Procedure Placement Using Temporal Ordering Information

Nikolas Gloy[†], *Trevor Blackwell*[†], *Michael D. Smith*[†], *and Brad Calder*[‡]

[†]Division of Engineering and Applied Sciences, Harvard University

[‡]Department of Computer Science and Engineering, University of California, San Diego

Abstract

Instruction cache performance is very important to instruction fetch efficiency and overall processor performance. The layout of an executable has a substantial effect on the cache miss rate during execution. This means that the performance of an executable can be improved significantly by applying a code-placement algorithm that minimizes instruction cache conflicts. We describe an algorithm for procedure placement, one type of code-placement algorithm, 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. We compare the performance of our algorithm with previously published procedure-placement algorithms and show noticeable improvements in the instruction cache behavior.

Keywords: code layout, profiling, conflict misses

1 Introduction

The linear ordering of procedures in a program's text segment fixes the addresses of each of these procedures and this in turn determines the cache line(s) that each procedure will occupy in the instruction cache. In the case of a direct-mapped cache, conflict misses result when the execution of the program alternates between two or more procedures whose addresses map to overlapping sets of cache lines. Several compile-time *code-placement* techniques have been developed that use heuristics and profile information to reduce the number of conflict misses in the instruction cache by a reordering of the program code blocks [5,6,7,8,11]. Though these techniques successfully remove a sizeable number of the conflict misses when compared to the default code layout produced during the typical compilation process, it is possible to do even better if we gather improved profile information and consider the specifics of the hardware configuration. To this end, we propose a method for summarizing the important temporal ordering information related to code placement, and we show how to use this information in a machine-specific manner that often further reduces the number of instruction cache conflict misses. In particular, we apply our new techniques to the problem of procedure placement in direct-mapped caches, where the compiler achieves an optimized cache line address for each procedure by specifying the ordering of the procedures and gaps between procedures in an executable.

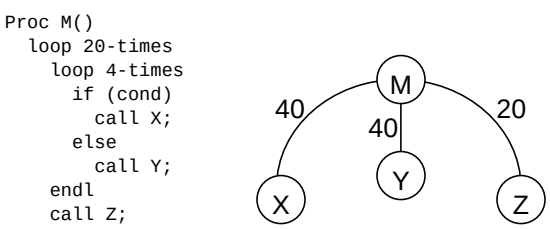
Code-placement techniques may reorganize an application at one or more levels of granularity. Typically, a technique focuses on the placement of whole procedures or individual basic blocks. We use the term *code block* to refer to the unit of granularity to which a code-placement technique applies. Though we focus on the placement of variable-sized code blocks defined by procedure boundaries, our techniques for capturing temporal information and using this information during placement apply to code blocks of any granularity.

The default code layout produced by most compilers places procedures in the order in which they were listed in the source files and preserves the order of object files from the linker command line. Therefore, it is left to chance which code blocks will conflict in the cache. Whenever execution alternates between code blocks that overlap in the cache, the number of conflict misses experienced during this execution grows by the amount of dynamic interleaving between the blocks. Furthermore, several studies have shown that, when a compile-time optimization changes the relative placement of code blocks, large changes can occur in the instruction cache miss rate [3,4]. This means that the performance of the program is affected not only by the intended effect of the optimization, but also by the resulting change in instruction cache miss rate. This makes it difficult to predict the final effect of the optimization on performance. In summary, code-placement techniques are important because they both reduce the instruction cache miss rate and enable the effective application of some important compile-time optimizations [4].

To reduce the instruction cache miss rate of an application, a code placement algorithm requires two capabilities: it must be able to assign code blocks to the cache lines; and it must have information on the relative importance of avoiding overlap between different sets of code blocks. There are only a few ways for the compiler to set the addresses of a code block. The compiler can manipulate the order in which code blocks appear in the executable, and it can leave gaps between two adjacent code blocks to force the alignment of the next code block at a specific cache line. The more interesting problem is determining how code blocks should overlap in the instruction cache.

The previous work on procedure placement has almost exclusively been based on summary profile statistics that simply indicate how often a code block was executed. Often this information is organized into a *weighted procedure call graph* (WCG) that records the number of calls that occurred between pairs of procedures during a profiling run of the program. Figure 1 contains an example of a WCG. This summary information is used to estimate the penalty resulting from the placement of these procedure pairs in the same cache locations. The aim of most existing algorithms is to place procedures such that pairs with high call counts do not conflict in the cache.

Counting the number of calls between procedures and summarizing this information in a WCG provides a way of recognizing procedures that are temporally related during the execution of a program. However, a WCG does not give us all the temporal information that we would like to have. In particular, the absence of an edge between two procedures does not necessarily mean that there is no penalty to overlapping the procedures. For example, the WCG in Figure 1 is produced both when the condition `cond` alternates between true and false (Trace #1 in Figure 1) and when the condition `cond` is true 40 times and then false 40 times (Trace #2). Assume for the purposes of this example that all procedures in Figure 1 require only a single cache line and that we have only three locations in our direct-mapped instruction cache. If one cache location is reserved for procedure M, we do not want the same code layout for the last two cache locations in both these execution traces. Trace #1 experiences fewer cache conflict misses when procedures X and Y are each given distinct cache line (Z shares a cache line with X or Y), while Trace #2 experiences fewer cache conflict misses when procedures X and Y share a cache line (Z is given its own cache line). The WCG in Figure 1 does not capture the temporal ordering information that is needed to determine which layout is best. A WCG summarizes only direct call information; no precise information is provided on the importance of conflicts between siblings (as illustrated in Figure 1) or on more distant temporal relationships.



(a) Example program

(b) Weighted procedure call graph

Call Trace #1: $M((XYXY)Z)^{20}$

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

(c) Possible call traces corresponding to the WCG

Figure 1. Example of a program that calls three leaf procedures. The weighted call graph (WCG) is obtained when the condition `cond` is true 50% of the time. Notice that this same WCG is obtained from many different call traces.

To enable better code layout, we want a measure of how much the execution of a program alternates between pairs of procedures (not just the pairs connected by an edge in the WCG). We refer to this measure as *temporal ordering information*. With this and other information concerning procedure sizes and the target cache configuration, we can make a better estimate of the number of conflict misses experienced by any specific layout.

We begin in Section 2 with a brief description of a well-known procedure-placement algorithm that sets a framework for understanding our new algorithm. We present the details of our algorithm in Sections 3 and 4. Section 3 describes our method for extracting and summarizing the temporal ordering information in a program trace, while Section 4 presents our procedure-placement algorithm that uses the information produced by this method. In Section 5, we explain our experimental methodology and present some empirical results that demonstrate the benefit of our algorithm over previous algorithms for direct-mapped caches. Section 6 describes how to modify our algorithm for set-associative caches. Finally, Section 7 reviews other related work in code layout and Section 8 concludes.

2 Procedure placement in Pettis & Hansen

Current approaches to procedure placement rely on greedy algorithms. We can summarize the differences between these algorithms by describing how each:

- selects the order in which procedures are considered for placement; and

- determines where to place each procedure relative to the already-placed procedures.

We begin with a description of the well-known procedure-placement algorithm by Pettis and Hansen [8]. As we will explain, our new algorithm retains much of the structure and many of the important heuristics found in the Pettis and Hansen approach. In addition to procedure placement, Pettis and Hansen also address the issues of basic-block placement and branch alignment. For the purposes of this paper, we use the acronym PH when referring to our implementation of the procedure placement portion of their algorithm.

Pettis and Hansen reduce instruction-cache conflicts between procedures by placing the most frequent caller/callee procedure pairs at adjacent addresses. Their approach is based on a WCG summary of the profile information. They use this summary information both to select the next procedure to place and to determine where to place that procedure in relationship to the already-placed procedures.

For our implementation of PH, we produce an undirected graph with weighted edges, which contains essentially the same information as a WCG. There is one node in our graph for each procedure in the program. An edge $e_{p,q}$ connects two nodes p and q if p calls q or q calls p. The weight $W(e_{p,q})$ given to $e_{p,q}$ is equal to the total number of control-flow transitions between procedure p and q in the analyzed instruction trace. These transitions correspond to call and return points, and they quantify the amount of dynamic interleaving between the procedures. Therefore, $W(e_{p,q})$ in our graph is exactly twice the weight of the edge $e_{p,q}$ in a typical WCG; each call is paired (typically) with a return. The extra factor of two does not change the procedure placement produced by PH.

In PH, we use this graph to select both the next procedure to place and determine the relative placement for this procedure. 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 pairs 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.

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* [8]. 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' and B' the reverse of each chain. The four possibilities are AB, AB', A'B and A'B'. To choose the best one of these, 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. Our implementation of PH chooses the merged chain that minimizes the distance (in bytes) between p and q.

3 Summarizing temporal ordering information

Any algorithm that aims to optimize the arrangement of code blocks needs a *conflict metric* which quantifies the importance of avoiding conflicts between sets of code blocks. Ideally, the metric would report the number of cache conflict misses caused by mapping a set of code blocks to overlapping cache lines. We do not expect to find a metric that gives the exact number of resulting cache conflict misses, and we do not need one. We simply need the metric to be a linear function of number of conflict misses.[1] Section 5.3 shows that the metric used in our algorithm exhibits strong correlation with the instruction cache miss rate.

As discussed in the previous section, PH uses an edge weight $W(p,q)$ based on the dynamic count of calls and returns between two procedures p and q as its conflict metric. This simple metric drives the merging of nodes. Unfortunately, this metric has several drawbacks, as illustrated in Section 1.

To understand how to build a better conflict metric, it is helpful to review the actions of a cache when processing an instruction stream. Assume for a moment that we are tracking code blocks with a size equal to the size of a cache line. For a direct-mapped cache, a code block b maps to cache line $l = (Addr(b)\ DIV\ line\_size)\ MOD\ cache\_lines$. This code block remains in the cache until another code block maps to the same cache line. In terms of code layout, it is important therefore to note which other code blocks are referenced temporally nearby to a reference to b. Ideally, none of the blocks referenced between consecutive references to b map to the same cache line as b. In this way, we get *reuse* of the initial fetch of block b and do not experience a conflict miss during the second reference to b.

1. Clearly, any difference between the training and testing data sets will also affect the metric's ability to predict cache conflict misses in the testing run.

Since the reuse of a code block can be prevented by a single other code block in direct-mapped caches, we construct a data structure that summarizes, for each procedure *p* and *q*, the frequency of finding *q* between two consecutive references to *p*. It is convenient to build this data structure as a weighted graph, where the nodes represent individual code blocks. We refer to this graph as a *temporal relationship graph* (TRG). In PH, the conflict metric is simply the edge weight $e_{p,q}$ between two nodes *p* and *q* in the WCG. A TRG is 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 2 presents the TRG resulting from the execution that produced trace #2 in Figure 1.

To summarize the temporal locality of the code blocks in a trace (and to help build the corresponding TRG), we maintain and analyze an ordered set, *Q*, of recently-referenced code-block identifiers (e.g. procedure names). The code blocks in *Q* are ordered as they appeared in the trace. There is a bound on the maximum size of *Q* because its entries eventually become irrelevant and can be removed. There are two ways in which a code block identifier *p* can become irrelevant. First, we need only the latest reference to *p* in *Q*. Any code blocks that are executed after the most recent reference to *p* can only have an effect on that reference and not on any earlier reference. Second, *p* can become irrelevant if a sufficiently large amount of (unique) code has been executed since *p*'s last reference and evicted *p* from the cache. In some sense, this eviction is due to the limited capacity of the cache and not because of any timely interleaving. If *T* is the set of code block identifiers reached since the last reference to *p* and *S*(*T*) is the sum of the sizes of the code blocks referenced in *T*, exactly how big *S*(*T*) needs to grow before *p* becomes irrelevant for capacity concerns depends on the cache mapping of the code. Assuming that the code layout maximizes the reuse of the members of *T*, they will be mapped to mostly non-overlapping addresses, and their cache footprint will be nearly equal to *S*(*T*). Therefore, *p* becomes irrelevant only after *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.

A TRG is built from the information in Q. In particular, we process the trace one code block identifier at a time. At each processing step, we take the next identifier *p* from the trace and place it at the most recent end of *Q*. We next analyze *Q* to summarize the interleaving information related to this latest reference to *p*. If there is a previous reference to *p* in *Q*, we increment the weight on the edge $e_{p,q}$ for each *q* between the two references to *p*. If node *p* does not exist in the TRG, we create it; if the edge $e_{p,q}$ does not exist, we create it with a weight of 1. If on the other hand there is no previous reference to *p* in *Q*, we make no modifications to the TRG during this processing step. No modification is necessary since there is no previous reference to *p* that we could exploit for reuse reasons. We then perform maintenance on Q: If there is a previous occurrence of *p* in *Q*, remove it. If not, we remove the oldest members of *Q* until the removal of the next least-recently-used identifier would cause the total size (in bytes) of remaining code blocks in *Q* to be less than twice the cache size. The process then repeats until we have processed the entire trace. Figure 3 illustrates the processing of part of the trace that produced the TRG in Figure 2. After processing, we are left with a TRG whose edge weights $W(e_{p,q})$ record the number of times *p* and *q* occurred within a sufficiently small distance to be present in *Q* at the same time, independent of how *p* and *q* are related in the program's call graph.

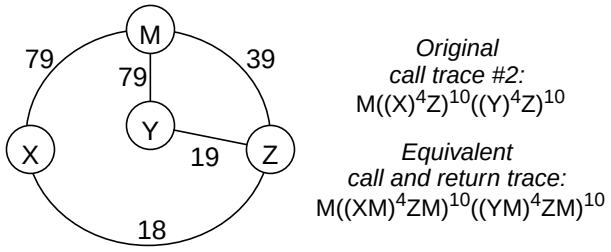


Figure 2. The TRG resulting from the execution trace #2 given in Figure 1. TRG construction is based on the ordering of calls and returns in this execution trace. All of the edges from the WCG still remain, except that their weights are nearly doubled. The weights are not quite doubled since an edge is incremented only when interleaving occurs, i.e. we see *p* then *q* then *p* again. The extra two edges indicate that there is interleaving in this trace between the procedure pairs (X,Z) and (Y,Z), but not between (X,Y).

4 Our placement algorithm

Given the discussion in Sections 2 and 3, it should be clear that we could use TRG constructed in Section 3 within the procedure-placement algorithm described by Pettis and Hansen [8]. We have found however that extra temporal ordering information alone is not sufficient to guarantee lower instruction cache miss rates. To get consistent improvements, we also make two key changes to the way we determine where to place each procedure relative to the already-placed procedures. The first involves the use of procedure size and cache configuration information that allows us to make a more informed procedure-placement decision. The second involves the gathering of temporal ordering information at a granularity finer than the procedure unit; we use this more detailed information to overcome problems created by procedures that are

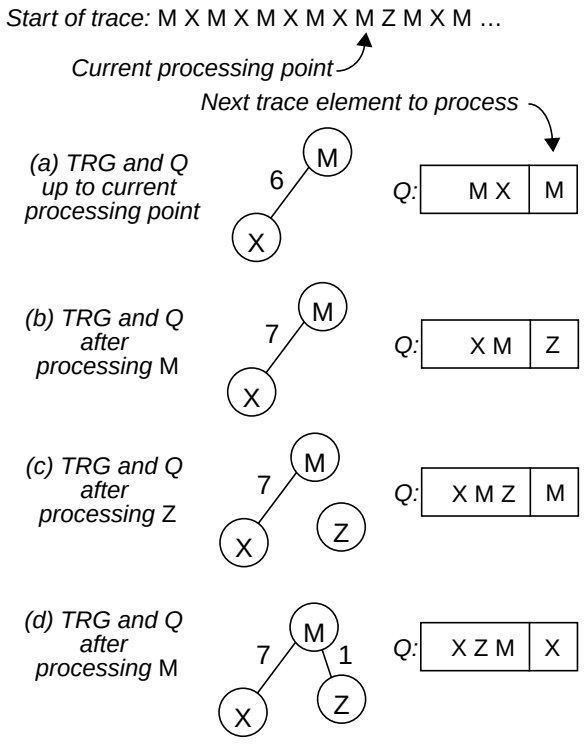


Figure 3. The TRG build process using execution trace #2 as an example. The processing of *Q* in (a) causes the edge weight $W(e_{M,X})$ to be incremented because X occurs between M and its previous occurrence in *Q*. The processing of *Q* in (b) does not add any new edges to the TRG because there is 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.

larger than the cache size. For efficiency reasons, we also consider only *popular* (i.e. frequently executed) procedures during the building of a relationship graph, as was proposed by Hashemi et al. [5].

The rest of this section outlines our procedure-placement algorithm. Section 4.1 begins with a description of the TRGs required for our algorithm and how we iterate through the procedure list selecting the order in which procedure are processed by the main outer loop. Section 4.2 focuses on the portion of our algorithm's main loop that places a procedure relative to the procedures already processed using cache configuration and procedure size information. This placement decision simply specifies a cache-relative alignment among a set of procedures. The determination of each procedure's starting address (i.e. its placement in the linear address space) occurs only after all popular procedures have been processed. Section 4.3 presents the details of this process, and Section 4.4 comments on the practicality of this approach.

4.1 TRGs and the main outer loop

Our algorithm uses two related TRGs. One selects the next procedure to be placed (TRG_{select}); and other aids in the determination of where to place this selected procedure (TRG_{place}). In PH, these two graphs are initially the same. In our algorithm, the graphs differ in the granularity of the code blocks processed during TRG build. While a code block in TRG_{select} corresponds to a whole procedure, a code block in TRG_{place} corresponds to a statically-determined *chunk* of a procedure. In the next section, we discuss why chunking is important.

For our benchmarks, we have found that a chunk size of 256 bytes works well. TRG_{place} thus contains $\lceil (\text{sizeof}\langle p \rangle)/chunksize \rceil$ nodes for each procedure *p* in a program. It is straightforward to modify the algorithm in the previous section to generate both TRGs simultaneously. Though we record temporal information concerning parts of procedures, our procedure-placement algorithm places only whole procedures. We use the finer-grain information only to find the best relative alignment of the whole procedures as explained below.

Though TRG_{select} contains more edges per node than the relationship graph built in PH (due to the additional temporal ordering information), we process TRG_{select} in exactly the same greedy-merging manner as the relationship graph discussed in Section 2. The only other difference in our "working" relationship graph is that TRG_{select} contains only popular procedures.

4.2 Determining cache-relative alignments

In PH, the data structure for the nodes in the working graph is a linear list (or a chain) of the procedures. The building of a chain is more restrictive in terms of selecting starting addresses for placed procedures than it needs to be however. The only constraint that we need to maintain is that the placed procedures are mapped to addresses that result in a cache layout with a small conflict cost.

Instead of chains, we use a data structure for nodes in TRG_{select} that comprises of a set of tuples. Each tuple consists of a procedure identifier and an offset, in cache lines, of the beginning of this procedure from the beginning of the cache. For a node containing only a single procedure, the offset is zero. When two nodes, each containing a single procedure, are merged together, our algorithm modifies the offset of the second procedure to ensure that the cost metric of the placement of these two procedures in the cache is minimized. The algorithm in Figure 4 presents the pseudo-code for the merging of two nodes containing any number of already-placed procedures.

```
typedef NODE = {(id, offset), ...};
typedef CACHE = array [#_cache_lines] of {id, ...};

NODE merge_nodes (NODE n1, NODE n2) {
  CACHE c1, c2;

  // Initialize cache c1 with chunks from node n1.
  foreach (id, offset) pair p in n1 {
    for (i = p.offset; i < p.offset+sizeof(p.id); i++)
      int lineIndex = i mod #_cache_lines;
      c1[lineIndex] := c1[lineIndex] ∪ {p.id};
    }

  // Initialize c2 using n2 -- code not shown.

  // Find the best relative offset of n2 with respect
  // to n1 that yields the lowest metric.
  int best_offset := 0, best_metric := INFINITY;
  foreach i in (0..#_cache_lines)
    foreach j in (0..#_cache_lines) {
      int metric := 0;
      foreach id p1 in c1[(j+i) mod #_cache_lines]
        foreach id p2 in c2[j]
          metric += weight_on_edge(p1,p2);
      if (metric < best_metric)
        best_metric := metric; best_offset := i;
    }

  // Update id's in n2 with the best relative offset.
  foreach (id,offset) pair p in n2
    p.offset += best_offset;

  return (n1 ∪ n2);
}
```

Figure 4. Pseudo-code for the merging of two nodes from the temporal relationship graph TRG_{select}. Procedure chunks within a node are identified by unique *id*'s. An *offset* for a chunk id records the cache-line index corresponding to the beginning of that chunk. Offsets are always in units of cache lines.

Three items are note-worthy concerning the *merge_nodes* routine in Figure 4. First, when we merge two nodes, we leave the relative alignment of all the procedures within each node unchanged. We do not backtrack and undo any previous decisions. Though the ability to rearrange the entire set of procedures in the two nodes being merged might lead to a better layout, this flexibility would noticeably increase the computational complexity of the algorithm. We assume that the selection order for procedure placement has guaranteed that we have already avoided the most expensive, potential cache conflicts. As our experimental results show, this greedy heuristic works quite well in practice.

Second, *merge_nodes* calculates a cost metric for each potential alignment of the layout in the first node with respect to the layout in the second node. If we fix the layout of the first node to begin at cache line 0, we can offset the start of the second node's layout by any number between 0 and the number of lines in the cache. We evaluate each of these relative offsets using the fine-grained temporal information in TRG_{place}. Each cost estimate is calculated only for procedure-piece conflicts between nodes and not for the intra-node conflicts which do not change. With the fine-grained information in TRG_{place}, we are able to find a low-cost, cache-relative placement for procedures even if they are larger in size than the cache. Without this information on relationships between procedure chunks, all cache-relative placements for these procedures would look equally good.

Third, if the cost-metric calculation produces several relative offsets with the same cost, our algorithm selects the first of these offsets. In the simplest case, if we merge two nodes each containing a single procedure (call them *p* and *q*) and the total size of these two procedures is less than the cache size, the merging of these nodes will result in a node that is equivalent to the chain created by PH. In other words, *merge_nodes* selects the first empty cache line after procedure *p* to begin procedure *q* since that is the first zero-cost location for *q*.

4.3 Producing the final linear list

The merging phase of our algorithm ends when there are no more edges left in TRG_{select}.[2] The final step in our algorithm produces a linear arrangement of all of the program procedures given the relative alignment decisions contained in remaining TRG_{select} nodes. We describe a simple algorithm that is concerned only with the reduction of cache misses. We are aware that the spatial and temporal locality of code pages is also an important performance factor, and it is possible to alter the algorithm described below to select a linear ordering of procedures that reduces paging problems.

To begin, we select a procedure *p* with a cache-line offset of 0 (any starting offset will do). This is the first procedure in our linear layout. To find the next procedure in the linear layout, we search the nodes for a procedure *q* whose cache-relative offset results in the smallest positive gap in cache lines between the end of *p* and the start of *q*.

To understand the general case, assume that the cache contains *N* cache lines and that procedure *p* is the last procedure in the linear layout. If *p* ends at the cache-relative offset p_{EL}, we choose a procedure *q* which starts at cache-relative offset q_{SL} as the next procedure in the linear layout. We choose *q* such that it produces the smallest positive *gap* value among all unconsidered popular procedures. The formula for *gap* is:

$$gap = \begin{cases} q_{SL} - p_{EL} & \text{if } (q_{SL} > p_{EL}) \\ q_{SL} - (p_{EL} - N) & \text{otherwise} \end{cases}$$

2. Unlike PH, our "working" graph, TRG_{select}, is not necessarily reduced to a single node. TRG_{select} contains only popular procedures, and it is possible to have the only connection between two popular procedures be through an unpopular procedure.

Whenever we produce a gap between two popular procedures, we search the unpopular procedures for one or more that fill the gap. Once we determine an address for each popular procedure in the linear address space, we simply append any remaining un-placed, unpopular procedures to the end of our linear list.

4.4 Practicality

In our on-going work, instead of processing traces we generate the TRGs during program execution using instrumentation techniques. Our instrumented executables run approximately 25-times slower than the corresponding non-instrumented executable.

The running time of the placement algorithm is dominated by the time spent in the routine *merge_nodes*. Let P be the number of popular procedures and C the number of cache lines. The routine *merge_nodes* is called at most P times. For each call, the two outer loops iterate C times and the two inner loops iterate less than P times. A crude upper bound on the running time is thus P^3C^2. This may seem excessive, but in practice neither P nor C grow very large—typical values are in the range 30–150 for P and 256–1024 for C. For our benchmarks and evaluation environment in Section 5, the running time of the algorithm varied between tens of seconds and a few minutes.

5 Experimental evaluation

In this section, we compare three different procedure-placement algorithms. In addition to PH and our algorithm (GBSC), we present results for a recently published procedure-placement algorithm, an algorithm by Hashemi et al. [5] which we refer to as HKC. Like our algorithm, HKC also extends PH to use knowledge of the procedure sizes, the cache size, and the cache organization. HKC uses a weighted call graph but not any additional temporal information. The key advantage of HKC over PH is that HKC records the set of cache lines occupied by each procedure during placement, and it tries to prevent overlap between a procedure and any of its immediate neighbors in the call graph. In addition, already mapped procedures are allowed to move in the mapping as long as the new location's cache lines do not conflict with prior decisions.

5.1 Evaluating code placement algorithms

We normally expect code optimizations to behave similar 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 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.

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, PH, HKC, and GBSC greedily choose 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 not distributed with more than a handful of input sets for each benchmark application.

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. 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 $\widehat{w}$ according to the equation $\widehat{w} = w \cdot \exp(sX)$, 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.

A large enough value for s will cause the layout to be effectively random, as the perturbed graphs will bear little

Program Name	All procedures		Popular procedures		Training trace		Testing trace		Miss rate of default layout	Average Q size
	size	count	size	count	input	length	input	length		
gcc	2277 K	2005	351 K	136	recog.i	33 M	global.i	45 M	4.86%	11.8
go	590 K	3221	134 K	112	11x11 board, level 4, no stones	20 M	9x9 board, level 6, 4 stones	17 M	3.34%	16.0
ghostscript	1817 K	372	104 K	216	14-page presentation	37 M	3-page paper	38 M	2.63%	18.7
m88ksim	549 K	460	21 K	31	dcrand, limited to 50M basic blocks	50 M	dhry, limited to 50M basic blocks	50 M	2.92%	8.5
perl	664 K	271	83 K	36	scrabbl.pl, reduced dictionary	77 M	primes.pl, reduced input file	146 M	4.19%	7.1
vortex	1073 K	923	117 K	156	persons.250, reduced iterations	42 M	persons.1k, reduced iterations	82 M	6.29%	26.4

Table 1: Details of our benchmark applications. We report sizes in bytes and trace lengths in basic blocks. A benchmark's "average Q size" reports the average number of procedures that were present in Q during the building of the TRG.

relationship to the profile data. Low 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 [2] 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, and that values of s as high as 2.0 do not degrade the average performance very much.

5.2 Methodology

We have implemented the PH, HKC, and GBSC procedure-placement algorithms such that they can be integrated into two different environments: a simulation environment based on ATOM [10]; and a compiler environment based on SUIF [9]. The results in Section 5.3 are based on the ATOM environment, but we have used the SUIF environment to verify that our algorithms produce runnable, correct code.

Table 1 lists the benchmarks used in our study. Except for *ghostscript*, they are all from the SPECint95 benchmark suite. We use only five of the eight SPECint95 benchmarks because the other three (*compress*, *ijpeg*, and *xlisp*) are uninteresting in that all have small instruction working sets that do equally well under any reasonable procedure-placement algorithm. We compiled go and perl using the SUIF compiler (version 1.1.2), while all other benchmarks were compiled using gcc 2.7.2 with the -O2 optimization flag. We chose the input data sets to keep the traces to a manageable size. All of the reported miss rates in this and the next section are based on the simulation of an 8 kilobyte direct-mapped cache with a line size of 32 bytes. We use the training input to drive the procedure-placement algorithms, and then simulate the instruction-cache performance of the resulting optimized executable using the testing input. We also experimented with smaller cache sizes and obtained similar results.

5.3 Results

The graphs in Figure 3 show our experimental results for PH, HKC, and GBSC. Each graph shows the results for a single benchmark. For each of the three algorithms, there is a set of sorted points showing the results over a set of 40 experiments, all based on our single pair of training and testing data sets. As described in Section 5.1, we use randomization to obtain 40 slightly different WCGs or TRGs that may result in slightly different placements. For each point, the X-coordinate reports the cache miss rate for that experiment while the Y-coordinate gives the fraction of all placements that had an equal or smaller miss rate. Consequently, if the points for one algorithm are to the left of the points for another algorithm, then the first algorithm gives better results. We notice that our algorithm gives clearly better results than the other two for all benchmarks except for *m88ksim* and *perl*. For these two benchmarks, the ranges of results overlap. Each graph also contains a table reporting the miss rate obtained by each technique without any randomization. Except for *m88ksim*, GBSC yields the lowest miss rate. In *m88ksim*, *dcrand* is a poor training set for *dhry*, and thus few conclusions can be drawn from the non-randomized results. The miss rates for train/test same (*dcrand*) are: 0.13% for GBSC, 0.19% for HKC, 0.23% for PH.

In Section 3, we said that a useful conflict metric should be strongly correlated with the number of cache misses. Figure 6 examines this issue by showing the relationship between conflict-metric values and cache miss rates. Each plot in Figure 6 contains 80 points, where each point corresponds to a different placement of the *go* benchmark. These placements are based on the GBSC algorithm; however we varied the output of this algorithm

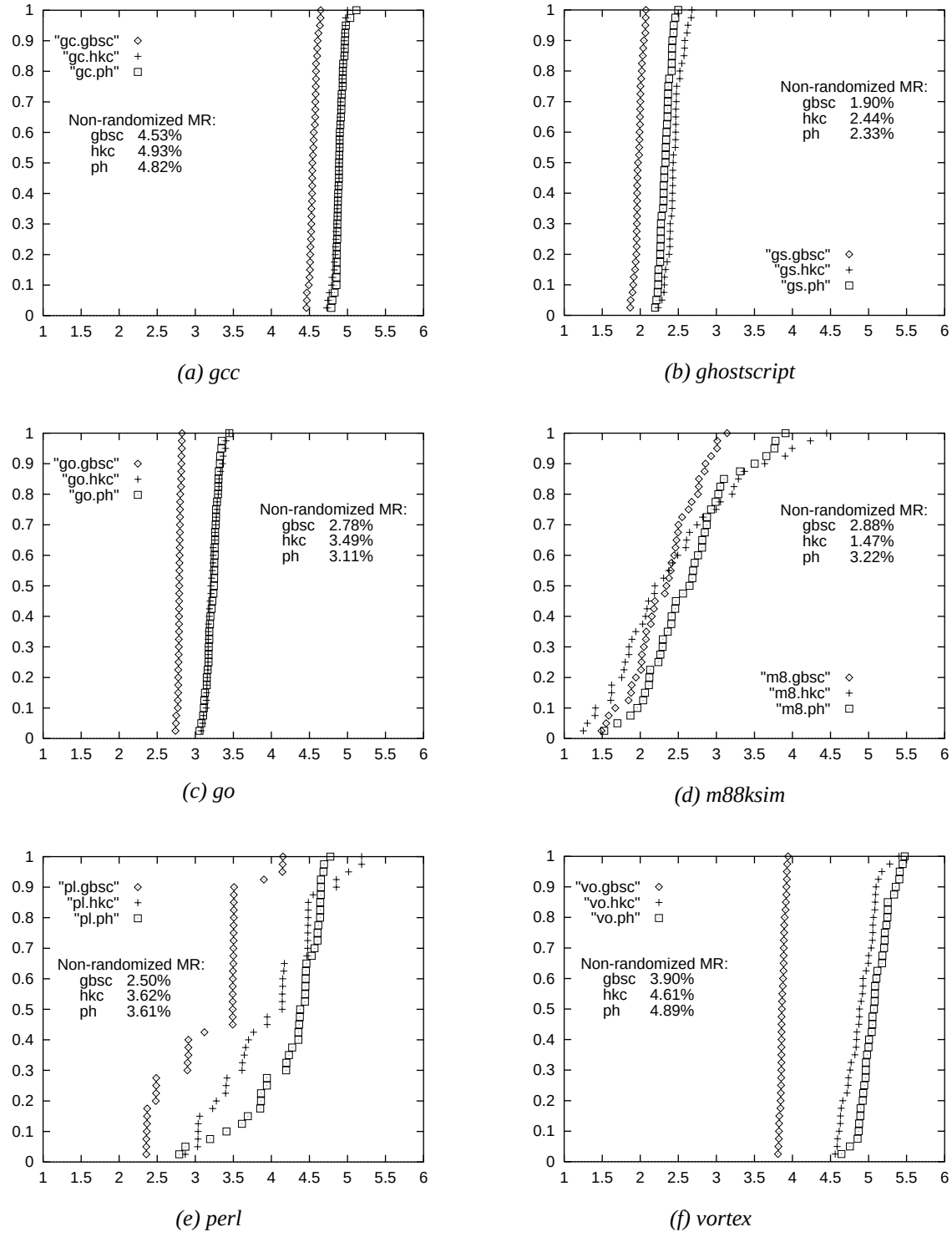


Figure 5. Instruction cache miss rates for our benchmarks. Each graph plots the miss rates corresponding to the layouts produced by PH, HKC, and our new procedure-placement algorithm (GBSC). Each data point in the graphs represents the result for a single placement based on the random perturbation of the profile data. Cache miss rates vary along the x-axis, and the y-axis gives the fraction of all placements that had an equal or smaller miss rate. We also report the miss rate (MR) for each algorithm using non-perturbed profile data.

to produce a placement with a range of different miss rates. We accomplished this by randomly selecting 0–50 procedures in the GBSC placement and randomly changing their cache-relative offsets. The metric value plotted corresponds to the resulting placement. The top of Figure 6 shows that our conflict metric, based on the fine-grained information in TRG_{place}, shows a linear relationship with the actual number of cache misses; all the points in the graph are close to the diagonal. On the other hand, the bottom of Figure 6 shows that a metric based only on a WCG is not always a good predictor of cache misses.

6 Extensions for set-associative caches

We have described a code-placement technique that targets direct-mapped caches. In particular, we assumed that a single occurrence of a procedure *q* between two occurrences of a procedure *p* is sufficient to displace *p*. This assumption is not necessarily true for set-associative caches, especially for those that implement a LRU policy. To extend our approach for set-associative caches, we construct a new data structure that replaces TRG_{place}, and we modify the cost-metric calculation in *merge_nodes*. This section focuses on 2-way set-associative caches; the implementation for other associativities follows directly.

Instead of a graph representation for TRG_{place}, it is now more convenient to view the temporal-relationship structure as a database *D* that records the number of times that a code-block pair {*r*,*s*} appears between consecutive occurrences of another code block *p* in a program trace. We can still use our ordered set approach to build this database. However, when we process the temporal associations related to the next code block *p* in the trace, we associate *p* with all possible selections of two identifiers from the identifiers currently in *Q* (up to any previous occurrence of *p* as before). We do this because two unique references are required to guarantee no reuse. Thus, the database simply records the frequency of each association between *p* and the pair {*r*,*s*}, accessed as *D*(*p*,{*r*,*s*}). If *r*, *s*, and *p* all occupy the same set in a two-way set-associative cache, then we estimate that *D*(*p*,{*r*,*s*}) of the program references to *p* will result in cache conflicts due to the displacement of *p* by intervening references to **both** *r* and *s*. We use this information in *merge_nodes* to check the cost of the association between a code block in node *n1* against all pairs of code blocks in *n2* and vice-versa. While implementing this extension, we found it also necessary to modify some of the other heuristics that were relatively unimportant for direct-mapped caches but were found to be important for procedure placement in set-associative caches.

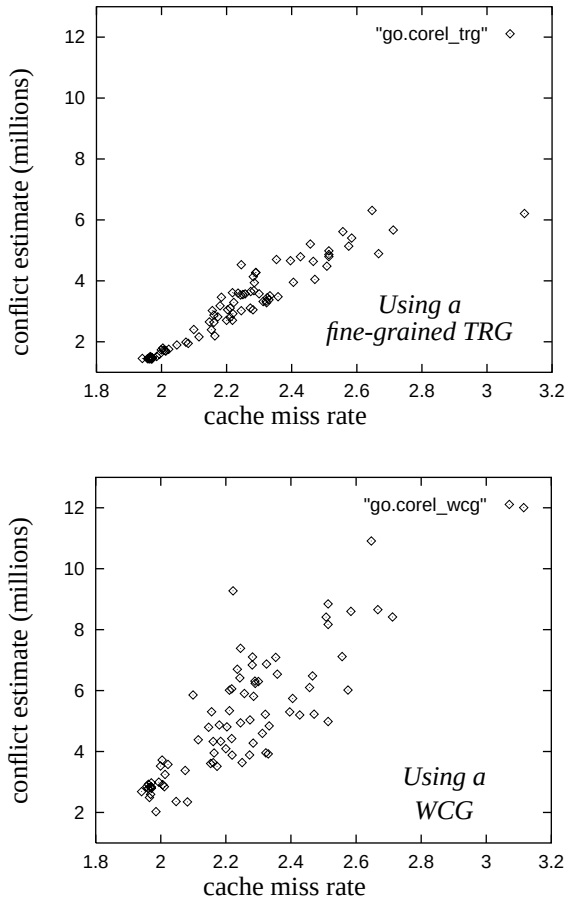


Figure 6. *Correlation between conflict metric and cache misses. Data points are 80 randomized layouts for the* go *benchmark. The X-coordinate of a point is the cache miss rate for that layout; the Y-coordinate is the sum of the conflict metrics for the indicated method over the entire placement.*

7 Related work

Much of the prior work in the area of compile-time code placement is related to early work in reducing the frequency of page faults in the virtual memory system and more recent work at reducing the cost of pipeline penalties associated with control transfer instructions. However, we limit our discussion here to studies that directly address the issue of code placement aimed at reducing instruction cache conflict misses. Some of the earliest work in this area was done by Hwu and Chang [6], McFarling [7], and Pettis and Hansen [8]. 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 [7] 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, this paper 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 from the instruction cache.

Torrellas, Xia, and Daigle [11] propose a code-placement technique for kernel-intensive applications. 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 sub-area 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.

The history mechanism we use to analyze the temporal behavior of a trace is similar to the problem of profiling paths in a procedure call graph. Ammons et al. [1] 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 sufficient temporal ordering information.

8 Conclusion

We have presented a practical method for extracting temporal ordering information from a trace. We then described a procedure-placement algorithm that uses this information along with the knowledge of the cache lines each procedure occupies to predict accurately which placements will result in the least number of conflict misses. The results show that these two factors combined allow us to obtain better instruction cache miss rates than previous procedure-placement techniques. Other code-placement techniques, such as "procedure splitting" [8] and branch alignment [12], are orthogonal to the problem of placing whole procedures and can therefore be combined with our technique to achieve further improvements. The success of our experiments indicates that it is worthwhile to continue research on the temporal behavior of applications. In particular, we plan to develop similar techniques to optimize the behavior of applications in other layers of the memory hierarchy.

9 Acknowledgments

We would like to thank the anonymous reviewers for providing helpful comments. Trevor Blackwell is funded in part by BNR's External Research program. Brad Calder is funded in part by a UC MICRO grant no. 97-018 and a Digital Equipment external research grant no. US-0040-97. Michael D. Smith is funded in part by a NSF Young Investigator award (grant no. CCR-9457779), a DARPA grant no. NDA904-97-C-0225, and research gifts from AMD, Digital Equipment, HP, and Intel.

10 References

[1] G. Ammons, T. Ball, and J. Larus. "Exploiting Hardware Performance Counters with Flow and Context Sensitive Profiling," *Proc ACM SIGPLAN'97 Conf. on Programming Language Design and Implementation*, pp. 85-96, June 1997.

[2] T. Blackwell. "Applications of Randomness in System Performance Measurement." Ph.D. Thesis, Department of Computer Science, Harvard University, 1997.

[3] W. Chen, P. Chang, T. Conte, and W. Hwu. "The Effect of Code Expanding Optimizations on Instruction Cache Design," Technical Report CRHC-91-17, Coordinated Science Lab, University of Illinois, Urbana, IL, April 1991.

[4] N. Gloy, M. Smith, and C. Young. "Performance Issues in Correlated Branch Prediction Schemes," *Proc. 28th Annual IEEE/ACM Intl. Symp. on Microarchitecture*, pp. 3-14, November 1995.

[5] A. Hashemi, D. Kaeli, and B. Calder. "Efficient Procedure Mapping Using Cache Line Coloring," *Proc ACM SIGPLAN'97 Conf. on Programming Language Design and Implementation*, pp. 171-182, June 1997.

[6] W. Hwu and P. Chang. "Achieving High Instruction Cache Performance with an Optimizing Compiler," *Proc. 16th Annual Intl. Symp. on Computer Architecture*, pp. 242–251, May 1989.

[7] S. McFarling. "Program Optimization for Instruction Caches," *Proc. Third Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pp. 183–191, April 1989.

[8] K. Pettis and R. Hansen. "Profile Guided Code Positioning," *Proc. ACM SIGPLAN'90 Conf. on Programming Language Design and Implementation*, pp. 16–27, June 1990.

[9] M. Smith. "*Extending SUIF for Machine-dependent Optimizations*," *Proc. First SUIF Compiler Workshop*, Stanford, CA, pp. 14–25, January 1996.

[10] A. Srivastava and A. Eustace, "ATOM: A System for Building Customized Program Analysis Tools," *Proc. SIGPLAN '94 Conf. on Programming Language Design and Implementation*, pp. 196–205, June 1994.

[11] J. Torrellas, C. Xia, and R. Daigle. "Optimizing Instruction Cache Performance for Operating System Intensive Workloads," *Proc. First Intl. Symp. on High-Performance Computer Architecture*, pp. 360–369, January 1995.

[12] C. Young, D. Johnson, D. Karger, and M. Smith. "Near-Optimal Intraprocedural Branch Alignment," *Proc ACM SIGPLAN'97 Conf. on Programming Language Design and Implementation*, pp. 183-193, June 1997.