In Proceedings of the 35th International Symposium on Microarchitecture, November, 2002

Compiling for Instruction Cache Performance on a Multithreaded Architecture

Rakesh Kumar   Dean M. Tullsen

Department of Computer Science and Engineering
University of California, San Diego
{rakumar,tullsen}@cs.ucsd.edu

Abstract

Instruction cache aware compilation seeks to lay out a program in memory in such a way that cache conflicts between procedures are minimized. It does this through profile-driven knowledge of procedure invocation patterns. On a multithreaded architecture, however, more conflicts may arise between threads than between procedures on the same thread. This research examines opportunities for the compiler to optimize instruction cache layout on a multithreaded architecture. We examine scenarios where (1) the compiler has knowledge about multiple programs that will be or are likely to be co-scheduled, and where (2) the compiler has no knowledge at compile time of which applications will be co-scheduled. We present solutions for both environments.

1 Introduction

Instruction cache aware compilation [8, 10, 13, 7, 17, 9, 14, 6] seeks to lay out a program in memory in such a way that instruction cache conflicts between procedures are minimized. It does this through profile-driven knowledge of procedure invocation patterns, taking procedures that are frequently invoked near in time and placing them in memory such that they do not conflict in the instruction cache.

On a conventional, single-threaded processor these techniques exploit the typically highly predictable pattern of procedure call invocations to anticipate conflicts. However, on a hardware multithreaded architecture, such as simultaneous multithreading (SMT) [21, 20], conflicts also arise between threads on the same processor due to time-sharing of the instruction fetch unit. In many cases, these inter-thread conflicts are higher than intra-thread conflicts, as demonstrated in Figure 1. For these pairs of applications, intra-thread conflicts account for 20.47% of all misses on average, while inter-thread conflict misses account for 21.21%. It is not surprising to see more inter-thread conflicts, as an SMT processor's fetch unit switches between threads as often as every cycle, while procedure calls (the most common cause of intra-thread conflicts) typically happen at a much lower frequency. This research will demonstrate that not only do existing layout techniques fail to address inter-thread conflicts, but in many cases they actually exacerbate the problem.

Unlike intra-thread conflicts, inter-thread conflicts are typically non-repeatable and difficult to predict, requiring different techniques than those that work on single-thread workloads. However, this research demonstrates that we can build on and extend existing single-thread instruction cache optimization mechanisms to effectively reduce these inter-thread conflicts.

We consider three scenarios where instruction cache layout can be applied to a multithreaded workload:

(1) We have knowledge of all or some programs likely to be co-resident, and have control over the code generation of each of those programs. Workloads where this would apply include a very static workload (e.g., a server environment, long-running simulations, or an embedded environment), applications that are typically tied together through a pipe (gzip and ghostview, for example) or other communication mechanism, or any multithreaded application where threads follow relatively different paths through the code.

(2) A program is compiled with some knowledge of what application(s) will be co-resident with it, but the compiler has no control over the other programs. In this case, the program can still be compiled to avoid conflicts with the existing programs. This includes all of the scenarios above (but where we lack the ability to remap all programs). It also works when one application (e.g., gzip, a web browser, or the operating system) expects to be paired with several different partners at various times. It could also be applied on a system with support for run-time or load-time code modification, which can optimize the one program being introduced into the running set.

(3) We have no knowledge of which jobs will be coscheduled. Here, we assume the operating system does

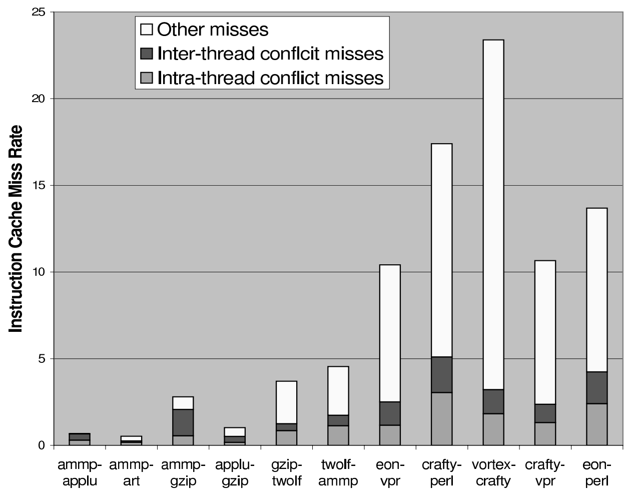

Figure 1. Percentage of instruction cache misses due to conflict misses, classified as inter-thread or intra-thread conflicts. The cache is a 32 KB direct-mapped cache running two threads on an SMT processor.

have some control over the mapping of logical pages to cache lines (the same assumption that standard instruction-cache layout optimizations typically must make). This would be the case when the physically-addressed cache is bigger than the system page size, or if the system has some simple support for cache page coloring [14]. In this, the most general case, we compile each program in such a way that the operating system will be able to map applications to minimize conflicts between jobs in the running set.

Between these techniques, we account for essentially any workload that is susceptible to inter-thread instruction cache conflicts.

This paper is organized as follows. Section 2 discusses previous and related work. Section 3 provides background information on simultaneous multithreading and instruction cache layout research. Section 4 describes the measurement methodology. Section 5 describes techniques for coordinated compilation of programs likely to be co-resident, and Section 6 presents mechanisms for compiling one program given some knowledge of other programs it is likely to be run with. Section 7 discusses compiler and OS mechanisms which reduce instruction cache misses even in the case when the mix of applications which will be run together is not known *a priori*. Section 8 shows that these techniques work across a range of cache associativities and sizes. We conclude in Section 9.

2 Related Work

This work builds on a large body of existing work in compile-time and run-time code-placement techniques to reduce the number of cache conflicts for a single program. Most of the prior work in the area of compile-time code placement is concerned with reordering of popular code-blocks such that mutual interference in the instruction cache is minimized. Some of the earliest work in this area was done by Hwu and Chang [8] , McFarling [10], and Pettis and Hansen [13].

Hwu and Chang describe an algorithm for improving instruction cache performance using inlining, basic block reordering and procedure reordering compiler optimizations. They use a Weighted Call Graph (WCG) and a proximity heuristic to address the problem of basic block placement. McFarling examines improving instruction cache performance by not caching infrequently used instructions. He represents the program as a DAG of procedures, loops and conditionals. The algorithm tries to partition the graph, concentrating on the loop nodes, so that the height of each partitioned tree is less than the size of the cache. Pettis and Hansen also describe a number of techniques for improving code layout that include basic block reordering, procedure splitting, and procedure reordering.

More recent work was done by Hashemi *et al.* [7], Torrellas *et al.* [17], Kalamatianos and Kaeli [9], Sherwood *et al.* [14] and Gloy and Smith [6]. Hashemi *et al.* improve the Pettis and Hansen algorithm by computing which lines are occupied by which procedures. Torrellas *et al.* propose an algorithm designed for mapping operating system code to increase performance. Kalamatianos and Kaeli define a structure called a Conflict Miss Graph (CMG) to keep track of temporal ordering information. Sherwood *et al.* propose a color mapping at compile time for code and data pages, which can then be used by the operating system to guide its allocation of physical pages. Gloy and Smith use a Temporal Relationship Graph (TRG) that summarizes the information about interleaving of procedures in a program trace. Their algorithm, also known as Temporal Profile Conflict Modeling (TPCM), applies the maximum-overlap strategy to place the procedures in the cache greedily. The optimizations presented in this work extend TPCM in various ways to accommodate a multithreaded processor. The details of the algorithm are discussed in Section 3.

Dynamic schemes monitor the behavior of the system and perform runtime optimizations based on this behavior. Chen and Leupen [5] present a technique for just-in-time code layout which loads procedures into the text segment in the order in which they are invoked. Bugnion *et al.* [3], examine compiler-directed page coloring for arrays on multiprocessors. The Impulse project [4] provides a compiler controlled memory controller. The impulse address space can be remapped by a new strided address calculation or an indirection vector for an array. Yamada *et al* [22] propose a similar scheme for L1 cache. Bershad *et al.* [2] introduce a hardware device called the Cache Miss Lookaside (CML) buffer that records and summarizes the cache-miss history.

They use CML to detect conflicts caused by page mapping and remove conflicts by dynamic remapping of pages. Sherwood, *et al.* [14] add a page remap field to the TLB which allows a page to be remapped to a different color in the physically indexed cache while keeping the same physical page in memory.

Our work is unique in addressing the problem of inter-thread cache conflicts on a multithreaded processor.

3 Background

A simultaneous multithreading processor [21, 20] is a hardware multithreading architecture which has the ability to issue instructions from multiple threads to the execution units in a single cycle. All threads compete for all processor resources each cycle, maximizing utilization of the processor. One of the fully shared resources is the instruction cache. This architecture has been shown to significantly increase the throughput of the processor, for relatively small hardware cost.

SMT processors are poised to appear in the marketplace. The Alpha 21464 [11] was originally announced to feature SMT, and Intel has announced that both desktop and server processor lines will feature simultaneous multithreading [12].

Temporal Profile Conflict Modeling has been shown to be an effective code placement algorithm for I cache performance [6], and represents the starting point for this research. TPCM is based on temporal profile data and accurately models cache mapping conflicts. It uses a Temporal Relationship Graph (TRG) to find a cache-relative alignment for each procedure that minimizes cache conflicts. Then it produces a procedure ordering and layout that may include gaps between procedures to achieve the desired cache-relative alignment. This ordering can be used by the compiler, linker or other code reordering tool to produce an executable with the correct procedure addresses.

Given a trace of code block references, a TRG for the trace can be defined to be a weighted undirected graph, with a node for each code block, and where an edge between nodes A and B has a weight I(A,B) where I(A,B) is the number of times that two successive occurrences of A are interleaved with at least one reference to B, and vice versa.

TPCM can be applied to code blocks of any granularity. However, Gloy and Smith apply it to procedures to obtain a procedure TRG and simultaneously generate a chunk TRG as well (corresponding to procedure chunks of fixed size). To reduce overhead, they select a set of popular procedures and focus only on these during the trace processing.

TPCM uses a working graph derived from the procedure TRG. The working graph structure and the edge weights are copied from the procedure TRG. The heaviest edge in the working graph *E(A,B)* is repeatedly found and the two nodes joined by this edge, A and B, are combined. At this point, A and B are aligned relative to each other by choosing the relative offset (in cache lines) between A and B that causes the least number of estimated conflicts.

The correct relative offset is found by considering all possible relative layouts in cache. For each value of displacement d (less than the number of cache lines), the cache line locations for all procedure chunks are computed. If we add the chunk-TRG edge-weight for each pair of chunks (I,J) mapped to a particular cache line, such that I is a node in chunk-TRG(A) and J is a node in chunk-TRG(B), this gives us the estimated cost of mapping conflicts for this cache line. This sum computed over all cache lines is the estimated cost of combining A and B using displacement d. The value of displacement which results in a minimum cost as well as maximum overlap is chosen as the relative offset for the merge of A and B.

The working graph is updated to adjust for the merging of A and B into node AB. Merging continues until there are no more edges left. A complete layout is generated from procedure alignments simply by leaving an appropriate amount of empty space before each procedure.

On a multithreaded architecture, procedure interleavings represent only one source of conflict misses. The other is the inter-thread switching within the fetch unit. If we can create an estimate of the frequency of “interleavings” between procedures of different threads, we can incorporate new edges into our TRG algorithm to account for these.

An estimate for the number of conflicts that a code-block A belonging to program P1 will have due to code block B belonging to another program P2 for a given cache layout of A and B can be expressed as:

$$Conflicts(A,B) = \%ET(A) \times \%ET(B) \times switching$$

This is a probabilistic estimate based on the likelihood that both procedures are being run at once. Here %ET(A) is the execution time of A expressed as a percentage of the total execution time of P1. *switching* is the number of times execution of program P1 was interleaved with execution of P2. This expression serves as the basis for the calculation of inter-program conflicts in subsequent discussion.

4 Methodology

Table 1 summarizes the benchmarks used. All come from the SPEC2000 benchmark suite. These benchmarks were chosen such that about half have “good” instruction cache behavior (less than 3% miss rate) and half have “bad” I cache behavior (by SPEC2000 standards). The multithread workloads are composed of sets of programs taken from this list. The two-thread workloads, for example, include bad-bad, good-bad, and good-good combinations.

Program	Hit-rate	Training Input	Reference Input
ammp	99.64	ammp.in (train)	ammp.in (ref)
art	100.00	c756hel.in (train)	c756hel.in (ref)
applu	100.00	applu.in (train)	applu.in (ref)
gzip	99.93	input.combined	input.log
twolf	98.99	train	ref
crafty	86.91	crafty.in (train)	crafty.in (ref)
eon	94.23	cook (train)	cook (ref)
perl	92.85	scrabbl	makerand
vortex	84.82	lendian.raw	lendian3.raw
vpr	96.84	train	ref

Table 1. Benchmarks simulated, included single-thread I cache hit rates for a 32 KB DM cache.

Parameter	Value
Fetch Bandwidth	2 threads 8 instructions total
Functional Units	3 FP, 6 Int (4 load/store)
Instruction Queues	32-entry FP, 32-entry Int
Inst Cache	32KB, direct-mapped, 32-byte lines
Data Cache	64KB, 2-way, 64 byte lines
L2 Cache(on-chip)	1 MB, 4-way, 64-byte lines
L3 Cache(off-chip)	4 MB
Latency(to CPU)	L2 6 cycles, L3 18 cycles, Memory 80 cycles
Pipeline depth	9 stages
Min branch penalty	7 cycles
Branch predictor	4K gshare
Instruction Latency	Based on Alpha 21164

Table 2. Processor details.

The pairings are otherwise arbitrary. Although the previous section describes a wide set of environments where these techniques might be particularly appropriate (multithreading applications, threads communicating through pipes, etc.), we focus our attention in this paper on the most difficult scenario, where multiple applications run without communication, giving us no basis for predicting how they interact. This paper examines configurations larger than two threads, but the majority of our results and most of the discussion of techniques focus on the two-thread case.

The baseline processor simulated is an 8-issue simultaneous multithreading superscalar processor configured as specified in Table 2. We assume a 32KB direct-mapped instruction cache. Heavy cycle time pressure in modern processors is applying significant downward pressure on L1 cache size and associativity (which will be coupled with deeper memory hierarchies), making this a liberal estimate of L1 cache resources for this study. The Itanium processor, for example, features a 16K instruction cache (4-way set-associative), and cycle time constraints will continue to put pressure on both those numbers for L1 caches. These techniques become more critical as caches shrink. Other cache sizes and associativities are considered in Section 8.

Processor simulation uses the SMTSIM [18] instruction-level simulator, which emulates unaltered Alpha executables, and models all typical sources of processor latency and conflict. The programs are run on the reference data sets and trained on the SPEC-supplied training data sets. Simulations are run for 1 billion instructions.

We use ATOM [16] to generate the procedure call-traces and other profile information as input to our code mapping algorithms. For all the algorithms, the TRG is composed of popular procedures only. Our definition of an unpopular procedure is one whose execution time is less than 1% of the total execution time.

Most of our algorithms require the number of context-switches (*switching*) as a parameter. On an SMT processor, a context switch (from the perspective of the I cache) happens every time the fetch unit goes from fetching one thread to another. Since our baseline SMT processor fetches from up to two threads per cycle, we assume a level of switching of once per cycle. The techniques described in this paper are by no means specific to SMT. By simply changing the switching factor, we can accommodate a coarse-grain multithreaded processor [1] or even a single-threaded processor with infrequent, OS-initiated context switches. However, the benefits of our techniques are greatest in the SMT environment.

This type of study represents a methodological challenge in accurately reporting performance results. In multithreaded experimentation, every run consists of a potentially different mix of instructions from each thread, making relative IPC a questionable metric. This paper will use weighted speedup, a metric proposed by Tullsen and Brown[19]. Weighted speedup is given by:

$$WS = \frac{1}{number\ of\ threads} \sum_{threads} \frac{IPCnew}{IPCbaseline}$$

Weighted speedup much more accurately reflects system-level performance improvements, and makes it more difficult to create artificial speedups by changing the bias of the processor towards certain threads. We refer the reader to [19] and [15] for more detailed discussion of the metric.

5 Coordinated Compilation

This section describes techniques applicable any time the compiler or code generator has access to multiple jobs likely to be co-executed. This might be a regular or long-running workload, a special-purpose processor or environment, or a set of applications that typically run together (e.g., gunzip and postscript, or a web browser and acrobat). In this case,

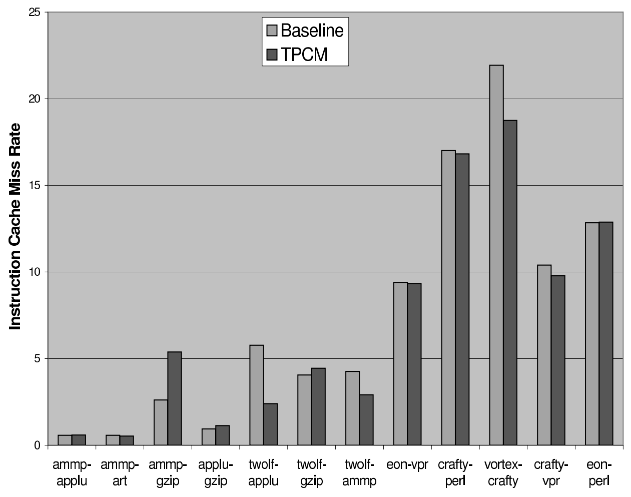

Figure 2. Miss rates when both the programs of every pair have a layout generated by TPCM without accounting for inter-thread conflicts.

we assume the ability to compile (or re-layout) both (or all) of the programs at the same time. This creates two separate executables with reduced instruction cache conflicts between them.

The light-gray bars of Figure 2 shows the miss rates for 12 benchmark-pairs. The aggregate instruction cache miss rates for these pairs vary from 0.5% (ammp-art and ammp-applu) to 21.9% (vortex-crafty). Figure 2 also shows the results when TPCM is applied to each of the programs in the workload individually. In this and all future figures, the baseline result represents the applications with no special code-layout optimizations, and TPCM refers to the multithreading-oblivious optimization TPCM applied to each program independently. For some benchmark-pairs (ammp-applu, ammp-gzip, applu-gzip, twolf-gzip), the miss rate after applying TPCM is more than the baseline case, implying that any reductions in intra-thread conflicts are more than counteracted by increases in inter-thread conflict due to the remapping. For ammp-gzip, the miss rate increases by more than 100%. This happens because when both applications are optimized to spread out and use the whole cache effectively, oblivious to other coscheduled threads, it can increase the likelihood of conflict between them.

For our techniques, which account for inter-thread conflicts, we assume the same profile-generated information typically used by code layout optimizations for each program, but no special knowledge of interactions between the programs. We modify the TPCM algorithm described in Section 3. We begin by generating TRGs for both programs. The TRGs represent the switching activity between all nodes (procedures) of the same program. We then add edges between nodes of the two graphs to indicate expected switching activity between procedures/cache lines of the two programs. The weights of the edges added represent an expected amount of switching-caused conflicts, as derived in Section 3.

We do not add edges between all possible pairs of nodes, but only those that will have the highest edge weights, potentially causing the most conflicts. We show that as few as one added edge can improve performance.

Once the two TRGs are combined into a single graph, we iterate over the combined TRG in decreasing order of weights and generate the cache-relative offset for all the nodes in a manner similar to TPCM. This cache-relative alignment can be used to dynamically remap cache accesses by each program during execution. Then each program is mapped individually according to the offsets computed for each node belonging to that program. This is accomplished simply by leaving appropriate gaps between programs.

Though the algorithm and results are presented for two programs only, it can be trivially generalized to multiple programs as well. If there are m programs that run together, all the m TRGs are combined by drawing edges between nodes selected on the basis of SWITCHING values. Again the new edges create a single graph, and we can apply TPCM on it to generate a cache layout as before.

Figure 3 shows that we can get significant improvements with even a single added edge between the TRGs (M-TPCM-1 in that figure, which denotes multithreading-aware TPCM, with one added edge). Making one connection gives us an improvement over the baseline in all but one case. The relative decrease in miss rate is as high as 68.6% (twolf-applu). Average improvement is 8.8% over raw-execution and 9.7% over TPCM. However, for several pairs, TPCM (applied on individual threads) still outperforms our technique with a single connection. This happens due to two factors. One is the disconnect with the profiled data set. Second is the limited information from only adding a single edge — adding a small number of edges has something of a randomizing effect, and we may create other important inter-thread conflicts if they aren't specifically represented by a node edge. We address the second factor by adding more edges.

When adding 4 inter-thread TRG edges, we get relative miss rate improvements up to 68.45% (twolf-applu). Average improvement is 9.94% over the baseline and 10.91% over TPCM. Also, the improvements are more than TPCM (applied on individual threads) for all but one (vortex-crafty) pairs. For 16 added connections, the average improvement is 4.53% over raw-execution and 5.57% over TPCM. With 16 edges, we over-constrain the mapping, and produce less desirable layouts for some applications.

Figure 4 shows the impact of our algorithm on overall performance. Average speedups with 4 connections is

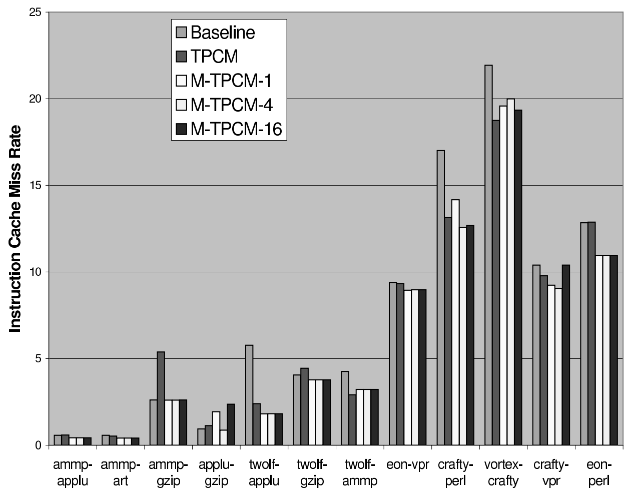

Figure 3. Instruction cache miss rates when multithreading-aware layout is applied by adding 1, 4, or 16 graph edges.

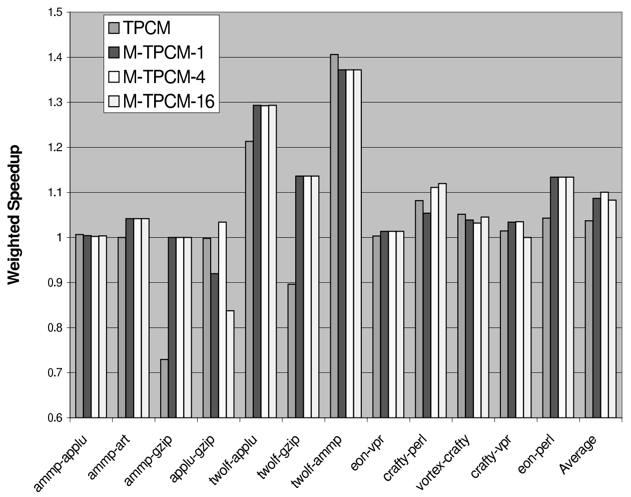

Figure 4. Weighted speedups when multithreading-aware layout is applied by adding 1, 4, or 16 graph edges.

11.34% (maximum 37.2%) over the baseline and 6.41% (max 37.11%) over TPCM alone.

These results demonstrate that when multiple applications are known at compile time to likely be co-resident, we can improve the overall performance by taking into account profile-generated information about these threads.

Previous work in code layout typically uses profile-generated information, generally requiring something that looks like a dynamic call graph, which is relatively easy to collect in many systems. These algorithms do not use more detailed information because they do not need it. For example, if procedure A calls procedure B, B can only evict lines of A once (causing one conflict miss each), no matter how many times lines in B are executed.

We have constrained our algorithms to use the same type of information as the single-thread techniques to more directly compare them; however, that does not show our techniques in the best light, because we can take great advantage of lower-level detail. For inter-thread conflicts, it matters whether a line in procedure B is executed once or a million times, because the opportunities for inter-thread conflicts are much greater in the latter case. Even if we constrain our algorithm to only remap code at a procedure granularity, we can still use this data. If we are forced to map two procedures on top of each other, we can then do it in such a way that the most popular lines in each procedure do not conflict. As a specific case study, we generated a more detailed profile for the eon-vpr pair. Doing some basic remapping of procedures with this extra information allows us to increase the combined hit rate from 90.60%(baseline) to 91.04%. The hit rate for M-TPCM without using this extra information was 90.71%. Further results in this paper rely only on the coarse-grain profile information, again to most easily compare with prior work.

The next section shows that gains are still possible when only one program is available for recompilation, as long as some runtime information is known about the overall workload.

6 Informed Single-Thread Compilation

This section assumes the ability to compile, recompile, or remap only a single program, as long as profile information is know about each (or perhaps some) of the programs that will be co-resident. This would apply if some of the programs come precompiled, one program has many co-resident mates (e.g., gunzip) and can't be co-compiled with each, or in a dynamic compilation or load-time application of these techniques, where typically only one new job at a time enters the jobmix.

In this case, we will start out with one program (or more) and its initial layout generated by the compiler, and we will lay out the second program so as to minimize conflicts between the two (as well as minimize same-program conflicts in the second program). We will refer to the initial program which we cannot remap as the *static* program, and the other program as the *new* program.

We assume that we have the TRG and profile information corresponding to the new program. We also assume that we have information about size and execution time of procedures of the static program. To generate a cache layout for the new program, we need to find the best cache-relative displacement for every procedure in the new program.

To determine the order in which we choose the procedures for placement, we define a metric called interleave-

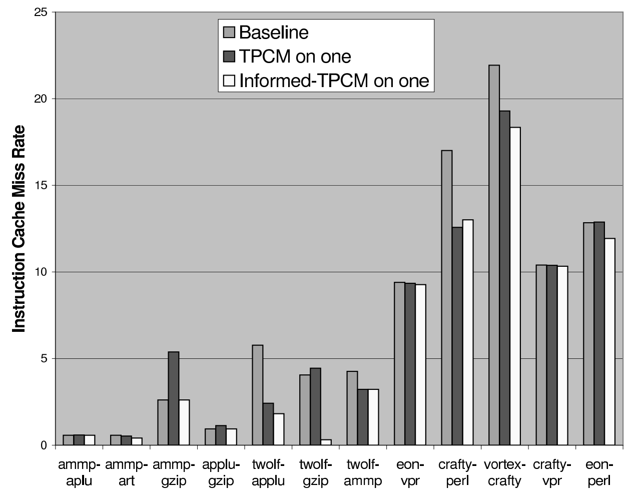

Figure 5. Miss rates when first program of every pair has a naive layout and a multithreading-aware layout is generated for the second program.

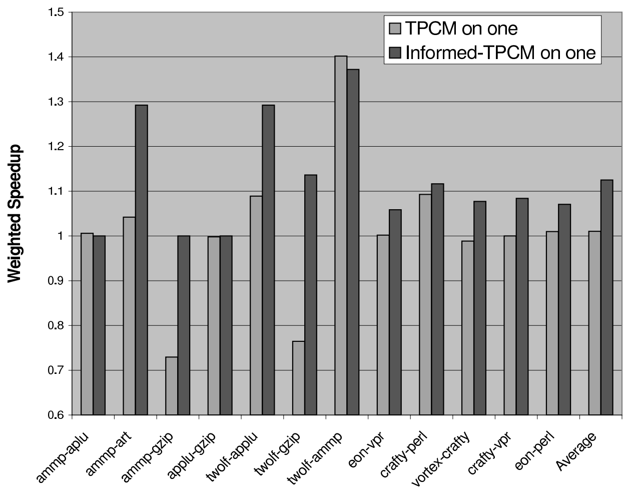

Figure 6. Impact on execution performance when first program of every pair has a naive layout and a multithreading-aware layout is generated for the second program.

value. Interleave-value is the maximum number of estimated cache conflicts which a code-block belonging to the new program can have due to all procedures in the static program. If A is a node belonging to some TRG, then the interleave-value of A can be given as:

$$Interleave(A) = \%ET(A) \times switching$$

We find the interleave-value for all the nodes belonging to TRG (new) and sort the nodes in decreasing order of these values. We take the node with the highest interleave-value and find the cache-relative offset for it by calculating cost for all possible placements in the cache and choosing the displacement with the least cost (the expected number of conflicts, considering both the static program and previously mapped procedures of the new program). Cost due to code-blocks belonging to the same (new) program is calculated using standard TPCM techniques. While calculating costs due to conflicts with code-blocks belonging to the static program we again use the estimated value for switching conflicts derived in Section 3.

Once we have placed a procedure, we remove its node from the list and choose the node with the next-highest interleave value. We continue until we have found the cache-relative offset for all nodes belonging to the new program. This algorithm chooses the order for placement only considering inter-thread conflict potential, but this works because the order is determined by execution time, which is also an effective ordering for minimizing intra-thread conflicts.

Though the algorithm is again given for two programs only, it is also easily generalized to the case where one program is added to a set of multiple pre-compiled programs. The case where a set of *multiple* programs is added to an existing set is less straightforward, but can most easily be adapted from this case by adding them one at a time.

Figure 5 shows the effectiveness of this technique. In each case, the "static" program has no layout optimization applied. The baseline represents no optimization to the new program. *TPCM* applies multithreading-oblivious TPCM to the new program, and *informed-TPCM* applies our multithreading-aware technique to the new program. We observe that applying TPCM leads to deterioration in cache miss rate for many pairs (ammp-applu, ammp-gzip, applu-gzip, twolf-gzip and eon-perl). Our algorithm results in relative miss-rate improvements up to 92.3% (twolf-gzip). Average miss-rate improvement is 21.9% over the baseline, and 18.5% over TPCM.

Figure 6 shows the performance results. Our algorithm results in speedups up to 37.2%. Average speedup is 12.5% over the baseline and 11.1% over TPCM.

This section demonstrates that we need compilation access to only one of the co-resident applications to get significant improvements in instruction-cache performance, as long as we have some knowledge about the future co-resident workload.

The next section shows that performance improvements can be achieved even when we lack that information.

7 Compiler Support for Dynamic Cache Conflict Avoidance

The previous section relaxed the constraint that we need co-compilation access to co-resident applications. In this section we relax the constraint that we have any pre-runtime knowledge of application co-residency. The only requirement we add is a minimal amount of either operating system or architectural support (similar to what traditional TPCM assumes). Thus, each program is compiled using only information known about the application itself. But it does assume that the other applications running were also compiled with a compiler that followed the same conventions described here.

The key to these techniques is the realization that code layout techniques like TPCM do not rely on forcing equal access to each cache line, but only separate procedures prone to interleaving. So, for example, if the three most popular procedures are not typically interleaved, they can be mapped to the same portion of the cache without affecting performance. This insight allows us to tweak the TPCM algorithm to intentionally create highly *uneven* mappings. That is, certain portions of the cache are accessed more heavily than others, but in a predictable way.

But we also must ensure that the threads do not each stress the cache in the same way. We assume that the operating system has some control over the mapping of virtual address to instruction cache lines. In a system with physically addressed instruction caches that are larger than the page size, code layout techniques require operating system support to ensure that a page the compiler thought would be mapped onto page 2 of the cache does indeed occupy that page. The OS might do this through a page-coloring page allocation mechanism. We assume the same type of support here. This technique could be adapted to a virtually-addressed cache by simply adding a small bit of hardware to the cache lookup logic which *xor*s certain bits of the virtual address with the hardware thread id, forcing the same virtual address on different threads to map to different regions of the cache. More details on hardware remapping can be found in [2] and [14].

This represents the most general scenario where we have no compile-time information about other threads that a particular thread might run with. All we know is the maximum number of threads (or, perhaps, the most common number) that will run at once. We can exploit this by creating a biased, or uneven, layout for each thread. As long as these programs are biased in a predictable way, the operating system can map them into the instruction cache appropriately. For example, in a system with four threads, we would compile each program so that the virtual addresses that correspond to the first quadrant of the cache are accessed more heavily than the other quadrants. When the operating system allocates physical pages, it does it in such a way that co-resident threads map their heaviest quadrant to a different quadrant in the physically-addressed cache. That is, thread 0 might be constrained such that virtual pages are mapped to physical pages with the same address modulo the cache size. Thread 1 virtual pages are mapped to physical pages with the same address plus

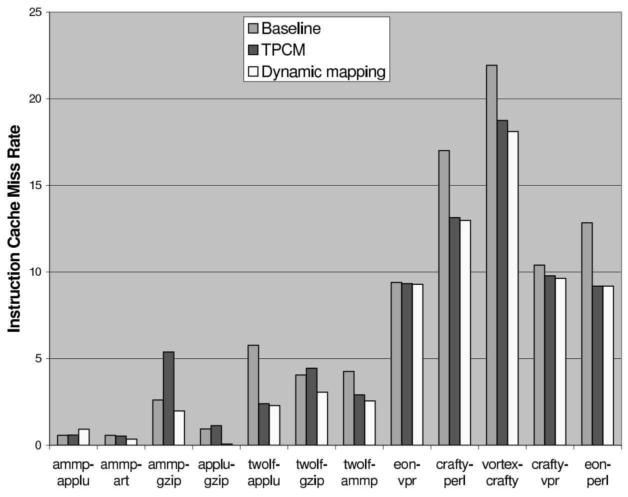

Figure 7. Miss rates when both the programs of every pair are mapped dynamically according to our algorithm

cache_size/number_threads, modulo the cache size. This decreases the chance that popular procedures belonging to different programs conflict with each other.

To determine the order in which we choose procedures for placement, we extend the definition of interleave-value to include the total number of estimated cache conflicts which a code-block belonging to a program can have due to other procedures in the same program, as well as expected conflicts with other programs. This implies that we'll be placing procedures with high execution time (and thus high conflict cost) earlier in the algorithm than standard TPCM.

Hence, if A is a node belonging to some TRG, then the interleave-value of A can be given as:

$$Interleave(A) = in\_edge\_wts + \%ET(A) \times switching$$

where *in_edge_wts* is the sum of the weights of all edges incident on A.

We choose the nodes in decreasing order of interleave values and find the best cache-relative offset using a method similar to the ones used in the prior sections.

However, to ensure that the generated layout is top-heavy, we assume that every cache-line except those belonging to the first 1/T of cache, where T is the maximum number of threads, offer a constant bias-resistance to all the code-blocks mapped to that line. This bias-resistance is added to the cost of mapping a code-block to a particular cache-line.

The cache layout which is generated is top-heavy because the popular procedures (i.e. with high interleave values and execution time) get placed early, at the top. The less popular procedures then only get placed in the lower portions of the cache if there is sufficient interaction with those procedures already placed in the first section. Since

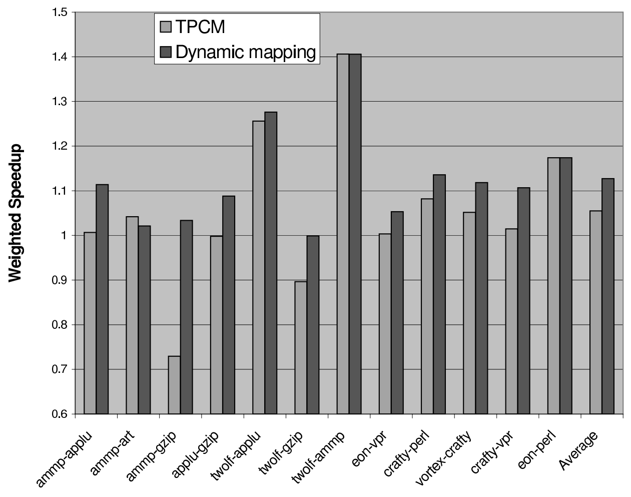

Figure 8. Impact on execution performance when both the programs of every pair are mapped dynamically according to our algorithm

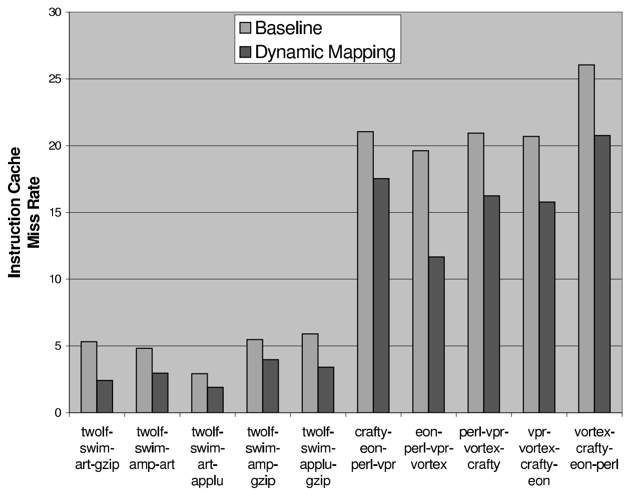

Figure 9. Miss rates when all the four programs are mapped dynamically according to our algorithm

the layout is generated using a TPCM-based algorithm, the number of conflicts with the procedures belonging to the same program is still kept low.

This algorithm can easily be used for dynamically mapping the cache accesses because the operating system is not required to know profile information about the programs. However, if not all programs are compiled in this manner, the OS could use some runtime or profile information to identify which parts of the instruction cache those programs would use most heavily.

Figure 7 shows the results of applying top-heavy compilation with dynamic mapping. Our algorithm results in improvement in miss rate for all but one case (ammp-applu). Relative miss rate reductions are up to 92.6% (applu-gzip). Average reduction is 24.7% over the baseline and 22.3% over TPCM.

Figure 8 shows the impact on overall performance. Average speedup is 12.7% (max 41%) over the baseline and 6.6% over TPCM.

When four threads are running together (Figure 9), this algorithm results in relative miss rate reductions over the baseline of 26.8% on average. Performance improvements are more modest (3-6%), as the system becomes more tolerant of instruction cache misses with more threads.

Because we depart from traditional TPCM to generate program layouts, it would be expected that we sacrifice some single-thread performance (i.e., in the case where only one thread is being run) to achieve this higher multiple-thread performance. In fact, this is not the case. In 8 out of 12 cases (and the average case), miss rates are lower for our threads than with regular TPCM, when running alone. The traditional TPCM algorithm does not accurately account for the total number of cache conflicts when it calculates interleave values between procedures. The total number of potential conflict misses are a factor of both the number of interleavings, and the static sizes of the two procedures. TPCM only accounts for the first in selecting the order in which procedures get placed. We do not directly account for this, but because execution time is somewhat correlated with static code size, we end up with a better ordering of procedures for placement.

Relative to standard TPCM, then, this technique provides both higher multiple-thread performance and higher single-thread performance.

One concern about this technique is that it could constrain the operating system's ability to schedule jobs. If a job has its heavy region mapped to the second quadrant, and the jobmix changes significantly, we would not want to later schedule it with another job that mapped its heavy region to the same quadrant. However, it is hard for the OS to change these mappings. If the mapping is done in hardware, rather than using virtual memory, the OS has more freedom to change a job's mapping over time. Otherwise, there may be some cases where it is best to physically move some physical pages if two jobs are likely to be co-scheduled together for a long period.

8 Generality of Results

This work has focused on direct-mapped caches because those are most prone to conflict miss problems. But conflict misses are by no means exclusive to those caches. This section will show that these techniques continue to be important even with associative caches, as well as with different cache sizes.

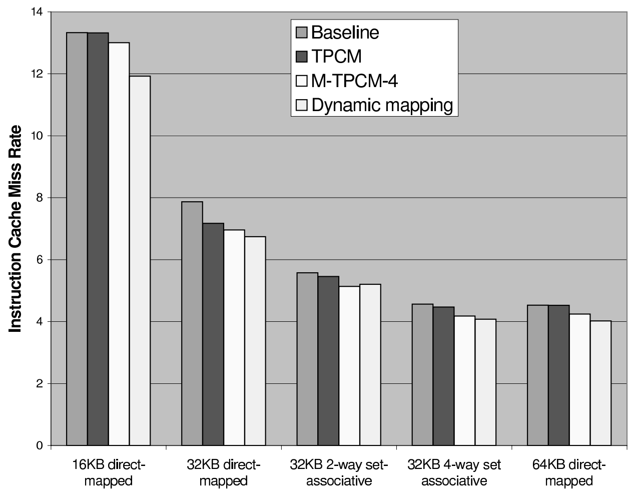

Figure 10. Miss Rates for various Cache Configurations

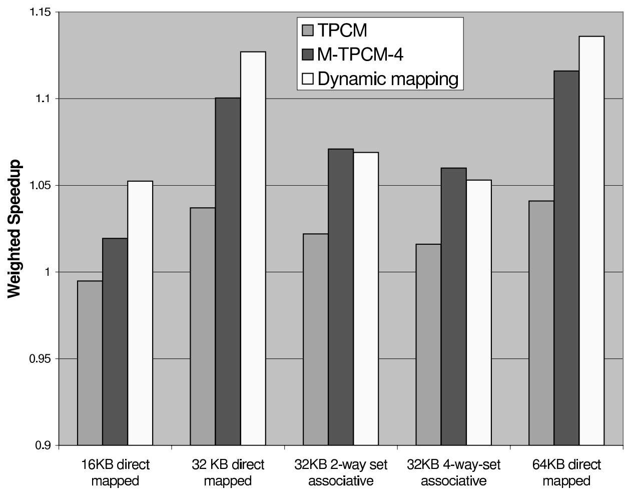

Figure 11. Impact on Performance for various Cache Configurations

The techniques discussed so far adapt easily to associative caches. For M-TPCM, we can treat a 32 KB 2-way set associative cache as a 32 KB direct-mapped cache and arrive at a good mapping. For dynamic mapping, we would want to treat it as we would a 16 KB direct-mapped cache, to insure that the heavy portions of two threads are not mapped together.

Figure 10 shows the reduction in miss rates using both the multithreading-aware M-TPCM technique which lays out both threads, and the dynamic mapping results. Note that for these results, the relative magnitude of the miss rate reductions, even with the associative caches, is maintained. Similarly, Figure 11 shows that speedups are still available for associative instruction caches with these techniques.

9 Conclusions

On a simultaneous multithreading processor with a shared instruction cache, inter-thread conflict misses can be more important than same-thread conflict misses, rendering traditional cache layout optimizations ineffective.

This paper demonstrates techniques that can be applied when multiple programs are compiled at the same time in anticipation of being co-scheduled on the processor. In that case, our best compilation technique results in an average 11% performance improvement over a system with no instruction cache mapping applied, nearly double the improvement provided by a remapping algorithm which ignores inter-thread conflicts.

In the case where the compiler has access to only one of the co-resident threads, we demonstrate techniques for compiling the one program given some profile-generated knowledge of the other program(s).

We also demonstrate a dynamic technique that can map each program independently at compile time, and with minor OS support, map them into the cache at runtime so as to minimize conflicts. This technique achieves an average 13% improvement with two threads and 27% improvement with four, over no remapping.

These techniques provide a set of optimizations that can be applied to virtually any workload that is prone to instruction-cache conflict misses between thread, whether or not the compiler has access to, or even knowledge of, co-resident threads.

Acknowledgments

The authors would like to thank the anonymous reviewers for their helpful comments, and Tryggve Fossum of Intel for suggesting this research direction. This research was supported by NSF grant CCR-0105743, a Powell Foundation Faculty Fellowship, and a CalRA fellowship.

References

[1] A. Agarwal, J. Kubiatowicz, D. Kranz, B.-H. Lim, D. Yeung, G. D'Souza, and M. Parkin. Sparcle: An evolutionary processor design for large-scale multiprocessors. *IEEE Micro*, June 1993.

[2] B. Bershad, D. Lee, T. Romer, and J. Chen. Avoiding conflict misses dynamically in large direct-mapped caches. In *Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 158–170, Oct. 1994.

[3] E. Bugnion, J. Anderson, T. Mowry, M. RosenBlum, and M. Lam. Compiler-directed page coloring for multiprocessors. In *Seventh International Conference on Architectural*

Support for Programming Languages and Operating Systems, Oct. 1996.

[4] J. Carter, W. Hsieh, L. Stoller, M. Swanson, L. Zhang, E. Brunvand, A. Davis, C. Kuo, R. Kuramkote, M. Parker, L. Schaelicke, and T. Tateyama. Impulse:building a smart memory controller. In *Fifth International Symposium on High-Performance Computer Architecture*, Jan. 1999.

[5] J. Chen and B. Leupen. Improving instruction locality with just-in-time code layout. In *USENIX Windows NT Workshop*, Aug. 1997.

[6] N. Gloy, T. Blackwell, M. Smith, and B. Calder. Procedure placement using temporal ordering information. In *MICRO-30 International Symposium on Microarchitecture*, Dec. 1997.

[7] A. Hashemi, D. Kaeli, and B. Calder. Efficient procedure mapping using cache line coloring. In *ACM SIGPLAN'97 Conference on Programming Language Design and Implementation*, pages 171–182, June 1997.

[8] W. Hwu and P. Chang. Achieving high instruction cache performance with an optimizing compiler. In *16th Annual International Symposium on Computer Architecture*, pages 242–251. ACM, 1989.

[9] J. Kalamantianos and D. Kaeli. Temporal-based procedure reordering for improved instruction cache performance. In *Fourth International Symposium on High-Performance Computer Architecture*, pages 244–253, Feb. 1998.

[10] S. McFarling. Program optimization for instruction caches. In *Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 183–191, Apr. 1989.

[11] Compaq chooses SMT for alpha. *Microprocessor Report*, 13(16), Dec. 1999.

[12] Intel embraces multithreading. *Microprocessor Report*, 15(9), Sept. 2001.

[13] K. Pettis and R. Hansen. Profile guided code positioning. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 16–27. ACM, ACM, June 1990.

[14] T. Sherwood, B. Calder, and J. Emer. Reducing cache misses using hardware and software page placement. In *ICS-99 International Conference on Suprcomputing*, June 1999.

[15] A. Snavely and D. Tullsen. Symbiotic jobscheduling for a simultaneous multithreading architecture. In *Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, Nov. 2000.

[16] A. Srivastava and A. Eustace. Atom:a system for building customised program analysis tools. In *PLDI-94 Programming Languages Design and Implementation*, pages 196–205, 1994.

[17] J. Torrellas, C. Xia, and R. Daigle. Optimizing instruction cache performance for operating system intensive workloads. In *Proceedings of the First International Symposium on High-Performance Computer Architecture*, pages 360–369, Jan. 1995.

[18] D. Tullsen. Simulation and modeling of a simultaneous multithreading processor. In *22nd Annual Computer Measurement Group Conference*, Dec. 1996.

[19] D. Tullsen and J. Brown. Handling long-latency loads in a simultaneous multithreading processor. In *34th International Symposium on Microarchitecture*, Dec. 2001.

[20] D. Tullsen, S. Eggers, J. Emer, H. Levy, J. Lo, and R. Stamm. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *23rd Annual International Symposium on Computer Architecture*, May 1996.

[21] D. Tullsen, S. Eggers, and H. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *22nd Annual International Symposium on Computer Architecture*, June 1995.

[22] Y. Yamada, J. Gyllenhaal, G. Haab, and W. Hwu. Data relocation and prefetching for large data sets. In *MICRO-27 International Symposium on Microarchitecture*, pages 118–127, Dec. 1994.