Hardware Identification of Cache Conflict Misses

Jamison D. Collins Dean M. Tullsen

Department of Computer Science and Engineering
University of California, San Diego
La Jolla, CA 92093-0114

Abstract

This paper describes the Miss Classification Table, a simple mechanism that enables the processor or memory controller to identify each cache miss as either a conflict miss or a capacity (non-conflict) miss. The miss classification table works by storing part of the tag of the most recently evicted line of a cache set. If the next miss to that cache set has a matching tag, it is identified as a conflict miss. This technique correctly identifies 87% of misses in the worst case.

Several applications of this information are demonstrated, including improvements to victim caching, next-line prefetching, cache exclusion, and a pseudo-associative cache. This paper also presents the Adaptive Miss Buffer (AMB), which combines several of these techniques, targeting each miss with the most appropriate optimization, all within a single small miss buffer. The AMB's combination of techniques achieves 16% better performance than any single technique alone.

1. Introduction

A number of cache-based architectural optimizations are aimed at particular types of cache misses. Caches that mimic associativity (e.g., victim caches and pseudo-associative caches) all target conflict misses. Prefetching mechanisms tend to be more effective with capacity misses. Each of these mechanisms suffer because they end up being applied to all misses, due to our inability to distinguish conflict misses from capacity misses on the fly.

This paper presents a technique to classify misses as either conflict misses or capacity misses (we'll group compulsory and capacity misses together and call them capacity misses for simplicity). This technique can be used to filter misses seen by hardware structures that handle only one type of miss well. Even better, it can be used to apply different mechanisms to different types of misses, as appropriate. We show that this technique can correctly identify over 85% of misses as either conflict or capacity; it is accurate for both direct-mapped and associative caches of different sizes. One possible application of this would restrict the misses that can write into a victim buffer. This potentially protects two critical resources, the entries themselves, hopefully leaving high-probability entries in the buffer longer, and the buffer access ports, increasing its availability.

Cache conflict misses are identified through the use of the Miss Classification Table (MCT). The MCT has one entry corresponding to each set of the cache. For each set, the MCT stores the tag of the cache line most recently evicted from the cache. If the next miss to that set has the same tag, it is labeled a conflict miss. This implies that the line may have been a hit with a slightly more associative cache.

Armed with this information, we can choose to treat differently those lines evicted by a conflict miss, or those lines that come into the cache on a conflict miss. We do not need to store the entire tag in the MCT; however, the more evicted-line tag bits we save, the fewer false hits we receive and the more accurate our classification. We show that we still get nearly the full accuracy by only keeping 8 bits per entry.

We apply this technique to the following cache architectures: victim caches, cache prefetching, cache exclusion, and pseudo-associative caches. For victim caches, we achieve a modest speedup using miss classification filtering. We significantly increase the prefetch accuracy of a simple next-line prefetcher, allowing a very simple scheme to approach the accuracy (if not coverage) of more complex mechanisms. We show a simple approach to cache exclusion which decreases the overall cache miss rate over more expensive schemes. The pseudo-associative cache uses its knowledge of miss type to change the line replacement algorithm. We also describe a mechanism for combining these techniques, targeting each miss with the optimization most likely to produce a performance benefit. This approach, called the Adaptive Miss Buffer, demonstrates the real potential of miss classification, targeting a different op-

timization for each type of cache miss. It achieves as much as a 16% speedup over any single technique, while using the same buffer structure.

This paper is organized as follows. Section 2 discusses related work. Section 3 describes the miss classification table and other hardware support. Section 4 describes our measurement and evaluation methodology. Section 5 describes the performance of four applications of miss classification filtering on previous architectures and a new architecture that combines several techniques. Section 6 concludes.

2. Related Work

This research applies miss classification to several previously proposed and implemented cache structures. Victim buffers [10] store recently evicted cache lines in a small buffer, often transforming cache conflict misses into victim buffer hits. Hardware cache prefetching [14, 10, 3] attempts to predict future memory access patterns from prior access history, moving those cache lines expected to be accessed in the future into the lower levels of the cache hierarchy. Cache exclusion identifies either instructions [20] or addressable regions of memory [8] that have poor data locality (for the particular cache) and do not allow those accesses to evict other lines from the cache. A pseudo-associative cache [1] uses an alternate entry in the cache as a backup location for a line evicted from its primary location. The hit time to the primary location is the same as a direct-mapped cache, but the hit rate (including hits to the secondary location) is similar to an associative cache.

Stone [17] describes a *shadow directory*, which he attributes to J. Pomerene, that keeps some number of evicted line addresses per cache set. It is similar to our miss classification table, but there are no performance results available to evaluate the technique. He suggests it be used to favor *shadow misses* over other misses (he calls them transient misses) in the cache replacement algorithm of a set-associative cache. We add to this the conflict bit, which allows the retention of the miss classification while the line remains in the cache. We examine many more applications of the technique.

Architectures have been proposed that dynamically attempt to classify other aspects of references, including temporal or spatial locality [5, 11] or migratory behavior [4, 16].

3. Classifying Misses

Miss classification identifies the following conflict miss scenario. Cache line B is accessed, resulting in a cache miss, and evicts line A from the cache. The next miss to the same cache set is an access to line A. The second miss is a conflict miss which can be identified simply by saving the tag for A when it was evicted. On the subsequent miss to this cache set, the tag of the most recently evicted line from that set is compared with the tag of the newly accessed line. If they are identical, we identify this as a conflict miss. This is slightly different than the classic definition of a conflict miss [6], but is much easier to identify on the fly. This definition actually filters a more interesting subset of conflict misses. It identifies conflict *near-misses* — direct-mapped misses that would have been caught by a 2-way cache, or 4-way misses that would have been caught be a 5-way cache. These are the misses that a victim buffer, for example, handles most effectively. Direct-mapped conflict misses that would have required 8-way associativity to catch are unlikely to be aided significantly by a victim buffer.

The extra tags would be best stored in a structure separate from the cache, because it is accessed much less frequently and without the access time pressure. We call this structure the Miss Classification Table (MCT). The extra tag comparison is only done on cache misses. The MCT tag comparison will not affect the critical-path access time of the cache, nor will it even affect the access time of post-cache structures (e.g., victim or prefetch buffer), because we only use the conflict information after those structures are queried.

The MCT has one entry per cache set, and thus is direct-mapped regardless of the associativity of the cache (we could store multiple evicted tags per set to identify higher-order conflict misses, but we do not consider that optimization). In addition, we need not store the entire tag to identify conflict misses with high accuracy.

The accuracy with which the MCT identifies conflict and capacity misses (assuming complete tags) is shown in Figure 1. It correctly identify 88% of conflict misses and 86% of capacity misses on a 16KB direct-mapped data cache and 91% and 92%, respectively, on a 64KB DM cache. In fact, it is an overstatement to label the rest of the misses as inaccurate, because in some cases our definition of conflict miss may be more useful than the classic definition.

Figure 2 shows the impact of saving only the lower bits of the evicted tag. This shows that very little accuracy is lost with only 8 bits stored. This, however, can be highly sensitive to the working set size of the program, so the SPEC benchmarks may not be representative. However, 10-12 bits should be sufficient for most applications. This graph is for a 16 KB DM cache, showing the average for all benchmarks. The results for the other cache configurations converge in very similar ways. With fewer bits stored, more misses are classified as conflict misses, which is why conflict accuracy starts out artificially high and capacity accuracy starts low. This graph shows that even a *single* bit per cache set could be effective for an architecture that targets

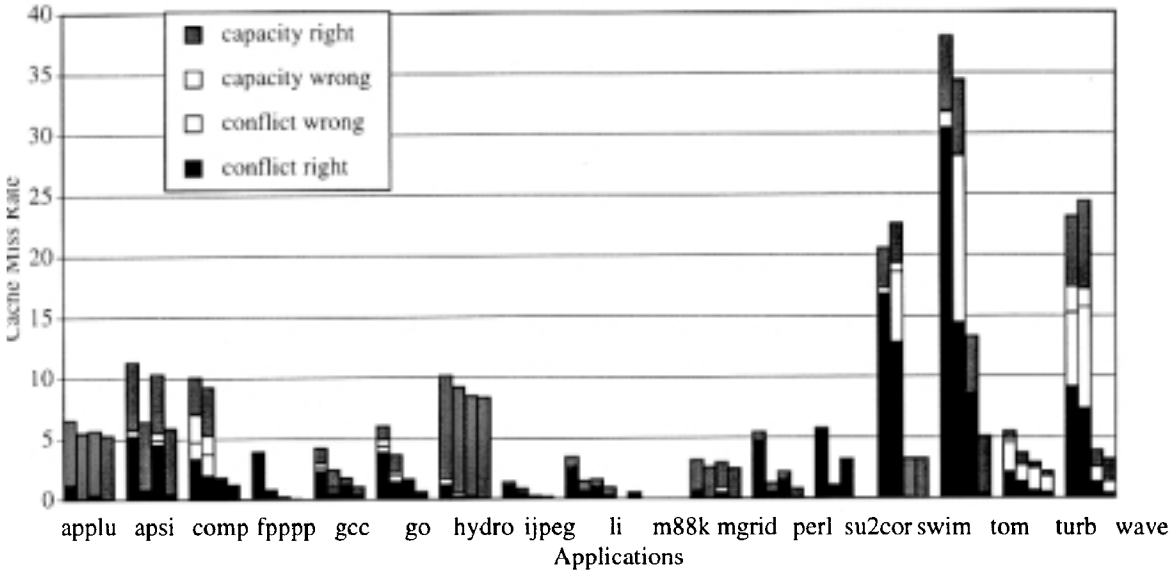

Figure 1. The accuracy of miss classification. Results are shown for four cache configurations. The four bars, left to right, are for a 16KB DM cache, 16KB 2-way cache, 64KB DM cache, and 64KB 2-way cache.

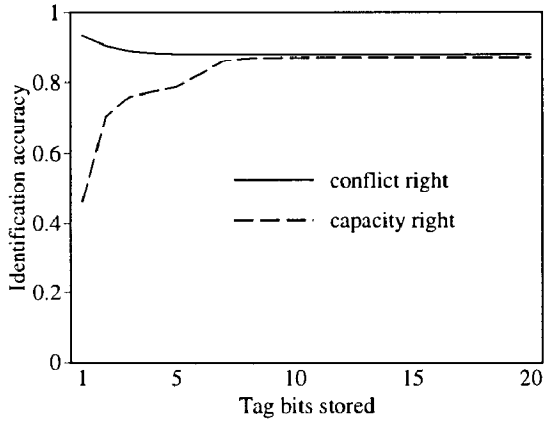

Figure 2. The accuracy of miss classification when less than the total tag of the evicted line is saved. Results are shown for a 16KB DM cache.

conflict misses, allowing nearly half of capacity misses to be excluded while misidentifying very few conflict misses.

In some cases, we are more interested in the nature of the evicted line than the classification of the new miss that caused the eviction. If we store one extra bit (call it a *conflict bit*) in the MCT per cache line, we can remember which lines came in on conflict misses.

In a direct-mapped cache, this enables the following possible filters: *in-conflict* — the evicted line originally came in as a conflict miss, *out-conflict* — the evicted line is being forced out by a conflict miss, *and-conflict* — both misses were conflict misses, and *or-conflict* — either the new or evicted line were conflict misses. A set associative cache would have even more possible scenarios. We typically tried all four filters for each architecture. When results were similar, we present the *out-conflict* result, which does not require the extra bits.

This then provides two mechanisms for changing the bias of our classification, depending on how we plan to use the information. Fewer saved tag bits and an algorithm that uses the *or-conflict* filter will err on the side of conflict misses. An algorithm that stores more bits and uses the *and-conflict* filter will err on the side of capacity misses.

The miss classification table is quite small. If we store 10 bits per entry, the MCT contains 1.25 KB of storage for a direct-mapped 64 KB cache with 64 byte lines (half that for a 2-way set-associative cache). If we also store a conflict bit per cache line, the total storage overhead goes up to only 1.4 KB.

4. Methodology

The utility of conflict miss classification is demonstrated by applying it to a variety of cache architectures and measuring the performance on a detailed emulation-driven processor simulator, SMTSIM [19]. SMTSIM runs Compaq Alpha binaries and models an out-of-order processor pipeline, including execution and memory access along

wrong paths following branch mispredictions. The processor model used in this study has a 7-stage pipeline, two 32-entry instruction queues, and 8-instruction fetch and issue, including four load-store functional units.

The processor simulates a three-level memory hierarchy, including first-level instruction and data caches, an off-chip second level cache (20 cycles from the processor in the absence of contention) and main memory (100 cycles from the cpu without contention). Contention for cache banks and buses between caches is also simulated. All simulations (except Figure 1) assume a 16 KB direct-mapped L1 data cache (multi-ported via 8-way banking) and a 1 MB 2-way L2 unified cache. The L1 data cache configuration was chosen to create an interesting mix of conflict and capacity misses for the simulated workload (based on Figure 1). For larger workloads (e.g., databases, OLTP, graphics), we expect much larger and associative caches to still experience both types of misses in interesting quantities. All caches have 64-bye lines. The caches are non-blocking with up to 16 misses in-flight at once. When the miss limit is exceeded, further misses stall the pipeline, but prefetches are discarded.

Our application suite comes from the SPEC95 benchmarks. We run the reference data sets, starting measured simulation 1 billion instructions into execution, then measuring the next 300 million instructions (starting with a cold cache). We only carry a subset of the SPEC95 benchmarks used in the previous section forward, eliminating those that don't have at least a somewhat interesting mix of conflict and capacity behavior. However, we show the initial results to demonstrate that even the "uninteresting" behavior is classified accurately. This still includes a number of the irregular C applications for which the overall impact of the memory subsystem on performance is not high. This is in contrast to much of the previous work on, for example, prefetching and cache exclusion, which focus on regular numeric applications. This limits the magnitude of our gains, but also tests the applicability of our techniques on much "messier" applications.

We will apply the various architectural techniques exclusively to the data cache in the following sections; however, they should, in general, also apply to the instruction cache.

We will model a variety of flavors of a cache assist buffer, which will serve at different times as a victim buffer, prefetch buffer, cache bypass buffer, or the adaptive miss buffer. In each case the structure is very similar. In most cases it will have eight fully-associative entries and have two read and two write ports. It can produce a word to the CPU in one cycle. A full cache line read or write requires a port for two cycles. A line swap with the data cache requires two ports for two cycles. The buffer is only accessed after the data cache misses, but can provide data with a single additional cycle of latency (assuming no contention). The size of this structure was chosen to ensure single-cycle access.

All results in the next section store the entire tag in the MCT.

5. Applications of Conflict Miss Filtering

This section demonstrates the miss classification table by examining it in the context of four cache memory architectures. Also, a new technique that combines these techniques in a single mechanism is presented, called the adaptive miss buffer.

5.1. Victim buffer caches

The victim buffer [10] is a small buffer that holds data recently evicted from the cache. The victim cache is probed when the main cache misses, and when the data is found it can be returned much more quickly than a full cache miss. It targets conflict misses, and is most effective when just a few cache sets are heavily contended for.

Normally, a victim cache hit requires a swap of the two affected lines, the newly evicted line now becoming the first entry in the victim buffer (and thus the last to be evicted from the victim cache). The victim cache can be organized as a FIFO from which entries can be taken out of the middle. This provides LRU eviction because lines are consumed out of the victim cache as soon as they are accessed.

Although the victim cache serves conflict misses almost exclusively, the application of conflict-miss filtering is not necessarily straight-forward. We want to put data in the victim cache that *will* experience a conflict miss in the near future. Therefore, by using a filter to place cache lines in the victim cache, we are using the current miss classification as a *prediction* of future miss classification. If a cache line came into the cache originally as a conflict miss, or if it was forced out as a conflict miss, we expect that to be a reasonable predictor that it will be accessed next as a conflict miss. Both of those assumptions are true for most applications, but we did see exceptions.

The victim cache provides several policy options. When we get a miss, we can choose not to place the evicted line in the victim cache if we don't believe it will be useful. This should increase the likelihood useful data in the victim cache stays there. This is most useful when the victim cache is small.

When we get a victim cache hit, we can choose not to swap the line with the cache, but just have the victim cache provide the data to the CPU. Allowing victim cache hits without swaps implies (but does not actually require) that you now enforce some kind of LRU organization that accounts for cache hits. This violates the FIFO nature of the victim cache, but at the size we are simulating (eight entries), a traditional fully-associative organization is not

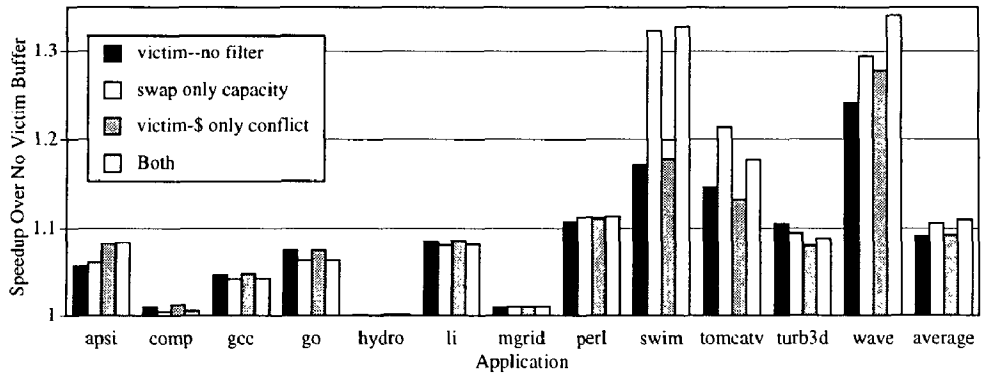

Figure 3. The performance of victim cache policies using the conflict classification.

complex. Eliminating swaps puts considerably less pressure on both the victim cache and the data cache, which are both occupied during a swap. However, this gain can be lost if the rate of victim cache accesses rises significantly. This can happen when oft-accessed data remains in the victim cache because of our decision not to swap.

The full range of possible permutations of policies (even the subset likely to be profitable) is quite large, and we will only show a subset of well-performing or interesting policies — this is true for each of the cache architectures we examine. Figure 3 shows the performance of the victim cache policies. The first bar is a traditional victim cache. The second does not swap lines on a victim cache hit for a conflict miss. This policy eliminated a great deal of heavy ping-ponging of cache lines between the main cache and the victim cache, recognizing that conflict misses were the primary culprit. The third bar bypasses the victim cache when a line is being evicted from the main cache as a capacity miss. The last bar combines the two techniques. Each of these policies use the *or-conflict* algorithm, the most liberal identification of conflict misses.

The speedup (about 3% on average, for the combined policy over a traditional victim cache) was gained primarily by placing less pressure on the victim and data caches by reducing swaps and victim cache fills. In most cases more selective use of the victim cache did not increase overall hit rates — the victim cache, even at eight entries, was not being overly hurt by contention for entries. Therefore, bypassing the victim cache with low-probability entries did not in general prevent useful lines from being evicted, but did serve to eliminate a large number of unnecessary victim cache fills, with very little loss in hit rate. The hit rate data and rate of victim cache swaps (on victim cache hit) and fills (on victim cache misses) is shown in Table 1. We see that the no-fill option cut victim cache fills by more than half. The no-swap option dramatically decreased the number of swaps. Although the numbers of swaps are relatively low to begin with, they are very expensive, occupying both the cache bank and the victim buffer.

Policy	D$ HR	V$ HR	Total	swaps	fills
no V cache	88.2	0	88.2	0	0
V cache	88.2	6.4	94.7	1.7	6.6
filter swaps	82.5	12.1	94.6	0.1	6.6
filter fills	88.1	6.2	94.3	1.7	2.6
filter both	80.8	13.6	94.4	0.1	2.6

Table 1. Hit rates and rate of swaps and fills (as a percentage of all accesses) for the various victim cache configurations.

5.2. Cache Prefetching

Hardware cache prefetching predicts future memory access patterns based on current or past access patterns, and attempts to move data likely to be accessed in the near future closer to the processor. While all misses can benefit from prefetching, we expect capacity misses to be more amenable to prediction via pattern analysis than conflict misses.

Hardware prefetchers range from very simple next-line prefetchers to more sophisticated stride [3] or even repeated-pattern based predictors [9]. We examined both a next-line prefetcher and a stride predictor (results not shown here) based on Chen and Baer's reference prediction table (RPT) [3]. However, for most of the benchmarks we use, particularly the irregular applications, the simple next-line prefetcher actually provides higher coverage of misses. But it does so at the expense of a very large number of wasted prefetches (prefetches that are lost from the buffer before they are, if ever, accessed by the program). Those wasted prefetches will be the target of our approach. The RPT scheme can potentially benefit from miss classification by removing the noise from the access stream created by the conflict misses, particularly when the predictor is able to follow a limited number of address streams. However, for this paper we will focus on the next-line prefetcher. The next-line prefetcher simply prefetches the next cache line on a cache miss (assuming the next line is not already in

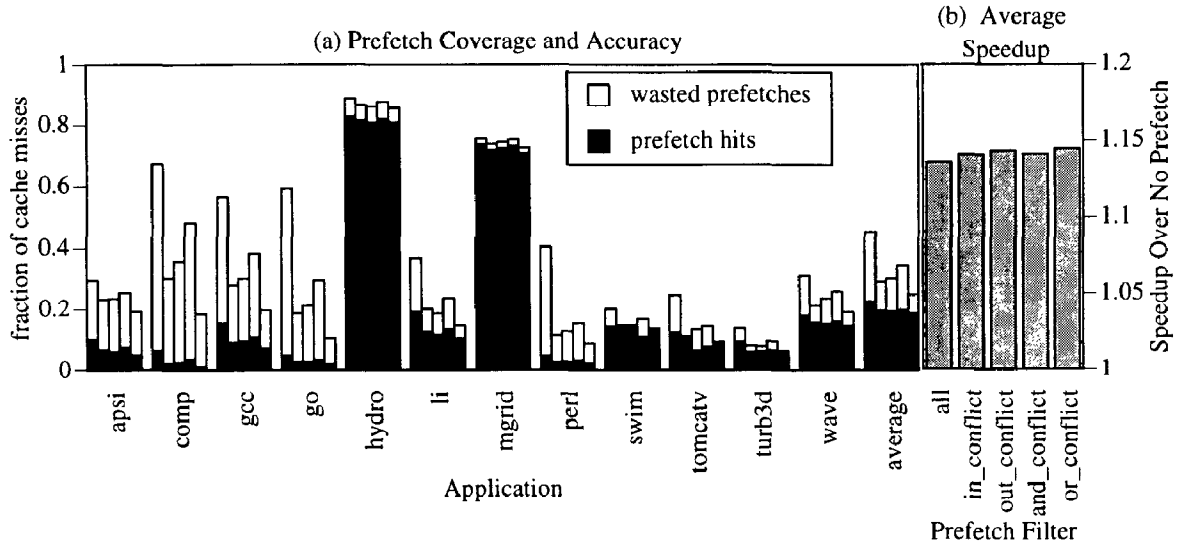

Figure 4. The performance of various next-line prefetch strategies. The first bar is a conventional next-line prefetcher. The others, left to right, are only prefetch capacity misses, where the filter used is in-conflict, out-conflict, or-conflict, and and-conflict, respectively. (b) shows the average speedup over no prefetching.

the cache) into a buffer which looks very much like our victim cache or a stream buffer [10]. On a hit in the prefetch buffer, the line is moved into the cache and the next line is prefetched.

A next-line prefetcher, even when augmented with our miss classifier, is much simpler than the RPT prefetcher, which must be read and updated on every memory access. With our prefetcher, the next-line calculator and the miss classification table are only accessed on misses.

Figure 4 shows the effect of applying capacity-miss filtering to a simple next-line prefetcher. The bars, left to right, are (1) a standard next-line prefetcher with no filtering, (2) prefetching only capacity misses, achieved by ignoring the *in-conflict* misses, (3) ignoring the *out-conflict* misses, (4) ignoring the *and-conflict* misses, and (5) ignoring the *or-conflict* misses. The or-conflict filter is the most discriminating, because it chooses not to prefetch if there is even a hint of a conflict miss. In all cases, filtered prefetching provided significantly higher prefetch accuracy (fewer wasted prefetches) by eliminating low-probability prefetches. In one case (*swim*), the filtered prefetching actually provided higher coverage by eliminating useless prefetches from the prefetch buffer, preventing useful prefetches from being overwritten.

In a memory system with sufficient memory bandwidth, coverage is ultimately a more important factor than accuracy; however, in some memory-bound scenarios accuracy will be important. The speedup results shown are for a system with a slower memory bus (between the L1 and L2 caches) than modeled in the rest of the paper. However, even under those conditions the performance advantage is not significant. For prefetching, the payoff is not in *not* prefetching conflict misses, but in finding something better than prefetching to do for the conflict misses, as seen in Section 5.5.

This section has shown that prefetch filtering using the miss classification table can significantly decrease the amount of useless prefetch traffic, increasing prefetch accuracy by about 25%.

5.3. Cache Exclusion

Tyson, et al. [20] and Johnson and Hwu [8] have demonstrated that not all data accesses are good candidates for caching. Higher overall hit rates can be achieved by not allocating space for certain cache lines, even on load misses. Tyson uses a table, indexed by program counter, to track hit/miss frequency, and excludes from the cache accesses predicted to miss with high likelihood. Johnson and Hwu record the frequency of access to 1KB regions of memory, and prevent a cache line from a low-access region from replacing one from a high-access region. We model the Johnson and Hwu memory access table (MAT) in this section, and compare it with exclusion algorithms based on the MCT.

Both the Tyson and Johnson schemes require tables that are updated on every access. A processor with 4 load/store units must be able to do 4 reads to the structure, 4 increments/decrements, and 4 writes each cycle. The MAT also

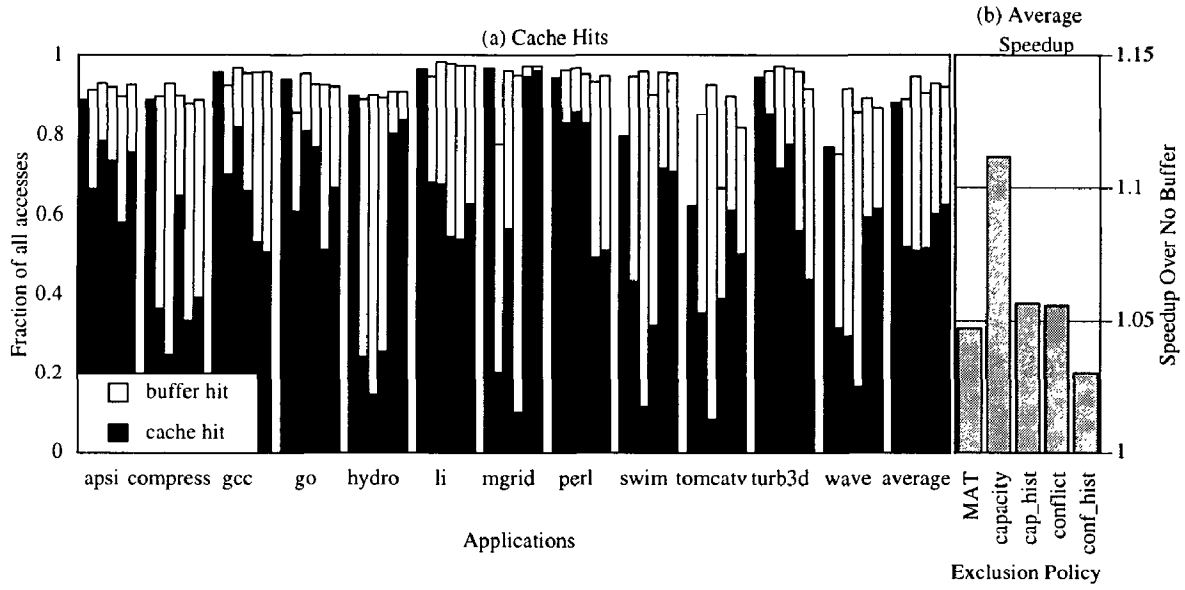

Figure 5. The performance of various cache-exclusion policies. The first bar in each group has no extra buffer, the second uses Johnson and Hwu's memory access table, the third through sixth use the *conflict*, the *conflict history*, *capacity*, and *capacity history* filter mechanisms to select lines to go into the bypass buffer. (b) shows the average speedup over no exclusion.

requires tag-match comparisons for each access, and on a miss must access two entries in the table. If we can achieve equal or better performance with our miss classification table, which is only used on cache misses, we can greatly simplify this structure. We simulate a 1K-entry direct-mapped MAT. It was shown that excluded accesses do typically have some short-term spatial locality [8]; therefore, we bring excluded lines into a small 16-entry bypass buffer, which again looks much like our victim/prefetch buffer, where they remain until bumped out by other lines. The Johnson algorithm was originally studied with a much larger buffer, and we found it to do poorly with an 8-entry buffer, which is why we use the slightly larger structure here.

The best policy was not obvious in this case. Capacity accesses are more likely to have the property of short but temporary bursts of activity; however, filtering for conflict misses ensures that there is a problem to be solved with this cache line. We found the former factor to be more important and that capacity misses were the best candidates for the bypass buffer.

Filtering for capacity misses required a slight modification to the MCT algorithm. The problem is that no line can be classified as a conflict access unless it has been in the cache once. But if we redirect all capacity misses to the bypass buffer, none ever get classified as conflict. Therefore whenever a line gets put into the bypass buffer, we put its tag into the MCT entry for the index in the cache where it would normally go. That way, if the line causes a miss later (after it is evicted from the bypass buffer), it has the opportunity to be classified as a conflict miss.

For this study, we examined more complex exclusion algorithms (examining possible compromises between the simplicity of the MCT and the complexity of the MAT), yet failed to beat the performance of the most simple filter. The simulated policies are *MAT* — the Johnson and Hwu scheme, *capacity* — put any miss identified as a capacity miss in the bypass buffer, *capacity history* — exclude misses from a region with a history of capacity misses (using a structure somewhat similar to the MAT), *conflict* — put any miss identified as a conflict miss in the bypass buffer, and *conflict history* — put accesses from a region with a history of conflict misses into the bypass buffer. In each case (except MAT), the *out-conflict* filter is used.

Figure 5 shows the performance of miss classification filtering for cache exclusion. Simply excluding capacity misses provided the best performance, both outperforming the MAT scheme and our simpler variants of the MAT scheme. This scheme provided both a higher overall hit rate and higher performance, although it does pay a slightly higher price in more buffer accesses.

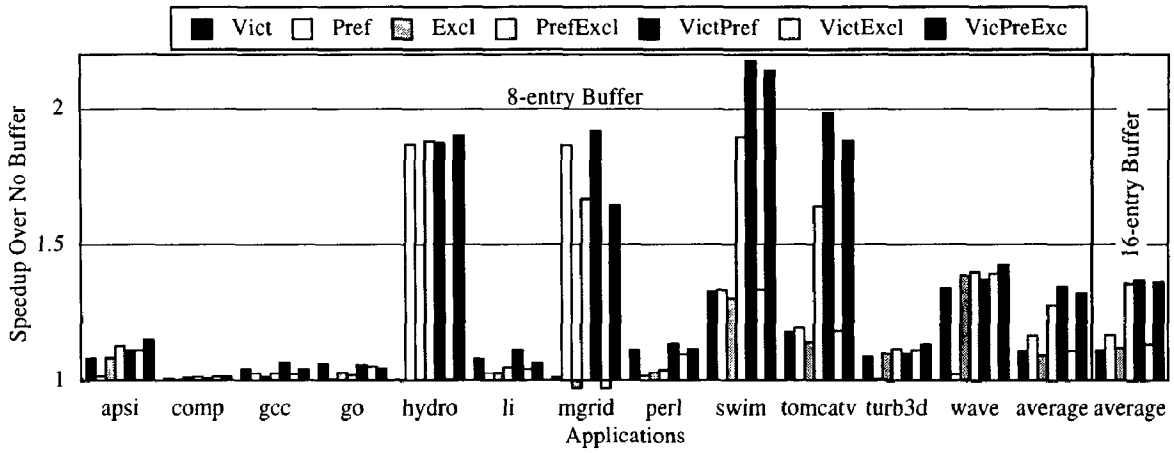

Figure 6. The performance of the adaptive miss buffer using various combinations of victim, prefetch, and exclusion policies.

5.4. Pseudo-Associative Cache

A pseudo-associative cache [1] uses an alternate entry in the cache as a backup location for a line evicted from its primary location. The secondary location has a longer hit time than the primary, and a hit to the secondary triggers a cache line swap between the two locations. Like the victim cache, this works most effectively with conflict misses, and we can improve its effectiveness by protecting conflict-miss cache lines in the cache. A wide variety of policies are possible, but here we only describe one particular algorithm which we found to be effective.

The traditional pseudo-associative cache can be modified to use the MCT to discard cache lines that are less likely to profit from the associativity, namely capacity misses. In this scheme, the MCT entry at a particular index holds the tag of the line most recently evicted from that index, even if the line was in its secondary position. A new line gets its conflict bit set only if it matches the tag in its primary location. When a line needs to be chosen for eviction, and exactly one of the two candidates has its conflict bit set, the other line is evicted and the first line's conflict bit is reset, regardless of primary/secondary location or the LRU bit. In this way we give a temporary advantage to lines that come in as conflict misses. If both lines have their conflict bits set, traditional LRU is used and the conflict bit of the kept line is not cleared.

This technique improved the average performance of the pseudo-associative cache by 1.5%, with individual gains as high as 7%. Our modified pseudo-associative cache ran only 0.9% slower than a true 2-way set-associative cache. In fact, three of the programs, tomcatv, turb3d and wave all outperformed the 2-way cache. Overall, the average miss rate was improved from 10.22% in the base pseudo-associative cache to 9.83%.

5.5. Adaptive Miss Buffer

So far in this section, we have achieved small gains through not applying optimizations where they were not appropriate, but the real power in miss classification is the opportunity to apply the best optimization to each type of miss individually. Each of the first three schemes use a very similar structure to hold data that is not appropriate to put in the main cache. It would be straightforward to combine these schemes in a single buffer which treats each miss in a manner most appropriate for that type of miss.

The advantage of maintaining a single buffer rather than multiple buffers is that the access time remains the same as any of the other cache architectures alone. Having multiple buffers would add another level of associativity to the access which would certainly affect access times.

Combining these policies requires extra bits to remember how a cache line entered the buffer, because we may do something different on a buffer hit depending on whether the line came in, for example, as a prefetch or a victim swap. In some cases lines will transition — for example, if we are combining prefetching and exclusion, a hit on a prefetched line may still leave the line in the buffer, but mark it as an exclusion line.

When combining policies, we stayed with the best filter found so far. For example, VictPref victim caches (but doesn't swap) conflict misses and prefetches capacity misses. PrefExcl does not do anything with conflict misses, because both do best with capacity misses; however, we found that a variant of PrefExcl which prefetched capacity misses and excluded conflict misses also performed well, but that result is not shown. For an 8-entry buffer, the best combination was VictPref (Figure 6), which more than doubled the overall gain of any single policy. With more room in the buffer (see the 16-entry result), the policy which does

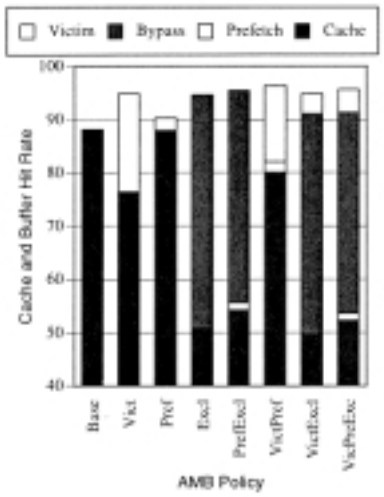

Figure 7. The average data cache and buffer hit rate components for the adaptive miss buffer policies.

everything (VicPreExc, which excludes and prefetches bypass misses, and victim-caches conflict misses) becomes more attractive.

The single-policy results shown use the variant which gave the best performance, which in each case did involve miss classification filtering. All multiple-policy results shown use the *out-conflict* filter.

The hit rate statistics (Figure 7) show that the AMB is indeed deriving its performance by optimizing the coverage of each type of miss. On average a factor of 1.4 improvement (30% reduction) in total miss rate is achieved over the best individual policy, and for the memory-critical applications, it was even higher. On tomcatv, a 1.7X improvement in miss rate was achieved (VictPref over Vict). Even that is deceiving, because while the prefetch hit rates show prefetching to be ineffective for the combined policy, the "wasted" prefetches (because they failed to stay long enough in the buffer) end up pre-filling the L2 cache quite effectively, so the average access time of the remaining misses was also much lower with VictPref than with Vict alone. So even on the most demanding of our application (tomcatv has a 38% miss rate with no buffer), we found that almost all of the misses are at least partially covered by the adaptive miss buffer, despite its small size.

5.6. Other Applications of Miss Classification

This paper has examined a few potential applications of miss classification, but has certainly not exhausted the possibilities. This section presents some other possible applications.

Highly associative caches Many real workloads will still experience conflict misses with 4-way or higher-associative caches (unfortunately, this is not in general true of the workloads used in this paper). In that case, the cache may benefit from using miss classification as part of the cache line replacement algorithm. For example, a bias against capacity misses will ensure that accesses that stride through memory (characterized by a capacity miss followed by a short burst of activity) will move out of the cache set quickly once they are no longer being used. This is the same application suggested by Stone [17] and Pomerene.

Runtime conflict avoidance The cache miss lookaside buffer [2, 13] counts cache misses by their page in memory. This allows the operating system to alter the virtual-to-physical page mapping of two pages that map to the same region of the cache and are both experiencing high miss rates. Miss classification would allow this technique to only count conflict misses. Reallocation could be avoided when the majority of misses are capacity misses (in which case reallocation typically would not help).

Multithreaded architectures Multithreaded [18, 7] processors, or other architectures that allow multiple threads to dynamically share a cache [15, 12], are particularly prone to high levels of conflict, even with associative caches. In addition, this problem cannot be solved with software techniques because the conflicts are produced by competition with other threads.

All of the techniques described in this paper would apply to an even greater extent with multithreaded caches. But multithreaded processors enable another dimension to the solution through control of job scheduling. Jobs which produce an inordinate number of conflict misses when scheduled together can be identified as bad candidates for co-scheduling in the future.

6. Conclusions

This paper describes the miss classification table which enables the processor to dynamically distinguish between conflict and capacity (non-conflict) cache misses. The miss classification table works by storing all or part of the tag of the most recently evicted line of a cache set. If the next miss to that cache set has a matching tag, it is identified as a conflict miss. This technique correctly identifies at least 87% of misses. In addition, a single bit per cache line (the conflict bit) enables the preservation of that miss classification information during the line's lifetime in the cache. The miss classification table is small and simple. It can require as little as 8-10 bits per cache set and need be accessed only on cache misses.

We demonstrate the utility of this information by applying it to victim cache design, cache prefetching, a cache exclusion mechanisms, and pseudo-associative caches. In each case, the architecture benefits from applying different policies to different types of misses. It does so in some cases by eliminating accesses unlikely to benefit from the particular architecture.

Three of these techniques can be combined into a single architecture, which we call the adaptive miss buffer. The adaptive miss buffer uses the victim/prefetch/exclusion buffer in a different way depending on the classification of each miss. This uses a single structure to optimize buffer performance for the elimination of both conflict and capacity misses. This greatly increases the effectiveness of a cache-assist buffer, providing twice the performance gain of any single optimization using the same size buffer.

Acknowledgments

We would like to thank the anonymous reviewers for their useful comments. This work was funded in part by NSF CAREER grant No. MIP-9701708, NSF grant No. CCR-980869, and equipment grants from Compaq Computer Corporation.

References

[1] A. Agarwal and S. D. Pudar. Column-associative caches: a technique for reducing the miss rate of direct-mapped caches. In *20th Annual International Symposium on Computer Architecture*, pages 179–190, San Diego, CA, May 1993. ACM.

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

[3] T.-F. Chen and J.-L. Baer. Effective hardware-based data prefetching for high-performance processors. *IEEE Transactions on Computers*, 44(5):609–623, May 1995.

[4] A. L. Cox and R. J. Fowler. Adaptive cache coherency for detecting migratory shared data. In *20th Annual International Symposium on Computer Architecture*, pages 98–108, San Diego, CA, May 1993. ACM.

[5] A. Gonzalez, C. Aliagas, and M. Valero. A data cache with multiple caching strategies tuned to different types of locality. In *International Conference on Supercomputing*, pages 338–347, June 1995.

[6] M. D. Hill. *Aspects of Cache Memory and Instruction Buffer Performance*. PhD thesis, University of California, Berkeley, 1987.

[7] H. Hirata, K. Kimura, S. Nagamine, Y. Mochizuki, A. Nishimura, Y. Nakase, and T. Nishizawa. An elementary processor architecture with simultaneous instruction issuing from multiple threads. In *19th Annual International Symposium on Computer Architecture*, pages 136–145, May 1992.

[8] T. L. Johnson and W. W. Hwu. Run-time adaptive cache hierarchy management via reference analysis. In *24th Annual International Symposium on Computer Architecture*, pages 364–373, May 1997.

[9] D. Joseph and D. Grunwald. Prefetching using markov predictors. *IEEE Transactions on Computers*, 48(2):121–133, Feb. 1999.

[10] N. P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *17th Annual International Symposium on Computer Architecture*, pages 364–373, May 1995.

[11] V. Milutinovic, M. Tomasevic, B. Markovi, and M. Tremblay. A new cache architecture concept: the split temporal/spatial cache. In *Proceedings of 8th Mediterranean Electrotechnical Conference*, pages 1108–1111, May 1996.

[12] B. A. Nayfeh, L. Hammond, and K. Olukotun. Evaluation of design alternatives for a multiprocessor microprocessor. In *23rd Annual International Symposium on Computer Architecture*, pages 67–77, May 1996.

[13] T. H. Romer, D. Lee, B. N. Bershad, and J. B. Chen. Dynamic page mapping policies for cache conflict resolution on standard hardware. In *First Annual Symposium on Operating Systems Design and Implementation*, pages 255–266, Nov. 1994.

[14] A. J. Smith. Cache memories. *ACM Computing Surveys*, 14(3):473–530, Sept. 1982.

[15] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *22nd Annual International Symposium on Computer Architecture*, pages 414–425, June 1995.

[16] P. Stenstrom, M. Brorsson, and L. Sandberg. An adaptive cache coherence protocol optimized for migratory sharing. In *20th Annual International Symposium on Computer Architecture*, pages 109–118, San Diego, CA, May 1993. ACM.

[17] H. S. Stone. *High-Performance Computer Architecture*. Addison Wesley, 1987.

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

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

[20] G. Tyson, M. Farrens, J. Matthews, and A. R. Pleszkun. A modified approach to data cache management. In *28th Annual International Symposium on Microarchitecture*, pages 93–103, Dec. 1995.