Analysis of Temporal-Based Program Behavior for Improved Instruction Cache Performance

John Kalamatianos, *Student Member*, *IEEE*, Alireza Khalafi, *Student Member*, *IEEE*, David R. Kaeli, *Member*, *IEEE*, and Waleed Meleis, *Member*, *IEEE*

Abstract—In this paper, we examine temporal-based program interaction in order to improve layout by reducing the probability that program units will conflict in an instruction cache. In that context, we present two profile-guided procedure reordering algorithms. Both techniques use *cache line coloring* to arrive at a final program layout and target the elimination of *first generation* cache conflicts (i.e., conflicts between caller/callee pairs). The first algorithm builds a call graph that records local temporal interaction between procedures. We will describe how the call graph is used to guide the placement step and present methods that accelerate cache line coloring by exploring aggressive graph pruning techniques. In the second approach, we capture global temporal program interaction by constructing a *Conflict Miss Graph (CMG)*. The CMG estimates the worst-case number of misses two competing procedures can inflict upon one another and reducing higher generation cache conflicts. We use a pruned CMG graph to guide cache line coloring. Using several C and C++ benchmarks, we show the benefits of letting both types of graphs guide procedure reordering to improve instruction cache hit rates. To contrast the differences between these two forms of temporal interaction, we also develop new characterization streams based on the Inter-Reference Gap (IRG) model.

Index Terms—Instruction caches, program reordering, temporal locality, conflict misses, graph coloring, graph pruning.

1 Introduction

Cache memories are found on most microprocessors designed today. Caching the instruction stream can be very beneficial since instruction references exhibit a high degree of spatial and temporal locality. Still, cache misses will occur for one of three reasons [1]:

1. first time reference,
2. finite cache capacity, or
3. memory address conflict.

Our work here is focused on reducing memory address conflicts by rearranging a program on the available memory space. Analysis of program interaction can be performed at a range of granularities, the coarsest being an individual procedure [2]. We begin by considering the procedure Call Graph Ordering (CGO) associated with a program. The CGO captures *local* temporal interaction by weighting its edges with the number of times one procedure follows another during program execution.

We also consider the interaction of basic blocks contained within procedures by identifying the number of cache lines touched by each basic block in our Conflict Miss Graph (CMG). We do not attempt to move basic blocks or split procedures [3] though. We weight CMG edges by measuring global temporal interaction between procedures occurring in a finite window containing as many entries as there are cache lines. Program interaction outside this window is not of interest because of the finite cache effect. We use these graphs as input to our coloring algorithm to produce an improved code layout for instruction caches.

• *The authors are with the Department of Electrical and Computer Engineering, Northeastern University, Boston, MA 02115. E-mail: kaeli@ece.neu.edu.*

For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number 108231.

To characterize the temporal behavior captured by these graphs, we extend the Inter-Reference Gap (IRG) model [4]. We define three new IRG-based streams that describe different levels of procedure-based temporal interaction. We show how we can use them to compare the temporal content between the CGO, CMG, and the *Temporal Relationship Graph* (TRG) (as described by Gloy et al. in [5]).

There has been a considerable amount of work done on code repositioning for improved instruction cache performance [3], [5], [6], [7], [8], [9], [10]. In the following section, we discuss some of this work as it relates to our work here.

1.1 Related Work

Pettis and Hansen [3] employ procedure and basic block reordering, as well as procedure splitting based on frequency counts to minimize instruction cache conflicts. The layout of a program is directed by traversing call graph edges in decreasing edge weight order using a closest-is-best placement strategy. Chains are formed by merging nodes, laying them out next to each other until the entire graph is processed.

A number of related techniques have been proposed, focusing on mapping loops [6], operating system code [10], *traces* [9], and *activity sets* [7]. Two other approaches discussed in [8] and [11] reorganize code based on compile-time information.

The profile-guided algorithms described above use calling frequencies to weight a graph and guide placement [3], [6], [9], [10]. Our first approach also uses calling frequencies, but improves performance by intelligently placing procedures in the cache by coloring cache lines. The second algorithm described in this paper captures global temporal information and attempts to minimize

```
Input: graph G(P,E), P = all procedures (nodes),
E = all edges;
Sort E in descending order based on weight;
Eliminate all E < threshold T;

FOR_EACH (remaining edge between procedures Pi and Pj) {

  SELECT(state of Pi, state of Pj)

  CASE I: (Pi is unmapped and Pj is unmapped) {
    Arbitrarily map Pi and Pj, forming a
    compound node Pij
  } END_CASEI
  CASE II: (both Pi and Pj are mapped, but they reside
            in two different compound nodes Ci and Cj) {
    Concatenate the two compound nodes Ci and Cj,
    minimizing the distance between Pi and Pj;
    If there is a color conflict, shift Ci in the
    color space until there is no conflict;
    If a conflict can not be avoided, return Ci to
    its original position;
  } END_CASEII
  CASE III: (either Pi or Pj is mapped, but not both) {
    same as CASE II
  } END_CASEIII
  CASE IV: (both Pi and Pj are mapped and they belong
            to the same compound node Ck) {
    If there is no conflict, return them in position;
    Else {
       Move the procedure closest to an end of the
       compound node Ck (e.g. Pi) to that end of Ck (outside
       of the compound node);
       If there is still a conflict, shift Pi in the color
        space until no conflict occurs with Pj;
       If conflict can not be avoided, leave Pi to its
         original position;
    } END_ELSE
  } END_CASEIV
  END_SELECT
 Update the unavailable set of colors;

} END_FOR_EACH
Fill in holes created by CASES II, III, and IV;
```

Fig. 1. Pseudocode for our cache line coloring algorithm.

conflicts present between procedures that do not immediately follow each other during execution. It is similar in spirit with the approach described in [5], with some differences that will be highlighted in Section 5. Also, our graph coloring algorithm works at a finer level of granularity (cache line size instead of cache size [6], [11]), and can avoid conflicts encountered when either forming chains with the closest-is-best heuristic [3] or dealing with subgraphs having a size larger than the cache.

This paper is organized as follows. In Sections 2 and 3, we describe our graph construction algorithms. Section 2 describes an improved graph pruning technique. In Section 4, we report simulation results. Section 5 reviews the IRG temporal analysis model and presents new methods for characterizing program interaction.

2 Call Graph Ordering

Program layout may involve two steps: 1) Constructing a graph-based representation of the program and 2) using the graph to perform layout of the program on the available memory space. A Call Graph is a procedure graph having edges between procedures that call each other. The edges are weighted with the call/return frequency captured in the program profile. Each procedure is mapped to a single vertex, with all call paths between any two procedures condensed into a single edge between the two vertices in the graph. Edge weights can be derived from profiling information or estimated from the program control flow [8], [12]. In this paper, we concentrate on profile-based edge weights.

After constructing the Call Graph, we lay out the program using cache line coloring. We start by dividing the cache into a set of colors, one color for each cache line. For each procedure, we count the number of cache lines needed to hold the procedure, record the cache colors used to map the procedure, and keep track of the unavailable-set of colors (i.e., the cache lines where the procedure should not be mapped to).

We define the popular procedure set as those procedures which are frequently visited. The popular edge set contains the frequently traversed edges. The rest of the procedures (and edges) will be called *unpopular*. Unpopular procedures are pruned from the graph. Pruning reduces the amount of work for placement and allows us to focus on the procedures most likely to encounter misses. A discussion of the base pruning algorithm used can be found in [2].

Note, there is a difference between popular procedures and procedures that consume a noticeable portion of a program's overall execution time. A time consuming procedure may be labeled unpopular because it rarely switches control flow to another procedure. If a procedure rarely switches control flow, it causes a small number of conflict misses with the rest of the procedures.

The algorithm sorts the popular edges in descending edge weight order. We then traverse the sorted popular edge list, inspect the state (i.e., mapped or unmapped) of the two procedures forming the edge and map the procedures using heuristics. Fig. 1 provides a pseudocode description of the cache color mapping. A more complete description can be found in [2]. This process is repeated until all of the edges in the popular set have been processed. The unpopular procedures fill the holes left from coloring using a simple depth-first traversal of the unpopular edges joining them.

The algorithm in Fig. 1 assumes a direct-mapped cache organization. For associative caches, our algorithm breaks up the address space into chunks, equal in size to $(number\ of\ cache\ sets * cache\ line\ size)$. Therefore, the number of sets represents the number of available colors in the mapping. The modified color mapping algorithm keeps track of the number of times each color (set) appears in the procedure's unavailable-set of colors. Mapping a procedure to a color (set) does not cause any conflicts as long as the number of times that color (set) appears in the unavailable-set of colors is less than the degree of associativity of the cache.

Next, we look at how to efficiently eliminate a majority of the work spent on coloring by using an aggressive graph pruning algorithm.

2.1 Pruning Rules for Procedure Call Graphs

Pruning a call graph is done using a fixed threshold value (selected edge weight) [2]. In this section, we present pruning rules that can reduce the size of the graph that is

```
C = number of lines in the cache;
N = number of procedures P (nodes) in graph G;
NodeWeight_i = Sum of all incident edges on
procedure Pi in Graph G;
Sort procedures based on increasing
NodeWeight order;

DO_WHILE (at least one procedure is pruned) {
   FOR_EACH unpruned procedure Pi {
      num_i = number of neighbors of Pi;
      size_i = number of cache lines comprising Pi;
      sum_size_i = sum of the sizes of the num_i
                   neighbors of Pi;
      C* = (num_i * (size_i - 1)) + sumsize_i;
      if (C* < C) {
        Prune procedure Pi and all edges
        incident on Pi;
        N--;
      } END_IF
   } END_FOR_EACH
} END_DO_WHILE
```

Fig. 2. Pseudocode for our $C*$ pruning rule

used in cache coloring. They are specifically designed to reduce the number of first-generation cache conflicts.

We assume we are using a direct-mapped cache containing C cache lines. The program is represented as an undirected graph (P, E) where nodes $i \in P$ represent procedures and each edge $(i, j) \in E$ represents a procedure call in the program. The number of cache lines spanned by each procedure is $size_i$. For each edge (i, j), $weight[i, j]$ is the number of times procedures i and j follow one another in the control flow (in either order).

A procedure mapping M is an assignment of each procedure i to $size[i]$ adjacent cache lines within the cache (with wraparound). The cost of a procedure mapping is the sum of all $weight[i, j]$ for all procedures i and j such that $(i, j) \in E$ and i and j overlap in the cache. An optimal mapping is one that is less costly than any other mapping.

Note that the cost of a mapping depends only on the number of immediately adjacent procedures whose mappings in the cache conflict. Conflicts between procedures that do not call one another are not considered. Furthermore, the cost of assigning two adjacent procedures i and j to conflicting cache lines is a constant, equal to the number of conflicts, even though the actual number of replaced cache lines may be smaller.

2.2 C^* Pruning Rule

Consider a cache mapping problem P. It is possible to determine in some cases that a particular node i will be able to be mapped to the cache without causing any conflicts, regardless of where in the cache all adjacent nodes are eventually mapped. In this case, i and all edges connected to i can be deleted from the graph, creating a new cache mapping problem P' with one less node. Fig. 2 provides pseudocode for our pruning algorithm.

This pruning rule is a generalization of the rules described in [13]to perform graph coloring. The graph coloring problem is to assign one of K colors to the nodes of the graph such that adjacent nodes are not assigned the same color. If a node has $K - 1$ neighbors, it can be deleted because, regardless of how its neighbors are eventually colored, there will definitely be at least one color left over that can be assigned to it. The deleted nodes are then colored in the reverse order of their deletion.

The remaining (nonprunable) graph is passed to our coloring algorithm. Once coloring has been performed, each pruned node must be mapped. The nodes are laid out in the opposite order of their deletion.

3 Conflict Miss Graphs

Next, we consider cache misses which can occur between procedures many procedures away in the call graph, as well as on different call chains [14]. We capture temporal information by weighting the edges of a procedure graph with an estimation of the worst case number of conflict misses that can occur between any two procedures. We then use the graph to apply cache line coloring to place procedures in the cache address space. We call this graph a *Conflict Miss Graph* (CMG).

The complete algorithm is described in [14]. We summarize it here and will contrast it with the CGO in Section 5, using Inter-Reference Gap analysis.

3.1 Conflict Miss Graph Construction

The CMG is built using profile data. We assume a worst-case scenario where procedures completely overlap in the cache address space every time they interact. Given a cache configuration, we determine the size of a procedure P_i in cache lines. We also compute the number of unique cache lines spanned by every basic block executed by a procedure, l_i. We identify the first time a basic block is executed, and label those references as *globally unique* accesses, gl_i.

The CMG is an undirected procedure graph with edges being weighted according to our worst case miss model [14]. The edge weights are updated based on the contents of an N-entry table, where N is the number of cache lines. The table is fully-associative and uses an LRU replacement policy. Every entry (i.e., cache line) in the table is called *live*. A procedure that has at least one live cache line is also called live. When P_i is activated, we update the edge weights between P_i and all procedures that have at least one live cache line *and* were activated since the last activation of P_i (if this last activation is captured in the LRU table). The LRU table allows us to estimate the finite cache effect.

We increment the CMG edge weight between P_i and a live procedure P_j by the minimum of: 1) the accumulated number of unique live cache lines of P_j (since P_i's last occurrence) and 2) the number of unique cache lines of P_i's current activation (excluding cold-start misses). A detailed example of updating CMG edge weights can be found in [14].

CMG edge weights are more accurate than CGO edge weights because CGO edge weights do not record 1) the number of cache lines that may conflict per call, and 2) the interaction between procedures that do not directly call each other.

4 Experimental Results

We use trace-driven simulation to quantify the instruction cache performance of the resulting layouts. Traces are generated using ATOM [15] on a DEC 3000 AXP workstation running Digital Unix V4.0. All applications are

TABLE 1
Attributes of Traced Applications

Program	Input	Instr. in million	Exe Size in KB	# Static Procs	Pop Procs		Unpop Procs
					% Exe Size	% Procs	% Exe Size
perl	primes	12	512	671	4.9 (25.1K)	5.2	3.5 (18.1K)
flex	fixit	19	112	170	14.8 (16.6K)	17.6	6.5 (7.3K)
bison	objparse	56	112	158	22.4 (25.1K)	22.1	6.1 (6.9K)
pC++2dep	sample	19	480	665	9.5 (45.7K)	16.3	10.3 (49.5K)
f2dep	f77_3	33	456	760	7.7 (35.2K)	9.0	4.4 (20.2K)
dep2C++	sample	31	560	1338	4.8 (27.1K)	1.7	1.7 (9.5K)
gs	tiger	34	496	1410	12.9 (64.0K)	11.2	9.3 (46.3K)
ixx	layout	48	472	1581	5.7 (27.2K)	5.1	2.4 (11.7K)

The attributes include the number of executed instructions, the application executable size, the number of static procedures, the percentage of program's size occupied by popular procedures, the percentage of procedures that were found to be popular, and the percentage of unactivated procedures that were used to fill memory gaps left after applying coloring.

TABLE 2
Instructon Cache Performance for Static DFS, CGO, and CMG-Based Reordering

Program	I-Cache Miss Rate			Reduction		# I-Cache Misses		
	DFS	CGO	CMG	DFS	CGO	DFS	CGO	CMG
perl	4.72%	4.60%	3.77%	.95	.83	588,123	572,650	469,329
flex	0.53%	0.45%	0.45%	.08	.00	100,488	85,538	85,478
bison	0.04%	0.04%	0.05%	-.01	-.01	21,798	21,379	26,792
pC++2dep	4.72%	5.46%	3.68%	1.04	1.78	895,261	1,035,639	698,003
f2dep	4.98%	4.71%	4.18%	.80	.53	1,680,610	1,588,516	1,411,639
dep2C++	3.92%	3.46%	3.11%	.81	.35	1,205,076	1,063,682	957,102
gs	3.45%	2.09%	2.08%	1.37	.01	1,176,335	712,230	709,643
ixx	5.83%	4.42%	2.57%	3.26	1.85	2,843,330	2,154,747	1,251,022
Avg.	3.52%	3.15%	2.48%					

Column 1 lists the application. Columns 2-4 show the instruction miss rates. The next two columns show the percent improvement over each by our algorithm. The last three columns show the number of instruction cache misses.

compiled with the DEC C V5.2 and DEC C++ V5.5 compilers. The same input is used to train the algorithm and gather performance results. We simulated an 8KB, direct-mapped, instruction cache with a 32-byte line size (similar in design to the DEC Alpha 21064 and 21164 instruction caches). Our benchmark suite includes perl from SPECINT95, flex (a generator of lexical analyzers), gs (a ghostscript postscript viewer), and bison (a C parser generator). It also includes PC++2dep (C++ front-end written in C/C++), f2dep (a Fortran front-end written in C/C++), dep2C++ (a C/C++ program translating Sage internal representation to C++ code), and ixx (an IDL parser written in C++).

Table 1 presents the static and dynamic characteristics of the benchmarks. Column 2 shows the input used to both test and train our algorithms. Columns 3-5 list the total number of instructions executed, the static size of the application in kilobytes, and the number of static procedures in the program. Column 6 presents the percentage of the program that contains popular procedures in the CMG, while column 7 contains the percentage of procedures that were found to be popular (for CMG). The last column presents the percentage of unactivated procedures used to fill in the gaps left from the color mapping.

To prune the CMG graph, we form the popular set from those procedures that are connected by edges that contribute up to 80 percent of the total sum of edge weights in the CMG [14]. Notice that the pruning algorithm reduces the size of the CMG by 80-97 percent, and reduces the size of the executable by 77.7-94.5 percent of the executable. This allows us to concentrate on the important procedures in the program.

4.1 Simulation Results

We compare simulation results against the ordering produced by the DEC compiler (static DFS ordering of procedures) and CGO using a fixed threshold value for pruning (no aggressive pruning was employed).

Table 2 shows the instruction cache miss rates. In all cases, the same inputs were used for both training and testing. The first column denotes the application, while columns 2-4 (7-9) show the instruction cache miss rates (number of cache misses) for DFS, CGO, and CMG, respectively. Columns 5 and 6 show the relative improvement of CMG over DFS and CGO, respectively.

As we can see from Table 2, the average instruction cache miss rate for CMG is reduced by 30 percent on average over the DFS ordering and by 21 percent on average over the CGO ordering. CMG improves performance against both static DFS and CGO over all benchmarks except bison, flex, and gs. Bison and flex already have a very low miss rate and no further improvement can be achieved. Gs has a large number of popular procedures that cannot be mapped in the cache with significant reduction of the miss rate.

Next, we apply the C^* pruning rule to CGO for four benchmarks, bison, flex, gs, and perl. We have also tried to apply this approach to CMG, but found we were unable to significantly reduce the size of the graph. As shown in

TABLE 3
The Results of Applying the C^* Pruing Rules to Call Graphs for Four Applications

Program	Input	Visited Procs.	Pruned 1st pass	Pruned 2nd pass	Pruned 3rd pass
bison	objparse	125	125	0	0
flex	fixit	97	61	5	2
gs	tiger	532	524	5	0
perl	primes	209	113	4	0

Passes refer to pruning iterations over the graph. The algorithm is finished when no more nodes can be pruned in the graph.

TABLE 4
The Reults of Applying the C^* Pruning Rules to Call Graphs for Four Applications

Program	CGO Miss Rate	first order	higher order	C* Miss Rate	first order	higher order
bison	.04	1316	19812	.14	0	79888
flex	.45	55004	30280	.51	42467	55233
gs	2.09	530908	658066	2.39	25663	791567
perl	4.60	99327	473070	4.45	92914	462056

Cache parameters are the same as those used above.

TABLE 5
Miss Ratios When Using Different Test and Training Inputs

Program	Training input	Test input	Trace instr.	Static DFS Miss rate	CGO Miss rate	CMG Miss rate
bison	objparse	cparse	35.6M (56.1)	0.05%	0.04%	0.06%
f2dep	f77_3	f77_8	17.3M (33.7)	4.40%	4.07%	3.64%
flex	fixit	scan	23.2M (19.1)	0.49%	0.43%	0.38%
gs	tiger	golfer	15.5M (34.1)	3.39%	2.49%	2.51%
ixx	layout	widget	52.7M (48.7)	5.89%	4.45%	2.54%
perl	primes	jumble	71.9M (12.4)	4.36%	4.61%	4.16%

The inputs are described in columns 2-3 while the sizes of their corresponding traces are presented in column 4. Columns 5-7 include the instruction cache miss ratios.

Tables 3 and 4, the pruning rule deletes all 125 nodes from bison and completely eliminates all first-generation conflict misses. Similarly, most of the nodes are pruned from the other benchmarks, accompanied by a significant drop in the number of first-generation conflict misses. However, this drop is, most of the time, followed by an increase in the total number of misses. By deleting nodes and edges that do not contribute to first-generation conflict misses, the coloring algorithm is deprived of information that can be used to prevent higher order misses. In the case of the bison benchmark, the nodes are inserted into the mapping with complete disregard for higher order conflicts.

These results suggest that node pruning rules such as C^* can be useful as part of a cache conflict reduction strategy, but only when paired with other techniques that prevent higher order cache conflicts from canceling out the benefits of reducing first-order conflicts.

4.2 Input Training Sensitivity

Since our procedure reordering algorithm is profile-driven, we tried different training and test input files, as shown in Table 5. Column 2 (3) has the training (test) input, while column 4 shows the size of the traces in millions of instructions for the test and the training inputs (the last one in parentheses). The last three columns present the miss ratios for each of the algorithms simulated.

As we can see from Table 5, although the performance of both the CGO and the CMG approaches drop compared to the simulations using the same inputs, the relative advantage of CMG against CGO and static DFS still remains. In fact, the performance gain is of the same order for all benchmarks, i.e., CGO and CMG achieve similar performance for bison and gs, while CMG significantly improves the miss ratio of ixx.

5 Temporal Locality and Procedure-Based IRG

Next, we characterize the temporal interaction exposed by CGO, CMG, and the Temporal Relationship Graph (TRG) [5] using an extended version of the *Inter-Reference Gap* (IRG) model [4].

5.1 Procedure-Based IRG

In [4], Phalke and Gopinath define the IRG for an address as the number of memory references between successive references to that address. An IRG stream for an address in a trace is the sequence of successive IRG values for that address and can be used to characterize its temporal locality. Similarly, we can measure the temporal locality of larger program granules such as basic blocks, cache lines or procedures. The accuracy of the newly generated IRG

TABLE 6
Frequency Distribution of the IRPG, IRILG, and IRALS Sequences for Two Procedure Pairs (Fourth and 12th in Calling Frequency) of the ixx Benchmark

IRPG value	4th %	12th %	IRILG value	4th % (%)	12th % (%)	IRALS value	4th % (%)	12th % (%)
1	100	41.3	1	0.0 (0.0)	0.0 (0.0)	1	100 (100)	79.3 (52.0)
2	0.0	0.0	2-10	100 (100)	66.1 (62.4)	2	0.0 (0.0)	20.6 (0.0)
3-5	0.0	27.8	10-50	0.0 (0.0)	6.6 (0.0)	3-5	0.0 (0.0)	0.0 (0.0)
> 5	0.0	30.9	> 50	0.0 (0.0)	27.2 (0.0)	> 5	0.0 (0.0)	0.0 (0.0)

stream depends on the interval granularity. In this work, we set the program unit under study to be a procedure while we vary the interval definition.

The original IRG model exploits the temporal locality of a single procedure, but not the temporal interaction between procedures. Therefore, we redefine the IRG value for a procedure pair A, B as the number of unique activated procedures between invocations of A and B. We will refer to this value as the *Inter-Reference Procedure Gap* (IRPG). The CGO edge weights record part of the $IRPG$ stream since they capture the $IRPG$ values of length 1.

In the TRG, every node represents a procedure and every edge is weighted by the number of times procedure A follows B and vice versa *only* when both of them are found inside a moving time window. The window includes previously invoked procedures and its size is proportional to the size of the cache. The window's contents are managed as a LRU queue. The temporal interaction recorded by a TRG can be characterized by the *Inter-Reference Intermediate Line Gap* ($IRILG$) whose elements are equal to the number of unique cache lines activated between successive A and B invocations. The decision of when to update a TRG edge depends on the size of the window or, equivalently, on the values present in the $IRILG$ stream. The edge weight is simply the count of all $IRILG$ elements with a value less than the predefined window size. The TRG captures temporal interaction at a more detailed level than CGO because the $IRILG$ stream is richer in content than the $IRPG$ stream.

The CMG edge weight between procedures A and B is updated only when A and B follow one another inside a moving time window proportional in size to the cache size. A CMG edge weight is updated whenever the $IRILG$ value is less than the window size. In both the TRG and the CMG, procedures interact as long as they are found inside the time window. A CMG, however, replaces procedures in the time window based on the age of individual lines in a procedure [14], while a TRG manages replacement on an entire procedure basis [5].

In addition, a CMG edge weight between A and B is incremented by the minimum of the unique live cache lines of the successive invocations of A and B. The TRG simply counts the number of times A and B follow each other. We define the *Inter-Reference Active Line Set* ($IRALS$) for a procedure pair as the sequence of the number of unique live cache lines referenced between any successive occurrences of A and B. Each $IRALS$ element value is computed according to the *Worst Case Miss* analysis presented in Section 3. A CMG edge weight is equal to the sum of the IRALS values whose corresponding $IRILG$ values are less than the window size.

Table 6 shows the $IRPG$, $IRILG$, and $IRALS$ element frequency distribution for two edges in the ixx benchmark. The selected edges have the fourth and 12th highest calling frequency in the CGO popular edge list and are labeled as e_4 and e_{12}, respectively. We classify the stream values in the ranges shown in columns 1, 4, and 7, and present per stream frequency distributions in columns 2-3, 5-6, and 8-9. The numbers in parentheses indicate how closely CGO approximates the temporal information captured by the stream under consideration. For example, while 66.1 percent of $IRILG$ elements for e_{12} lie in the range between 2 and 10 unique cache lines, only 62.4 percent of them are recorded in the CGO.

The three different approaches to edge weighting have significant impact on the global edge ordering and the final procedure placement.

Fig. 3 compares the ordering of the common popular edges of CGO and CMG in ixx. A point at location (3, 4) means that the popular edge under consideration was found third in CGO and fourth in CMG ordering. Points on the straight line correspond to edges with the same relative position in both edge lists. Points lying above (below) the straight line indicate edges with a higher priority in the CGO (CMG) edge list. Notice that very few edges fall below the straight line due to the artificially inflated edge index in the CMG edge list (which is much larger than the CGO one) and the different pruning algorithms used. Although a lot

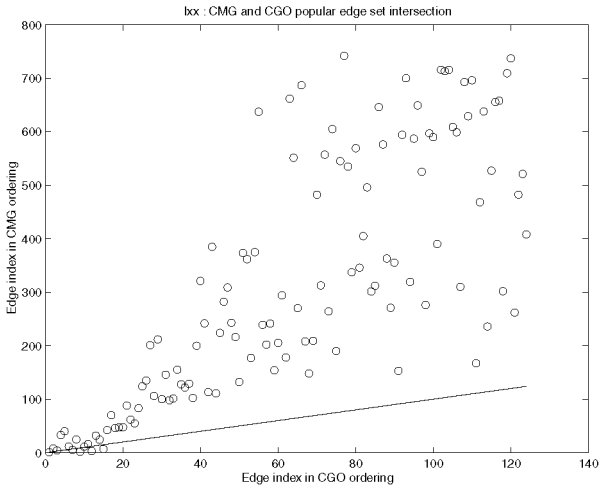

Fig. 3. Relative edge ordering for the intersection of the CMG and CGO popular edge orderings in ixx.

TABLE 7
Intersections of the CMG and CGO Popular Procedure and Edge Sets

Program	CGO pop Procs	CMG pop Procs	CGO ∩ CMG Procs	CGO pop Edges	CMG pop Edges	CGO ∩ CMG Edges
bison	43	35	35	44	67	31
f2dep	19	69	19	15	685	15
flex	28	30	28	26	94	23
gs	94	158	94	105	1478	105
ixx	109	82	82	182	773	124
perl	20	35	20	18	345	18

of highly weighted edges maintained their relative positions, the significant performance improvement for CMG came from edges that were promoted higher in the edge list ordering.

Table 7 shows the intersection between the CMG and CGO popular procedure and edge sets. Columns 2-4 (5-7) list the CGO and CMG popular procedure (edge) sets along with their intersection. The numbers shown in Table 7 are sensitive to the pruning algorithm, but they are compared to better illustrate the differences between the CMG and CGO approach. Although one procedure set is always the superset of the other, the CMG edge list is always larger than the CGO edge list.

6 Conclusions

The performance of cache-based memory systems is critical in today's processors. Research has shown that compiler optimizations can significantly reduce memory latency and every opportunity should be taken by the compiler to do so.

In this paper, we presented two profile-guided algorithms for procedure reordering which take into consideration not only the procedure size but the cache organization as well. While CGO attempts to minimize first-generation conflicts, CMG targets higher generation misses. Both approaches use pruned graph models to guide procedure placement via cache line coloring. The CMG algorithm improved instruction cache miss rates on average by 30 percent over a static depth first search of procedures, and by 21 percent over CGO.

We also introduced three new sequences ($IRPG$, $IRILG$, and $IRALS$) based on the IRG model, to better characterize the contents of each graph model.

Acknowledgments

We would like to acknowledge the contributions of H. Hashemi and B. Calder to this work. This research was supported by the U.S. National Science Foundation CAREER Award Program, by IBM Research, and by Microsoft Research.

References

[1] M.D. Hill and A.J. Smith, "Evaluating Associativity in CPU Caches," *IEEE Trans. Computers,* vol. 38, no. 12, pp. 1,612-1,630, Dec. 1989.

[2] A.H. Hashemi, D.R. Kaeli, and B. Calder, "Efficient Procedure Mapping Using Cache Line Coloring," *Proc. Int'l Conf. Programming Language Design and Implementation,* pp. 171-182, June 1997.

[3] K. Pettis and R. Hansen, "Profile-Guided Code Positioning," *Proc. Int'l Conf. Programming Language Design and Implementation,* pp. 16-27, June 1990.

[4] V. Phalke and B. Gopinath, "An Inter-Reference Gap Model for Temporal Locality in Program Behavior," *Proc. Int'l Conf. Measuring and Modeling Computer Systems,* pp. 291-300, May 1995.

[5] N. Gloy et al. , "Procedure Placement Using Temporal Ordering Information," *Proc. Int'l Conf. Microarchitecture,* pp. 303-313, Dec. 1997.

[6] S. McFarling, "Program Optimization for Instruction Caches," *Proc. Int'l Symp. Architectural Support for Programming Languages and Operating Systems,* pp. 183-191, Apr. 1989.

[7] K. Gosmann et al. , "Code Reorganization for Instruction Caches," *Proc. Hawaii Int'l Conf. System Sciences,* pp. 214-223, Jan. 1993.

[8] A.H. Hashemi, D.R. Kaeli, and B. Calder, "Procedure Mapping Using Static Call Graph Estimation," *Proc. Workshop Interaction Between Compiler and Computer Architecture,* Feb. 1997.

[9] W.W. Hwu and P.P. Chang, "Achieving High Instruction Cache Performance with an Optimizing Compiler," *Proc. Int'l Symp. Computer Architecture,* pp. 242-251, May 1989.

[10] J. Torrellas, C. Xia, and R. Daigle, "Optimizing Instruction Cache Performance for Operating System Intensive Workloads," *Proc. First Int'l Symp. High-Performance Computer Architecture,* pp. 360-369, Jan. 1995.

[11] A. Mendlson, S. Pinter, and R. Shtokhamer, "Compile-Time Instruction Cache Optimizations," *Proc. Int'l Conf. Compiler Construction,* pp. 404-418, Apr. 1994.

[12] D.W. Wall, "Predicting Program Behavior Using Real or Estimated Profiles," *Proc. ACM SIGPLAN '91 Conf. Programming Language Design and Implementation,* pp. 59-70, Toronto, Ontario, Canada, June 1991.

[13] G.J. Chaitin et al. , "Register Allocation and Spilling via Graph Coloring," *Proc. ACM SIGPLAN '86 Symp. Compiler Construction,* pp. 98-105, New York, 1982.

[14] J. Kalamatianos and D.R. Kaeli, "Temporal-Based Procedure Reordering for Improved Instruction Cache Performance," *Proc. Int'l Conf. High Performance Computer Architecture,* Feb. 1998.

[15] Digital Equipment Corporation, *ATOM User Manual,* Mar. 1994.

John Kalamatianos received his BS degree in computer engineering and informatics from the University of Patras, Greece, in 1992, and his MS degree in electrical and computer engineering from Northeastern University, Boston, in 1995. He is currently working toward his PhD degree at Northeastern University. His research interests are primarily in the area of computer architecture and compiler optimizations with particular emphasis on cache and memory hierarchy design, branch and value prediction, prefetching, compiler optimizations for ILP, code reordering, and specialization. He is a student member of the ACM and the IEEE.

Alireza Khalafi received the BE degree in electronics engineering from the Polytechnic University of Tehran, Iran, in 1993, and the MS degree in electrical engineering from the University of Tehran, Iran, in 1996. He is currently a PhD candidate at Northeastern University, Boston. From 1994 to 1996, he was a hardware design engineer at the Iranian Power Research Center in Tehran. His research interests include compiler optimization, computer architecture, dynamic profiling, testing and fault simulation, and hardware synthesis. He is a student member of the IEEE.

David R. Kaeli received his BS in electrical engineering from Rutgers University, his MS in computer engineering from Syracuse University, and his PhD in electrical engineering from Rutgers University. He is currently an assistant professor in the Department of Electrical and Computer Engineering at Northeastern University, Boston. Prior to 1993, he spent 12 years at IBM, the last seven of which were at the IBM T.J. Watson Research Center in Yorktown Heights, New York. In 1996, he received a U.S. National Science Foundation CAREER award. He is currently the director of the Northeastern University Computer Architecture Research Laboratory (NUCAR). Dr. Kaeli's research interests include computer architecture and organization, compiler optimization, VLSI design, trace-driven simulation, and workload characterization. HE is a member of the IEEE and the ACM.

Waleed M. Meleis received the BSE degree in electrical engineering from Princeton University in 1990 and the MS and PhD degrees in computer science and engineering from the University of Michigan in 1992 and 1996. He is an assistant professor in the Department of Electrical and Computer Engineering at Northeastern University. His research interests are in combinatorial optimization and its application to computer architecture and compilers.