equation

$$\overline{w} = w \times \exp(sX) \tag{4}$$

where X is a random variable, normally distributed with mean 0 and variance 1, and s is a scaling factor, which determines the magnitude of the random perturbations. Using multiplicative rather than additive noise is attractive for two reasons. First, additive noise can cause weights to become negative, for which there is no obvious interpretation. Second, the method is inherently self-scaling in the sense that reasonable values for s are independent of the initial edge weights.

Large values of s will cause the layout to be effectively random, as the perturbed graphs will bear little relationship to the profile data. Small values of s will cause only statistically insignificant differences in edge weights, and we can then observe the range of results produced by these small changes. We use $s = 0.1$ in our experiments. Blackwell [1998] shows that for several code-placement algorithms, values of s as low as 0.01 elicit most of the range of performance variation from the system, while values of s as high as 2.0 do not degrade the average performance very much.

4. PROCEDURE PLACEMENT USING TEMPORAL-ORDERING DATA

In this section, we present the basics of our code-placement algorithm. We focus the discussion on minimizing the cache conflicts in a single level of instruction cache. Section 4.1 introduces our profiling method that captures information on the temporal ordering of a program's execution. We organize this profile information as a *Temporal Relationship Graph* (TRG).

Section 4.2 describes our procedure-placement algorithm, which we refer to as *Temporal Profile Conflict Modeling* (TPCM) because it is based on temporal profile data and accurately models cache mapping conflicts. It uses a TRG to find a cache-relative alignment for each procedure that minimizes cache conflicts. Then, it outputs a procedure ordering that may include gaps between procedures to achieve the desired cache-relative alignment for each procedure. This ordering can be used by a linker or other code reordering tool to produce an executable with the correct procedure addresses.

Section 4.3 presents cache simulation results that show the improvement in miss rates that our TPCM algorithm achieves over the PH algorithm. Section 4.4 discusses the application of TPCM to set-associative caches. Section 4.5 shows that a TRG yields a stronger linear relationship between conflict-metric values and cache miss rates than does a WCG. Finally, Section 4.6 briefly investigates the interplay between procedure splitting and placement.

4.1 Temporal-Ordering Information

Our procedure-placement algorithm uses a TRG to make decisions about which mapping conflicts should be avoided and which mapping conflicts can be allowed. In the following paragraphs, we define a TRG for any code-block granularity. We use a broad definition for two reasons.

First, our placement algorithm will use two kinds of TRGs: one for procedures; and one for *procedure chunks*, which are fixed-size pieces of procedures. A procedure is divided into chunks by starting at the beginning of the procedure, adding basic blocks to the current chunk in order of increasing addresses, and starting a new chunk whenever the size of the current chunk exceeds the fixed chunk size.

Second, we would ideally like to gather temporal-ordering information for code blocks that are a cache line in size. Under this granularity, there is an exact match between the temporal relationship (as defined below) of two code blocks and the number of conflict misses caused by them. Since a procedure may occupy many cache lines and not all of these cache lines may be executed on any particular invocation, the temporal-ordering information gathered at the granularity of procedures may not provide an exact count of the number of conflict misses. However, for practical purposes, a TRG for procedures, computed as if the procedure bodies were uniformly executed, still captures the essential information necessary for procedure placement. A TRG of procedure chunks simply gives us more precise information about conflict misses, since the procedure chunks are closer to idealized code blocks than procedures.

Given a trace of code block references, for any two code blocks P and Q, let $R(P, Q)$ be the number of times that two successive occurrences of P are interleaved with at least one reference to Q, or vice versa. We say that $R(P, Q)$ is the *temporal relationship* of P and Q because it measures the degree to which references to P and Q are temporally interleaved in the trace. We define the TRG for the trace to be a weighted graph, with a node for each code block, where the edge between P and Q has weight $R(P, Q)$.

Recall from Section 2 that a conflict miss for a code block P occurs whenever two nearest references to P are separated by a reference to another code block Q which maps to the same cache line(s) as P. This is the source of our definition of

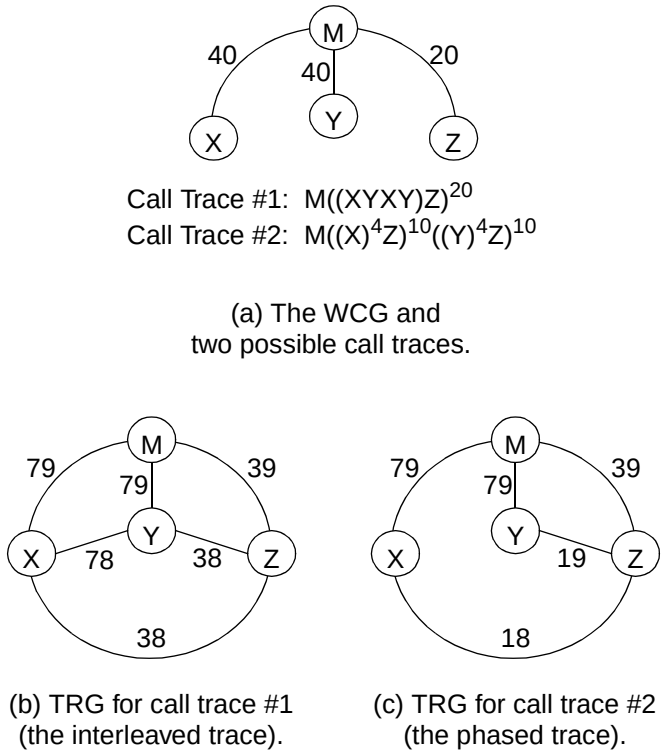


Fig. 3. Comparing the WCG with the TRGs obtained from the interleaved and phased call traces in Figure 1. For call trace 1, the execution of X and Y is interleaved, and corresponding TRG has an edge between X and Y. For call trace 2, the execution is phased (no interleaving), and the corresponding TRG does not have an edge between X and Y. Remember that the TRGs are built from *execution* traces where a reference to procedure P appears both when P is called and when control returns to P.

$R(P, Q)$. When the code blocks are all exactly the size of a cache line, then $R(P, Q)$ gives the exact number of conflict misses for P if Q is the only mapping conflict. As mentioned earlier, when code blocks are entire procedures, we may not get an exact count of conflict misses, since a procedure may occupy many cache lines and the set of these cache lines touched may vary between invocations. But $R(P, Q)$ still measures the degree to which execution of P and Q is interleaved, which we show in Section 4.5 is well correlated with the number of conflict misses P and Q will suffer if they have a mapping conflict. Not only is $R(P, Q)$ a better predictor of conflict misses than the number of calls between P and Q (which is the WCG edge weight), but the TRG is also more general than a WCG because it can contain edges connecting any pair of code blocks for which there is some interleaving during the program execution.

Figure 3 compares the WCG for the program in Figure 1 with the TRGs corresponding to the interleaved and phased call traces. These TRGs capture the differences in the temporal ordering. In particular, the edge between procedures X and Y expresses the degree to which their execution is interleaved.

```
Q  := emtpy;
QS := 0;        // total size of elements in Q

for each reference R in trace
    if R in Q
        k := position of R in Q;
        for i := k-1 downto 0
            weight(Q[i],R) += 1;
        QS -= sizeof(Q[k]);
        delete Q[k];
    endif

    add R to front of Q;
    QS += sizeof(R);

    // last(Q) gives the oldest entry in Q
    while QS-sizeof(last(Q)) >= 2*cache_size
        QS -= sizeof(last(Q));
        delete last(Q);
    endwhile
endfor
```

Fig. 4. Pseudocode for the trace of processing algorithm. The TRG is constructed by incrementally computing the edge weights stored as $weight(x, y)$.

4.1.1 *Trace-Processing Algorithm for Building a TRG.* Given a trace of references to code blocks, the following algorithm processes the trace one reference at a time to build the corresponding TRG. Notice that if the code blocks are procedures, the trace contains a reference to a procedure P both when P is called and when control returns to P.

We maintain an ordered set Q of recently referenced code block identifiers (e.g., procedure names). At each step, the code blocks appear in Q in the same order as their most recent occurrences in the trace. Let $Q[i]$ be the ith entry of Q, such that $Q[0]$ is the most recent entry. To process the next reference to code block b in the trace, we search Q for an occurrence of b. If Q does not contain b, we simply add b to the front of Q and move on to the next reference in the trace. If Q does contain b, say in position k, then we know that $Q[k-1], \ldots, Q[0]$ are code blocks that were referenced between the most recent previous reference to b and the current reference to b. Therefore, for any code block c in that range, we increment the weight of the TRG edge between b and c.

For each block b, Q has to store only the most recent occurrence of b (and thus the index k in the previous paragraph is unique). This is sufficient because, after a reference to b, it will be present in the cache, regardless of earlier references to it.

Furthermore, we can drop the oldest entry b in Q whenever the sum of the sizes of the code blocks in Q exceeds a certain limit. If a sufficiently large amount of (unique) code has been executed since the last reference to b, then b will most likely have been evicted from the cache. This eviction is due to the limited capacity of the cache, and not due to an avoidable conflict with a particular other code block. If T is the set of code block identifiers reached since the last reference to b and if $S(T)$ is the sum of the sizes of the code blocks referenced in T, exactly how big

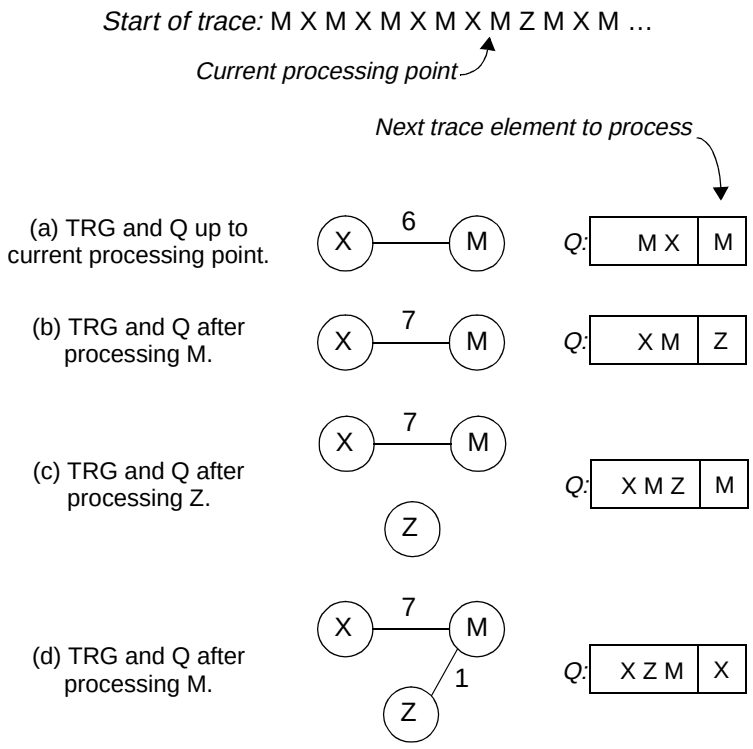


Fig. 5. The TRG build process using the phased trace (call trace #2) as an example. The processing of Q in (a) causes the edge weight $W(e_{M,X})$ to be incremented because X occurs between M and its previous occurrence in Q. The processing of Q in (b) does not add any new edges to the TRG because there is no previous occurrence of Z. The node Z and the edge $e_{M,Z}$ are added during the processing of Q in (c). The processing of Q in (d) would increment $e_{M,X}$ and add $e_{X,Z}$. The configuration of Q assumes that the total size of X, M, and Z is less than twice the size of the target instruction cache.

$S(T)$ needs to grow before b is evicted depends on the cache mapping of the code. Assuming that the code placement minimizes mapping conflicts between members of T, they will be mapped to mostly nonoverlapping addresses, and their cache footprint will be nearly equal to $S(T)$. Therefore, b becomes irrelevant when $S(T)$ is greater than the cache size. Empirically, we have found that a bound on Q of twice the cache size works quite well; the extra factor of 2 is a safeguard against discarding potentially valuable information.

The pseudocode in Figure 4 summarizes the trace-processing algorithm. Figure 5 shows several steps in the processing of our example phased trace.

4.1.2 *Practicality Issues.* As stated before, this algorithm can be applied to code blocks of any granularity. For the algorithm in the next section, we apply it to procedures to obtain a *procedure TRG*, and we simultaneously apply it to generate a *chunk TRG*, for the reasons stated earlier. For maximum accuracy, the chunk size could be set to the cache line size (32 bytes in our experiments). The smaller the chunks, the greater the number of nodes and edges in the chunk TRG.

We found that the achievable cache miss rate varies only slightly between different chunk sizes, and there is no benefit to using chunks smaller than 256 bytes for our benchmarks. Therefore we use this value as a trade-off between accuracy and efficiency.

To reduce overhead even further, we follow the suggestion of Hashemi et al. [1997] to select a set of popular procedures and focus only on these during the trace processing. In our experiments, we select the procedures accounting cumulatively for 99% of the total dynamic procedure calls. Thus, we avoid inflating the size of our data structures with information about infrequently executed procedures.

4.2 Procedure-Placement Algorithm

The structure of our algorithm is a greedy graph-merging process very similar to the PH algorithm. This is not a coincidence, since we intended to keep our algorithm as similar to PH as possible. That way, the difference in results can be more easily traced to the benefits of our more-detailed profile information and our more-informed cache-relative alignment. Thus, the outer loop of our algorithm repeatedly chooses the largest-weight edge in a procedure graph and combines the adjacent nodes. As in PH, once a set of procedures has been merged into a single node, their relative positions are fixed.

Instead of the simple "closest-is-best" heuristic of PH, our algorithm determines the exact cache mapping of the procedures within a node. In particular, for each cache line, we calculate which procedure chunks[4] are mapped to that cache line. We use these data to find the best cache-relative alignment for combining two nodes. Up to this point, each procedure is placed by determining its cache-relative alignment, i.e., by fixing its starting address modulo the cache size. In this phase, we are concerned only with the cache mapping, and therefore only this part of a procedure's address is relevant. Thus there is no linear ordering of procedures; this leaves us the most freedom for avoiding mapping conflicts. However, eventually we will have to find complete addresses for the procedures, either to produce an optimized executable or for the purpose of a complete memory system simulation.

In the following sections, we present the details of these aspects of our algorithm, as well as a complexity analysis.

4.2.1 *Order of Procedure Processing.* Our placement algorithm uses a working graph derived from the procedure TRG. The working graph structure and edge weights are copied from the procedure TRG. Each node of the working graph represents a list of pairs (p, o), where p is a procedure identifier and o is a relative offset expressed in cache lines. The list in each node initially contains a single pair whose procedure identifier comes from the corresponding procedure TRG node and whose offset is 0.

We repeatedly find the heaviest edge $e_{P,Q}$ in the working graph and combine the two nodes P and Q that are connected by this edge. At this point, we align nodes P and Q relative to each other by choosing the relative offset (in cache lines)

[4]Even though the calculation is based upon procedure chunks, our algorithm does not split any procedures during the placement process. Recall that we use a TRG of procedure chunks simply because it provides us with more precise information. We briefly address the effect of procedure splitting in Section 4.6.

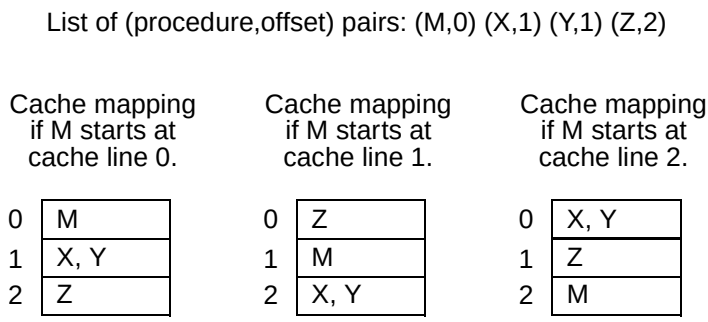


Fig. 6. An example of a node containing four procedures. Assume that each procedure is the same size as a cache line. For a three-line cache, we show all three possible cache mappings that correspond to these offsets. The mapping conflicts are identical for all cases, and thus we can choose any of these mappings when we combine this node with another node.

between P and Q that causes the least number of estimated conflicts. We explain the details of choosing the best alignment in the next section. From this point on, the relative positions of all the procedures in the combined node PQ remain fixed.

As in the PH algorithm, we update the working graph to adjust for the merging of P and Q into PQ. We replace any pair of edges $e_{P,R}$ and $e_{Q,R}$ to a common node R by a single edge $e_{PQ,R}$ and add the individual weights so that $W(e_{PQ,R}) = W(e_{P,R}) + W(e_{Q,R})$. We replace any single edge $e_{P,S}$ or $e_{Q,S}$ with $e_{PQ,S}$ and finally delete the edge $e_{P,Q}$. We continue merging nodes in this fashion until there are no more edges left.

Like the PH algorithm, our algorithm is "greedy" in the sense that, at each step, it aligns procedures to achieve the best placement given the information available at that point. It does not backtrack, i.e., a placement decision cannot be changed once it has been made. Our algorithm processes the procedures in decreasing order of importance. This makes it more likely that the most important conflicts can be avoided, since there are fewer placement constraints when fewer procedures have been aligned. Since the computational complexity of finding a globally optimal solution is exponential in the number of procedures placed, researchers in procedure placement typically employ some kind of greedy approach.

4.2.2 *Procedure Alignment.* As the merging process adds procedures to a node, it generates a cache-relative offset for each procedure in this node. These offsets define a cache-relative alignment for the merged procedures that is fixed for the rest of the placement process. The node as a whole however is not bound to any specific cache line or memory address. As Figure 6 shows, we can shift the node as a whole by any number of cache lines without changing the mapping conflicts within the node.

Given two nodes P and Q that we want to combine into a single node, we can choose a displacement d to shift Q before combining it with P. We want to choose this displacement to minimize the estimated number of cache conflicts. As we have seen, this merging process will not change the mapping conflicts within Q and therefore preserves the effect of all previous decisions we have made to produce Q. The problem of combining two nodes reduces to finding the best value for the

displacement d, i.e., the value which results in the minimum number of expected conflict misses.

For each value of d, we can easily compute the cache line location for all the procedure chunks, since we know the cache size and the procedure offsets. Thus for each cache line, we have a set C_P of procedure chunks from P, and a set C_Q of procedure chunks from Q that map to that cache line. If we add up the chunk-TRG edge weight $W(e_{i,j})$ for each pair $i \in C_P$, $j \in C_Q$, this gives us the estimated cost of the mapping conflicts for this cache line. This sum computed over all cache lines is the estimated cost of combining P and Q using displacement d. We try out all values $d \in \{0, \ldots, cachelines - 1\}$ and choose the value resulting in the minimum estimated cost.

There may be several displacements with the same minimum cost. As an added heuristic, we choose the displacement that results in the maximum number of empty cache lines and thus has the maximum amount of overlap between nodes. For identical cost, a placement with more overlap is better because it leaves more cache lines unoccupied (or lightly occupied) for future placement decisions.

Figure 7 shows the three steps that the algorithm performs on the TRG for the phased call trace from our earlier example in Figure 1. We can see that procedures X and Y end up on the same cache line, because there is no TRG edge between them and because the "maximum overlap" heuristic chooses this placement. This is desirable because it saves an empty cache line so that procedure Z can be placed in a conflict-free position.

4.2.3 *Generating a Complete Layout from Procedure Alignments.* When there are no more edges left in the graph, each procedure has been assigned a starting cache line. This is an address, expressed in cache lines, modulo the cache size. This means that the memory address of each procedure is only partially specified. Thus, we have to find a memory layout for the program that results in each procedure's starting address mapping to the correct cache line.

As long as we are concerned only with the first-level cache behavior of the program, we could arrange the procedures in any order and achieve the correct cache mapping simply by leaving an appropriate amount of empty space before each procedure. Suppose the first free cache line following the previous procedure is LE, and suppose that our algorithm has determined that the current procedure should start at cache line LS; then we need to leave a gap of LG cache lines, where

$$LG = \begin{cases} LS - LE & \text{if } (LE \leq LS) \\ LS - LE + cachelines & \text{otherwise.} \end{cases} \tag{5}$$

To ensure that the next procedure starts at a cache line boundary, we may have to insert a small additional gap. If the size of the previous procedure is S bytes and the cache line size is CLS bytes, then the required gap (in bytes) is

$$G' = \begin{cases} CLS - (S \bmod CLS) & \text{if } (S \bmod CLS \neq 0) \\ 0 & \text{otherwise.} \end{cases} \tag{6}$$

Clearly, the total amount of gap depends on the order in which we arrange the procedures. If we are only interested in the first-level cache, then this aspect is not relevant. If we are concerned about the total size of the resulting executable, then we can try to find an ordering of the procedures that minimizes the total gap

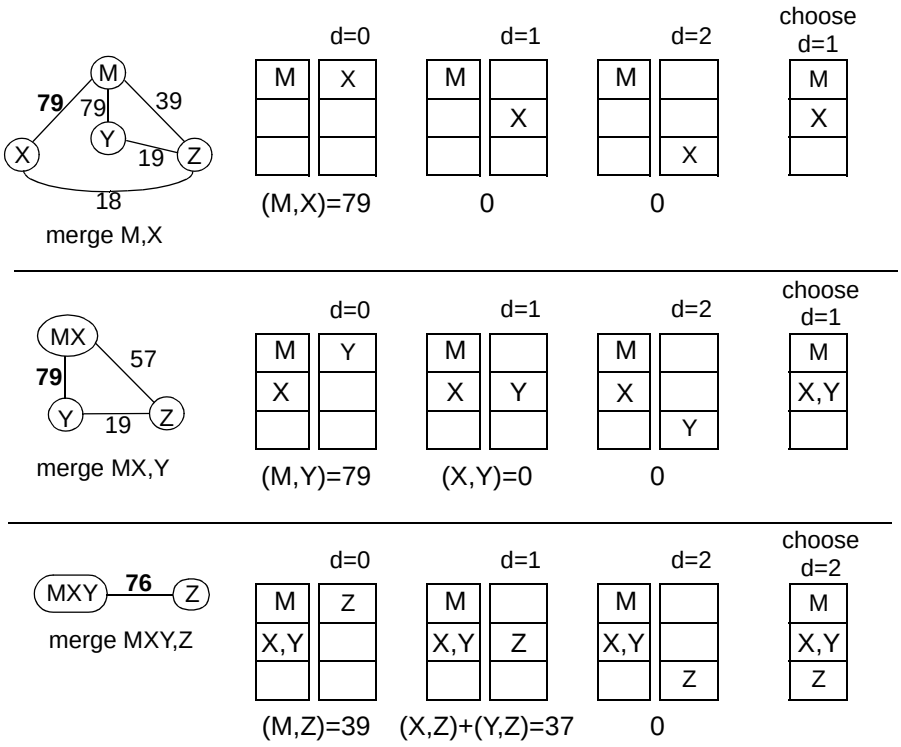


Fig. 7. Procedure alignment steps applied to the TRG for the phased call trace in Figure 1. On the left, we show the current state of the working graph; the heaviest edge in this graph decides which nodes are merged next. The procedure alignment requires computing cost estimates for all possible displacements (d). We choose the displacement which gives the minimum cost and maximum overlap. The combined node for this displacement is shown on the right.

amount. We can also fill gaps with *unpopular* procedures that have been excluded from the alignment process.

While the total gap amount has no impact on first-level cache misses, it does have an effect on higher levels of the memory hierarchy. In Section 6, where we include these aspects in our evaluation, we shall return to this topic.

4.2.4 *Simplification of the Graph-Merging Process.* When we look at the sizes of the two nodes being merged at each step, we notice that, in almost all cases, one of the nodes consists of a single procedure. Table III shows the distribution of node sizes. Based on this observation, we find that we can simplify our implementation slightly by explicitly restricting it to handling only the case of adding a single procedure to a node of multiple procedures. Instead of searching for the edge of maximal weight between any nodes in the working graph, we search the edges only from the multiprocedure node to individual procedures. We verified empirically that this does not change our results. In Section 6, we use this simplification to simultaneously optimize for cache alignment and spatial locality. It is an open research problem to explain why the merging process proceeds in this fashion.

Table III. Distribution of Node Sizes (in procedures) During the Graph-Merging Process. In almost all cases, we merge a large node with a small node consisting of a single procedure.

Benchmark	Size of the smaller node in procedures						
	1	2	3	4	5..9	10..15	>15
gcc	321	12	2	0	0	1	0
go	207	8	5	0	0	0	0
ghostscript	273	2	3	0	0	1	0
latex	128	2	3	4	0	1	0
perl	57	0	0	0	0	1	0
porky	343	1	0	0	0	0	0
vortex	190	0	0	0	0	0	0

4.2.5 *Complexity Analysis.* To analyze the complexity of our algorithm, Figure 8 provides a pseudocode summary of the descriptions in the previous sections, including the simplification introduced in Section 4.2.4. Let P be the number of popular procedures, and let C be the number of cache lines. The algorithm contains many nested loops; we will first explain their purpose and then estimate their iteration counts to arrive at a bound on the total number of steps.

The function `PlaceProcedures` begins by finding the edge of maximum weight in the working graph G, chooses one of its nodes as the compound node, and then processes G in loop (2). This loop iterates at most P times, because each iteration merges a procedure into the compound node, and this loop must terminate when all procedures have been merged (or when there are no more edges left). To find the best displacement for combining the two nodes, we call the function `FindBestDisplacement`. Inside this function, loop (4) iterates over all C possible displacements. For each value of d, loop (7) in function `ComputeCost` calculates the cost estimate over C cache lines.

For each cache line k, let R_k be the number of procedure chunks from node $N1$ (i.e., the compound node) on that cache line, and let S_k be the corresponding number from node NX (i.e., the single-procedure node). We add up $R_k \cdot S_k$ weights for the total cost estimate for cache line k. The largest value that R_k can have is achieved at the very end, when all procedures have been placed. At this point, all procedure chunks are distributed over C cache lines. Let SC be the chunk size in cache lines, and let SP be the average procedure size in cache lines. There are approximately $P \cdot \frac{SP}{SC}$ chunks, and each of them occupies SC cache lines; hence on the average, each cache line is occupied by $\left(P \cdot \frac{SP}{SC} \cdot SC\right) \div C = \frac{P \cdot SP}{C}$ chunks, and hence we use this value to approximate R_k. Similarly, we use $\frac{SP}{C}$ to approximate S_k . Since the average procedure size SP is generally less than the cache size C, we estimate S_k to be $O(1)$.

Thus, loop (7) takes at most

$$\sum_{k=0}^{C-1} R_k \cdot S_k \leq C \cdot \frac{P \cdot SP}{C} = P \cdot SP$$

steps. Loop (4) iterates C times and hence takes $O(P \cdot SP \cdot C)$ steps. Since loop (6) in function `BuildChunkSetArray` requires about SP steps and since loop (5) performs at most P iterations, loop (5) takes $O(P \cdot SP)$ steps. Thus each

```
      function PlaceProcedures(procedureTRG PTRG)
          derive working graph G from PTRG;
(1)       find e(N1,N2) := maximum weight edge in G;
(2)       while G has edges
(3)           find e(N1,NX) := maximum weight edge in G adjacent to N1;
              d := FindBestDisplacement(N1,NX);
              insert the single procedure p in NX into N1 with offset d;
              update G to reflect merging of NX into N1;
          endwhile
          LayoutProcedures(N1);

      function FindBestDisplacement(node N1, node NX)
          chunks1 := BuildChunkSetArray(C, N1);
          chunks2 := BuildChunkSetArray(C, NX);
          best_d := 0; best_cost := INFINITY;
(4)       foreach d in [0..C-1]
              cost := ComputeCost(C, d, chunks1, chunks2);
              if (cost < best_cost) then
                  best_cost := cost; best_d := d;
          endfor
          return best_d;

      function BuildChunkSetArray(int cachesize, node N)
          // builds an array of sets of chunks from N, one set for each cache line
          foreach i in [0..cachesize-1]
              chunks[i] := {};
(5)       foreach (proc,offs) in N
(6)           foreach j in [0..Size(proc)-1]
                  proc_chunk := j / chunk_size_in_cache_lines;
                  chunk_offs := (j + offs) MOD cachesize;
                  chunks[chunk_offs] := chunks[chunk_offs] UNION {proc_chunk};
              endfor
          return chunks;

      function ComputeCost(int cachesize, int d, chunkSetArray a1, chunkSetArray a2)
          cost := 0;
(7)       foreach k in [0..cachesize-1]
              foreach ch1 in a1[k]
                  foreach ch2 in a2[(k+d) MOD cachesize]
                      cost += weight from e(ch1,ch2) in CTRG;
          return cost;

      function LayoutProcedures(node N)
(8)       while N not empty
(9)           find first (proc,offs) pair p in N that minimizes LG;
              set starting address of p.proc;
              remove p from N;
          endwhile
```

Fig. 8. Pseudocode for our code-placement algorithm. PTRG and CTRG represent the procedure and chunk TRGs respectively. The function `Size` returns the size of its argument in cache lines. The function `PlaceProcedures` assumes that the PTRG is connected; if not, one would just make several calls to `PlaceProcedures`, one for each connected component.

call of `FindBestDisplacement` takes $O(P \cdot SP \cdot C) + O(P \cdot SP) = O(P \cdot SP \cdot C)$ steps. Finding the maximum edge in statement (3) takes $O(P)$ steps in the worst case, since any one node in the graph can have at most P edges. We thus multiply $O(P \cdot SP \cdot C)$ by the P iterations of the outermost loop (2) to arrive at a complexity of $O(P^2 \cdot SP \cdot C)$.

The cost of statement (1), which finds the initial maximum edge, is at most $O(P^2)$, since there can be at most P^2 edges in the working graph. To complete the placement process, we produce a procedure layout from the generated alignment data as shown in function `LayoutProcedures`. This requires $O(P^2)$ steps, since loop (8) iterates $O(P)$ times and statement (9) requires at most $O(P)$ steps. Thus the overall complexity for function `PlaceProcedures` is $O(P^2 \cdot SP \cdot C)$.

4.3 Results

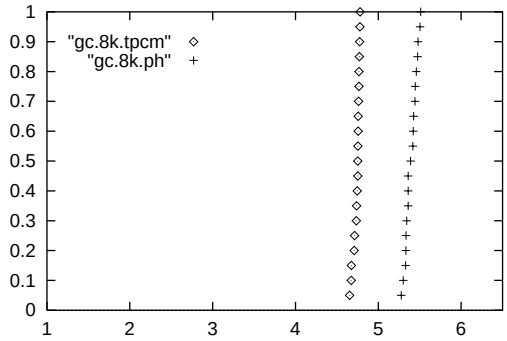
We evaluate our placement algorithm by comparing the instruction cache miss rates it achieves to those obtained by the PH algorithm. Following the methodology proposed by Blackwell [1998] as summarized in Section 3.2, we conduct many randomized experiments for each benchmark and each placement. Each experiment consists of generating a randomized perturbation of the TRG or WCG as applicable, and simulating the instruction cache references for an 8KB direct-mapped cache. Figures 9, 10, and 11 show the cumulative distribution of the resulting cache miss rates.

As an example of how to read one of these graphs, consider the results for *go* (Figure 9(b)) using PH. The lowest "+" with a miss rate of 4.0 has a vertical axis value of (approximately) 0.45. This means that 45% (or 9) of the PH-based placements done for *go* yielded a miss rate below 4.0 on the testing data set.

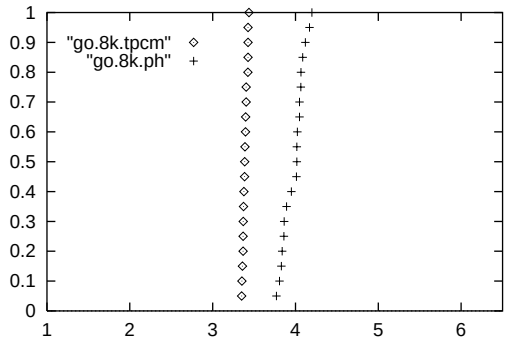
We see that our TPCM algorithm always gives better results than the PH algorithm. The cumulative distribution shows this by placing the line formed by the data points of TPCM clearly to the left of the line corresponding to the PH algorithm, i.e., at lower miss rates. The slope of the line indicates the variation between different code placements resulting from the randomized profile data. We see that the PH data points show more variation. This is because the PH algorithm does not have precise information on mapping conflicts and therefore sometimes causes costly mapping conflicts to occur. The variation is particularly pronounced for the *perl* benchmark; this is due to the fact that this benchmark has one procedure that is very frequently executed and is larger than the cache size of 8KB. We recall that the PH heuristic of adjacent placement becomes useless when the total of the procedures involved is greater than the cache size.

The distribution of the miss rates of the randomized experiments shows that the ranges of results produced by the different algorithms do not overlap. Thus it is valid to summarize the miss rates by computing their average. In Figure 12, we show the average miss rates (for the same 20 randomized experiments as in Figures 9–11) for cache sizes of 4KB, 8KB, and 16KB. The graphs also show the average miss rate for completely random procedure ordering. This number represents the miss rate we can expect from the default procedure ordering produced by a compiler that does not perform code placement.

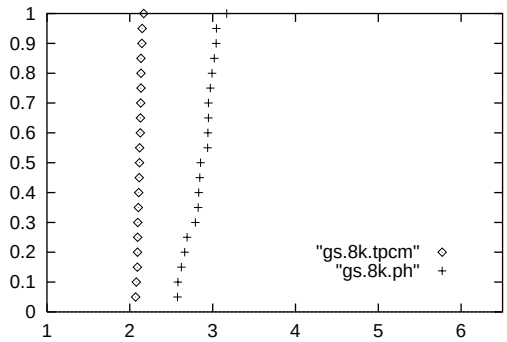
For a 4KB cache, TPCM improves the miss rate by 6% to 29% over PH, with a geometric mean of 13%. For an 8KB cache, TPCM improves the miss rate by



(a) Results for *gcc*.

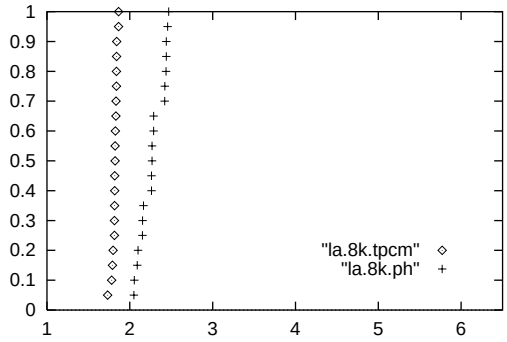


(b) Results for *go*.

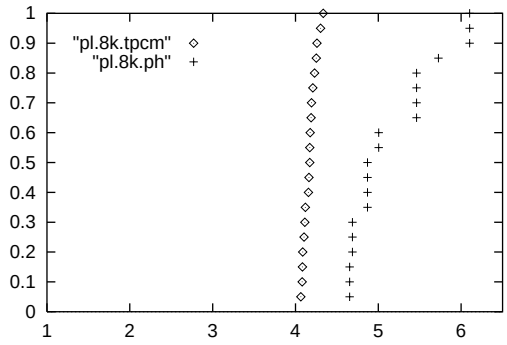


(c) Results for *ghostscript*.

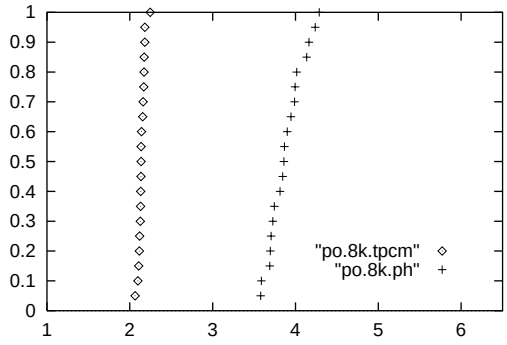
Fig. 9. Comparison of the TPCM and PH algorithms. We simulated an 8KB direct-mapped instruction cache. For each algorithm, the graph shows the cumulative distribution of 20 randomized experiments; all experiments used a scaling factor s of 0.1. The cache miss rate (in percent) is on the horizontal axis; the fraction of the cumulative distribution is shown on the vertical axis.



(d) Results for *latex*.

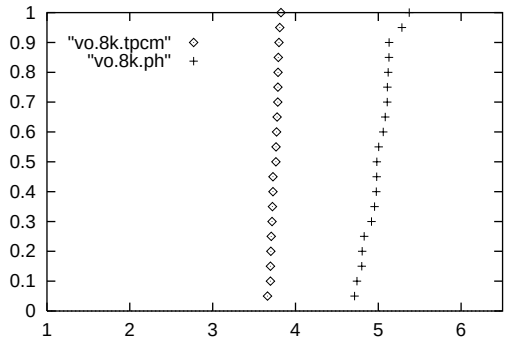


(e) Results for *perl*.



(f) Results for *porky*.

Fig. 10. Comparison of the TPCM and PH algorithms (continued). We simulated an 8KB direct-mapped instruction cache. For each algorithm, the graph shows the cumulative distribution of 20 randomized experiments; all experiments used a scaling factor s of 0.1. The cache miss rate (in percent) is on the horizontal axis; the fraction of the cumulative distribution is shown on the vertical axis.



(g) Results for *vortex*.

Fig. 11. Comparison of the TPCM and PH algorithms (continued). We simulated an 8KB direct-mapped instruction cache. For each algorithm, the graph shows the cumulative distribution of 20 randomized experiments; all experiments used a scaling factor s of 0.1. The cache miss rate (in percent) is on the horizontal axis; the fraction of the cumulative distribution is shown on the vertical axis.

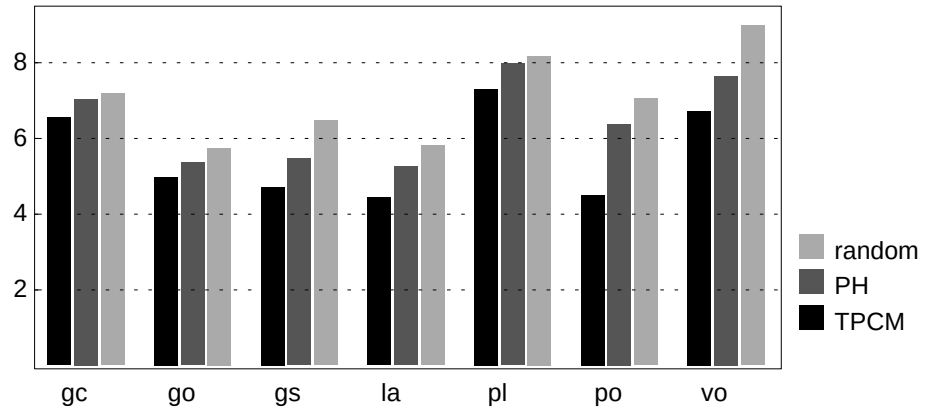
12% to 44% over PH, with a geometric mean of 23%. For the 16KB cache, the improvement ranges from 18% to 44%, with a geometric mean of 31%. In fact, code placement using TPCM allows us to achieve the same average miss rate on an 8KB cache as we would get on a 16KB cache without any code placement.

The improvement of TPCM over PH is greater for larger caches, because there are more opportunities for avoiding mapping conflicts. TPCM is better at taking advantage of these opportunities. The benefit decreases for smaller caches, because their behavior is dominated by capacity misses, which are not affected by code placement. On the other hand, the miss rates in absolute terms get much smaller for larger caches and thus have less impact on overall application performance. However, the benchmarks used in this evaluation are relatively small compared to applications such as database, CAD, or office productivity software.

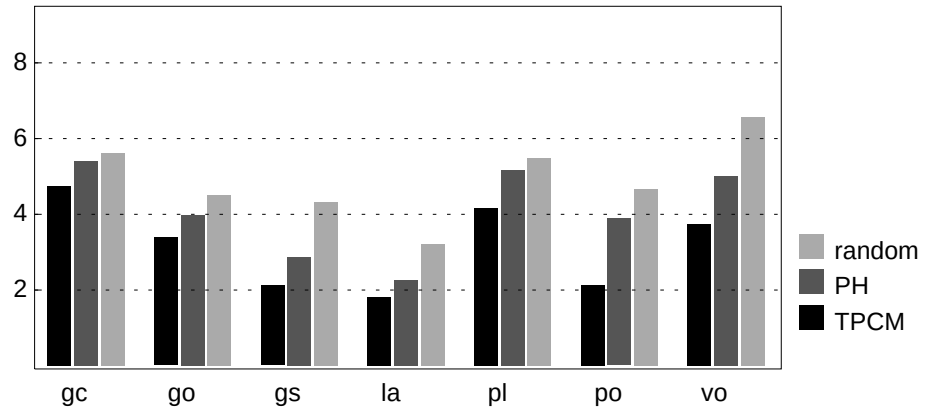
4.4 Application to Set-Associative Caches

Our code-placement technique is based on the assumption that the instruction cache is direct-mapped. This is reflected in the modeling of the cache mapping and the calculation of the conflict estimates. However, the resulting code placement still makes sense for associative caches. The code-placement algorithm attempts to remove as many mapping conflicts as possible, and the associative cache has the ability to remove some additional mapping conflicts.

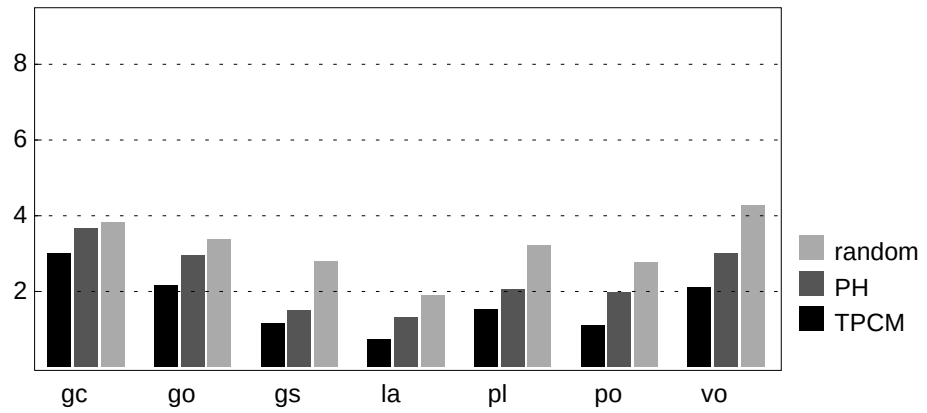
If our algorithm is successful at removing all major mapping conflicts for a particular program, then each cache line contains at most one important code block, and thus an N-line set in an N-way associative cache contains at most N important code blocks. Thus, neither cache will experience a significant number of conflict misses, and there is no benefit from the associative cache. If there is a large amount of frequently executed code with a sufficiently high degree of interleaving in its execution, then the code-placement algorithm will not be able to avoid all conflict misses, and an associative cache can help. In the presence of unavoidable conflicts, our algorithm is still able to see the relative costs of different placements. The re-



(a) Results for a 4KB cache.



(b) Results for an 8KB cache.



(c) Results for a 16KB cache.

Fig. 12. Average cache miss rates for direct-mapped caches of (a) 4KB, (b) 8KB, and (c) 16KB. Each average is computed over 20 randomized experiments. The vertical axis shows the miss rates in percent.

Table IV. Average Miss Rates for Direct-Mapped and Set-Associative Caches. All caches are 8KB in size; the associative caches use the LRU replacement policy.

Benchmark	Placement	Direct-mapped	Associative	
			two-way	four-way
gcc	TPCM	4.70	4.71	
	PH	5.40	4.92	
	Random	5.61	5.01	4.83
go	TPCM	3.37	3.44	
	PH	3.98	3.63	
	Random	4.51	3.77	3.54
ghostscript	TPCM	2.15	2.06	
	PH	2.87	2.27	
	Random	4.33	3.17	2.38
latex	TPCM	1.88	1.72	
	PH	2.27	1.83	
	Random	3.21	2.28	1.72
perl	TPCM	4.08	3.18	
	PH	5.17	3.41	
	Random	5.48	3.57	2.44
porky	TPCM	2.15	2.04	
	PH	3.89	2.84	
	Random	4.67	3.32	2.50
vortex	TPCM	3.78	3.90	
	PH	5.01	4.48	
	Random	6.57	5.15	4.46

sulting placements tend to distribute conflicts evenly across the cache, which makes it more likely that an associative cache can avoid some of the conflicts.

A code-placement algorithm that is less effective than TPCM tends to leave more mapping conflicts. If the remaining conflicts can be removed by the associative cache, the less efficient placement algorithm combined with the associative cache is just as effective as TPCM combined with the associative cache. In general, the associative cache helps to reduce the difference between the TPCM placement and the PH placement.

Table IV compares the average miss rates for these two placements for direct-mapped and two-way associative caches. It also shows the average miss rates for random procedure placement for direct-mapped, two-way, and four-way associative caches. These results support our discussion in the previous paragraphs about the interaction between code placement and cache associativity. In all cases, the difference between the PH and TPCM placements is less pronounced for the two-way associative cache than for the direct-mapped cache. For some benchmarks (*gcc*, *go*, *ghostscript*, *latex*) there is very little difference between PH and TPCM for the two-way associative cache. This indicates that the number of mapping conflicts left by the PH placement is small enough so that the associative cache can remove most of them. We also notice that, for these benchmarks, the TPCM placement shows little difference between the direct-mapped and the two-way associative cache; this means that this placement succeeds in avoiding almost all mapping conflicts.

For the *perl* benchmark, there is a significant difference between the direct-mapped and two-way associative caches, for both placement algorithms. This benchmark is particularly interesting because the four-way associative cache results

in a much lower miss rate for the random placement. This means that both the PH and TPCM placements are unable to avoid some significant mapping conflicts, while the greater associativity is much more helpful in avoiding these conflicts. We believe that this is caused by this benchmark having some very large procedures that are larger than the cache and are conflicting with themselves. Procedure placement cannot avoid these conflicts, since it can only move procedures as a whole. This analysis is supported by our results for hot-cold splitting in Section 4.6, where we remove infrequently executed parts of procedures and compact the frequently used code. This placement is particularly effective for the *perl* benchmark, because a large part of its conflict misses are caused by a few large procedures.

Finally, the *porky* and *vortex* benchmarks have a sufficiently large amount of code and interleaving of execution so that even with the two-way associative cache, the TPCM placement is still significantly better than the PH placement.

In three cases (*gcc*, *go*, and *vortex*), we notice that the miss rates for the TPCM placement are slightly higher for the two-way associative cache than for the direct-mapped cache. This seems quite counterintuitive, but it can occur when the fixed mapping of code blocks to cache lines is more effective than the LRU replacement policy of the two-way associative cache.

4.5 Comparison of WCG and TRG

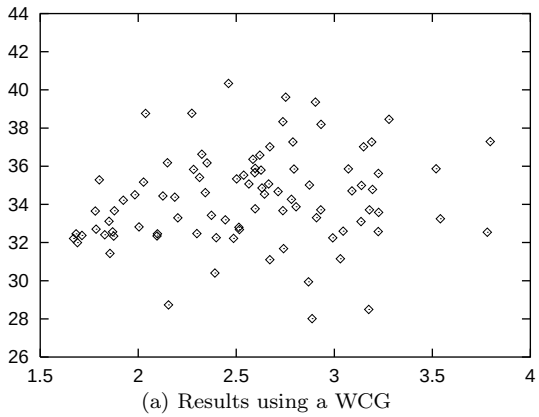
Given a program layout and a cache configuration, we can identify all the mapping conflicts for each cache line. We can then use either a TRG or a WCG to assign a predicted cost to each pair of code blocks that exhibit a mapping conflict. The sum of all these costs should be a predictor of cache misses.

To show that a TRG is a more accurate predictor, we perform the following experiment. We run our procedure-placement algorithm (from Section 4.2) and then generate 90 different procedure placements by starting with the output of the algorithm and moving between 5 and 70 procedures to a random cache line. For each randomized procedure placement, we compute the total conflict cost as described above, using both a WCG and a TRG. We also simulate the cache behavior using the procedure placement to obtain the actual instruction cache miss rate. When we plot the resulting data points as an X-Y plot, better correlation between predicted and actual miss rates causes the data points to be closer to a diagonal line. Note that we do not expect the predicted cost to match the actual number of cache misses; it is sufficient for it to be approximately linear in the actual miss rate. Figure 13 shows these graphs for the *latex* benchmark, and they confirm that the TRG-based conflict estimate is a more accurate predictor of actual cache misses. The other benchmarks yield similar graphs.

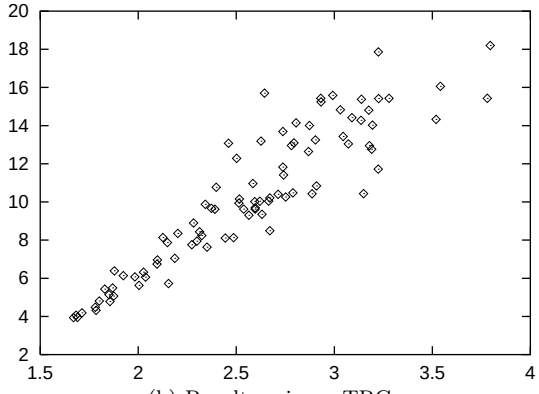
4.6 Impact of Procedure Splitting

While our work focuses on techniques that involve the placement of entire procedures, there are other code-placement techniques that are orthogonal to this approach and can improve the benefits of whole-procedure code placement. These techniques involve rearranging the code inside procedures, with the goal of allowing the instruction fetch unit to work more efficiently.

One simple and effective example is hot-cold splitting [Cohn and Lowney 1996; Pettis and Hansen 1990], also known as "fluff removal." The "cold" part (or "fluff")



(a) Results using a WCG



(b) Results using a TRG

Fig. 13. Estimated cost vs. actual cache miss rate for the *latex* benchmark. Each point in each graph corresponds to one of 90 randomized procedure placements. The vertical axis shows the estimated conflict cost, which is computed by adding up edge weights for conflicting procedures. The horizontal axis shows the actual instruction cache miss rate for the same trace that was used to generate the profile data, for an 8KB direct-mapped cache. In the graph for the TRG, the points are a better approximation to a straight line; hence the TRG-based cost estimate is a more accurate predictor of actual conflicts.

of a procedure consists of those basic blocks whose execution frequency is below some threshold, based on profile data. Hot-cold splitting moves the cold parts of a procedure out of line and thus compacts the hot parts of the procedure. The benefit is the increased density of executed code, which improves cache line and page utilization. One possibility is to move the cold parts to the end of the procedure. It is even better, if slightly harder to implement, to place the cold parts from all procedures in some contiguous area of the executable. This reduces the effective footprint of the procedures, which makes it easier for a procedure-placement algorithm to avoid conflicts.

To evaluate the impact of hot-cold splitting on the effectiveness of procedure placement, we conducted the following experiment. We modified our code placement and memory system simulation to remove from each procedure all basic blocks

Table V. L1 Cache Miss Rates for Procedure Placement with and without Hot-Cold Splitting. To avoid problems with profile-time fluff code vs. run-time fluff code, we used the same trace for profiling and cache simulation. Therefore, the improvement of hot-cold splitting may be optimistic.

	PH			TPCM		
Benchmark	without	with	change	without	with	change
gcc	5.4%	4.6%	15%	4.7%	3.9%	18%
go	1.9%	1.8%	2%	1.5%	1.4%	3%
ghostscript	2.3%	1.4%	38%	1.5%	1.1%	26%
latex	2.2%	1.9%	16%	1.7%	1.2%	27%
perl	4.7%	1.4%	70%	2.9%	0.4%	86%
porky	4.3%	4.0%	7%	2.3%	2.0%	13%
vortex	4.9%	3.3%	32%	3.7%	2.3%	38%

that are not touched during the testing trace. This reduces the procedure sizes and changes the way procedures are divided into chunks. We can then generate a new TRG and compute a procedure placement for the compacted procedures. The memory system simulator that we use to compute the instruction cache miss rates also takes into account the removal of fluff code. These modifications allow us to evaluate the effect of hot-cold splitting with a procedure-placement algorithm. We conducted this experiment with both the PH and TPCM algorithms. As the results in Table V show, both placement techniques benefit from hot-cold splitting approximately to the same degree. The considerable improvement achieved by this simple technique shows that it is definitely worthwhile to combine hot-cold splitting with procedure-placement techniques.

5. PROCEDURE PLACEMENT FOR MULTILEVEL CACHES

In the previous section, we presented a code-placement technique that optimizes cache-mapping conflicts in a single instruction cache. It achieves a substantial improvement in miss rates over the PH algorithm. But most of today's high-performance computer systems use multiple levels of cache in their memory hierarchy. The reason for the growing number of cache levels is the same as the reason for the existence of a memory hierarchy: ideally we would like to have a very large memory with a very fast access speed. However, memory implementations that are fast enough for the processor are very expensive and cannot be made very large. To bridge the gap between the processor speed and the speed of a large memory, we need some intermediate levels at various points in the trade-off between access speed, size, and cost.

Clearly we would like to achieve a code placement that makes all levels of cache work efficiently. A shortcoming of the placement technique we described in the previous section is that by targeting only at a single level of cache, we have left the other-level cache mappings to chance. Thus, there is an opportunity for improving this aspect of the code placement. In this section, we show how to extend our TPCM algorithm to simultaneously optimize code placement in multiple levels of the cache hierarchy. Our results show that we can reduce cache misses in all cache levels while retaining virtually all of the first-level cache performance.

5.1 Impact of L1 Placement on the L2 Cache

Since the PH algorithm does not involve the cache size in its calculations, the layout it produces is suitable for any cache size. Thus, in a multi-level cache, all caches will

simultaneously benefit from the PH placement. This is not the case for our TPCM placement technique, or for any other code-placement technique that models the cache mapping for a particular cache size.

In TPCM, we constrain the procedure addresses modulo the L1 cache size. The remaining part of each procedure's address is determined by the linear ordering of the procedures and the necessary gaps between them. The linear ordering is not designed to avoid L2 mapping conflicts, and thus it is left to chance which conflicts occur. We still manage to avoid some L2 conflicts, because we "pack" the popular procedures into a contiguous sequence, whereas a completely random procedure placement spreads them across the entire text segment. For example, if the total length of this sequence is twice the L2 cache size, then each popular procedure can suffer a mapping conflict with at most one other popular procedure. Our results in Section 5.3 confirm this analysis. The L2 cache miss rates of our single-cache TPCM placement are better than those for random procedure ordering, but worse than those for the PH technique.

5.2 Improved Algorithm for Multilevel Caches

A natural approach to extending the scope of our code placement to multilevel caches is to model the mapping conflicts in all caches and to find a placement that simultaneously optimizes the estimated costs in all caches. In our discussion, we focus on two levels of cache, but the technique is completely general and can be extended to a cache hierarchy of more than two levels in a straightforward fashion.

If we are modeling mapping conflicts in both the L1 and L2 cache, the cost metric we want to minimize now has two components: the cost of L1 and L2 mapping conflicts. When we align a new procedure, we choose an L1 displacement and an L2 displacement that align the new procedure relative to the procedures that have already been placed. Let C_1 and C_2, respectively, be the size of the L1 and L2 caches, in cache lines.[5] As Figure 14 shows, any L2 displacement d_{L2} corresponds to a unique L1 displacement $d_{L1} = d_{L2} \bmod C_1$. Conversely, any L1 displacement d_{L1} corresponds to several L2 displacements $d_{L2} = d_{L1} + k \cdot C_1$, for $k = 0, \ldots, (C_2/C_1) - 1$.

5.2.1 *Finding a Good Choice.* In general, it is not always possible to achieve a simultaneous minimum for the two components. Because of the smaller size of the L1 cache, L1 conflicts are much more frequent, and harder to avoid, than L2 conflicts. Therefore, we should give priority to the L1 cache and first choose the L1 displacement that minimizes the estimated cost of L1 conflicts, using the same search algorithm described in Section 4.2. Any displacement $d_{L2} = d_{L1} + k \cdot C_1$ is equivalent to d_{L1} as far as the L1 cache mapping is concerned. We compute the L2 costs for each displacement $d_{L1}, d_{L1} + C_1, d_{L1} + 2C_1, \ldots, d_{L1} + ((C_2/C_1) - 1)\, C_1$ and choose the displacement with the minimum cost.

To allow ourselves more freedom in choosing the L2 displacement, we select all L1 displacements that have a cost that is close, but not necessarily identical, to the minimum-cost L1 displacement. An L1 displacement is included if it is less

[5]Our algorithm assumes that the cache line size is constant across all levels of the memory hierarchy. Further research is required to understand how to handle a hierarchy with multiple different line sizes.

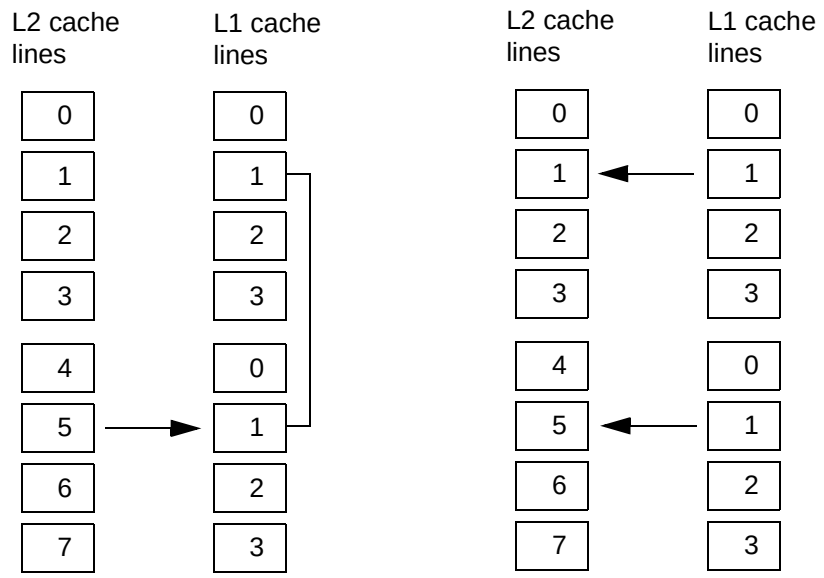


Fig. 14. Mapping of L1 and L2 cache line addresses. To provide a simple example, this figure shows an L1 cache of 4 lines and an L2 cache of 8 lines. An L2 cache line address of 5 corresponds to the unique L1 cache line address 1. But an L1 cache line address of 1 can result from an L2 cache line address of either 1 or 5. This means that when we choose an L1 displacement first, we are still left with a choice of several L2 displacements.

than or equal to $(1 + \alpha) \cdot cost_{min}$, where α is a small number, typically less than 0.1. We then try all equivalent L2 displacements (as defined above) for each of these L1 displacements. We found that this results in better L2 miss rates without noticeably affecting L1 miss rates. In Section 5.3, we show experimental results for a value of $\alpha = 0.05$ and compare this to using only a single minimum-cost L1 displacement. The former technique works significantly better, but we found that the results are not particularly sensitive to the exact value of α. When we increased α beyond 0.1, we noticed a gradual increase in L1 miss rates without a noticeable reduction in L2 miss rates.

5.2.2 *Using the Correct TRG.* It is important to remember that our trace-processing algorithm that generates the TRG is given the cache size as a parameter and ignores any interaction between references that are separated by other references touching more than two cache sizes worth of code. If we use a TRG that was generated for a small L1 cache to place procedures in a large L2 cache, the following may happen. There are some procedures whose execution is interleaved to a considerable degree, but they are separated by enough code so that they are never present in the L1 cache at the same time. The TRG does not contain any edges connecting these procedures, and therefore the code-placement algorithm mistakenly assumes that there is no cost associated with overlapping these procedures in the L2 cache. Thus, we can compute only meaningful L2 costs using a TRG which was generated for the L2 cache size. Since the L2-based TRG contains a superset of the information in the L1-based TRG, one might initially think that we could

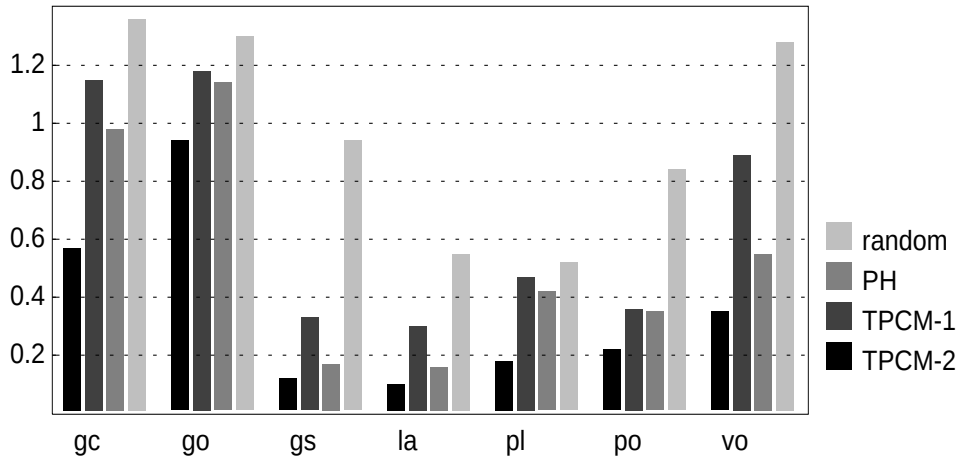


Fig. 15. L2 cache miss rates (in percent) for a 64KB L2 cache and an 8KB L1 cache. The miss rate shown on the vertical axis is the ratio of L2 misses to total instruction references.

use this same TRG for computing both L1 and L2 costs. In our experiments, we found that we get slightly better results for the L1 cache if we use the L1-based TRG for the L1 code placement, and the L2-based TRG for the L2 code placement. Thus more information is not necessarily better. The longer-term L2-based TRG can sometimes distort the trade-off between different placements based on a conflict which cannot be avoided (due to capacity concerns) in the smaller L1 cache.

5.3 Results

We evaluate our combined L1 and L2 placement algorithm (TPCM-2) by comparing it to the PH algorithm as well as a random procedure ordering. Additionally, we also show the L2 cache miss rates resulting from our L1-only placement from Section 4 (TPCM-1). Figure 15 shows the average L2 cache miss rates for each placement and for each benchmark. It is not necessary to show the L1 miss rates because they are the same as in Section 4.3. For the PH and random placements, there is only a single code placement regardless of the number of cache levels. For our combined L1 and L2 placement, the L1 cache miss rates remain unchanged by the extensions described in this section.

As expected, our L1-only placement performs worse than the PH placement, because it leaves L2 mapping conflicts to chance. But our combined L1/L2 placement achieves significantly better L2 (and L1) miss rates than the PH placement.

In the previous section, we claimed that we can improve the procedure placement by searching among L2 alignments corresponding to any L1 alignment within 5% of the minimum-cost L1 alignment. Table VI presents the results for our two-level placement approach with and without this improvement. The data support the claims in the previous section.

The numbers we present are L2 miss rates calculated as the ratio of L2 misses to total instruction references. An alternative would be to use the ratio of L2 misses to L2 references. But this calculation may be misleading because L2 references

Table VI. L2 Cache Miss Rates (in percent) for a 64KB L2 Cache and an 8KB L1 Cache under Two Variants of TPCM-2. The miss rates shown are the ratio of L2 misses to total instruction references. The "choose best" scheme is a version of TPCM-2 where the algorithm only chooses among L2 displacements corresponding to a single minimal-cost L1 displacement. The "choose any within 5% of best" scheme is a version of TPCM-2 where the algorithm chooses among the L2 displacements corresponding to any L1 displacement that is within 5% of the minimal-cost L1 displacement.

Placement	gcc	go	ghost-view	latex	perl	porky	vortex	G. mean
choose best	0.63	1.00	0.18	0.12	0.20	0.28	0.37	0.31
choose any within 5% of best	0.57	0.94	0.12	0.10	0.18	0.24	0.35	0.26

are equal to L1 misses, and hence a smaller number of L1 misses would cause the same absolute number of L2 misses look like a worse L2 miss ratio. While the L2 miss rates in our results are small numbers around 1%, they are still important to program performance because the penalty for an L2 cache miss is usually many tens of cycles. For example, on the Digital Alphaserver 4100 5/400 using an Alpha 21164 processor at 400MHz, the penalty of an L1 cache miss is 5 to 9 cycles [Kawaf et al. 1996], but the penalty of an L2 cache miss is more than 45 cycles [Cvetanovic and Donaldson 1996].

6. OPTIMIZING FOR SPATIAL LOCALITY

In the previous two sections, we focused on the cache lines occupied by a program's procedures, and we assumed that any placement of the procedures in a sequence (including gaps between procedures where necessary) that achieves the determined cache mapping was equally desirable. However, the procedure sequence affects the spatial locality of instruction memory references in a way that impacts the performance of a paged virtual memory system. Depending on the procedure sequence, the number of distinct pages occupied by a set of procedures may vary considerably.

One way of quantifying this effect is by measuring the working set size of the program, i.e., the average number of pages touched during an interval of a certain length. We recall from Section 2.2 that during intervals for which the working set size is less than the size of the TLB, the entire working set of the program can be accessed without TLB misses. Therefore, if our code placement increases the working set size of the program, it will increase the number of TLB misses and incur a run-time penalty. A special case of the working set size is the total number of pages touched over the execution of the program. If this number is increased, the program will have to page in more pages and will attempt to occupy more physical memory.

Because of these effects, a code-placement technique that encompasses all relevant levels of the memory hierarchy has to optimize a program's working set size as well as its cache-mapping conflicts. In this section, we show how to integrate the placement algorithm of the previous sections with a placement algorithm that optimizes for spatial locality, and hence the working set size.

6.1 Combining Optimizations for Spatial Locality and Cache Mapping

The "closest-is-best" heuristic used by the Pettis and Hansen algorithm [1990] is not only useful for avoiding cache-mapping conflicts, but is also intuitively appealing for placing related procedures on the same page. Suppose we have a model of the temporal relationship between procedures (such as a TRG). If we place the most strongly related procedures as close to each other as possible, we expect that this tends to minimize the number of pages occupied by a set of related procedures. This is also the basic approach of Hatfield and Gerald [1971] and Ferrari [1975] (see Section 7.5 for a discussion of this work). Thus we find that the simple procedure-placement technique proposed by Pettis and Hansen [1990] simultaneously addresses the cache mapping and the spatial locality.

Our procedure-placement algorithm from Sections 4 and 5 specifies the cache-line index bits for each procedure address, but leaves the rest of the address unspecified. One possibility is to use a PH-like algorithm to compute a procedure sequence, which is a layout of procedures in which the starting address of the first is set arbitrarily to zero. This sequence contains the gaps necessary to achieve the cache alignment determined earlier. We have found that this simple approach requires a large amount of gap space between procedures, and hence results in an inefficient packing of procedures into pages. The problem is a lack of integration between the cache-alignment phase and procedure-sequencing phase. The cache-alignment phase overly constrains the sequencing phase, without any knowledge of how its decisions will affect the ability of the later sequencing phase to pack procedures into pages efficiently.

An obvious solution is to integrate the two processes so that we simultaneously find a good cache mapping (*cache alignment*) and a good sequence (*procedure sequencing*).

6.1.1 *Constructing a Completely Specified Program Layout.* We describe an algorithm that gradually constructs a program layout that not only specifies the cache-relative alignment for each procedure, but also explicitly places the procedures in a sequence chosen to optimize spatial locality.

Recall from Sections 4 and 5 that we are inserting individual procedures into a growing set of procedures. For any procedure in this set, the cache-relative alignment decision has already been made and remains fixed from that point on. We keep this approach, but in addition to deciding on the cache-relative alignment of procedures, we also decide where to insert the procedure into a procedure sequence. This sequence may contain gaps as necessary to achieve the desired cache alignment. We may insert procedures into gaps left by earlier placement decisions if this is desirable based on the cache mapping and spatial locality. When we insert a procedure into the sequence, we completely fix its address relative to all other procedures in the sequence.

6.1.2 *Metric for Spatial Locality.* To make placement decisions that optimize for spatial locality, we introduce the following metric. Let $P_1, \ldots, P_k$ be the procedure sequence $\mathcal{P}$, and let d_j be the displacement from the start of the first procedure P_1 to the start of P_j. Let $W_{TRG}(X, Y)$ be the weight of the TRG edge between procedures X and Y. The metric for inserting a new procedure Q into sequence $\mathcal{P}$

at displacement c is

$$M(\mathcal{P}, Q, c) = \sum_{j=1}^{k} (|c - d_j| \cdot W_{TRG}(P_j, Q)) \tag{7}$$

A placement that minimizes this metric tends to place strongly related procedures as close to each other as possible to minimize the distance $|c - d_j|$. Since strongly related procedures appear together in many working sets, placing them close together in a small set of pages means that these working sets occupy fewer pages, compared to a placement where strongly related procedures are placed far apart and hence scattered across many pages.

6.1.3 *Extending Our Placement Algorithm to Optimize Spatial Locality.* Assume that we have already processed k procedures, resulting in a partial program layout that places these procedures in a sequence $\mathcal{P}$. As described in Section 4, we choose the next procedure to process by finding the maximum-weight edge in the working graph from the combined node of k procedures to a procedure outside this node. Let Q be the next procedure, and let C_1 and C_2, respectively, be the number of cache lines in the L1 and L2 caches.

First, we compute the L1 cache conflict estimates between Q and already placed procedures for all displacements of Q between 0 and $C_1 - 1$. We consider the set D_1 of all displacements with a L1 cache cost estimate within 5% of the minimum value, as discussed in Section 5.2.1.

For each displacement $d_1 \in D_1$, any displacement $d_2 = d_1 + i \cdot C_1$, where $0 \leq i < C_2/C_1$, results in the same L1 cache mapping. We compute the L2 cache conflict estimates for all d_2 derived in this fashion and select the set D_2 containing those displacements with a L2 cache cost estimate within 5% of the minimum value.

For any $d_2 \in D_2$, any displacement $d = d_2 + j \cdot C_2$ results in the same L1 and L2 cache mappings. For each such value of d that does not place Q more than C_2 cache lines outside of the sequence of previously placed procedures and does not cause overlap with a previously placed procedure, we compute the spatial locality metric $M(\mathcal{P}, Q, d)$. We choose the value of d that minimizes this metric.

The 5% bound used in generating D_1 and D_2 allows the algorithm to try several placements with conflict cost estimates near the minimum. If we consider only the single best cache-relative alignment, we may arbitrarily reject placements whose cache cost estimate is only slightly above the minimum. By choosing among several good L1 and L2 cache-relative displacements, we improve our chances of finding a placement with good spatial locality. If we increase the bound substantially beyond 5%, we run the risk of accepting placements that have a noticeably poor cache mapping.

6.1.4 *Complexity Analysis.* Figure 16 shows the pseudocode for the functions `FindBestDisplacement` and `LayoutProcedures` that replace the corresponding functions in Figure 8. These functions together with the other functions from Figure 8 comprise the placement algorithm that we refer to as TPCM-2S.

Let P be the number of procedures. Computing the spatial locality metric at statement (4) takes at most $O(P)$ steps. The loop (3) iterates $O(P)$ times, because the size of the procedure sequence in $N1$ is unlikely to exceed $P \cdot C_2$, and the

```
      function FindBestDisplacement(node N1, node NX)
          // build chunk set arrays for each node and cache combination
          chunks_L1_N1 := BuildChunkSetArray(C1, N1);
          chunks_L1_NX := BuildChunkSetArray(C1, NX);
          chunks_L2_N1 := BuildChunkSetArray(C2, N1);
          chunks_L2_NX := BuildChunkSetArray(C2, NX);

          // calculate L1 placement costs
          best_L1_cost := INFINITY;
          foreach d1 in [0..C1-1]
              cost_L1[d1] :=
                  ComputeCost(C1, d1, chunks_L1_N1, chunks_L1_NX);
              if (cost_L1[d1] < best_L1_cost) then
                  best_L1_cost := cost_L1[d1];
          endfor

          // consider only the top 5% of the L1 displacements
          best_d = 0; best_metric := INFINITY;
(1)       foreach d1 in [0 .. C1-1]
              if (cost_L1[d1]/best_L1_cost <= 1.05) then
                  // calculate L2 placement costs for this L1 displacement
                  best_L2_cost := INFINITY;
                  foreach i := 0 .. (C2/C1)-1
                      d2 := d1 + i*C1;
                      cost_L2[d2] :=
                          ComputeCost(C2, d2, chunks_L2_N1, chunks_L2_NX);
                      if (cost_L2[d2] < best_L2_cost) then
                          best_L2_cost := cost_L2[d2];
                  endfor

                  // now consider only those within 5% of best_L2_cost
(2)               foreach i := 0 .. (C2/C1)-1
                      d2 := d1 + i*C1;
                      if (cost_L2[d2]/best_L2_cost <= 1.05) then
                          j := 0;
(3)                       do
                              d := d2 + j*C2;
(4)                           m := LocalityMetric(N1, NX, d);
                              if (m < best_metric) then
                                  best_metric := m; best_d := d;
                              j := j+1;
                          while (d < max displacement in N1)
                      endif
                  endfor
              endif
          endfor
          return best_d;

      function LayoutProcedures(node N)
          // empty
```

Fig. 16. Functions that replace like-named functions in Figure 8 to form algorithm TPCM-2S. Since `FindBestDisplacement` calculates displacements for procedure sequences, `LayoutProcedures` is unnecessary, and hence empty.

average procedure size is unlikely to exceed C_2. The number of displacements whose costs are within 5% of the minimum tends to grow very slowly with cache size, and therefore the impact of loops (1) and (2) on the computational complexity is insignificant. From Section 4.2.5, we recall that computing the conflict costs for the L1 placement requires $O(P \cdot SP \cdot C_1)$ steps. To this, we add $O(P^2)$ for the cost of statement (4) inside the loop (3). This results in $O(P^2 + P \cdot SP \cdot C_1)$ for each call to the function `FindBestDisplacement`. Thus, the overall complexity for placing all P procedures is $O(P^3 + P^2 \cdot SP \cdot C_1)$, which is larger than the complexity of the algorithm in Section 4.2.5. This is not surprising. TPCM-2S considers several layouts for each procedure within `FindBestDisplacement`, while TPCM and TPCM-2 perform layout and placement separately. TPCM-2S thus incurs an extra factor of P.

6.1.5 *Restricting the Number of Cache-Aligned Procedures.* In our experiments, we found that it is not possible to achieve a satisfactory working set size if we insist on cache-aligning all executed procedures. The cache alignment imposes enough of a restriction on procedure placement that we end up with too much gap space that cannot be filled. Fortunately, the order in which our algorithm processes procedures was chosen such that the most important procedures—those responsible for most cache misses—are placed first. Toward the end of the execution of the algorithm, we are placing procedures that have very little impact on the cache miss rate, and thus there is little benefit to finding an optimal cache mapping for them. Figure 17 shows how the sum of the TRG edge weights between processed procedures grows with the number of processed procedures. When we choose the top 50% to 60% of the executed procedures, we see that the TRG edge weights between them account for over 90% of total TRG edge weights, and therefore the cache placement of the remaining procedures has little influence on the total miss rate.

However, the total amount of gap space keeps growing as we add more cache-aligned procedures. In fact, by the time we have processed between 40% and 60% of the executed procedures, we reach a state where the gap space between the procedures is sufficient to accommodate all the remaining procedures. This suggests that we should limit the number of cache-aligned procedures so that we get most of the benefit of cache alignment, but avoid the penalty of too much gap space.

We modify our algorithm as follows. First, perform cache alignment and procedure sequencing until the number of procedures placed reaches a certain percentage of the popular procedures. This results in a procedure sequence with a significant amount of gap space. Then, place the remaining procedures solely based on spatial locality. This allows us to fill the gaps in the procedure sequence and thus achieve good overall spatial locality. In this second phase, we insert a procedure into the sequence by trying all displacements that place it directly adjacent to some procedure already in the sequence, and choose the displacement that minimizes our spatial locality metric.

The percentage of cache-aligned procedures is a parameter that represents a trade-off between cache miss rate and working set size. From our measurements above, we would expect that values between 50% and 60% should work well. As we shall see in the next section, this is the case, and for values in this range, we

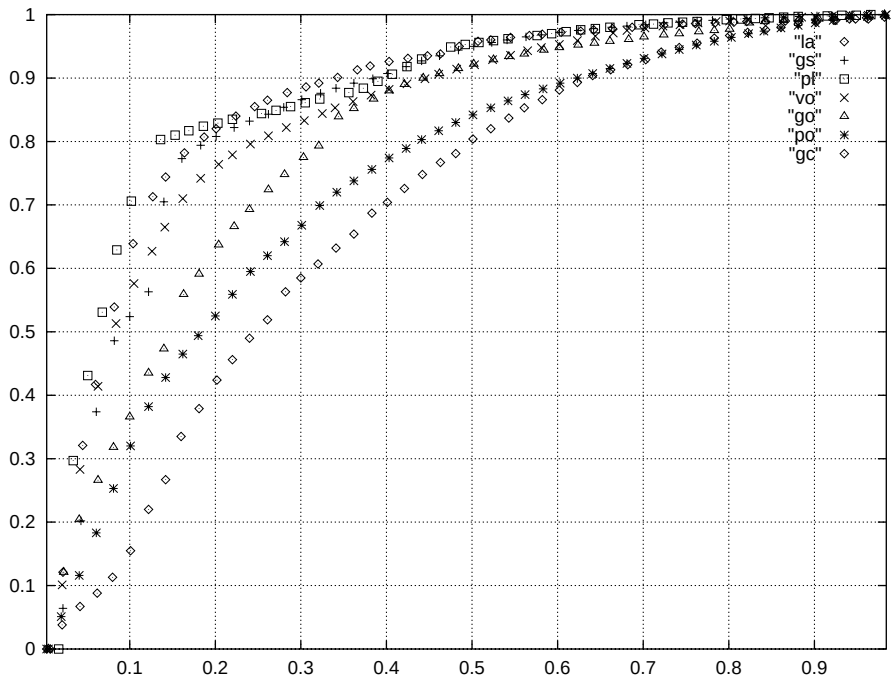


Fig. 17. Cumulative distribution of the sum of the TRG edge weights between processed procedures, for all seven benchmarks. The fraction of processed procedures is shown on the horizontal axis, and the ratio of the sum of the TRG edge weights for the processed procedures to the sum for all procedures is shown on the vertical axis.

can achieve very good performance for both the cache miss rate and the working set size.

6.2 Results

Tables VII and VIII shows the average working set sizes for three different interval lengths, with the percentage of cache-aligned procedures in the range between 40% and 90%. For most benchmarks, the L1 and L2 miss rates increase slightly as we decrease this percentage. For some benchmarks (*gcc* and *porky*), one of the two miss rates increases more strongly as we decrease the percentage. From Figure 17, we can see that this is the case because these benchmarks have a larger number of procedures with significant TRG edge weights. But for all benchmarks, choosing a percentage between 50% and 60% allows our placement algorithm to achieve working set sizes that are nearly identical to those of the PH placement, while still obtaining significantly better L1 and L2 miss rates.

The percentage of cache-aligned procedures is a parameter that allows us a trade-off between improved cache miss rates and smaller working sets. The correct choice of this parameter may depend on the relative run-time penalties for cache misses and TLB misses, as well as the relationship between TLB size and working set size.

In Figures 18–20, we compare our code placement (TPCM-2S) to the PH placement by showing the size of each individual working set over the entire trace. This

Table VII. Average Working Set Sizes and Cache Miss Rates for Different Placements. Where a percentage value is shown in the "placement" column, this refers to the version of our placement algorithm described in this section (TPCM-2S), which places the given percentage of the procedures based on the cache mapping. The placement of the remaining procedures is determined only by interprocedural spatial locality. The interval lengths (100K, 1M, 10M) are expressed in basic blocks, and each value in the table is the average of 10 randomized experiments.

Benchmark	Placement	Avg. working set size in 4KB pages			Cache miss rate in percent	
		100K	1M	10M	L2	L1
gcc	PH	65.9	130.7	248.9	0.98	5.40
	40%	65.5	128.0	249.9	0.78	4.98
	50%	65.7	127.6	249.8	0.71	4.90
	55%	65.5	128.3	250.7	0.65	4.86
	60%	65.1	127.2	249.9	0.67	4.87
	70%	67.2	128.9	249.8	0.62	4.82
	80%	66.9	128.3	250.8	0.61	4.73
	90%	68.2	130.6	253.0	0.57	4.72
go	PH	64.8	76.7	82.3	1.14	3.98
	40%	64.6	76.4	82.8	1.09	3.52
	50%	68.5	78.0	83.4	1.04	3.46
	55%	69.1	79.3	84.8	1.02	3.46
	60%	71.5	81.2	86.4	1.03	3.44
	70%	75.2	85.9	91.2	0.98	3.42
	80%	76.9	89.2	95.2	0.94	3.42
	90%	84.4	98.8	104.8	0.94	3.38
ghostscript	PH	31.8	50.6	61.1	0.17	2.87
	40%	32.3	51.0	61.5	0.17	2.21
	50%	33.5	51.2	61.8	0.16	2.14
	55%	34.0	51.9	61.8	0.17	2.15
	60%	34.7	52.4	62.3	0.14	2.14
	70%	37.2	55.3	63.9	0.13	2.13
	80%	39.1	58.2	67.2	0.13	2.09
	90%	40.3	60.2	69.5	0.12	2.11
latex	PH	30.0	41.7	54.3	0.16	2.27
	40%	29.8	42.2	54.9	0.10	1.90
	50%	33.1	44.5	55.3	0.11	1.89
	55%	33.6	45.5	56.5	0.11	1.87
	60%	36.9	48.1	58.2	0.11	1.86
	70%	39.0	52.1	61.8	0.11	1.84
	80%	41.9	57.0	67.8	0.11	1.82
	90%	42.4	58.7	70.7	0.10	1.83
perl	PH	22.3	23.9	33.1	0.42	5.17
	40%	21.9	23.5	32.5	0.18	4.10
	50%	22.3	23.9	32.8	0.18	4.04
	55%	22.5	24.2	33.5	0.15	4.03
	60%	22.7	24.3	33.6	0.17	4.01
	70%	24.0	25.6	34.7	0.24	4.02
	80%	23.6	25.1	34.0	0.23	4.09
	90%	24.1	25.7	35.0	0.18	4.07

Table VIII. Average Working Set Sizes and Cache Miss Rates for Different Placements (continued). Where a percentage value is shown in the "placement" column, this refers to the version of our placement algorithm described in this section (TPCM-2S), which places the given percentage of the procedures based on the cache mapping. The placement of the remaining procedures is determined only by interprocedural spatial locality. The interval lengths (100K, 1M, 10M) are expressed in basic blocks, and each value in the table is the average of 10 randomized experiments.

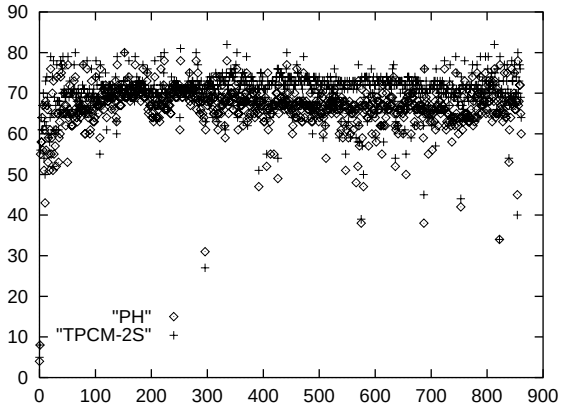
Benchmark	Placement	Avg. working set size in 4KB pages			Cache miss rate in percent	
		100K	1M	10M	L2	L1
porky	PH	45.8	71.8	106.1	0.35	3.89
	40%	42.7	68.7	104.4	0.34	2.88
	50%	42.4	68.9	105.3	0.30	2.62
	55%	42.0	67.7	103.7	0.30	2.55
	60%	41.6	67.6	104.5	0.28	2.47
	70%	41.7	67.5	104.0	0.25	2.32
	80%	42.0	67.7	104.2	0.25	2.16
	90%	41.7	67.4	103.5	0.22	2.10
vortex	PH	38.0	47.3	70.9	0.55	5.01
	40%	37.3	46.7	70.7	0.42	4.17
	50%	37.8	47.0	70.8	0.39	4.06
	55%	38.5	47.8	70.9	0.39	3.97
	60%	39.3	48.5	71.8	0.39	3.93
	70%	41.6	50.0	73.0	0.36	3.87
	80%	43.6	51.8	74.6	0.36	3.87
	90%	44.0	52.4	75.6	0.35	3.82

allows us to confirm that the relationship between the average working set size for the two placement algorithms is paralleled by the relationship between the sizes of each individual working set. Therefore, it is valid to compare the two placement algorithms based on the average working set size. We show the graphs for only three of our seven benchmarks (*go*, *ghostscript*, and *vortex*), since the behavior of the other benchmarks is similar.

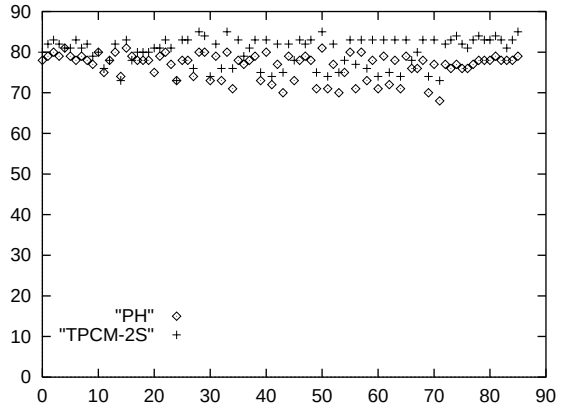
In Figure 21, we show the same type of graph for three different code placements: random procedure ordering, our cache-alignment algorithm from Section 5 (TPCM-2), and the algorithm presented in this section (TPCM-2S). The TPCM placements are better than the random procedure ordering because they place all the executed procedures together, while the random ordering mixes executed and unexecuted procedures. TPCM-2S outperforms TPCM-2 because TPCM-2 leaves more to chance what procedures are placed together on the same page.

6.3 Summary

We have successfully integrated a simple heuristic for optimizing interprocedural spatial locality into our cache-alignment algorithm. This allows us to achieve almost the same working set size as the PH placement. The PH placement is a simple instance of a general technique for optimizing working set sizes [Ferrari 1975; Hatfield and Gerald 1971], and thus we believe that reaching this level of performance is satisfactory. At the same time, we are able to retain most of the advantage of our optimization of cache misses.



(a) Interval = 100K basic blocks.



(b) Interval = 1M basic blocks.

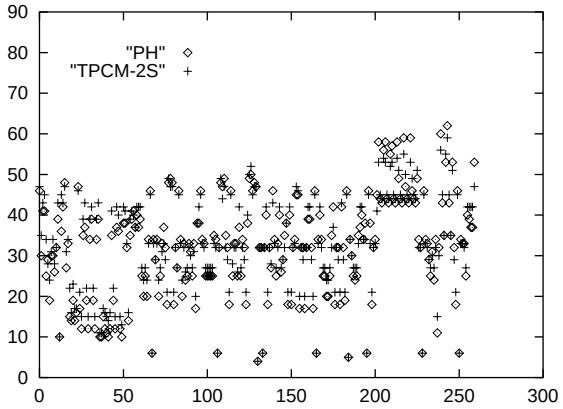
Fig. 18. Working set sizes for *go* at each interval in the testing trace. The vertical axis shows the working set size in 4KB pages, and the horizontal axis represents the intervals. The two data series shown in each graph correspond to the PH placement and the placement described in this section (TPCM-2S).

7. RELATED WORK

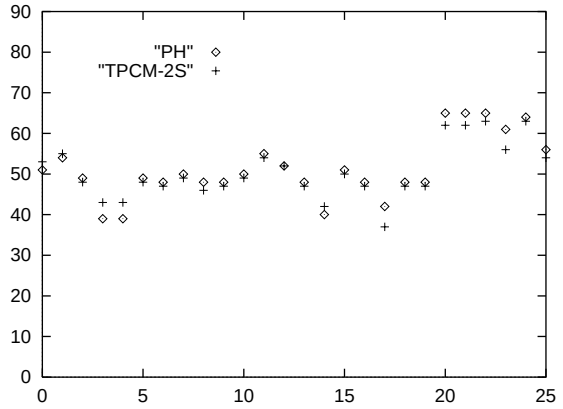
The initial motivation for this work came from a review of existing code-placement techniques aimed at reducing instruction cache conflicts. We found that all of the work of which we were aware was based on profile data that lacked temporal information. We summarize some of this work in Section 7.1.

We also provide a brief overview of some relevant work on intraprocedural code placement, dynamic code-placement schemes, methods for summarizing temporal information, and code placement aimed at working set reduction.

While our work focuses entirely on instruction memory references, we are aware that data memory references are also an important source of cache conflicts. Thus, it is also worthwhile to optimize the cache placement of data [Calder et al. 1998; Carr et al. 1994]. In fact, the recent work by Calder et al. [1998] applies the ideas in this article to the problem of data placement.



(a) Interval = 100K basic blocks.



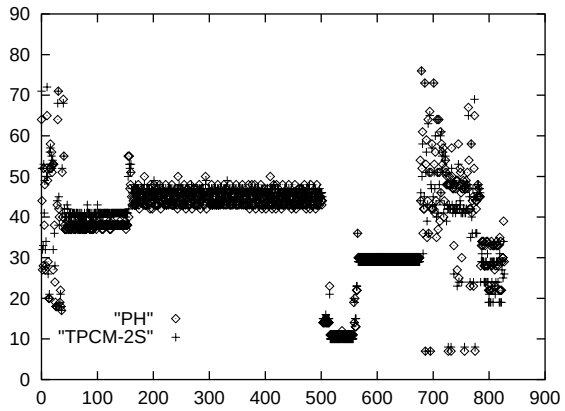
(b) Interval = 1M basic blocks.

Fig. 19. Working set sizes for *ghostscript* at each interval in the testing trace. The vertical axis shows the working set size in 4KB pages, and the horizontal axis represents the intervals. The two data series shown in each graph correspond to the PH placement and the placement described in this section (TPCM-2S).

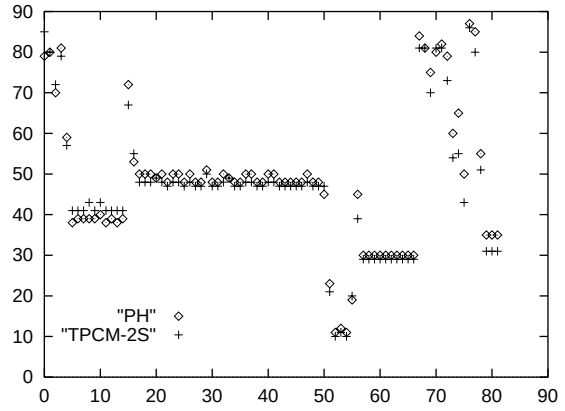
7.1 Similar Code-Placement Techniques

Some of the earliest work in this area was done by Hwu and Chang [1989], McFarling [1989], and Pettis and Hansen [1990]. Hwu and Chang use a WCG and a proximity heuristic to address the problem of basic-block placement. Their approach is unique in that they also perform function inline expansion during code placement to overcome the artificial barriers imposed by procedure call boundaries.

McFarling [1989] uses an interesting program representation (a DAG of procedures, loops, and conditionals) to drive his code-placement algorithm, but the profile information is still summarized in such a way that the temporal interleaving of blocks in the trace is lost. In fact, McFarling explicitly states that, because he is unable to collect temporal interleaving information, his algorithm assumes and optimizes for a worst-case interleaving of blocks. Finally, his algorithm is unique in its ability to determine which portions of the text segment should be excluded



(a) Interval = 100K basic blocks.



(b) Interval = 1M basic blocks.

Fig. 20. Working set sizes for *vortex* at each interval in the testing trace. The vertical axis shows the working set size in 4KB pages, and the horizontal axis represents the intervals. The two data series shown in each graph correspond to the PH placement and the placement described in this section (TPCM-2S).

from the instruction cache. The paper focuses only on a single instruction cache and thus ignores the other levels of the memory hierarchy. The rather complicated placement algorithm, which attempts to fold a tree structure of related code into the cache, may be difficult to adapt to multilevel caches.

Hashemi et al. [1997] try to improve the Pettis and Hansen algorithm by computing which cache lines are occupied by which procedures. Their goal is to use a more precise placement heuristic than the simple "closest-is-best" heuristic. However, they still use a simple WCG for profile information. Although this technique optimizes only a single instruction cache, it might be possible to extend it to address multiple levels of cache, because it has precise information on the cache mapping of the code. However, this technique ignores the issues of spatial locality and working set size.

Torellas et al. [1995] propose a code-placement technique for kernel-intensive ap-

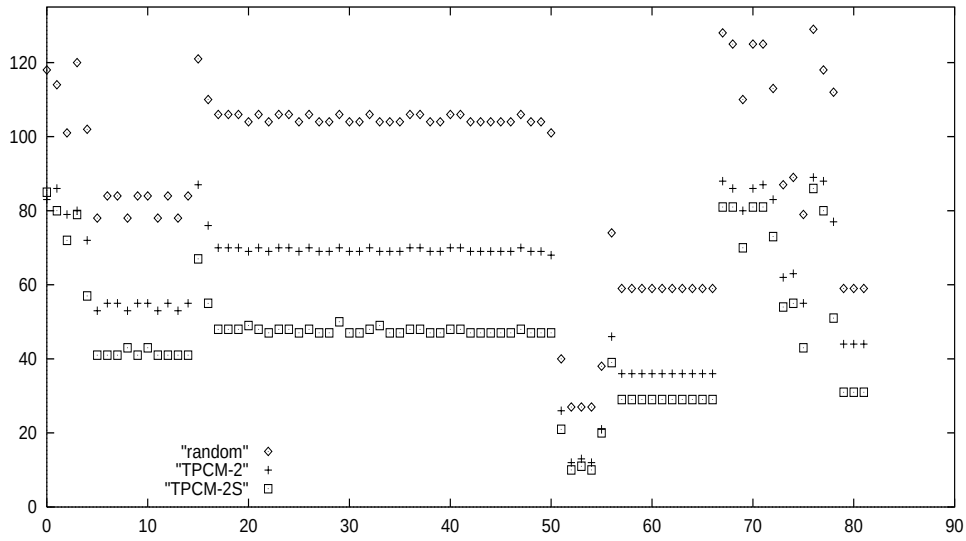


Fig. 21. Working set sizes for three different code placements: random procedure ordering, our cache-only placement (TPCM-2), and the placement algorithm presented in this section (TPCM-2S). Each point shows the working set size in 4KB pages for one interval of 1M basic blocks. The intervals are shown along the horizontal axis, and the working set sizes are shown on the vertical axis. This data is for the *vortex* benchmark and the "training" trace.

plications. Their algorithm considers the cache address mapping when performing code placement. They define an array of *logical caches*, equal in size and address alignment to the hardware cache. Code placed within a single logical cache is guaranteed never to conflict with any other code in that logical cache. Though there is a subarea of all logical caches that is reserved for the most frequently executed basic blocks, there is no general mechanism for calculating the placement costs across different logical caches. Their code placement is guided by execution counts of edges between basic blocks, and therefore does not capture temporal ordering information. There is no mention of multilevel caches and working set size, although the concept of logical caches could be extended to apply to multilevel caches. But this technique puts additional constraints on the code placement and may therefore make it more difficult to simultaneously optimize the working set size.

Soon after we published our initial work [Gloy et al. 1997] on procedure placement using temporal-ordering information, Kalamatianos and Kaeli [1998] published a similar approach. They define a structure called a *Conflict Miss Graph* (CMG) that is nearly identical in construction and contents to our TRG. They present a placement algorithm that is an extension of the work by Hashemi et al. [1997], extended to use the CMG. As one would expect, the differences between their algorithm and the one presented in Section 4 are small. Their work, however, optimizes for only the primary instruction cache.

7.2 Intraprocedural Code Placement

As we mentioned earlier, there are other inter-procedural techniques for code placement whose benefits are orthogonal to those gained through procedure placement. Section 4.6 looked at hot-cold splitting [Cohn and Lowney 1996; Pettis and Hansen 1990]. In their paper, Cohn and Lowney [1996] address not only the cache effect of hot-cold splitting but also the other optimizations it enables. Removing the cold code from the path makes it possible to optimize the hot code by removing partially dead code, eliminating unnecessary preserved registers and stack adjusts, and performing register lifetime splitting and copy propagation.

A similar technique is branch alignment [Calder and Grunwald 1994; Young et al. 1997]. It reorders basic blocks within each procedure to minimize branch penalties, which generally means minimizing the number of taken branches during execution. As a secondary effect, it improves the efficiency of a sequential instruction prefetching scheme, which further reduces instruction cache misses.

7.3 Dynamic Schemes

Dynamic schemes monitor the behavior of the system and perform optimizations at run time based on this behavior. Thus, they avoid the problems of profile-based optimizations: profiling is rarely undertaken (often because of apathy); and a program's behavior during profiling may differ from the actual run-time behavior (because of differences in the data sets).

Chen and Leupen [1997] present a technique for *just-in-time code layout*. Under this scheme, procedures are loaded into the text segment at run time in the order in which they are invoked. Their experiments show that the resulting procedure placement is at least as effective as the placement technique by Pettis and Hansen [1990].

Bershad et al. [1994] introduce a hardware device called the *Cache Miss Lookaside (CML) buffer* that records and summarizes a history of cache misses. They describe how one could modify the operating system's virtual memory system to use the CML data to detect conflicts caused by the page mapping and remove conflicts by dynamically remapping pages whenever large numbers of conflict misses are detected. Their experiments show that a CML buffer enables a large direct-mapped cache to perform nearly as well as a two-way set-associative cache of equivalent size and speed.

Prefetching schemes take a different approach to avoiding cache misses. Whereas the methods described change the addresses of code to remove conflicts, prefetching schemes leave the cache mapping of the code unchanged. They avoid the run-time penalty of conflicts by transferring code into the cache from the next level of the memory hierarchy before it is needed. Conflicting parts of the code still evict each other from the cache, but this does not force the processor to wait for the cache. Prefetching schemes have the advantage that they can also reduce compulsory and capacity misses, whereas code-placement schemes are limited to reducing conflict misses.

The simplest prefetching scheme is *next-N-line prefetching* [Smith 1982]. Under this scheme, the cache would prefetch the next N sequential lines after the line currently being executed. To cover branch targets that do not fall within these

N prefetched cache lines, *wrong-path prefetching* [Pierce and Mudge 1996] always prefetches the target address of control transfer instructions (with static addresses) in addition to the next N sequential lines. As an alternative to using the static branch address, *target-line prefetching* [Smith and Hsu 1992] uses a dynamically updated prediction table. For each cache line, this table provides the address of the successor line that occurred most recently. *Markov prefetching* [Joseph and Grunwald 1997] uses a dynamic prediction table to store the addresses of several of the next likely cache misses based on a recent miss address. When a cache miss occurs, the cache lines predicted to suffer misses can be prefetched. Finally, *cooperative prefetching* [Luk and Mowry 1998] combines the best of the hardware- and compiler-based approaches.

7.4 Temporal Profile Information

While there are some interesting approaches to summarizing temporal profile information [Ammons et al. 1997; Ball and Larus 1996; Young and Smith 1998], it appears that none of them are applicable to our kind of code placement. The history mechanism we use to analyze the temporal behavior of an execution trace is similar to the problem of profiling paths in a procedure call graph. Ammons et al. [1997] describe a way of implementing efficient path profiling. However, the data structure generated by this technique cannot be used in the place of our TRG, because it does not capture any interleaving information.

Quong [1994] describes an alternative way of summarizing temporal profile information. It is particularly interesting because this method has the opposite goal of our TRG: it removes the influence of code placement on the cache miss rate. It achieves this by assuming a stochastical model of the cache behavior and deriving a formula for the expected number of cache misses over all possible address mappings. For each code block, the expected number of cache misses for the entire trace can be computed based on the size of the gaps between successive references to the code block. The frequency count for the gap sizes can be recorded in a compact form by quantizing the gap sizes. The benefit of this scheme is that it provides an estimate of the cache miss rate for a program independent of its layout.

Phalke and Gopinath [1995] examine the temporal locality of memory references. They define the inter-reference gap (IRG) for an address in a trace as the time interval between successive references to that same address. The IRG stream for an address in a trace is the sequence of successive IRG values for that address, and each IRG stream is modeled using an order k Markov chain. They successfully apply this technique to page replacement policies and trace compaction.

7.5 Working Set Optimization

Hatfield and Gerald [1971] describe a method for reducing the working set for both data and instruction references. They divide the address space into relocatable sectors (which are smaller than pages) and define a *nearness matrix* C such that $C[i, j]$ is a count of control transfers or data references from sector i to sector j. The matrix is divided into square regions corresponding to pages. All entries in regions along the diagonal indicate references within the same page, and entries in regions near the diagonal indicate references to nearby pages. Reordering sectors corresponds to moving rows and columns in the matrix. The authors describe a method

of reordering sectors that maximizes the entries near the diagonal and achieves an improvement in spatial locality. (Ferrari [1975] discusses the working set problem in a similar but more abstract way by defining it in terms of a restructuring graph and a clustering algorithm.) Hatfield and Gerald report a reduction in working set size of about 25% when they apply this technique in a compiler.

We note the similarity between the nearness matrix and the WCG, as well as the analogy between reordering sectors to increase near-diagonal entries and the PH algorithm to achieve closeness between procedures connected by important WCG edges. It is for this reason that we used the working set sizes obtained by the PH algorithm as our baseline in Section 6.

8. CONCLUSIONS

We have presented a procedure-placement technique that advances the state of the art in two major ways. First, our technique for capturing temporal-ordering information provides substantially better information on the temporal behavior of a program than previous profiling techniques. Second, our procedure-placement algorithm addresses all relevant levels of the memory hierarchy.

To the best of our knowledge, all previously published code-placement techniques use profile data that do not capture any temporal information beyond counts of procedure calls. We analyzed which aspects of the flow of execution of a program affect the number of cache conflict misses and showed that the interleaving of the execution of different parts of the program is crucial to estimating cache conflict misses. We therefore developed a profiling technique that extracts precisely this information, in a form that is concise enough to be practical for code-placement optimizations.

By combining the temporal-ordering information with precise control over cache-mapping conflicts, we achieve a significant improvement in single-level instruction cache conflicts. But our analysis of second-level cache conflicts and working set sizes shows that it is not sufficient to focus only on this single level of the memory hierarchy. The placement decisions that are intended to improve this aspect of instruction fetch performance can actually incur penalties in the other levels of the memory hierarchy.

We showed that our placement algorithm can easily be extended to simultaneously optimize procedure placement in multilevel caches. We demonstrated this for a two-level cache, but the technique is equally applicable to caches with more than two levels.

Finally, we were able to integrate the cache conflict optimization with another code-placement technique that optimizes the working set size. While these two optimizations may not be able to achieve peak performance simultaneously, we were able to find a trade-off between them that achieves a good compromise.

As applications keep growing in size and as the impact of memory system efficiency on program performance increases, code placement will continue to become more important. This view is also supported by the increasing number of workstation vendors who are developing and shipping code-placement optimization tools.

ACKNOWLEDGMENTS

We would like to thank Trevor Blackwell and Brad Calder. Trevor showed us how to apply randomness to measure system performance better. Brad verified our implementation of the algorithm for procedure mapping using cache line coloring, and he worked with us on the algorithm described in Section 4. We also thank the anonymous reviewers for their numerous helpful suggestions.

REFERENCES

Ammons, G., Ball, T., and Larus, J. 1997. Exploiting hardware performance counters with flow and context sensitive profiling. In *Proceedings of ACM SIGPLAN '97 Conference on Programming Language Design and Implementation*. ACM, New York, 85–96.

Ball, T. and Larus, J. 1996. Efficient path profiling. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*. IEEE, Los Alamitos, California, 46–57.

Bershad, B., Chen, J. B., Lee, D., and Romer, T. 1994. Avoiding conflict misses dynamically in large direct-mapped caches. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, New York, 158–170.

Blackwell, T. 1998. Applications of randomness in system performance measurement. Ph.D. thesis, Harvard University, Cambridge, Massachusetts.

Calder, B. and Grunwald, D. 1994. Reducing branch costs via branch alignment. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, New York, 242–251.

Calder, B., Krintz, C., John, S., and Austin, T. 1998. Cache-conscious data placement. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, New York, 139–149.

Carr, S., McKinley, K., and Tseng, C. 1994. Compiler optimizations for improving data locality. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, New York, 252–262.

Chen, J. B. and Leupen, B. 1997. Improving instruction locality with just-in-time code layout. In *Proceedings of the USENIX Windows NT Workshop*. USENIX, Berkeley, California, 25–32.

Cohn, R. and Lowney, G. 1996. Hot cold optimization of large Windows NT applications. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*. IEEE, Los Alamitos, California, 80–89.

Cvetanovic, Z. and Donaldson, D. 1996. AlphaServer 4100 performance characterization. *Digital Technical Journal 8,* 4, 3–20.

Cytron, R. and Loewner, P. 1986. An automatic overlay generator. *IBM Journal of Research and Development 30,* 6 (Nov.), 603.

Denning, P. 1970. Virtual memory. *Computing Surveys 2,* 3 (Sept.), 153–189.

Ferrari, D. 1974. Improving locality by critical working sets. *Commun. ACM 17,* 11 (Nov.), 614–620.

Ferrari, D. 1975. Tailoring programs to models of program behavior. *IBM Journal of Research and Development 19*, 244.

Gloy, N., Blackwell, T., Smith, M. D., and Calder, B. 1997. Procedure placement using temporal ordering information. In *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*. IEEE, Los Alamitos, California, 303–313.

Hashemi, A., Kaeli, D., and Calder, B. 1997. Efficient procedure mapping using cache line coloring. In *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation*. ACM, New York, 171–182.

Hatfield, D. and Gerald, J. 1971. Program restructuring for virtual memory. *IBM Systems Journal 10,* 3, 168–192.

Hill, M. D. 1988. A case for direct-mapped caches. *Computer 21,* 12 (Dec.), 25–40.

Hwu, W. and Chang, P. 1989. Achieving high instruction cache performance with an optimizing compiler. In *Proceedings of the 16th Annual International Symposium on Computer Architecture*. IEEE, Washington, D.C., 242–251.

Joseph, D. and Grunwald, M. 1997. Prefetching using Markov predictors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*. ACM, New York, 252–263.

Kalamatianos, J. and Kaeli, D. 1998. Temporal-based procedure reordering for improved instruction cache performance. In *Proceedings of the 4th IEEE Symposium on High-Performance Computer Architecture*. IEEE, New York, 244–253.

Kawaf, T., Shakshober, D. J., and Stanley, D. 1996. Performance analysis using very large memory on the 64-bit AlphaServer system. *Digital Technical Journal 8,* 3, 58–65.

Luk, C.-K. and Mowry, T. 1998. Cooperative prefetching: Compiler and hardware support for effective instruction prefetching in modern processors. In *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture*. IEEE, Los Alamitos, California, 182–193.

McFarling, S. 1989. Program optimization for instruction caches. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, New York, 183–191.

Pettis, K. and Hansen, R. 1990. Profile-guided code positioning. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*. ACM, New York, 16–27.

Phalke, V. and Gopinath, B. 1995. An inter-reference gap model for temporal locality in program behavior. In *Proceedings of the ACM SIGMETRICS '95 International Conference on Measurement and Modeling of Computer Systems*. ACM, New York, 291.

Pierce, J. and Mudge, T. 1996. Wrong-path prefetching. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*. IEEE, Los Alamitos, California, 165–175.

Quong, R. 1994. Expected I-cache miss rates via the gap model. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*. IEEE, Los Alamitos, California, 372–383.

Ryder, K. 1974. Optimizing program placement in virtual systems. *IBM Systems Journal 13*, 292.

Smith, A. 1982. Cache memories. *Computing Surveys 14,* 3 (Sept.), 473–530.

Smith, J. and Hsu, W.-C. 1992. Prefetching in supercomputer instruction caches. *Supercomputing*, 588.

Smith, M. D. 1996. Extending SUIF for machine-dependent optimizations. In *Proceedings of the First SUIF Compiler Workshop*. Stanford University, Stanford, California, 14–25.

Srivastava, A. and Eustace, A. 1994. ATOM: a system for building customized program analysis tools. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*. ACM, New York, 196–205.

Torellas, J., Xia, C., and Daigle, R. 1995. Optimizing instruction cache performance for operating system intensive workloads. In *Proceedings of the First IEEE Symposium on High-Performance Computer Architecture*. IEEE, Los Alamitos, California, 360–369.

Young, C., Johnson, D., Karger, D., and Smith, M. D. 1997. Near-optimal intraprocedural branch alignment. In *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation*. ACM, New York, 183–193.

Young, C. and Smith, M. D. 1998. Better global scheduling using path profiles. In *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture*. IEEE, Los Alamitos, California, 115–123.

Received January 1999; revised June 1999; accepted November 1999