

Data Layout for Cache Performance 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.

It is shown that these techniques reduce data cache misses for a variety of cache architectures, multithreading environments, and cache latencies.

1 Introduction

High performance embedded architectures seek to accelerate performance in the most energy-efficient and complexity-effective manner. Cacheing and multithreading are two technologies that improve performance and energy efficiency at the same time. 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. This paper demonstrates techniques which allow the architecture to maintain these simpler caches, rather than necessitating more complex and power-hungry caches. It does so by solving the problem in software via the compiler and the runtime, rather than through more complex hardware.

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.

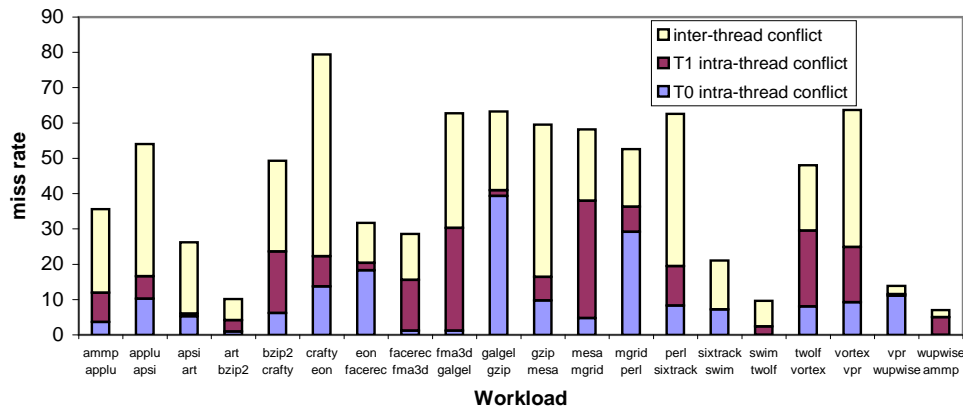


Fig. 1: Percentage of data cache misses that are due to conflict. The cache is 32 KB direct-mapped, shared by two contexts in an SMT processor. The program listed on top is T1.

However, in a multithreaded environment, such as simultaneous multithreading (SMT) [2, 3], CCDP can lose much of its benefit, or even reduce performance. In an SMT processor multiple threads run concurrently in separate hardware contexts. This architecture has been shown to be a much more energy efficient approach to accelerate processor performance than other traditional performance optimizations [4, 5]. 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, shows the percentage of conflict misses for various pairs of co-scheduled threads. This figure 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.

Inter-thread cache conflicts are not strictly confined to multithreaded architectures. We also see this phenomenon in multi-core architectures. Multi-cores may share on-chip L2 caches, or possibly even L1 caches [6, 7]. A data-layout strategy that reduces both intra-thread and inter-thread conflict misses will be helpful in those architectural scenarios as well. 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) In the most general case, 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. In this scenario, the compiler does not know which applications are going to be co-scheduled by the operating system or runtime system, and in fact the combination of co-scheduled threads may even change over the lifetime of a particular thread.

(2) In more specialized environments, 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) It shows 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) It proposes two extensions to CCDP that can identify and eliminate most of the inter-thread conflict misses for each of the above mentioned scenarios. We show as much as a 26% average reduction in misses after our placement optimization. (3) It shows 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) It presents 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. We show results for various caches and cache latencies, including 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. (6) Additionally, we extend these techniques to higher numbers of threads.

The rest of the paper is organized as follows. Section 2 discusses related work. Our simulation environment and benchmarks are described in Section 3. Section 4 and Section 5 provide algorithms and results for independent and co-ordinated data placement methods respectively. Section 6 shows that these techniques can work across a broad range of cache and processor configurations. We conclude in Section 7.

2 Related Work

Direct-mapped caches, although faster 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 [8–10] 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. A compiler-directed partitioning of the process address-

space for a real-time system is described in [13], such that no pre-emptible process will share cache location with other processes.

The Split Cache [14] 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.

Other works [15, 16] dynamically detect and remove conflict misses, without requiring any support from the compiler. These methods logically partition the cache into *pages*, and can recolor conflicting pages to reduce conflict misses. This research attempts to reduce cache conflict misses without specialized hardware, or reducing the ability of any single thread to use the entire cache.

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, 17, 18]. 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 [19] that partitioning the cache into per-thread local regions and a common global region can avoid some inter-thread conflict misses. Compiler directed cache partitioning for SMT processors has been explored by May, *et al.* [20]. However, static partitioning reduces the amount of cache memory available to a particular thread, which is undesirable. 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 [21]; however, these transformations yield good results only for regular loop structures. Lopez, *et al.* [22] 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. The TRG is constructed by keeping a queue of objects accessed in the recent past. The queue is examined at each memory reference to check if the newly accessed object has a previous occurrence. Accessing other objects between two successive accesses to the same object indicates a temporal conflict. A simple example of a TRG and a possible resulting cache mapping is shown in Figure 2.

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 [23] describe techniques, some similar to this paper, to minimize instruction cache conflicts on an SMT processor. However, the dy-

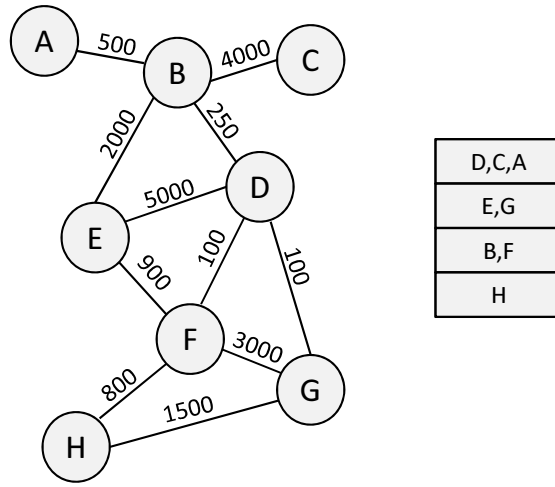


Fig. 2: A simplified Temporal Relationship Graph of the interleavings of 8 equal-sized objects (left), and a mapping of those objects (right) into a cache big enough to hold four objects, which minimizes conflicts between objects.

dynamic nature of the sizes, access patterns, and lifetimes of memory objects makes the data cache problem significantly more complex.

This paper contains several enhancements over a prior published version [24]. For example, this version contains more detailed descriptions of the compiler algorithms used, and several new results, particularly in Section 6.

3 Simulation Environment and Benchmarks

We run our simulations on SMTSIM [25], 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 are more severe, and because direct-mapped caches can 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 [26], a performance optimization that reduces the overall cost of any individual miss. The ICOUNT fetch policy gives fetch priority to that thread which has the fewest instructions in the front end (fetch/decode/rename/queue) stages of the pipeline,

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 1: SMT Processor Details

thus always seeking to provide an even mix of instructions in the instruction window to maximize parallelism. The *flush* policy recognizes that in the presence of very long memory latencies, it is better for a stalled thread to release all held resources for use by non-stalled threads.

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 our purposes, these benchmarks represent a more challenging environment to apply our techniques. In our experiments, we generate 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 $\{galgel\ zip\}$. 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 to ensure that we are executing in the main execution body of the application. Table 2 also lists the L1 hit rate of each application when run independently. All profiles (used to drive

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

Table 2: Simulated Benchmarks

the compiler and layout optimizations) are generated running the SPEC *train* inputs, and simulation and measurement with the *ref* inputs. We also profile and optimize for a much 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 [26] to report our results.

Weighted speedup (WS) is given by

$$WS = \frac{1}{\text{number of threads}} \sum_{\text{threads}} \frac{IPC_{\text{new}}}{IPC_{\text{baseline}}}$$

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

The next two sections handle two different execution scenarios. In this first 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 the following section, we handle the case where we have specific knowledge about which jobs will be co-scheduled.

In the current 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.

CCDP techniques tend to create balanced access across the cache. 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 [23] 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. 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 and 4 KB memory pages – so the cache can accommodate 8 memory pages. Physical pages whose page number is from the set $L = \{0, 1, 2, 3\}$ (modulo 8) map to the *top half* of the cache. Similarly, pages having page-numbers from the set $U = \{4, 5, 6, 7\}$ (modulo 8) map to the *bottom half*. The compiler creates an unbalanced partition by placing most of the heavily accessed objects in virtual pages having page-number from set L (modulo 8). During execution, before the OS allocates a physical page for the virtual page V of process p , it examines the virtual page number of V and the hardware context in which p is running. If the page-number of V is in the set L and p is running in context 0, the OS tries to allocate a physical page whose page number is from the set L . If p is running in context 1 instead, the OS tries to allocate a physical page having page-number from the set U . 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 a common 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`.

Reordering the objects in the stack segment can be more difficult. First, no symbol table entry is created for the objects that are allocated in stack frames. Second, addresses on the stack are specified relative to the stack or frame pointer – so the same variable can be assigned different virtual addresses at different points of execution, based on the current call stack. Hence, in this analysis we treat the whole stack segment as a single object. To place stack objects more finely could require adding significant padding to the stack, which would reduce spatial locality, and increase stack overflow events. As a result, our techniques are not expected to be particularly effective for stack objects. However, stack objects tend to be short-lived and accessed with high temporal locality. Therefore, our techniques tend to still be effective overall for two reasons. First, stack objects tend to have low miss rates, and second, short-lived objects tend to be ignored (marked as unimportant) in our placement algorithm, anyway.

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 [27]. Also found 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. The size of the chunk used for tracking conflicts is an important policy decision –

smaller chunks capture more information at the expense of a larger TRGSelect graph. 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 few millions of committed instructions, 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. This is possible because finding a suitable placement for all the identifiable objects is not necessary. Most of these objects are rarely accessed and/or have a very short life-time – hence their relative placement with respect to other objects has little effect on L1 miss rates. We classify objects as *unimportant* if their reference count is zero, or the sum of the weights of incident edges in TRGSelect graph lies below a certain threshold. In our experiment, that threshold was set to be either first percentile or fifth percentile – depending on the total number of objects enumerated for a particular workload. Object filtering follows Algorithm 1.

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. Since heap objects with the same `HeapTag` would be indistinguishable from one another to the customized memory allocator, making a placement decision based on the most prominent member of the group seems to be a logical choice.

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. Algorithm 2 describes the edge trimming more precisely. In practice, we have noticed that a TRGSelect graph with an upper threshold of 10 million edges can capture all the important temporal relationships between the object-chunk pairs.

```

1: mark all objects as important
2: if an object has a reference count of zero then
3:   mark the object is unimportant
4: end if
5: for all object  $o$  with nonzero reference count do
6:   TRGSum[ $o$ ]  $\leftarrow$  sum of TRG edge weights incident on this object
7: end for
8: if total number of objects is less than 4096 then
9:   find the objects below 1 percentile (based on TRGSum)
10:  mark these objects as unimportant
11: else
12:  find the objects below 5 percentile (based on TRGSum)
13:  mark these objects as unimportant
14: end if
15: for all non-unique HeapTag do
16:  add all objects having that HeapTag to a list  $L$ 
17:  find the object  $p$  in  $L$  having maximum reference count
18:  delete  $p$  from  $L$ 
19:  mark all objects in  $L$  as unimportant
20: end for

```

Algorithm 1: Object Filtering

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.

The various parameters that we have determined via experiment for Algorithm 2 are given in Table 3.

4.4 Building the TRGSelect graph

To build the TRGSelect graph, memory references in the profiling window are scanned sequentially and each of these memory references is attributed to some profiled object. Then, the access patterns with respect to other objects in the window are used to add appropriate edges (or increase edge-weights) in the TRGSelect graph, with the help of the following data structures:

1. set O_{TRG} : objects of a given executable, which is also the set of vertices of TRGSelect graph
2. set E_{TRG} : edges of the TRGSelect graph
3. queue TRGQueue: A FIFO queue which records a finite history of object access patterns in the profile

Ideally, TRGQueue (the reference history window) should be able to grow without any bound so that all interleavings are accurately reflected in TRGSelect. However, maintaining such an accurate history is space and time-prohibitive, and also quite unnecessary for the following reason. An object, after being brought into the cache, does not stay there indefinitely – it typically gets evicted after

```

1: if number of edges is TRGSelect > trigger_value then
2:   low_cutoff  $\leftarrow$   $A$  percentile of all edge weights
3:   high_cutoff  $\leftarrow$   $\gamma$  percentile of all edge weights
4:   for all edges  $E$  in TRGSelect graph do
5:     if  $E$  is marked as ‘spurious’ and its weight < high_cutoff then
6:       delete  $E$ 
7:     end if
8:   end for
9:   for all edges  $E$  in TRGSelect graph do
10:    if weight of  $E$  < low_cutoff then
11:      mark  $E$  as ‘spurious’
12:    else
13:      mark  $E$  as ‘important’
14:    end if
15:  end for
16: end if
17: if number of edges is TRGSelect > max_trg_size then
18:   aggressive_cutoff  $\leftarrow$   $\Theta$  percentile of all edge weights
19:   for all edges  $E$  in TRGSelect graph do
20:     if weight of  $E$  < aggressive_cutoff then
21:       delete  $E$ 
22:     end if
23:   end for
24: end if
25: trigger_value  $\leftarrow$  number of edges in TRGSelect + trigger_delta

```

Algorithm 2: Edge Filtering for a Particular Epoch

Parameter	Value
initial <code>trigger_value</code>	5×10^6
<code>trigger_delta</code>	10^6
<code>max_trg_size</code>	12×10^6
λ	30
γ	40
θ	30

Table 3: Parameters for Edge Filtering

a while due to a conflict or capacity miss. Thus, if consecutive accesses to an object are so far apart as to not appear in a large window, the particular interleavings not recorded are less important (that is, it would be very difficult to remove all interleavings and keep the object in the cache over a long time period, anyway). We have observed that we can trim `TRGQueue` after its size exceeds twice the targeted cache size without affecting the quality of placement. Algorithm 3 sketches the steps involved in building the `TRGSelect` graph.

4.5 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, which 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 (see algorithm 4).

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. If an object faces high resistance (thus signifying a high probability of conflict) with the objects already placed in the native cache block, it might be (fully or partially) placed in the foreign cache blocks. 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 targeting 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

```

1:  $O_{TRG} \leftarrow \Phi$ 
2:  $E_{TRG} \leftarrow \Phi$ 
3: add the stack object to  $O_{TRG}$ 
4: add the constant and global objects to  $O_{TRG}$  by scanning symbol-table
5: add the heap objects to  $O_{TRG}$  by scanning the memory-allocation calls
6: repeat
7:   scan the next memory reference  $m_r$ 
8:   find object  $o$  such that  $m_r$  accesses  $o$  and  $o \in O_{TRG}$ 
9:   if  $o$  is not the tail of TRGQueue then
10:     enqueue  $o$  to TRGQueue
11:     object  $p \leftarrow$  second-last object in TRGQueue
12:     while  $p \neq \text{NULL}$  or  $p \neq o$  do
13:       if  $\text{edge}(o, p) \in E_{TRG}$  then
14:          $\text{edge-weight}[\text{edge}(o, p)] \leftarrow \text{edge-weight}[\text{edge}(o, p)] + 1$ 
15:       else
16:          $E_{TRG} \leftarrow E_{TRG} \cup \text{edge}(o, p)$ 
17:          $\text{edge-weight}[\text{edge}(o, p)] \leftarrow 1$ 
18:       end if
19:        $p \leftarrow$  predecessor of  $p$  in TRGQueue
20:     end while
21:     if size of all objects in TRGQueue exceeds threshold  $S_{TRGQ}$  then
22:       prune TRGQueue
23:     end if
24:     if size of  $E_{TRG}$  exceeds threshold  $S_{ETRG}$  then
25:       prune  $E_{TRG}$  {see Algorithm 2}
26:     end if
27:   end if
28: until there are no more memory references to scan
29:
30: prune  $O_{TRG}$  {see Algorithm 1}

```

Algorithm 3: Building the TRGSelect Graph

```

1:  $\text{cost} \leftarrow 0$ 
2: for all cache block  $C$  that is going to be occupied by this object do
3:   let  $p$  be the chunk of this object to be placed in  $C$ 
4:   for all object-chunk pair  $q$  already placed in  $C$  do
5:      $\text{cost} \leftarrow \text{cost} + \text{edge weight between } p \text{ and } q \text{ in TRGSelect graph}$ 
6:   end for
7: end for
8: return  $\text{cost}$ 

```

Algorithm 4: Finding Cost of a Placement

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 `TRGSelect` graph 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. Thus, two objects with a heavy edge between them may still overlap in the cache, if only some chunks of those objects have interleaving access pattern.

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. Placement of important global objects usually creates large *holes* (a contiguous section in the address space where no object has been placed) in the data segment. When placing an unimportant object, we scan the data segment and place the object in the first available free region big enough to accommodate it.

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`

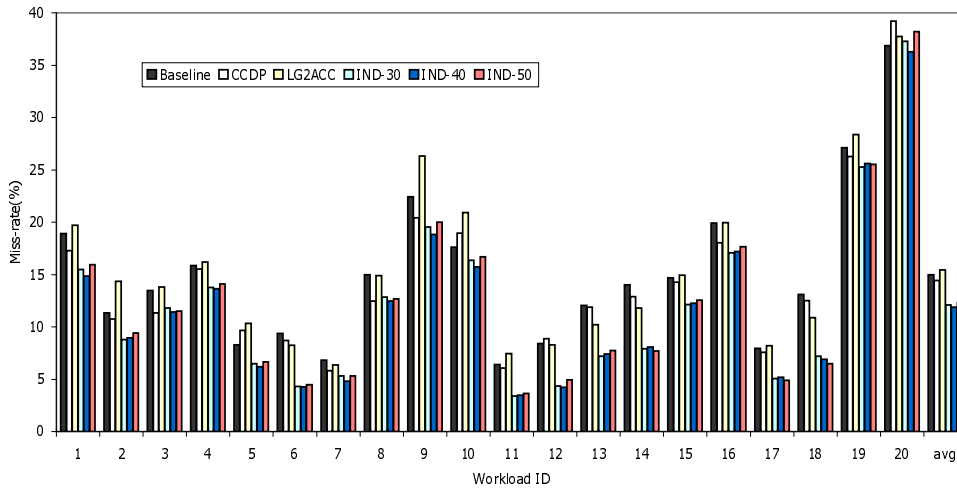


Fig. 3: Data Cache miss rate after Independent Placement (IND)

the location of the first block of the desired mapping. The object gets placed there, even if the size differs from the profiled run.

Our customized memory allocation/deallocation routines are closely based on the FastFit algorithm [28]. Every memory request of less than 4 KB in size is rounded to the next power of two. For every power of two from 16 to 4096, there is a corresponding linked list of memory blocks. When `malloc` receives an allocation request, it tries to satisfy it from the corresponding linked list. If no free block is available in the corresponding linked list, or the request is for more than 4 KB of memory, then memory is allocated from a *wilderness chunk* using the traditional FirstFit algorithm [29].

During execution, our customized `malloc` first computes the `HeapTag` for the requested heap object. If the `HeapTag` matches any of the recorded `HeapTags` for which a customized allocation should be performed, `malloc` returns a suitably aligned address. When the newly created heap object is brought in the cache, it occupies the blocks specified by the placement algorithm.

4.6 Independent Placement Results

The effects of data placement by IND on miss rate and weighted speedup are shown in Figure 3 and Figure 4, 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* [19]. 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,

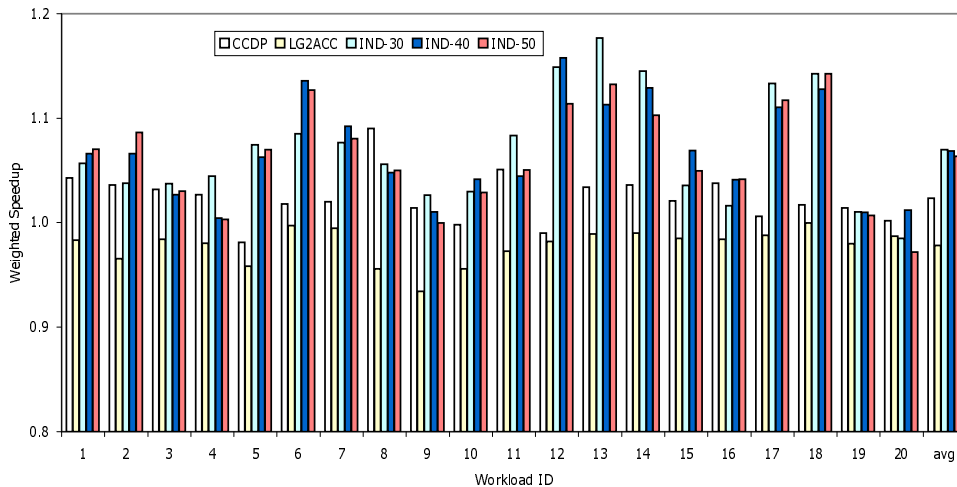


Fig. 4: Weighted Speedup after Independent Placement (IND)

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 3 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.

IND reduced overall miss rate by 19% on average, 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 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 or application-specific environments, programs 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.

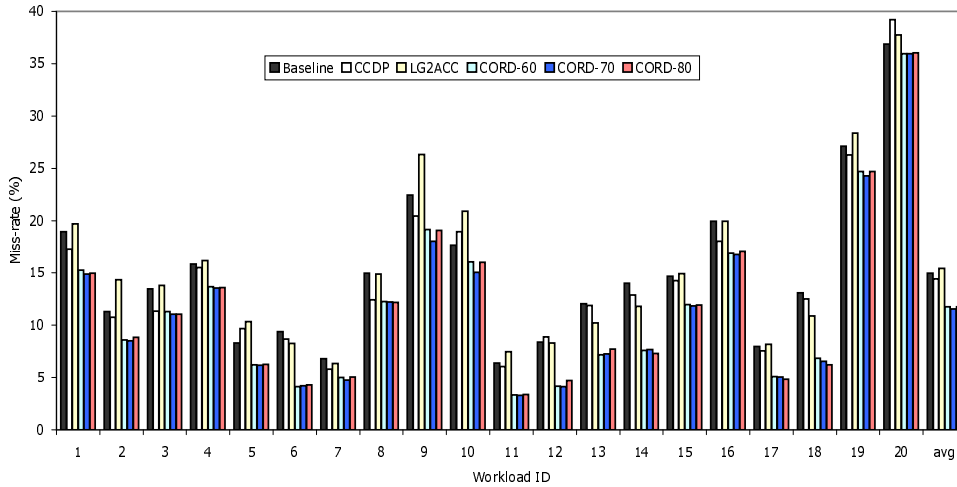


Fig. 5: Data Cache miss rate after Coordinated Placement (CORD)

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. The steps required to merge two `TRGSelect` graphs into a single, unified graph is outlined in Algorithm 5.

In our experiments we have found it sufficient to only put temporal edges between *important* objects (i.e., objects not marked as unimportant) 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 5 and 6. 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 being able to

```

1:  $O_{TRG,0} \leftarrow$  set of vertices in TRGSelect graph of application 0
2:  $O_{TRG,1} \leftarrow$  set of vertices in TRGSelect graph of application 1
3:  $E_{TRG,0} \leftarrow$  set of edges in TRGSelect graph of application 0
4:  $E_{TRG,1} \leftarrow$  set of edges in TRGSelect graph of application 0
5:
6:  $\{O_{TRG}$  and  $E_{TRG}$  are respectively the vertex and edge sets of the merged
   TRGSelect graph $\}$ 
7:  $O_{TRG} \leftarrow O_{TRG,0} \cup O_{TRG,1}$ 
8:  $E_{TRG} \leftarrow E_{TRG,0} \cup E_{TRG,1}$ 
9:
10: for all object  $o \in O_{TRG,0}$  do
11:   for all object  $p \in O_{TRG,1}$  do
12:      $LT_o \leftarrow$  life-time of  $o$ 
13:      $R_o \leftarrow$  number of memory references to  $o$ 
14:      $LT_p \leftarrow$  life-time of  $p$ 
15:      $R_p \leftarrow$  number of memory references to  $p$ 
16:      $LT_{o,p} \leftarrow$  overlapping life-time of  $o$  and  $p$  {computed from  $LT_o$ ,  $LT_p$  and
       relative rate of executions of two applications}
17:     if  $LT_{o,p} > 0$  then
18:        $R_o^n \leftarrow R_o \times \frac{LT_{o,p}}{LT_o}$ 
19:        $R_p^n \leftarrow R_p \times \frac{LT_{o,p}}{LT_p}$ 
20:        $E_{TRG} \leftarrow E_{TRG} \cup \text{edge}(o, p)$ 
21:       edge-weight[ $\text{edge}(o, p)$ ]  $\leftarrow 2 \times \min(R_o^n, R_p^n) \times s$  { $s$  being the scaling factor}
22:     end if
23:   end for
24: end for

```

Algorithm 5: Merging TRGSelect graphs

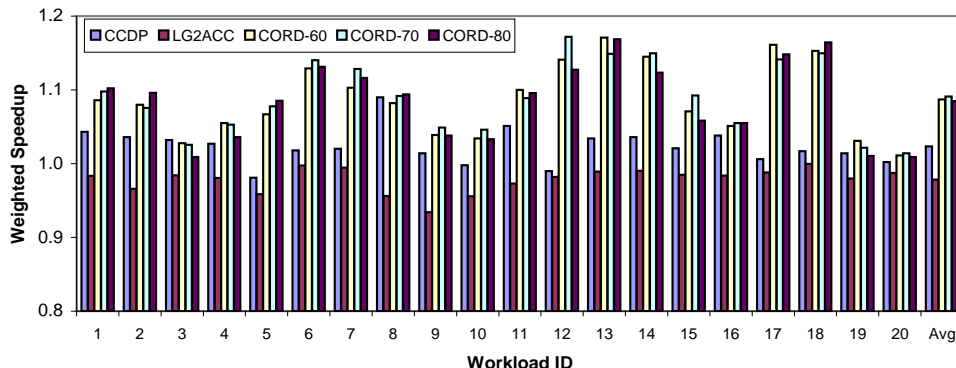


Fig. 6: Weighted Speedup after Coordinated Placement (CORD)

exploit more specific information about conflicting objects leads to better placement decisions. On the average CORD reduced miss rate by 26% and achieved 8.8% 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%.

We see in these figures, however, that overall the results are fairly insensitive to the scaling factor, which is closely tied to our estimates of relative execution speed of the two programs and the lifetimes of the objects. Thus, inexact estimates of any of these factors (expected rate of interleaving, relative execution rate, object lifetimes) should not have significant impact on the effectiveness of the technique.

6 Exploring Other Processor and Cache Configurations

Up to this point, we have demonstrated the effectiveness of our placement techniques for a single hardware configuration. We did extensive sensitivity analysis to understand how these techniques work as aspects of the architecture – such as cache sizes and organizations, cache latencies, and number of execution contexts in the processors – are modified. In this section we present and interpret the results of some of those experiments.

6.1 Effects of cache size and associativity

Cache associativity is the most interesting alternative, in large part because proposed CCDP algorithms do not accommodate associative caches. The naïve approach would model a set-associative cache as a direct-mapped cache with the

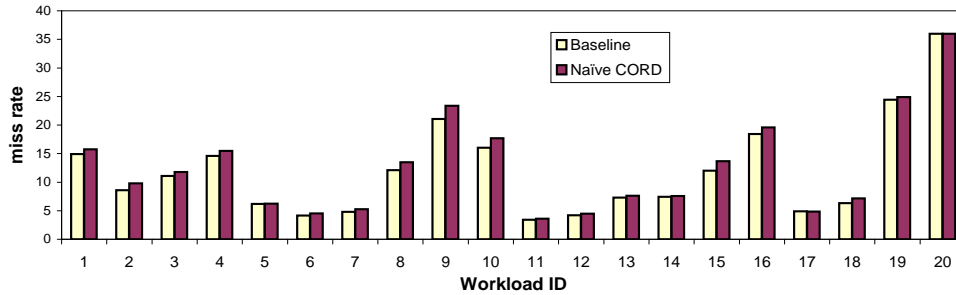


Fig. 7: Naïve Coordinated Data Placement with a Set-Associative Cache (32 KB, 2-way)

same number of sets. This has the benefit of retaining the correct model of line mapping – that is, two addresses that conflict in a 32 KB 2-way cache will also map to the same set in a 16 KB direct-mapped cache. However, this simplistic approach does not take into consideration false positives while enumerating conflicts, (i.e. this technique does not take into consideration the fact that k objects can share a single set in a k -way associative cache without causing any conflicts) and generally leads to sub-optimal object placement. The reason for the suboptimal placement is that the cost function is incorrect, and begins penalizing placements before they actually cause any misses. This can be particularly harmful for our independent placement (IND) algorithm, because a set-associative cache actually increases our ability to create unbalanced mappings – but only if the cost model allows us to.

But even in the case of coordinated placement, the incorrect cost function results in poor placement. This is shown in Figure 7, where we have modeled a 2-way 32 KB cache as a direct-mapped 16 KB cache, and the quality of the coordinated placement is actually consistently worse than the baseline placement.

For associative caches, any mapping function that used our `TRGSelect` graph would be an approximation, because we only capture 2-way conflicts. On the other hand, profiling and creating a hypergraph to capture more complex conflicts would be computationally prohibitive. However, we found the following heuristic to work well when using our existing `TRGSelect` graph. We have adjusted our default placement algorithm such that for a k -way set-associative cache, an object incurs placement cost only if it 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.

Average miss-rate reductions and weighted speedups for the variety of benchmark pairs is given in Figures 8 and 9 for different cache configurations. The split cache results (LG2ACC) are only shown for direct-mapped caches, because that technique is not applicable for set-associative caches. The results for set-associative caches, in particular, are indicative of the low incidence of conflict

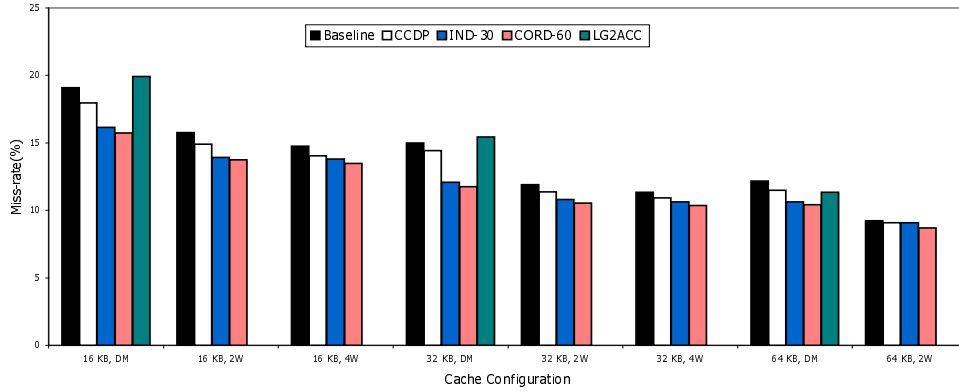


Fig. 8: Miss Rates for Different Cache Configurations. The set-associative results assume the new multithreaded set-associative cache placement algorithm.

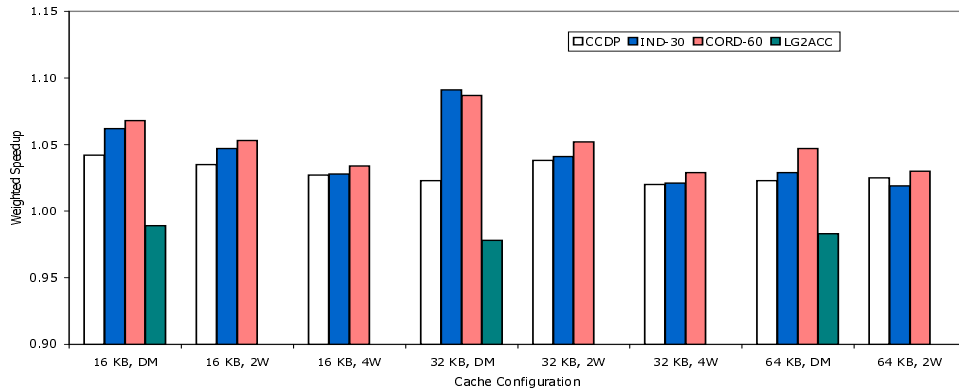


Fig. 9: Weighted Speedup for Different Cache Configurations

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%.

Although in some cases the performance gains are not high, they all still follow the trends we have seen so far. This is encouraging, since the low performance results are primarily the result of cache performance being good overall for those workloads. But because the trends remain the same, and we effectively reduce or nearly eliminate conflict misses in all cases, we have confidence that workloads that exhibit higher miss rates will be able to make good use of these techniques.

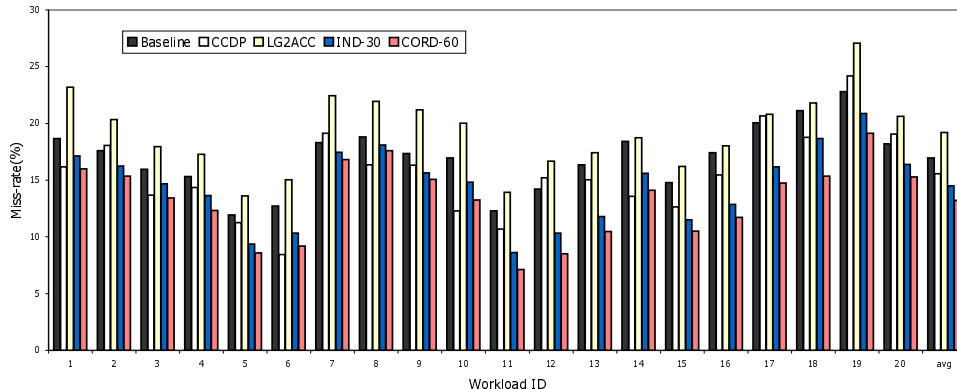


Fig. 10: Miss rate for a 4-context SMT processor

6.2 Increasing the number of execution contexts

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. Figure 10 and 11 shows the result of our placement techniques being applied to a four-threaded SMT processor.

For a 4-thread processor, IND-30 and CORD-60 reduced miss rates by 14% and 22% on the average; however, the actual weighted speedups were smaller (2.0% and 3.1% respectively), due to the SMT processors’ ability to tolerate cache misses in a latency-limited configuration like the one we simulate. However, the more threads running on a core, the more likely we are to saturate memory bandwidth (both instruction execution rate and misses per instruction go up significantly with more threads) – and in those scenarios the substantial reduction in miss rate would likely be translated directly into performance, as was demonstrated in the two-thread case. Moreover, there are other important advantages of reducing L1 miss rate, such as lowering net power dissipation.

6.3 Effects of cache miss penalty

Up to this point we have assumed the L1 miss penalty (to the L2 cache) to be 15 cycles, which is a reasonable figure for current microprocessors. However, in future multi-core processors, pressure on L2 bandwidth and the overhead of cache-coherence protocols will result in higher L1 miss penalty. In figure 12, we plot the weighted speedup of co-ordinated and independent placement techniques for a range of L1 miss penalties. Not surprisingly, weighted speedup resulting from our placement algorithms increase monotonically with miss penalty. Thus, as we increasingly pack more contexts (cores and thread contexts) onto the die, while

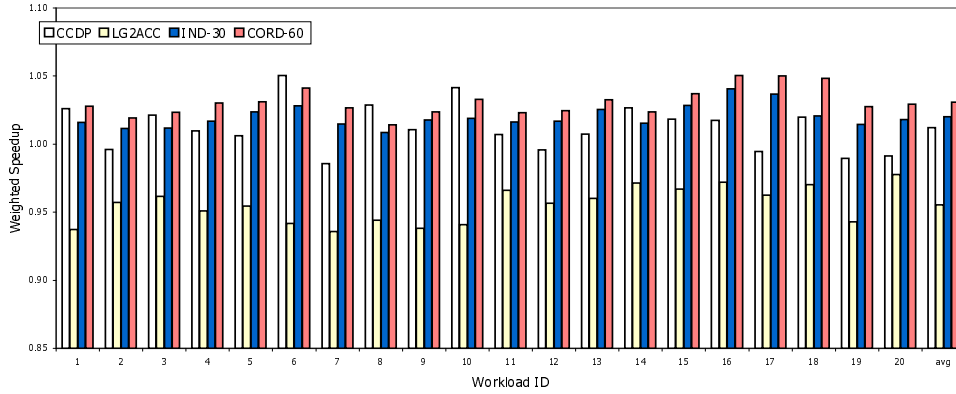


Fig. 11: Weighted Speedup for a 4-context SMT processor

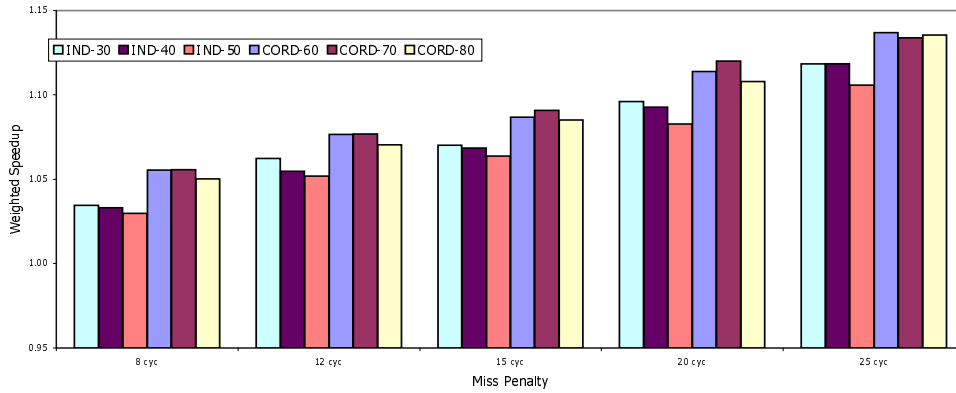


Fig. 12: Variation of Weighted Speedup with Miss Penalty

communication latencies across the die continue to increase, the importance of these techniques will only increase.

7 Conclusion

As we seek higher performance embedded and other 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 knowledge 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 various cache sizes, cache latencies, numbers of threads, and even set-associative caches.

Acknowledgments

This research was supported in part by NSF Grant CCF-0541434 and Semiconductor Research Corporation Grant 2005-HJ-1313.

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. (Dec 2004)
7. Dolbeau, R., Sez nec, A.: Cash: Revisiting hardware sharing in single-chip parallel processor. In: IRISA Report 1491. (Nov 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., Gonzalez, A.: Randomized cache placement for eliminating conflicts. *IEEE Transactions on Computer* 48(2) (1999)
10. Sez nec, A., Bodin, F.: Skewed-associative caches. In: International Conference on Parallel Architectures and Languages. (1993) 305–316
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. (1998) 38–49

13. Mueller, F.: Compiler support for software-based cache partitioning. In: Workshop on Languages, Compilers and Tools for Real-Time Systems. (1995) 125–133
14. Juan, T., Royo, D.: Dynamic cache splitting. In: XV International Conference of the Chilean Computational Society. (1995)
15. Bershad, B.N., Lee, D., Romer, T.H., Chen, J.B.: Avoiding conflict misses dynamically in large direct-mapped caches. In: Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, CA, USA (5–7 October 1994) 158–170
16. Sherwood, T., Calder, B., Emer, J.S.: Reducing cache misses using hardware and software page placement. In: International Conference on Supercomputing. (1999) 155–164
17. Nemirovsky, M., Yamamoto, W.: Quantitative study on data caches on a multi-streamed architecture. In: Workshop on Multithreaded Execution, Architecture and Compilation. (1998)
18. Hily, S., Sez nec, A.: Standard memory hierarchy does not fit simultaneous multithreading. In: Proceedings of the Workshop on Multithreaded Execution Architecture and Compilation (with HPCA-4). (1998)
19. Jos, M.G.: Data caches for multithreaded processors. In: Workshop on Multithreaded Execution, Architecture and Compilation. (2000)
20. May, D., Irwin, J., Muller, H.L., Page, D.: Effective caching for multithreaded processors. In: Communicating Process Architectures, IOS Press (2000) 145–154
21. Nikolopoulos, D.S.: Code and data transformations for improving shared cache performance on SMT processors. In: International Symposium on High Performance Computing. (2003) 54–69
22. 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)
23. Kumar, R., Tullsen, D.M.: Compiling for instruction cache performance on a multithreaded architecture. In: 35th Annual International Symposium on Microarchitecture. (2002)
24. Sarkar, S., Tullsen, D.M.: Compiler techniques for reducing data cache miss rate on a multithreaded architecture. In: Proceedings of the International Conference on High Performance Embedded Architectures and Compilers. (2008)
25. Tullsen, D.M.: Simulation and modeling of a simultaneous multithreading processor. In: 22nd Annual Computer Measurement Group Conference. (Dec 1996)
26. Tullsen, D.M., Brown, J.: Handling long-latency loads in a simultaneous multithreaded processor. In: 34th International Symposium on Microarchitecture. (Dec 2001)
27. Srivastava, A., Eustace, A.: Atom: a system for building customized program analysis tools. In: SIGPLAN Notices. Volume 39. (2004) 528–539
28. Grunwald, D., Zorn, B.G., Henderson, R.: Improving the cache locality of memory allocation. In: SIGPLAN Conference on Programming Language Design and Implementation. (1993)
29. Robson, J.M.: Worst case fragmentation of first fit and best fit storage allocation strategies. In: The Computer Journal. Volume 20(3). (1977)