Compiler Techniques for Reducing Data Cache Miss Rate on a Multithreaded Architecture

Subhradyuti Sarkar and Dean M. Tullsen

Department of Computer Science and Engineering
University Of California, San Diego

Abstract. High performance embedded architectures will in some cases combine simple caches and multithreading, two techniques that increase energy efficiency and performance at the same time. However, that combination can produce high and unpredictable cache miss rates, even when the compiler optimizes the data layout of each program for the cache.

This paper examines data-cache aware compilation for multithreaded architectures. Data-cache aware compilation finds a layout for data objects which minimizes inter-object conflict misses. This research extends and adapts prior cache-conscious data layout optimizations to the much more difficult environment of multithreaded architectures. Solutions are presented for two computing scenarios: (1) the more general case where any application can be scheduled along with other applications, and (2) the case where the co-scheduled working set is more precisely known.

1 Introduction

High performance embedded architectures seek to accelerate performance in the most energy-efficient and complexity-effective manner. Two technologies that improve performance and energy efficiency at the same time are caches and multithreading. However, when used in combination, these techniques can be in conflict, as unpredictable interactions between threads can result in high conflict miss rates. It has been shown that in large and highly associative caches, these interactions are not large; however, embedded architectures are more likely to combine multithreading with smaller, simpler caches. The techniques in this paper allow the architecture to maintain these simpler caches, solving the problem in software via the compiler, rather than necessitating more complex and power-hungry caches.

Cache-conscious Data Placement (CCDP) [1] is a technique which finds an intelligent layout for the data objects of an application, so that at runtime objects which are accessed in an interleaved pattern are not mapped to the same cache blocks. On a processor core with a single execution context, this technique has been shown to significantly reduce the cache conflict miss rate and improve performance over a wide set of benchmarks.

However, CCDP loses much of its benefit in a multithreaded environment, such as simultaneous multithreading (SMT) [2,3]. In an SMT processor multiple threads run concurrently in separate hardware contexts. This architecture has

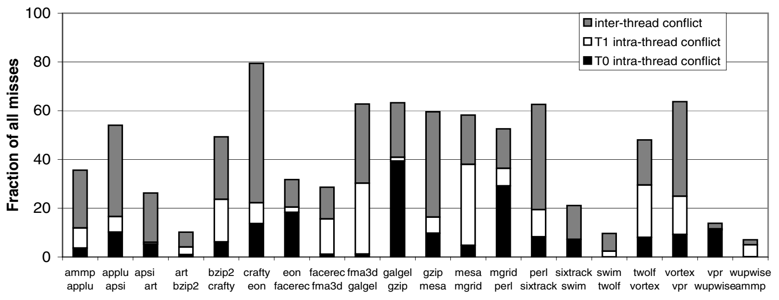

Fig. 1. Percentage of data cache misses that are due to conflict. The cache is 32 KB direct-mapped, shared by two contexts in an SMT processor.

been shown to be a much more energy efficient approach to accelerate processor performance than other traditional performance optimizations [4,5], and thus is a strong candidate for inclusion in high performance embedded architectures. In a simultaneous multithreading processor with shared caches, however, objects from different threads compete for the same cache lines – resulting in potentially expensive inter-thread conflict misses. These conflicts cannot be analyzed in the same manner that was applied successfully by prior work on intra-thread conflicts. This is because inter-thread conflicts are not deterministic.

Figure 1, which gives the percentage of conflict misses for various pairs of co-scheduled threads, shows two important trends. First, inter-thread conflict misses are just as prevalent as intra-thread conflicts (26% vs. 21% of all misses). Second, the infusion of these new conflict misses significantly increases the overall importance of conflict misses, relative to other types of misses.

This phenomenon extends beyond multithreaded processors. Multi-core architectures may share on-chip L2 caches, or possibly even L1 caches [6,7]. However, in this work we focus in particular on multithreaded architectures, because they interact and share caches at the lowest level.

In this paper, we develop new techniques that allow the ideas of CCDP to be extended to multithreaded architectures, and be effective. We consider the following compilation scenarios:

(1) First we solve the most general case, where we cannot assume we know which applications will be co-scheduled. This may occur, even in an embedded processor, if we have a set of applications that can run in various combinations.

(2) In more specialized embedded applications, we will be able to more precisely exploit specific knowledge about the applications and how they will be run. We may have *a priori* knowledge about application sets to be co-scheduled in the multithreaded processor. In these situations, it should be feasible to co-compile, or at least cooperatively compile, these concurrently running applications.

This paper makes the following contributions: (1) We show that traditional multithreading-oblivious cache-conscious data placement is not effective in a multithreading architecture. In some cases, it does more harm than good. (2) We propose two extensions to CCDP that can identify and eliminate most of the inter-thread conflict misses for each of the above mentioned scenarios. We have seen as much as a 26% average reduction in misses after our placement optimization. (3) We show that even for applications with many objects and interleavings, temporal relationship graphs of reasonable size can be maintained without sacrificing performance and quality of placement. (4) We present several new mechanisms that improve the performance and realizability of cache conscious data placement (whether multithreaded or not). These include object and edge filtering for the temporal relationship graph. (5) We show that these algorithms work across different cache configurations, even for set-associative caches. Previous CCDP algorithms have targeted direct-mapped caches – we show that they do not translate easily to set-associative caches. We present a new mechanism that eliminates set-associative conflict misses much more effectively.

2 Related Work

Direct-mapped caches, although faster, more power-efficient, and simpler than set-associative caches, are prone to conflict misses. Consequently, much research has been directed toward reducing conflicts in a direct-mapped cache. Several papers [8,9,10] explore unconventional line-placement policies to reduce conflict misses. Lynch, *et al.* [11] demonstrate that careful virtual to physical translation (page-coloring) can reduce the number of cache misses in a physically-indexed cache. Rivera and Tseng [12] predict cache conflicts in a large linear data structure by computing expected conflict distances, then use intra- and inter-variable padding to eliminate those conflicts. The Split Cache [13] is a technique to virtually partition the cache through special hardware instructions, which the compiler can exploit to put potentially conflicting data structures in isolated virtual partitions.

In a simultaneous multithreading architecture [2,3],various threads share execution and memory system resources on a fine-grained basis. Sharing of the L1 cache by multiple threads usually increases inter-thread conflict misses [2,14,15]. Until now, few studies have been conducted which try to improve cache performance in an SMT processor, particularly without significant hardware support. It has been shown [16] that partitioning the cache into per-thread local regions and a common global region can avoid some inter-thread conflict misses. Traditional code transformation techniques (tiling, copying and block data layout) have been applied, along with a dynamic conflict detection mechanism to achieve significant performance improvement [17]; however, these transformations yield good results only for regular loop structures. Lopez, *et al.* [18] also look at the interaction between caches and simultaneous multithreading in embedded architectures. However, their solutions also require dynamically reconfigurable caches to adapt to the behavior of the co-scheduled threads.

This research builds on the profile-driven data placement proposed by Calder, *et al.* [1]. The goal of this technique is to model temporal relationships between data objects through profiling. The temporal relationships are captured in a *Temporal Relationship Graph* (TRG), where each node represents an object and edges represent the degree of temporal conflict between objects. Hence, if objects P and Q are connected by a heavily weighted edge in the TRG, then placing them in overlapping cache blocks is likely to cause many conflict misses.

We have extended this technique to SMT processors and set associative caches. Also, we have introduced the concept of object and edge trimming - which significantly reduces the time and space complexity of our placement algorithm. Kumar and Tullsen [19] describe techniques, some similar to this paper, to minimize instruction cache conflicts on an SMT processor. However, the dynamic nature of the sizes, access patterns, and lifetimes of memory objects makes the data cache problem significantly more complex.

3 Simulation Environment and Benchmarks

We run our simulations on SMTSIM [20], which simulates an SMT processor. The detailed configuration of the simulated processor is given in Table 1. For most portions of the paper, we assume the processor has a 32 KB, direct-mapped data cache with 64 byte blocks. We also model the effects on set associative caches in Section 6, but we focus on a direct-mapped cache both because the effects of inter-thread conflicts is more severe, and because direct mapped caches continue to be an attractive design point for many embedded designs. We assume the address mappings resulting from the compiler and dynamic allocator are preserved in the cache. This would be the case if the system did not use virtual to physical translation, if the cache is virtually indexed, or if the operating system uses page coloring to ensure that our cache mappings are preserved.

The fetch unit in our simulator fetches from the available execution contexts based on the ICOUNT fetch policy [3] and the *flush* policy from [21], a performance optimization that reduces the overall cost of any individual miss.

It is important to note that a multithreaded processor tends to operate in one of two regions, in regards to its sensitivity to cache misses. If it is latency-limited (no part of the hierarchy becomes saturated, and the memory access time is dominated by device latencies), sensitivity to the cache miss rate is low, because of the latency tolerance of multithreaded architectures. However, if the processor is operating in bandwidth-limited mode (some part of the subsystem is saturated, and the memory access time is dominated by queuing delays), the multithreaded system then becomes very sensitive to changes in the miss rate. For the most part, we choose to model a system that has plenty of memory and cache bandwidth, and never enters the bandwidth-limited regions. This results in smaller observed performance gains for our placement optimizations, but we still see significant improvements. However, real processors will likely reach that saturation point with certain applications, and the expected gains from our techniques would be much greater in those cases.

Table 2 alphabetically lists the 20 SPEC2000 benchmarks that we have used. The SPEC benchmarks represent a more complex set of applications than represented in some of the embedded benchmark suites, with more dynamic memory usage; however, these characteristics do exist in real embedded applications. For

Table 1. SMT Processor Details

Parameter	**Value**
Fetch Bandwidth	2 Threads, 4 Instructions Total
Functional Units	4 Integer, 4 Load/Store, 3 FP
Instruction Queues	32 entry Integer, 32 entry FP
Instruction Cache	32 KB, 2-way set associative
Data Cache	32 KB, direct-mapped
L2 Cache	512 KB, 4-way set associative
L3 Cache	1024 KB, 4-way set associative
Miss Penalty	L1 15 cycles, L2 80 cycles, L3 500 cycles
Pipeline Depth	9 stages

Table 2. Simulated Benchmarks

ID	**Benchmark**	**Type**	**Hit Rate**(%)	**ID**	**Benchmark**	**Type**	**Hit Rate**(%)
1	*ammp*	FP	84.19	11	*gzip*	INT	95.41
2	*applu*	FP	83.07	12	*mesa*	FP	98.32
3	*apsi*	FP	96.54	13	*mgrid*	FP	88.56
4	*art*	FP	71.31	14	*perl*	INT	89.89
5	*bzip2*	INT	94.66	15	*sixtrack*	FP	92.38
6	*crafty*	INT	94.48	16	*swim*	FP	75.13
7	*eon*	INT	97.42	17	*twolf*	INT	88.63
8	*facerec*	FP	81.52	18	*vortex*	INT	95.74
9	*fma3d*	FP	94.54	19	*vpr*	INT	86.21
10	*galgel*	FP	83.01	20	*wupwise*	INT	51.29

our purposes, these benchmarks represent a more challenging environment to apply our techniques. In our experiments, we generated a k-threaded workload by picking each benchmark along with its $(k-1)$ successors (modulo the size of the table) as they appear in Table 2. Henceforth we shall refer to a workload by the ID of its first benchmark. For example, workload 10 (at two threads) would be the combination {*galgel gzip*}. Our experiments report results from a simulation window of two hundred million instructions; however, the benchmarks are fast-forwarded by ten billion dynamic instructions beforehand. Table 2 also lists the L1 hit rate of each application when they are run independently. All profiles are generated running the SPEC *train* inputs, and simulation and measurement with the *ref* inputs. We also profile and optimize for a larger portion of execution than we simulate.

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. In this paper we use weighted speedup [21] to report our results. 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 toward certain threads.

4 Independent Data Placement

In the next two sections, we consider two different execution scenarios. In this section, we solve the more general and difficult scenario, where the compiler actually does not know which applications will be scheduled together dynamically, or the set of co-scheduled threads changes frequently; however, we assume all applications will have been generated by our compiler. In this execution scenario, then, co-scheduling will be largely unpredictable and dynamic. However, we can still compile programs in such a way that conflict misses are minimized. Since all programs would essentially be compiled in the same way, some support from the operating system, runtime system, or the hardware is required to allow each co-scheduled program to be mapped onto the cache differently.

We have modified CCDP techniques to create an intentionally unbalanced utilization of the cache, mapping objects to a *hot* portion and a *cold* portion. This does not necessarily imply more intra-thread conflict misses. For example, the two most heavily accessed objects in the program can be mapped to the same cache index without a loss in performance, if they are not typically accessed in an interleaved pattern – this is the point of using the temporal relationship graph of interleavings to do the mapping, rather than just using reference counts. CCDP would typically create a more balanced distribution of accesses across the cache; however, it can be tuned to do just the opposite. This is a similar approach to that used in [19] for procedure placement, but applied here to the placement of data objects.

However, before we present the details of the object placement algorithm, we first describe the assumptions about hardware or OS support, how data objects are identified and analyzed, and some options that make the CCDP algorithms faster and more realizable.

4.1 Support from Operating System or Hardware

Our independent placement technique (henceforth referred to as IND) repositions the objects so that they have a *top-heavy* access pattern, i.e. most of the memory accesses are limited to the *top portion* of the cache. Now let us consider an SMT processor with two hardware contexts, and a shared L1 cache (whose size is at least twice the virtual-memory page size). If the architecture uses a virtual cache, the processor can xor the high bits of the cache index with a hardware context ID (e.g., one bit for 2 threads, 2 bits for 4 threads), which will then map the hot portions of the address space to different regions of the cache. In a physically indexed cache, we don't even need that hardware support. When the operating system loads two different applications in the processor, it ensures (by page coloring or otherwise) that heavily accessed virtual pages from the threads do not collide in the physically indexed cache.

For example, let us assume an architecture with a 32 KB data cache, 4 KB memory pages, and two threads. The OS or runtime allocates physical pages to virtual pages such that the three least significant bits of the page number are preserved for thread zero, and for thread one the same is done but with the third bit reversed. Thus, the mapping assumed by the compilers is preserved, but with each thread's "hot" area mapped to a different half of the cache. This is simply an application of page coloring, which is not an unusual OS function.

4.2 Analysis of Data Objects

To facilitate data layout, we consider the address space of an application as partitioned into several *objects*. An *object* is loosely defined as a contiguous region in the (virtual) address space that can be relocated with little help from the compiler and/or the runtime system. The compiler typically creates several objects in the code and data segment, the starting location and size of which can be found by scanning the symbol table. A section of memory allocated by a `malloc` call can be considered to be a single *dynamic object*, since it can easily be relocated using an instrumented front-end to `malloc`. However, since the same invocation of `malloc` can return different addresses in different runs of an application – we need some extra information to identify the dynamic objects (that is, to associate a profiled object with the same object at runtime). Similar to [1], we use an additional tag (henceforth referred to as `HeapTag`) to identify the dynamic objects. `HeapTag` is generated by xor-folding the top four addresses of the return stack and the call-site of `malloc`. We do not attempt to reorder stack objects. Instead the stack is treated as a single object.

After the objects have been identified, their reference count and lifetime information over the simulation window can be retrieved by instrumenting the application binary with a tool such as ATOM [22]. Also obtainable are the temporal relationships between the objects, which can be captured using a temporal relationship graph (henceforth referred to as `TRGSelect` graph). The `TRGSelect` graph contains nodes that represent objects (or portions of objects) and edges between nodes contain a weight which represents how many times the two objects were interleaved in the actual profiled execution.

Temporal relationships are collected at a finer granularity than full objects – mainly because some of the objects are much larger than others, and usually only a small portion of a *bigger* object has temporal association with the *smaller* one. It is more logical to partition the objects into fixed size chunks, and then record the temporal relationship between chunks. Though all the chunks belonging to an object are placed sequentially in their original order, having finer-grained temporal information helps us to make more informed decisions when two conflicting objects must be put in an overlapping cache region. We have set the chunk size equal to the block size of the targeted cache. This provides the best performance, as we now track conflicts at the exact same granularity that they occur in the cache.

4.3 Object and Edge Filtering

Profiling a typical SPEC2000 benchmark, even for a partial profile, involves tens of thousands of objects, and generates hundreds of millions of temporal relationship edges between objects. To make this analysis manageable, we must reduce both the number of nodes (the number of objects) as well as the number of edges (temporal relationships between objects) in the `TRGSelect` graph. We classify objects as unimportant if their reference count is zero, or the sum of the weights of incident edges is below a threshold.

If a `HeapTag` assigned to a heap object is non-unique, we mark all but the most frequently accessed object having that `HeapTag` as unimportant. Multiple objects usually have the same `HeapTag` when dynamic memory is being allocated in a loop and they usually have similar temporal relationship with other objects.

A similar problem exists for building the `TRGSelect` graph. Profiling creates a `TRGSelect` graph with a very large number of edges. Since it is desirable to store the entire `TRGSelect` graph in memory, keeping all these edges would not be practical. Fortunately, we have noted that in a typical profile more than 90% of all the edges are *light-weight*, having an edge weight less than one tenth of the heavier edges. We use the following epoch-based heuristic to periodically trim off the *potentially* light-weight edges, limiting the total number of edges to a preset maximum value. In a given epoch, edges with weight below a particular threshold are marked as *potentially light-weight.* In the next epoch, if the weight of an edge marked as *potentially light-weight* does not increase significantly from the previous epoch, it is deleted from the `TRGSelect` graph. The threshold is liberal when the total number of edges is low, but made more aggressive when the number of edges nears our preset limit on the number of edges.

In this algorithm, then, we prune the edges dynamically during profiling, and prune the objects after profiling, but before the placement phase. We find pruning has little impact on the quality of our results.

4.4 Placement Algorithm

For independent data placement, the cache blocks are partitioned into *native* and *foreign* sets. If we know the application is going to be executed on an SMT processor with k contexts, the top $\frac{1}{k}$ cache blocks are marked as *native*, and other cache blocks are marked as *foreign.* For any valid placement of an object in a native block, we define an associated cost. That cost is the sum of the costs for each chunk placed in the contiguous cache blocks. The cost of a chunk is the edge weight (interleaving factor) between that chunk and all chunks of other objects already placed in that cache block. If the cache block is marked as foreign, a bias is added to the overall cost to force the algorithm to only place an object or part of an object in the foreign section if there is no good placement in the native. The bias for an object is set to be λ times the maximum edge weight between a chunk belonging to this object and any other chunk in the `TRGSelect` graph. Varying this bias allows a tradeoff between combined cache performance, and uncompromised cache performance when running alone. Our basic placement heuristic is to order the objects and then place them each, in that order, into the cache where they incur minimal cost. Since some objects are fundamentally different in nature and size from others, we came up with a set of specialized placement strategies, each targetting one particular type of object. Specifically, we will separately consider constant objects, small global objects, important global objects, and heap objects.

An object which resides in the code segment is defined as a constant object. Constant objects are placed in their default location (altering the text segment might have adverse effects on the instruction cache). However when other objects are placed in cache, their temporal relationship with the constant objects is taken into consideration.

Small global objects are handled differently than larger objects, allowing us to transform potential conflicts into cache prefetch opportunities. A statically allocated object which resides in the data segment is defined as a global object. Furthermore, a global object is classified as *small* if its size is less than three-fourths of the block size. As in [1], we try to cluster the small global objects that have heavily-weighted edges in the TRG and place them in the same cache block. Accessing any of the objects in the cluster will prefetch the others, avoiding costly cache misses in the near future. Small global objects are clustered greedily, starting with the pair of objects with the highest edge weight between them.

After a cluster has been formed, nodes representing individual objects in the cluster are coalesced into a single node (in the `TRGSelect` graph). The cluster will be assigned a starting location along with other non-small objects in the next phase of the placement algorithm.

Next, we place the global objects. Our greedy placement algorithm is sensitive to the order in which the objects are placed. By experimentation, we have found the following approach to be effective. We build a `TRGPlace` graph from the `TRGSelect` graph, where chunks of individual objects are merged together into a single node (edge weights are adjusted accordingly). Next, the most heavily weighted edge is taken from the `TRGPlace` graph. The two objects connected by that edge are placed in the cache, and marked as placed; however, recall that the actual placement still uses the `TRGSelect` graph, which tracks accesses to the individual chunks.

In each subsequent iteration of the algorithm, an unplaced object is chosen which maximizes the sum of `TRGPlace` edge-weights between itself and the objects that have been already placed. In case of a tie, the object with a higher reference count is given preference.

Unimportant global objects are placed so as to fill holes in the address space created by the allocation of the important global objects.

Heap objects also reside in the data segment, however they are dynamically created and destroyed at runtime using `malloc` and `free` calls. Specifying a placement for heap objects is more difficult because a profiled heap object might not be created, or might have different memory requirements in a later execution of the same application with different input. Thus, we determine the placement assuming the object is the same size, but only indicate to our custom `malloc` the location of the first block of the desired mapping. The object gets placed there, even if the size differs from the profiled run.

During execution, our customized `malloc` first computes the `HeapTag` for the requested heap object. If the `HeapTag` matches any of the recorded `HeapTag`s for which a customized allocation should be performed, `malloc` returns a suitably aligned address from the available memory. When the newly created heap

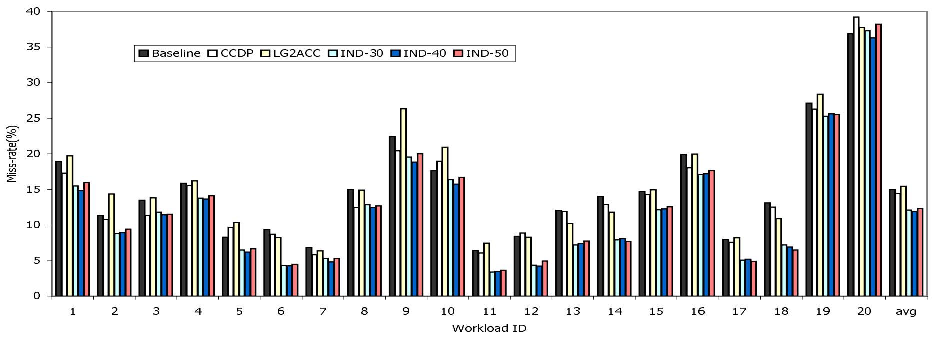

Fig. 2. Data Cache miss rate after Independent Placement (IND)

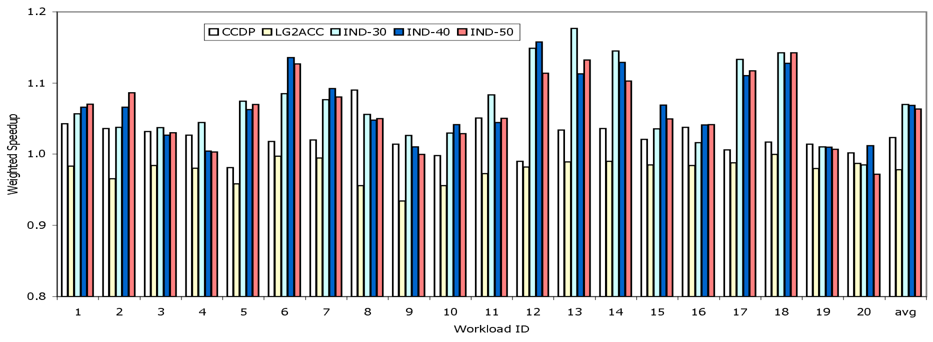

Fig. 3. Weighted Speedup after Independent Placement (IND)

object is brought in the cache, it occupies the blocks specified by the placement algorithm.

4.5 Independent Placement Results

The effects of data placement by IND on miss rate and weighted speedup are shown in Figure 2 and Figure 3 respectively. The Baseline series shows data cache miss rate without any type of placement optimization. CCDP shows the miss rate if traditional CCDP is performed on each of the applications. Since CCDP ignores inter-thread conflicts, for four workloads CCDP actually increases the miss rate over Baseline. LG2ACC shows the miss rate if L1 data cache is implemented as a *Double access local-global split cache* [16]. Split caches are designed to reduce conflicts in a multithreaded workload, though in our experiments the split cache was not overly effective. The final three series (IND-30, IND-40, IND-50) show the effect of co-ordinated data placement with λ (the placement bias) set to 0.30, 0.40 and 0.50 respectively. The figure shows that no single value of λ is universally better than others, though all of them yield improvement over traditional CCDP. For future work, it may be that setting λ individually for each application, based on number and size of objects, for example, will yield even better results. A careful comparison of Figure 2 and Figure 1 shows that the effectiveness of co-ordinated data placement is heavily correlated with the fraction of cache misses that are caused by conflicts. On workloads like {*crafty eon*} (workload 6) or {*gzip mesa*} (11), more than half of the cache misses are caused by conflicts, and IND-30 reduces the miss rate by 54.0% and 46.8%, respectively. On the other hand, only 6% of the cache misses in workload {*wupwise ammp*} (20) are caused by conflicts, and IND-30 achieves only a 1% gain.

On average, IND reduced overall miss rate by 19%, reduced total conflict misses by more than a factor of two, and achieved a 6.6% speedup. We also ran experiments with limited bandwidth to the L2 cache (where at most one pending L1 miss can be serviced in every two cycles), and in that case the performance tracked the miss rate gains somewhat more closely, achieving an average weighted speedup gain of 13.5%.

IND slightly increases intra-thread cache conflict (we still are applying cache-conscious layout, but the bias allows for some inefficiency from a single-thread standpoint). For example, the average miss rate of the applications, when run alone with no co-scheduled jobs increases from 12.9% to 14.3%, with λ set to 0.4. However, this result is heavily impacted by one application, *ammp* for which this mapping technique was largely ineffective due to the large number of heavily-accessed objects. If the algorithm was smart enough to just leave *ammp* alone, the average single-thread miss rate would be 13.8%. Unless we expect single-thread execution to be the common case, the much more significant impact on multithreaded miss rates makes this a good tradeoff.

5 Co-ordinated Data Placement

In many embedded environments, applications that are going to be co-scheduled are known in advance. In such a scenario, it might be more beneficial to co-compile those applications and lay out their data objects in unison. This approach provides more accurate information about the temporal interleavings of objects to the layout engine.

Our coordinated placement algorithm (henceforth referred to as CORD) is similar in many ways to IND. However, in CORD the cache is not split into *native* and *foreign* blocks, and thus there is no concept of *biasing*. In CORD, the `TRGSelect` graph from all the applications are merged together and important objects from all the applications are assigned a placement in a single pass.

5.1 Merging of `TRGSelect` Graphs

The `TRGSelect` graph generated by executing the instrumented binary of an application captures the temporal relationships between the objects of that application. However, when two applications are co-scheduled on an SMT processor, objects from different execution contexts will vie for the same cache blocks in the shared cache. We have modeled inter-thread conflicts by merging the `TRGSelect` graphs of the individual applications. It is important to note that we profile each application separately to generate two graphs, which are then merged probabilistically. While we may have the ability to profile the two threads running together and their interactions, there is typically little reason to believe the same interactions would occur in another run. The exception would be if the two threads communicate at a very fine granularity, in which case it would be better to consider them a single parallel application.

Assigning temporal relationship weights between two objects from different applications requires modeling interactions that are much less deterministic than interactions between objects in the same thread. We thus use a probabilistic model to quantify expected interactions between objects in different threads.

Two simplifying assumptions have been made for estimating the inter-thread temporal edge weights, which make it easier to quantify the expected interactions between objects in separate threads. (1) The relative execution speeds of the two threads is known a priori. Relative execution speed of co-scheduled threads typically remains fairly constant unless one of the threads undergoes a phase change – which can be discovered via profiling. (2) Within its lifetime, an object is accessed in a regular pattern, i.e. if the lifetime of an object o is k cycles, and the total reference count of o is n, then o is accessed once every $\frac{k}{n}$ cycles. Few objects have very skewed access pattern so this assumption gives a reasonable estimate of the number of references made to an object in a particular interval.

We use these assumptions to estimate the interleavings between two objects (in different threads). From the first assumption, along with the known lifetimes of objects, we can calculate the likelihood that two objects have overlapping lifetimes (and the expected duration). From the second assumption, we can estimate the number of references made to those objects during the overlap. The number of interleavings cannot be more than twice the lesser of the two (estimated) reference counts. We apply a scaling factor to translate this worst-case estimate of the interleavings during an interval, into an expected number of interleavings. This scaling factor is determined experimentally. To understand the point of the scaling factor, if the two objects are being accessed at an equal rate by the two threads, but we always observe a run of two accesses from one thread before the other thread issues an access, the scaling factor would be 0.50.

In our experiments we have found it sufficient to only put temporal edges between *important* objects of each application, which eliminates edge explosion.

5.2 Coordinated Placement Results

The miss-rate impact and weighted speedup achieved by CORD is shown in Figures 4 and 5. The three series CORD-60, CORD-70 and CORD-80 represents the result of independent data placement with scaling factor set to 0.6, 0.7 and 0.8 respectively. The scaling factor represents the degree of interleaving we expect between memory accesses from different threads accessing the same cache set.

In most of the workloads, the speedup is somewhat more than that obtained from independent placement, thus confirming our hypothesis that having access to more information about conflicting objects leads to better placement decisions. On the average CORD reduced miss rate by 26% and achieved 8.8%

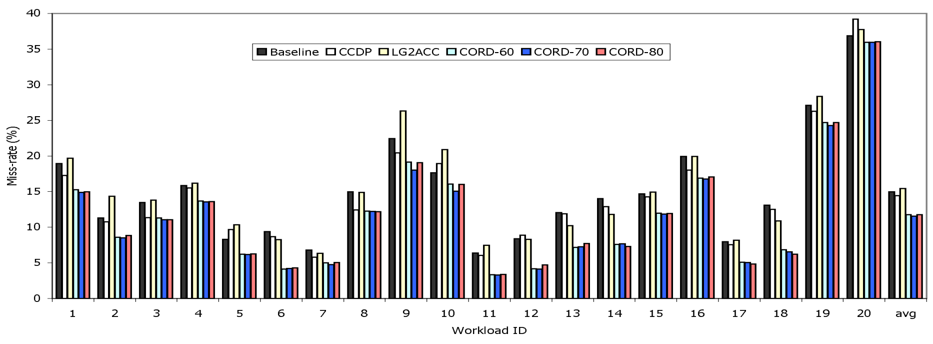

Fig. 4. Data Cache miss rate after Coordinated Placement (CORD)

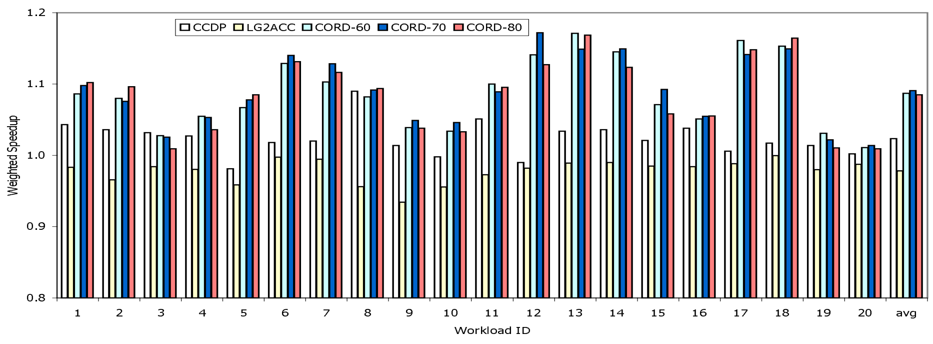

Fig. 5. Weighted Speedup after Coordinated Placement (CORD)

speedup. However, if one of these *optimized* applications is run alone (i.e. without its counterpart application) we do sacrifice single-thread performance slightly, but the effect is much less than the gain when co-scheduled. The amount of the single-thread loss depends somewhat on the scaling factor. The average Baseline miss rate was 12.9%. With coordinated placement, and a scaling factor of 0.7, the average single-thread miss rate goes up to 13.1%, but when the scaling factor is 0.8, the miss rate actually becomes 12.7%.

6 Exploring other Processor and Cache Configurations

We have demonstrated the effectiveness of our placement techniques for a single hardware configuration. We also did extensive sensitivity analysis, to understand how these techniques work as aspects of the architecture change. We lack space to show those results in detail, but include a brief summary here. We examined alternate cache sizes and organizations, different latencies, and different levels of threading.

Cache associativity is the most interesting alternative, in large part because proposed CCDP algorithms do not accommodate associative caches. The naive approach would model a set-associative cache as a direct-mapped cache with the same number of sets. It would over-state conflicts, but would hopefully produce a good mapping. In fact, we found that it did not.

Any mapping function that used our current TRGs would be an approximation, because we only capture 2-way conflicts. Profiling and creating a hypergraph to capture more complex conflicts would be computationally prohibitive. However, we found the following algorithm to work well, using our existing TRG. We have adjusted our default placement algorithm such that for a k-way set associative cache, an object incurs placement cost only if is placed in a set where at least k objects have already been placed. This new policy tends to fill up every set in the associative cache to its *maximum capacity* before potentially conflicting objects are put in the set that already contains more than k objects.

The actual results for set-associative caches are indicative of the low incidence of conflict misses for these workloads. However, we do see that our techniques are effective – we eliminate the vast majority of remaining conflict misses. For a 16 KB, 2-way cache, we reduce total miss rate from 15.8% to 13.8%.

Our placement techniques were designed to adapt easily to processors having more than two execution contexts. For co-ordinated data placement of k applications, k `TRGSelect` graphs must be merged together before placement. Independent data placement requires the cache be partitioned into k regions, where each region contains the *hot* objects from one of the applications. For a 4-thread processor, IND-30 and CORD-60 reduced miss rates by 14% and 22% on the average; however, the actual speedups were smaller, due to SMT's ability to tolerate cache latencies. However, there are other important advantages of reducing L1 miss rate, like lower power dissipation.

7 Conclusion

As we seek higher performance embedded processors, we will increasingly see architectures that feature caches and multiple thread contexts (either through multithreading or multiple cores), and thus we shall see greater incidence of threads competing for cache space. The more effectively each application is tuned to use the caches, the more interference we see between competing threads.

This paper demonstrates that it is possible to compile threads to share the data cache, to each thread's advantage. We specifically address two scenarios. Our first technique does not assume any prior knowledge of the threads which might be co-scheduled together, and hence is applicable to all general-purpose computing environments. Our second technique shows that when we do have more specific knowlege about which applications will run together, that knowledge can be exploited to enhance the quality of object placement even further. Our techniques demonstrated 26% improvement in miss rate and 9% improvement in performance, for a variety of workloads constructed from the SPEC2000 suite. It is also shown that our placement techniques scale effectively across different hardware configurations, including set-associative caches.

References

1. Calder, B., Krintz, C., John, S., Austin, T.: Cache-conscious data placement. In: Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (1998)
2. Tullsen, D.M., Eggers, S., Levy, H.M.: Simultaneous multithreading: Maximizing on-chip parallelism. In: Proceedings of the 22nd Annual International Symposium on Computer Architecture (1995)
3. Tullsen, D.M., Eggers, S.J., Emer, J.S., Levy, H.M., Lo, J.L., Stamm, R.L.: Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In: Proceedings of the 23rd Annual International Symposium on Computer Architecture (1996)
4. Li, Y., Brooks, D., Hu, Z., Skadron, K., Bose, P.: Understanding the energy efficiency of simultaneous multithreading. In: Intl. Symposium on Low Power Electronics and Design (2004)
5. Seng, J., Tullsen, D., Cai, G.: Power-sensitive multithreaded architecture. In: International Conference on Computer Design (September 2000)
6. Kumar, R., Jouppi, N., Tullsen, D.M.: Conjoined-core chip multiprocessing. In: 37th International Symposium on Microarchitecture (December 2004)
7. Dolbeau, R., Seznec, A.: Cash: Revisiting hardware sharing in single-chip parallel processor. In: IRISA Report 1491 (November 2002)
8. Agarwal, A., Pudar, S.: Column-associative caches: A technique for reducing the miss rate of direct-mapped caches. In: International Symposium On Computer Architecture (1993)
9. Topham, N., González, A.: Randomized cache placement for eliminating conflicts. IEEE Transactions on Computer 48(2) (1999)
10. Seznec, A., Bodin, F.: Skewed-associative caches. In: International Conference on Parallel Architectures and Languages, pp. 305–316 (1993)
11. Lynch, W.L., Bray, B.K., Flynn, M.J.: The effect of page allocation on caches. In: 25th Annual International Symposium on Microarchitecture (1992)
12. Rivera, G., Tseng, C.W.: Data transformations for eliminating conflict misses. In: SIGPLAN Conference on Programming Language Design and Implementation, pp. 38–49 (1998)
13. Juan, T., Royo, D.: Dynamic cache splitting. In: XV International Confernce of the Chilean Computational Society (1995)
14. Nemirovsky, M., Yamamoto, W.: Quantitative study on data caches on a multistreamed architecture. In: Workshop on Multithreaded Execution, Architecture and Compilation (1998)
15. Hily, S., Seznec, A.: Standard memory hierarchy does not fit simultaneous multithreading. In: Proceedings of the Workshop on Multithreaded Execution Architecture and Compilation (with HPCA-4) (1998)
16. Jos, M.G.: Data caches for multithreaded processors. In: Workshop on Multithreaded Execution, Architecture and Compilation (2000)
17. Nikolopoulos, D.S.: Code and data transformations for improving shared cache performance on SMT processors. In: International Symposium on High Performance Computing, pp. 54–69 (2003)
18. Lopez, S., Dropsho, S., Albonesi, D.H., Garnica, O., Lanchares, J.: Dynamic capacity-speed tradeoffs in smt processor caches. In: Intl. Conference on High Performance Embedded Architectures & Compilers (January 2007)

19. Kumar, R., Tullsen, D.M.: Compiling for instruction cache performance on a multithreaded architecture. In: 35th Annual International Symposium on Microarchitecture (2002)
20. Tullsen, D.M.: Simulation and modeling of a simultaneous multithreading processor. In: 22nd Annual Computer Measurement Group Conference (December 1996)
21. Tullsen, D.M., Brown, J.: Handling long-latency loads in a simultaneous multithreaded processor. In: 34th International Symposium on Microarchitecture (December 2001)
22. Srivastava, A., Eustace, A.: Atom: A system for building customized program analysis tools. In: SIGPLAN Notices, vol. 39, pp. 528–539 (2004)