Published in the Proceedings of HPCA-4, Feb 1-4, 1998 in Las Vegas, Nevada.

Temporal-based Procedure Reordering for Improved Instruction Cache Performance

John Kalamatianos & David R. Kaeli
Northeastern University
Department of Electrical and Computer Engineering
Boston, MA 02115, USA
E-mail: kalamat,kaeli@ece.neu.edu

Abstract

As the gap between memory and processor performance continues to grow, it becomes increasingly important to exploit cache memory effectively. Both hardware and software techniques can be used to better utilize the cache. Hardware solutions focus on organization, while most software solutions investigate how to best layout a program on the available memory space.

In this paper we present a new link-time code reordering algorithm targeted at reducing the frequency of misses in the cache. In past work we focused on eliminating first generation *cache conflicts (i.e., conflicts between a procedure, and any of its immediate callers or callees) based on calling frequencies. In this work we exploit procedure-level temporal interaction, using a structure called a* Conflict Miss Graph (CMG) *. In the CMG every edge weight is an approximation of the worst-case number of misses two competing procedures can inflict upon one another. We use the ordering implied by the edge weights to apply color-based mapping and eliminate conflict misses between procedures lying either in the same or in different call chains.*

Using programs taken from SPEC 95, Gnu applications, and C++ applications, we have been able to improve upon previous algorithms, reducing the number of instruction cache conflicts by 20% on average compared to the best procedure reordering algorithm.

1. Introduction

Cache memories are found on most microprocessors designed today. A cache exploits the temporal and spatial locality present in programs developed in high-level languages. 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: (i) first time reference, (ii) finite cache capacity, or (iii) memory address conflict. Our work here is focused on reducing memory address conflicts. Our goal is to rearrange a program on the available memory space in order to reduce the probability of conflict during run time. While we consider the interaction of basic blocks contained within procedures, we do not attempt to split procedures [11] or rearrange individual basic blocks [4, 11]. Instead, we perform our analysis on the procedure level, identifying pairs of procedures that would compete for the same cache address space if allowed to overlap. The goal is to avoid mapping two highly interacting procedures to the same portion of the cache. As an additional optimization we allow large procedures to overlap with temporally local procedures, allowing the overlap to occur in those portions of a large procedure which are infrequently accessed. We study procedure interaction occurring in a finite window, equal in size to the number of unique cache lines which could possibly live in the cache under study. Procedure interaction for procedures outside of this window is not of interest since it is known that the cache's finite capacity will be exhausted if a reference falls outside of this window.

In this paper we will show how to construct a *Conflict Miss Graph*. We will then use this graph as input to a color-based layout algorithm to produce an improved code ordering for instruction caches.

This paper is organized as follows. In Sections 2 and 3 we describe our graph construction algorithm. In Section 4 we describe our graph layout algorithm. In Section 5 we report simulation results while section 6 suggests further improvements and extensions to our scheme. In Section 7 we review past work on program positioning and we conclude the paper in Sections 8 and 9.

2. Worst Case Miss Analysis

We will begin by discussing what program information is needed to identify the causes of cache conflicts. Past research has focused on the interaction between procedures which directly call one another using their calling frequencies as a metric to determine pairs of procedures competing for the cache. However, misses can occur between procedures many procedures away on a call chain, as well as on different call chains [1]. In addition, the calling frequency does not accurately estimate the potential number of misses between two procedures. We need to capture more global temporal information to be able to address these conflicts.

To better capture this temporal information we estimate the worst case number of conflict misses that can occur between any two procedures. Our approach is not strictly guided by the control flow of the program (e.g. loops, procedure calls, etc.). It rather uses the liveness of a procedure's basic blocks to guide the conflict miss estimation algorithm. This worst case behavior weights edges in a procedure graph. We call this graph a *Conflict Miss Graph* (CMG). We use this graph to place procedures in the cache address space so that the overlap between heavily interacting procedures (or portions of procedures) is minimized. This mapping step is done by employing a cache line coloring algorithm similar to the one introduced in [2]. The CMG edge weights determine the ordering by which procedures will be considered. By utilizing the CMG information we manage to not only accurately predict procedure interaction at a finer level of granularity but explore higher order generation conflicts (as opposed to *first generation conflicts* for algorithms that use call graphs and calling frequencies for their procedure reordering). Cases where higher order conflicts can occur can be found in [9].

Next we describe our algorithm in full, while section 4 contains an description of our color-based procedure placement algorithm.

3. Conflict Miss Graph Construction

To estimate the worst case number of conflict misses that can occur between two procedures, we begin by assuming that on each activation, procedures completely overlap in the cache address space. To compute the resulting number of estimated conflicts, we utilize profile data taken from a trace of an application. Based on the cache configuration (cache size and line size), we determine the number of cache lines each procedure will occupy. We also compute the number of unique cache lines spanned by every basic block executed by a procedure. We identify the first time each basic block is accessed in the trace, and label those references as *globally unique* accesses. A single trace entry contains the procedure name (P_i), the number of unique cache lines accessed during the execution of a basic block (l_i), and the number of (globally unique) cache lines accessed during the execution of a basic block (gl_i). Notice that $gl_i \leq l_i$ for every record and that $gl_i \neq 0$ for only the first activation of each basic block.

The next step is to build the CMG, where every node is a procedure found in the profile, and every edge is weighted according to the worst case miss model. Every edge is undirected and models the temporal interaction between two procedures. Since we are not considering the interaction of basic blocks within a procedure's body, we do not generate self-loop edges (although they could be beneficial for procedures larger than the cache size).

Edge weights are incremented as we work through each record in the profile. There are two important questions raised at this step : (i) which procedures should be considered to interact and (ii) and how do we compute the worst-case number of misses between those procedures? Clearly, updating weights between any two procedures is not necessary because it is very unlikely that procedures would live for the entire execution of the application. Our worst-case model keeps a finite-size LRU stack of cache lines that have been activated so far (we refer to lines in the LRU stack as *live* cache lines). When a procedure P_i appears in the trace, we update the weights between P_i and the procedures that have at least one live cache line *and* were activated since the last activation of P_i in the profile (if this last activation is captured in the LRU stack). The LRU stack models a fully associative cache with the same size as the simulated one and is maintained using an LRU replacement policy. It allows us to estimate the finite cache effect since not all procedures in the program can fit in the instruction cache. We keep track of the number of unique cache lines that are currently active in the cache, and only consider conflicts with those that would fit in a fully associative cache considering a worst-case analysis. This helps us to approximate the liveness of procedures as they move through the cache.

When a procedure P_i is activated, the worst-case number of misses is calculated based on the accumulated number of unique live cache lines of each procedure (since P_i's last occurrence) and the number of unique live cache lines of P_i's current activation (excluding cold-start misses). For the rest of the paper we will denote the number of unique live cache lines of a procedure P_j with respect to P_i (i.e., those that have been activated since the last occurrence of P_i) as $\bigcup_{P_i}^{P_j}$. If P_i was not present in the LRU stack then $\bigcup_{P_i}^{P_j}$ is equal to the total number of unique live cache lines for P_j.

We will demonstrate our estimation model by process-

[1] A call chain is defined as the time-ordered sequence of procedures called.

ing the trace sample shown in Figure 1 assuming that our target cache is a direct mapped with 4 entries. We will show the corresponding state of a 4-entry LRU stack and its corresponding procedure list. The shaded box denotes the most recently activated cache line (i.e., the top of the stack). The profile information (see the first column in Figure 1) has the following four fields:

1. the name of the procedure activated,
2. the number of cache lines associated with this activation,
3. the number of globally unique cache lines associated with this activation, and
4. a list of the cache lines activated (listed in the order in which they were activated).

Next, we will step through the 9 procedure activations to illustrate the details of our algorithm.

On the first procedure activation, no edge weight is updated since P_1 is the only procedure in the procedure list. When P_2 is activated, the weight of the $P_2 - P_1$ edge is incremented by $min(l_2 - gl_2, l_1)$, which represents the following worst-case scenario: P_1 and P_2 are mapped onto the same cache address area, and P_2 executes basic blocks that map onto the exact same cache lines that were previously activated (and which are present in the cache) by P_1. Thus the worst-case number of misses that can occur is equal to the minimum number of accessed cache lines. Since we are interested in conflict misses, we subtract the gl_i value for P_2 to account for cold-start misses. In our example the $P_2 - P_1$ edge weight will not be modified since $min(l_2 - gl_2, l_1) = min(2 - 2, 1) = 0$ (all of the potential misses are cold-start). Notice also that the cache line activated last (P_{2b}) becomes the MRU in the LRU stack to accurately indicate the liveness on a cache line basis.

When P_1 is re-activated in the trace, we update the $P_2 - P_1$ edge weight since both procedures P_1 and P_2 are still live and procedure P_2 has been activated since procedure P_1's last activation. Accordingly, $\bigcup_{P_1}^{P_2} = 2$, and the estimated worst case number of misses is $min(l_1 - gl_1, \bigcup_{P_1}^{P_2}) = min(1-1, 2) = 0$. When procedure P_2 is activated a 2nd time, $\bigcup_{P_2}^{P_1} = 1$ because only one cache line of procedure P_1 was activated since the last occurrence of P_2. Thus the algorithm will increment the $P_2 - P_1$ edge weight by $min(l_2 - gl_2, \bigcup_{P_2}^{P_1}) = min(1 - 0, 1) = 1$. Procedure P_3 is activated next, so the edge weights for both $P_3 - P_2$ and $P_3 - P_1$ edges will be incremented by $min(l_3 - gl_3, \bigcup_{P_3}^{P_2}) = min(1 - 1, 2) = 0$ and by $min(l_3 - gl_3, \bigcup_{P_3}^{P_1}) = min(1 - 1, 1) = 0$, respectively. Notice that $\bigcup_{P_3}^{P_2} = 2$ because P_{2a} and P_{2b} cache lines are still live while $\bigcup_{P_3}^{P_1} = 1$ due to the only live cache line

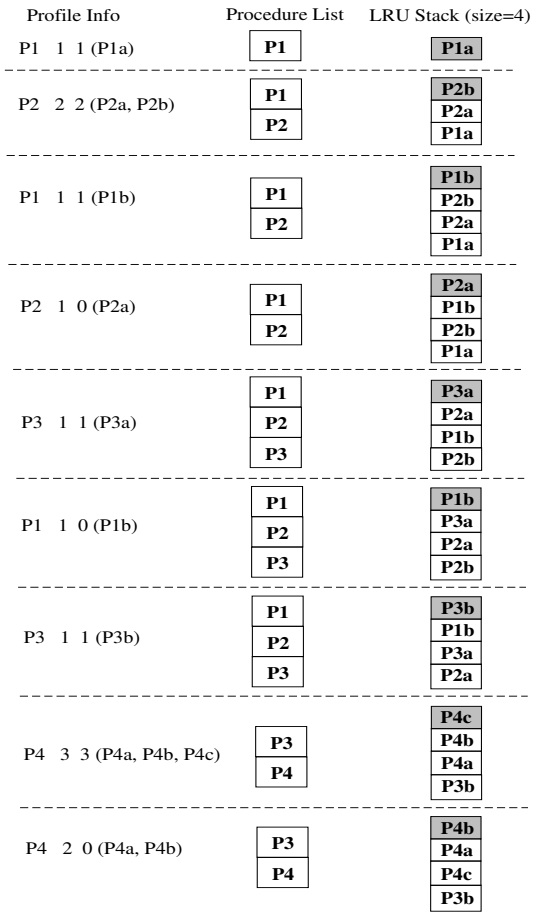

Figure 1. LRU stack and active procedure list contents using an LRU stack of size 4 for the profile information shown at the left column.

P_{1b}. Cache line P_{1a} was replaced from the LRU stack and is not considered. Also, $\bigcup_{P_i}^{P_j}$ is equal to the total number of unique live cache lines of P_j when procedure P_i is not present in the procedure list.

When procedure P_1 is reactivated, edge weights for both $P_3 - P_1$ and $P_2 - P_1$ will be incremented by $min(l_1 - gl_1, \bigcup_{P_1}^{P_3}) = min(1 - 0, 1) = 1$ and $min(l_1 - gl_1, \bigcup_{P_1}^{P_2}) = min(1 - 0, 1) = 1$, respectively. Notice that although procedure P_2 has two live cache lines (i.e., P_{2a} and P_{2b}), only one is considered for the edge weight (P_{2a}) since it is the only one activated by P_2 (the 4th trace entry) during the time window of the current and last appearance of P_1 (the 6th and 3rd trace entries, respectively). The subsequent P_3 activation increases only the $P_3 - P_1$ edge weight by $min(l_3 - gl_3, \bigcup_{P_3}^{P_1}) = min(1-1, 1) = 0$ because procedure P_2 has not been activated during the time span between the last two activations for procedure P_3. Following procedure P_3, the first appearance of procedure P_4 increments only

the $P_4 - P_3$ edge weight by $min(l_4 - gl_4, \bigcup_{P_4}^{P_3}) = min(3 - 3, 1) = 0$ since procedures P_2 and P_1 currently do not have any live cache lines. Finally, at the next occurrence of procedure P_4 we do not update the edge weight for $P_4 - P_3$ since procedure P_3 was not activated since the last activation of procedure P_4.

The motivation behind weighting the edges with the predicted worst case number of conflict misses is that we want to overcome the following limitations of the call graph model used in [2]: (i) only first generation conflicts were captured, (ii) call graph edge weights do not capture the number of cache lines that may conflict per call, and (iii) procedures lying in different call chains were not considered.

The Conflict Miss Graph captures the interaction between procedures lying on either the same, or different, call chain(s). For this reason, it performs a worst-case estimation of the number of conflict misses that can occur for every pair of procedures that may reside in the cache simultaneously. It is important to note that our goal is not to accurately compute the number of misses that can occur between any two procedures. We are rather pursuing a more accurate relative ordering by which procedures will be considered during color mapping.

Since we consider a worst-case scenario for estimating the number of conflict misses, the final values of the edge weights can be under or over the actual number of misses that would happen if we allowed procedures to overlap. Conflict misses between two procedures do not occur every time the control flow switches from one to another, even if these procedures completely overlap in the cache address space. The number of misses between them depends on their sizes, their relative placement and the placement of their activated basic blocks. Thus our edge weights may be pessimistic because we increment an edge weight every time two procedure activations follow each other at runtime. Although we found that this heuristic tends to inflate the edge weights, the ordering of edges *after pruning* is not affected.

Edge weights could also underestimate the number of conflict misses. This is because the LRU stack has a finite size so a procedure may be absent from the procedure list for a certain period and will miss updating its edge weights. Using a fixed mapping though, some of the procedure's cache lines may survive and cause extra misses which are not captured by the LRU stack model. This phenomenon did not affect the procedure ordering *after pruning*.

4. Procedure Placement Algorithm

We next present the cache line coloring algorithm used to layout the Conflict Miss Graph. The first step is to prune the CMG and this is done for the following reasons :

1. we want to concentrate on the highly interacting procedures,
2. we want to reduce the size of the problem, and
3. we may need some fluff procedures which we will use to fill gaps left in the memory space after color mapping.

To prune the graph, we select a threshold value by first sorting the edge weights in descending order, and then starting at the heaviest edge, sum the edge weights until their sum is equal to or greater than 80% of the sum of all edge weights. The value of the last edge added becomes the threshold value. Every edge weighing less than the threshold value is cut. The pruning step reduces the CMG into (usually) one strongly connected subgraph that includes all heavily interacting procedures. We decided to modify the threshold value calculation from the one in [2] because the CMG could not be effectively partitioned due to its high connectivity (in contrast to the call graph in [2], which tends to be easily partitioned into several subgraphs). This happens because the CMG captures much more temporal information than a program call graph. Consequently, procedures that seemed to be unrelated in a call graph, may now be connected.

The partitioned call graph in [2] was mapped in the cache by independently placing each one of its subgraphs without any conflicts. The idea there was to facilitate mapping by considering a subgraph at a time. In our case, although the pruned CMG does not usually fit in the cache, the increased accuracy of its edge weights imply an edge ordering that can avoid more conflict misses after the mapping.

After pruning is performed, any procedure that remains connected to any other procedure is considered a *popular procedure*. Any remaining edges are considered to be *popular edges*. We define *unpopular procedures* and *unpopular edges* as those not included in the popular set, but visited at least once during the program profile. We define *unvisited procedures* as those that do not appear in the program profile.

After the program's popularity has been decided, we process all of the popular edges, working in descending edge weight order. There are four possible cases when processing an edge in our algorithm. All four cases, along with a detailed example, can be found in [2]. The main idea is to merge procedures connected by edges into compound nodes that are mapped in the cache without any conflicts. The algorithm checks for conflicts between individual procedures or compound nodes by examining the cache lines that they occupy. The optimal assignment of cache lines to procedures forms a coloring problem similar to register allocation.

Although the major steps during the coloring remain the same as in [2] there exist a few differences. First the distinction between large and small compound nodes is specified by the number of colors occupied by the members of each compound node and not by the number of members. We believe that this heuristic provides us with a more accurate representation of the size of a compound node.

Second our coloring algorithm maps popular procedures first to the cache address space and then to main memory (instead of mapping procedures directly to the main memory address space). This allows for a more flexible implementation since one can adopt several strategies for mapping procedures into main memory (i.e. to decrease the probability of second order effects (e.g. TLB misses, etc.). Any holes left after mapping popular procedures are filled with unpopular and unvisited procedures using a best fit policy.

4.1. Procedure Overlapping based on Hot Regions

Although cache line coloring moves procedures in an attempt to avoid procedure overlap in the cache address space, it can fail to do so in the case where one of the procedures occupies a large number of cache lines. Figure 2 illustrates the problem.

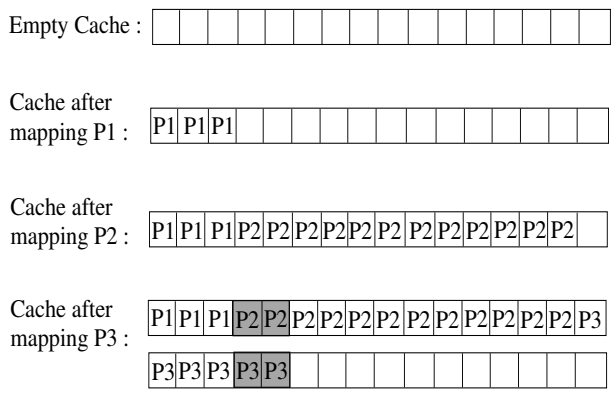

Figure 2. Procedures P_2 and P_3 overlap in a cache with 16 available cache lines.

Let us assume we have a direct mapped cache with 16 available cache lines, and procedures P_1, P_2, and P_3 which occupy 3, 12, and 6 lines (i.e. colors), respectively. In Figure 2 procedures P_1 and P_2 have already been mapped while P_3 is about to be merged with their newly formed compound node after considering the $P_3 - P_2$ edge. As we can see, procedure P_3 can not avoid color conflicts with P_2 regardless of its placement. According to the coloring algorithm, P_3 will be placed at the right end of the compound node. If we next encounter the $P_1 - P_3$ edge, we will find color conflicts between the two, but we will not move P_3 since its unavailable set does not allow it. Therefore, conflicts between procedures P_3 and P_1 will not be avoided.

We should have been able to avoid such a situation by reducing the number of colors considered during the coloring process for P_2. If we knew how control flows inside that procedure, we may have been able to specify a smaller number of colors for P_2 and allow P_3 to move later on in order to avoid overlapping with both P_1 and P_2. One way of dealing with this problem would be to reorder basic blocks, as implemented in [4, 11] and as suggested in [2]. This would produce more compact procedure bodies and facilitate the coloring step by reducing the actual number of colors required by each procedure. Although this approach is advantageous, it requires basic blocks to move inside a procedure's body, a step which not only introduces extra code but also complicates compilation. Our approach is to approximate the areas inside a procedure's body having the greatest potential of causing a large number of conflict misses if overlapped with another procedure. We call these areas the procedure's *Hot Regions* and we use access frequency counts for each cache line to identify them, as shown in Figure 3.

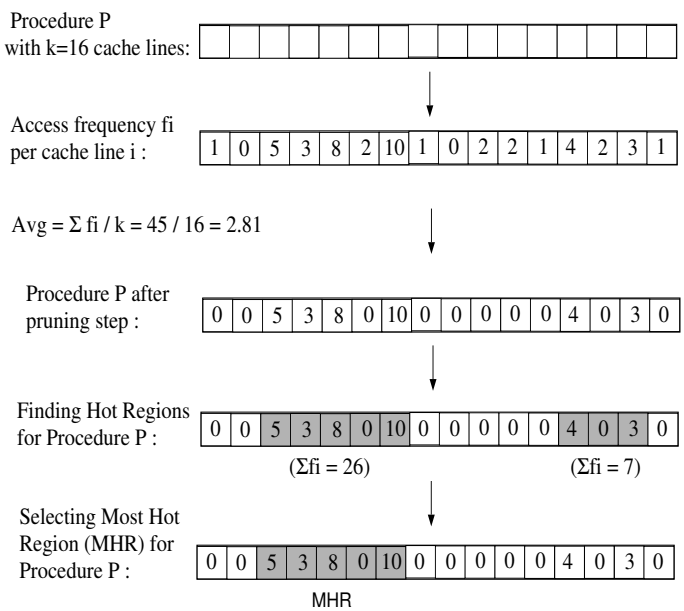

Figure 3. Finding the Hot Region for procedure P.

The algorithm for defining hot regions starts by annotating each cache line of a procedure's body with its frequency count. In order to compensate for loops that do not call other procedures we do not increment the frequency

count for a cache line when it is called more than once without transferring control flow to another procedure. For example if we have the following cache line access pattern ($P_{1a}, P_{2a}, P_{1b}, P_{1a}, P_{1c}, P_{1a}$), the frequency count for cache line P_{1a} of procedure P_1 will be 2 since only the first two activations of P_{1a} may cause a conflict miss (in our case with P_{2a}). The third activation will be a hit or a miss from a cache line of the same procedure (cache line P_{1c} can only replace P_{1a} if procedure P_1 is larger than the cache size, but we do not consider this case in our work here). Since we do not rearrange basic blocks, we can not avoid internal conflict misses for large procedures. Therefore, the frequency count generated provides an upper bound on the number of misses attributed to a cache line.

After acquiring the frequency counts, we compute their average and zero those counts that fall below the average (this is shown at the 3rd step of Fig. 3). This is done to uniquely identify rarely accessed areas in the procedure. The next step is to generate a list of hot regions within a procedure by grouping together the most frequently accessed cache lines. The grouping algorithm looks for sequences of hot cache lines that are separated by no more than 5 *cold* cache lines (i.e., lines with zero frequency count). The idea is to produce a number of compact hot regions within each large procedure (see the 4th step of Fig. 3). The amount of internal fragmentation allowed for a hot region can be an input parameter in our algorithm although we found that limiting fragmentation to 5 intervening cold cache lines works well in practice. Finally, we sum the frequency counts within each hot region and select the region with the highest sum (we refer to this region as the *Most Hot Region* (MHR)). The coloring algorithm will assign the number of colors contained in the MHR to this procedure.

In our current implementation, hot regions are found only for those procedures that occupy two-thirds or more of the cache address space. Smaller procedures do not impose severe constraints on the coloring algorithm and are not considered during this step. For those procedures that have a size larger than the cache address space, we keep frequency counts for a number of cache lines equal to the cache size. For example, for a direct mapped cache containing 256 cache lines and a procedure P_1 with 257 lines, accesses to cache lines 0 and 256 of P_1 will be tallied in a single count.

5. Methodology

To evaluate the merit of our algorithm, we compare our approach to that described in [2]. We use trace-driven simulation to quantify the instruction cache performance of our procedure reordering algorithm. The traces are generated using the tracing tool ATOM [1]. All applications are compiled with the DEC C V5.2 and DEC C++ V5.5 compilers on a DEC 3000 AXP workstation running Digital Unix V4.0. 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). Hence, the number of available colors and the size of the LRU stack for the cache lines will be 256. We evaluate a variety of applications, including C integer applications from the SPEC suites, Gnu applications, and C++ applications. From SPEC95 we include perl. Flex is a C program which generates C language lexical analyzers and bison is a C parser generator. pC++2dep is part of a C++ front-end written in C/C++ that transforms C++ programs into the *dep* intermediate format. f2dep is a parser that implements a Fortran compiler front end and is also written in C/C++. dep2C++ is a C/C++ program that translates the *dep* internal representation back to C++ code. All three applications are part of the pC++/Sage++ class library V1.7 [5]. ixx is a an IDL parser written in C++ and distributed as part of the Fresco X11R6 library. Ghostscript (gs) is an interpreter for postscript written in C.

Table 1 presents the static and dynamic characteristics of the benchmarks used. The second column shows the input used to both generate the profile and drive the simulations. The third column lists the total number of instructions executed. The fourth column shows the static size of the application while the fifth column contains the number of static procedures in the program. The sixth column has the percentage of the program that contains popular procedures and the seventh column contains the percentage of procedures that were found to be popular. The last column contains the percentage of unactivated procedures used to fill in the gaps left over after color mapping. Notice that our pruning step reduces the size of the CMG by 78-98%, and the size of the executable considered for cache coloring by 65-94%. This allows us to concentrate on the important procedures in the program.

5.1. Simulation Results

To evaluate the performance of our procedure reordering algorithm, we compare simulation results against the ordering produced by the DEC compiler (the current version of the DEC C/C++ compiler does a static DFS ordering of procedures) and the call graph based reordering algorithm presented in [2] (this algorithm was shown to outperform the Pettis and Hansen algorithm, previously considered to be the top reordering algorithm).

Table 2 shows the instruction cache miss rates for the standard ordering (DFS), call graph ordering (CGO), and our algorithm (CMG). The first column denotes the application. The second column shows the instruction cache

Program	Input	Instr. in million	Exe Size in KB	# Static Procs	Pop Procs		Unpop Procs
					% Exe Size	% Procs	% Exe Size
perl	primes	12	512K	671	29.4 (150.5K)	5.2	3.5 (18.1K)
flex	fixit	19	112K	170	33.3 (37.4K)	17.6	6.5 (7.3K)
bison	objparse	56	112K	158	22.4 (25.1K)	22.1	6.1 (6.9K)
pC++2dep	sample	19	480K	665	35.1 (168.8K)	16.3	10.3 (49.5K)
f2dep	f77_3	24	343K	474	25.0 (85.7K)	9.4	5.6 (19.4K)
dep2C++	sample	31	560K	1338	22.0 (123.7K)	1.7	1.7 (9.5K)
gs	tiger	34	496K	1410	12.9 (64K)	11.2	9.3 (46.3K)
ixx	layout	48	472K	1581	5.7 (27.2K)	5.1	2.4 (11.7K)

Table 1. Attributes of traced applications. The same input is used to train the algorithm and gather performance results. The attributes include the number of instructions in the instruction trace, the executable size of the application, 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 main memory gaps left from the color mapping.

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	4.92%	4.47%	4.21%	.71	.26	1,160,052	1,054,076	994,847
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.	3.51%	3.12%	2.49%					

Table 2. Instruction cache performance for the original mapping, call graph ordering and conflict miss algorithm. The first column lists the application. The next three columns show the instruction miss rates. The next two show the percent improvement over each by our algorithm. The last three columns show the number of instruction cache misses.

Program	First Order Misses			Higher Order Misses		
	DFS	CGO	CMG	DFS	CGO	CMG
perl	587,556	572,094	468,611	311	300	462
flex	97,600	85,183	78,413	2,635	99	6,809
bison	21,478	21,119	26,522	64	4	14
pC++2dep	759,517	927,967	577,776	135,488	107,416	119,981
f2dep	1,072,326	983,796	994,564	87,470	70,024	27
dep2C++	1204,820	1,063,335	956,834	0	91	12
gs	1,152,782	711,960	709,356	23,297	14	31
ixx	2,843,074	2,154,491	1,250,766	0	0	0

Table 3. Origin of conflict misses for the original mapping, call graph ordering and conflict miss algorithm. The first column lists the application. The next three columns show first order misses. The last three columns show higher order misses.

miss rates produced using the DEC compiler. The third column shows the instruction cache miss rates produced using CGO. The fourth column presents the results of our algorithm. The last three columns show the number of cache misses for each reordering algorithm.

As we can see from Table 2, the average instruction cache miss rate is reduced by 29% on average over the DFS ordering, and by 20% on average over the CGO ordering. CMG significantly improves the hit rates for all of the benchmarks over using the CGO ordering, except for bison, gs and flex. Flex and bison have working sets which fit in the small instruction cache under study so our optimizations are less effective (only bison experiences a minor increase in misses). That is not the case with gs which has a working set formed by a large number of small-sized procedures calling each other frequently. Both CMG and CGO eliminate a significant amount of conflicts compared to static DFS. However, CMG can not further improve the miss rate because the remaining number of misses is evenly distributed between many pairs of heavily interacting procedures which do not fit in the cache without color conflicts.

The rest of the benchmarks present an improvement mainly because of the better temporal locality data captured during the edge weighing in the Conflict Miss Graph. In certain cases (perl, pC++2dep, ixx), a number of misses were eliminated because a few large procedures were successfully approximated by their MHRs and the coloring process was able to avoid cache conflicts between them and the rest of the procedures.

Table 3 presents a categorization of the conflict misses as first order (those that occur between procedures that call each other) and higher order misses. As we can see in most of the benchmarks, first order misses dominate (pC++2dep is the only one where second order misses are significant). Our algorithm improves upon CGO mainly on first order misses due to the more accurate edge weight. The only exception is f2dep where the decrease of the miss rate comes from the elimination of higher order misses (first order are slightly increased). Also notice the presence of the opposite situation : a decrease in the first order class can cause an increase in the higher order (flex, pC++2dep). Both phenomena are a side effect of pruning. In flex and f2dep some of the edges estimating higher order misses are cut after pruning (although each one has a weight close to the threshold) and are not considered during the coloring. As a consequence their corresponding procedures are allowed to overlap, thus the increase in the number of second order misses. Similarly, in f2dep edges modeling first order misses are cut and cause an increase in the number of first order misses after the mapping. However, in both situations the overall miss rate drops (compared to CGO) because edge weighing is able to promote the most important procedures for the coloring stage even in the presence of a necessary step such as pruning.

6. Discussion

Next we discuss how to extend our algorithm to set associative caches and how our procedure reordering algorithm can be combined with other code reordering techniques to further improve the performance of instruction caches.

6.1. Procedure Reordering for Associative Caches

The results presented in this paper have been generated, modeling a direct-mapped instruction cache. Since our algorithm uses an algorithm similar to the original cache line coloring algorithm, placement in a set-associative can be done as proposed in [2]. The idea is to treat the degree of associativity as an extra dimension in the mapping on the cache address space. The number of sets will still represent the number of available colors but the placement algorithm will have to be slightly modified. Now, every procedure will keep track of the number of times a certain color (cache set) appears in its unavailable set. Hence, assigning a certain color to a procedure will not generate any conflicts if and only if that color appears in its unavailable set a number of times less than the degree of associativity of the cache.

6.2. Further Optimizations

The approach adopted in this paper does not fully exploit the potential of the cache line coloring and the increased accuracy of our conflict miss estimation model. The coloring step can further benefit from procedure splitting and basic block reordering. Basic block reordering can improve the accuracy during the coloring of procedures because it can create compact versions of procedure bodies, where the most frequently accessed portion can be remapped independently of the rest of the procedure. This can effectively reduce the number of colors used by a single procedure and will increase the number of colors available for other procedures to use.

Procedure splitting can improve the ability of coloring to work with only the portions of procedures that are truly causing conflicts. For example, if we choose to keep procedures intact, and if a procedure X calls procedure Y from one call site and procedure Z from another, cache line coloring will try to avoid overlapping the entire body of X with both Y and Z. Instead, if we split procedure X into three pieces X1, X2, and X3 (X1 represents the portion of X that will interfere with Y, X2 represents the portion of

X that interferes with Z, and X3 is the remaining portion of X), placing X1 with respect to Y is only constrained by the size of X1, and the same is true for placing X2 with respect to Z.

7. Comparison to previous work

There has been a considerable amount of work done on code positioning for improved instruction cache performance [2, 3, 4, 6, 7, 8, 9, 10, 11]. We next discuss some of this work, as it relates to our algorithm.

In [9], McFarling proposes a basic block remapping algorithm which captures control flow in the form of a *Directed Acyclic Graph*. The algorithm partitions the graph, paying special attention to loop nodes, with the goal of fitting each subgraph in the cache without any conflicts. The partitioning is done based on frequency counts assigned to edge weights.

Pettis and Hansen [11] employ procedure ordering, basic block reordering, and procedure splitting based on frequency counts to minimize instruction cache conflicts. They build a call graph by traversing edges in decreasing edge weight order using a closest-is-best placement strategy. They form chains by merging nodes, laying them out next to each other until the entire graph is processed.

Torrellas et al. [6] propose an algorithm for repositioning operating system codes. They identify the most frequently executed paths spanning several procedures, and then attempt to lay them out contiguously in the cache. They also try to fit loops in the cache. They manage to avoid a large number of cache conflicts for the most frequently executed basic blocks by partitioning the cache address space.

Hwu and Chang [4] suggest using basic block reordering, procedure reordering, and in-lining to improve instruction cache performance. They group frequently executed sequences of basic blocks into *traces* (this process is assisted by procedure in-lining). Then they lay out a procedure's body with these traces, based on execution frequencies. Their next step is to place procedures in a greedy (based on call frequencies as edge weights), depth-first, order. The goal is to create a small, compact, area approximating a procedure's *working set*.

Gossman et al. [7] define spatial and temporal costs to program modules with sizes smaller than the cache (activity sets). A search function is used to guide an iterative process to find a minimum of the cost function. Their search function is chosen so that a large number of combinations are covered on each experiment.

Two other approaches discussed in [3] and [10] reorganize code, based on compile-time information. In [10], code replication is performed based on the control flow of the graph (augmented with loop and procedure call information). The authors also attempt to partition the graph into subgraphs, smaller or equal in size to the cache using control-flow prediction heuristics. In [3] a similar approach to [2] is presented, where a call graph is constructed statically and weighted based on *program estimation*.

In [8] several basic block and procedure reordering algorithms are compared. These algorithms range from breadth-first and depth-first traversals to inter-procedural basic block reordering. They utilize execution frequencies in order to properly reorder basic blocks.

Most of the algorithms described above utilize execution or calling frequencies to weight the graph [4, 6, 8, 9, 11]. Our edge weighting is more accurate because it considers cache line interaction. The graph traversal based on those weights can avoid conflicts created by a depth-first or breadth-first traversal of a graph due to global information captured about procedure interaction. Also, our coloring algorithm works at finer level of granularity (cache line size instead of cache size [9, 10]) and can avoid conflicts encountered when either forming chains with the closest-is-best heuristic [11] (see [2] for an example) or dealing with subgraphs having a size larger than the cache.

Some algorithms remap basic blocks inside [4, 8, 11] or outside [6, 8, 9, 10] a procedure boundary. Although basic block reordering can improve spatial and temporal locality, we do not currently use it since it is a complex, error-prone algorithm that requires correction code to ensure proper control flow. For the same reason we did not consider procedure splitting [11]. The proposed approach does not explicitly consider the control flow of the program, however the conflict miss model implicitly takes into account control structures such as loops by not updating edge weights when a procedure's cache lines are continuously activated.

Heuristic-based static approaches do not suffer from the profiling and feedback overhead but they fail to capture part of the program activity in the control flow graph (i.e, it is difficult to predict the targets of indirect branches).

8. Acknowledgments

We thank Brad Calder, Hooshang Hashemi, Mike Smith and the anonymous referees for their feedback. We also thank Shantanu Tarafdar for providing and supporting the graph library used in this work. This research is supported by NSF CAREER Award MIP-9501172 and by Microsoft Research.

9. Conclusions

Cache performance is critical for today's microprocessors. Research has shown that compiler optimizations can

significantly reduce the number of cache misses, and every opportunity should be taken by the compiler to do so.

In this paper we propose a new link-time algorithm for procedure reordering which is based on procedure-level temporal interaction. We developed a model that estimates the worst case number of conflict misses which may occur between any two procedures in a program. We then employed a color-based placement algorithm that keeps track of the cache lines assigned to a procedure in order to avoid potential cache misses.

Using cache line liveness to find procedure interaction, we built a Conflict Miss Graph, with procedures of the program as nodes, and worst case conflict miss estimates as edge weights. We pruned the graph to reduce the size of the problem, and perform cache line coloring to produce an improved program layout. Using our algorithm, we were able to improve instruction cache miss rates on average by 29% over a static depth first search layout of procedures, and by 20% over the best competing algorithm.

During this study we concentrated on procedure reordering and tried to avoid color overlap between procedure boundaries. We then relax this constraint in order to improve the layout of large procedures. Our approach can be further enhanced with other techniques such as basic block reordering and procedure splitting which facilitate the coloring of the graph.

References

[1] Digital Equipment Corporation, Maynard, Massachusetts. *ATOM User Manual*, March 1994.

[2] A. Hashemi, D. R. Kaeli, and B. Calder. Efficient Procedure Mapping using Cache Line Coloring. In *Proceedings of the Int. Conference on Programming Language Design and Implementation*, pages 171–182, June 1997.

[3] A. Hashemi, D. R. Kaeli, and B. Calder. Procedure Mapping using Static Call Graph Estimation. In *Proceedings of the Workshop on Interaction between Compiler and Computer Architecture*, February 1997.

[4] W. Hwu and P. Chang. Achieving High Instruction Cache Performance with an Optimizing Compiler. In *Proceedings of the Int. Symposium on Computer Architecture*, pages 242–251, May 1989.

[5] Indiana University. *Sage++ : A Class Library for building Fortran 90 and C++ restructuring tools*, November 1993.

[6] J.Torrellas, C.Xia, and R.Daigle. Optimizing Instruction Cache Performance for Operating System Intensive Workloads. In *Proceedings of the Int. Conference on High Performance Computer Architecture*, pages 360–369, January 1995.

[7] K.Gosmann, C.Hafer, and et.al. Code Reorganization for Instruction Caches. In *Proceedings of the Hawaii Int. Conference on System Sciences*, pages 214–223, January 1993.

[8] D. Lee. Instruction Cache Effects of different Code Reordering Algorithms. Technical Report FR-35, University of Washington, Seattle, October 1994.

[9] S. McFarling. Program Optimization for Instruction Caches. In *Proceedings of the Int. Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 183–191, April 1989.

[10] A. Mendlson, S. Pinter, and R. Shtokhamer. Compile-time instruction cache optimizations. In *Proceedings of the Int. Conference on Compiler Construction*, pages 404–418, April 1994.

[11] K. Pettis and R. Hansen. Profile-Guided Code Positioning. In *Proceedings of the Int. Conference on Programming Language Design and Implementation*, pages 16–27, June 1990.