Capturing Dynamic Memory Reference Behavior with Adaptive Cache Topology

Jih-Kwon Peir, Yongjoon Lee[*]
Computer & Information Science & Engineering Department
University of Florida
Gainesville, FL 32611
{peir,yongjoon}@cise.ufl.edu

Windsor W. Hsu[†]
Computer Science Division
University of California
Berkeley, CA 94720
windsorh@cs.berkeley.edu

Abstract

Memory references exhibit locality and are therefore not uniformly distributed across the sets of a cache. This skew reduces the effectiveness of a cache because it results in the caching of a considerable number of less-recently-used lines which are less likely to be re-referenced before they are replaced. In this paper, we describe a technique that dynamically identifies these less-recently-used lines and effectively utilizes the cache frames they occupy to more accurately approximate the global least-recently-used replacement policy while maintaining the fast access time of a direct-mapped cache. We also explore the idea of using these underutilized cache frames to reduce cache misses through data prefetching. In the proposed design, the possible locations that a line can reside in is not predetermined. Instead, the cache is dynamically partitioned into groups of cache lines. Because both the total number of groups and the individual group associativity adapt to the dynamic reference pattern, we call this design the adaptive group-associative cache. Performance evaluation using trace-driven simulations of the TPC-C benchmark and selected programs from the SPEC95 benchmark suite shows that the group-associative cache is able to achieve a hit ratio that is consistently better than that of a 4-way set-associative cache. For some of the workloads, the hit ratio approaches that of a fully-associative cache.

1 Introduction

The least-recently-used (LRU) replacement policy works extremely well for memory hierarchy caching schemes because of the locality of reference. However, for processor caches where access time and hardware complexity are major design issues, a global LRU replacement policy across the entire cache is impractical. Instead, the cache is typically organized into *sets* of cache lines within which the LRU replacement policy is used. In this study, we observe that such an organization is a poor approximation to the global LRU replacement policy because the more-recently-referenced cache lines are not evenly distributed across all the cache sets. This non-uniform distribution results in the caching of a significant number of less-recently-used lines which are less likely to be re-referenced before replacement. Such an effect is especially pronounced for the direct-mapped cache which, because of its fast access time, is often the cache topology of choice for first level caches [7]. For instance, results from trace-driven simulations show that on average, 40% of the cache frames in a direct-mapped data cache contain less-recently-used lines during the execution of the Transaction Processing Performance Council Benchmark C (TPC-C) [27], and that such lines have only about 1% chance of being reused before they are replaced.

In this paper, we describe a technique that dynamically identifies the underutilized cache frames in a direct-mapped cache and effectively uses them to store the data that are more likely to be re-referenced. In this technique, a history table is used to track the cache lines that have been referenced recently. When a cache miss occurs and the line being replaced has been referenced recently, it is moved into an alternate location within the cache. The alternate location is selected from among those that have not been accessed in the recent past. A small directory is used to keep track of the more-recently used lines that have been so displaced. In this design, the possible locations that a line can reside in is not predetermined as is the case in a set-associative cache. Instead, the cache is dynamically partitioned into groups of cache lines with the same index bits. Because the total number of groups and the individual group associativity adapt to the reference pattern, we call this design the *adaptive group-associative* cache.

To further improve cache performance, the underutilized cache frames may also be used to accommodate prefetched data. One of the major issues in data prefetching is that the

[*] Peir and Lee are supported in part by an NSF CAREER Award (MIP-9624498), a US Army Grant (DACA39-97-K-0037) and a donation from Intel Corporation.
[†] Hsu is also with the IBM Almaden Research Center.

prefetched data may not actually be used and will pollute the cache if they are brought in. With the proposed adaptive group-associative cache, the prefetched data can be confined to the underutilized cache frames, thereby reducing cache pollution without the need for a separate prefetch buffer.

Performance evaluation using trace-driven simulations of the TPC-C benchmark and selected programs from the SPEC95 benchmark suite shows that the group-associative cache is able to achieve a hit ratio that is consistently superior to that of a 4-way set-associative cache. For some of the workloads, the hit ratio approaches that of a fully-associative cache. The results also show that for most of the workloads, the miss ratio of the adaptive group-associative cache is more than 25% lower than those of two recently proposed enhanced cache organizations, namely the column-associative cache [2] and the victim cache [10]. Furthermore, by applying some simple data prefetching techniques to the group-associative cache, the miss ratio of TPC-C can be further reduced by about 20%.

The remainder of this paper is organized as follows. In the next section, we demonstrate that the direct-mapped, 2-way set associative, and even 4-way set associative caches do a poor job of tracking the global LRU replacement policy. In Section 3, we describe the proposed adaptive group-associative cache, present an example design, and discuss its performance advantages. In Section 4, we discuss simple data prefetching techniques for the group-associative cache. Section 5 contains the results of the performance evaluation as well as the comparison with the conventional, column-associative, and victim caches. A brief survey of related work is in Section 6. Section 7 concludes the paper.

2 Underutilized Cache Frames

The performance of a cache is determined both by the fraction of memory requests it can satisfy and the speed at which it can satisfy them. The simple direct-mapped cache provides a fast access time but tends to have a low hit ratio due to conflict misses [7]. In a direct-mapped cache, a line can only be located at a fixed position. This restrictive line placement means that lines that have been assigned the same cache frame have to replace one another, even when they have been referenced very recently. In other words, the direct-mapped cache is unable to always retain the set of more-recently-used lines, lines that, because of the locality of reference, are the most likely to be referenced again.

This inability to retain all the more-recently-used lines can be quantified by mapping the contents of a fully-associative, LRU-replacement cache to a direct-mapped cache. In this mapping, the proper index bits of each line in the fully-associative cache determines its location in the direct-mapped cache. A snapshot of such a mapping is shown in Figure 1. In a typical case, the more-recently-used lines are not evenly distributed across all the sets in the direct-mapped cache. For instance, several lines (*a*, *b*, *c*,

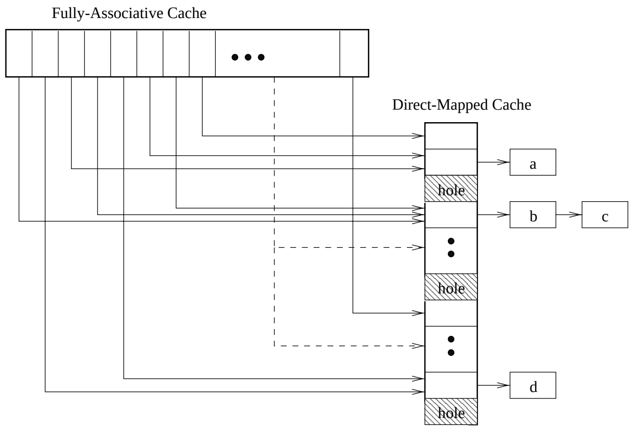

Figure 1: Mapping a Fully-Associative Cache to a Direct-Mapped Cache.

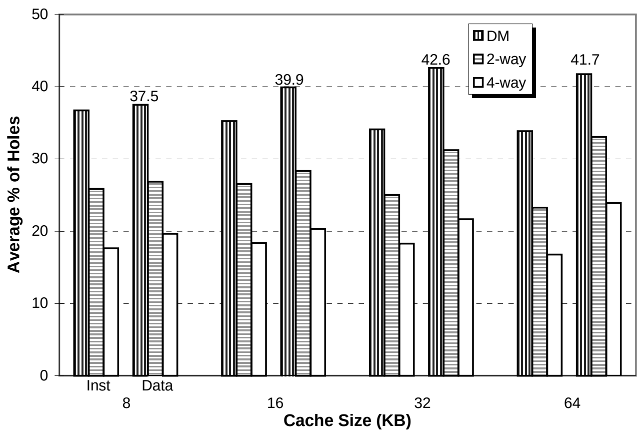

Figure 2: Average Percentage of Holes (TPC-C).

and *d*) cannot fit into the direct-mapped cache. As a result, the direct-mapped cache contains a number of empty frames or *holes*.

Figure 2 plots the average percentage of holes that exist in various cache configurations after each memory reference during the execution of TPC-C. The results show that a very significant number of holes exist in both the instruction and data caches. For the direct-mapped data caches, between 37.5% and 42.6% of the cache are holes. The corresponding ranges for the 2-way and 4-way set associative caches are about 27–33% and 20–24% respectively. Compared with the data caches, the instruction caches have somewhat fewer holes. About 34–37% of the direct-mapped instruction caches are holes. The corresponding numbers for the 2-way and 4-way set associative caches are about 24–27% and 17–18% respectively.

In real operation, these holes will be filled with less-recently-used lines which have a smaller chance of being reused before they are replaced. Simulations show that for TPC-C, the respective hit ratio to these less-recently-used lines is only 1.13%, 0.92%, 0.59%, and 0.30% for the 8KB, 16KB, 32KB, and 64KB direct-mapped data cache.

The existence of such holes also limits the performance impact of hit-ratio improvement techniques such as the column-associative cache and the victim cache. The column-associative cache is a direct-mapped cache in which each set has an alternate backup set that is accessed through a secondary hash function [2]. The secondary hash function is based on flipping the highest-order index bit used in the primary hash function. When a memory request misses in the primary set, the cache is accessed again with the secondary hash function. For maintaining the correct primary/secondary sequence, a *rehash* bit is included with each tag entry to indicate whether the line is accessed through the secondary hash function. In essence, the column-associative cache is effectively a 2-way set associative cache which still contains a significant number of holes. The victim cache is a separate fully-associative buffer that holds the recent victims of replacement, i.e. the lines that have been recently evicted from the direct-mapped cache [10]. In this approach, the less-recently-used lines that are evicted from the direct-mapped cache will filter through the victim cache, thereby polluting it and reducing its effectiveness. Moreover, a large number of holes remain in the direct-mapped cache.

3 Adaptive Group-Associative Caches

The basic idea behind the adaptive group-associative cache is to approach the fast access time of the direct-mapped cache while improving its hit ratio by identifying and using the existing *holes* to store the lines that have been recently replaced. There are thus three parts to this scheme. The first is to dynamically identify the holes. The second is to determine whether a line should be evicted or placed into a hole upon replacement. The third is to locate the *out-of-position* lines, i.e. the lines that have been displaced from their direct-mapped locations into holes. If the majority of holes are correctly identified and filled with more-recently-used lines, the group-associative cache can achieve a hit ratio approaching that of a fully-associative cache.

A straightforward approach to implementing the group-associative cache is to maintain two small directories. One directory, the *Set-reference History Table (SHT)*, tracks the sets that have been referenced recently. The other directory, the *Out-of-position Directory* or *OUT directory* for short, records the tags and locations of the lines that have been recently displaced from their direct-mapped positions. When a miss occurs in a set that is tracked by the SHT, the line to be replaced is not evicted from the cache but is instead moved to another location within the cache. The rationale for this is that the replaced line must have been referenced recently for its set to be recorded in the SHT. In order for this displaced line to be located, its address tag and new location or set-ID is entered into the OUT directory. When a line is neither recorded in the SHT nor the OUT directory, it is said to be *disposable*. Disposable lines are the candidates for eviction when a miss occurs.

In a group-associative cache, the lines with identical direct-mapped index bits belong to a *congruence group*. The total number of groups and the number of lines within each group adapt dynamically to the reference pattern. This is in contrast to conventional caches where the number of sets and the set-associativity are fixed. For instance, if the reference pattern is such that different lines with the same direct-mapped index bits are continually accessed, all the out-of-position lines may belong to the same congruence group. When there is more than one line in a congruence group, all but one must be located out of the direct-mapped location. The line in the direct-mapped position is located through the regular cache tag array. As is the case in the column-associative cache, a fast access time can be achieved in this case. The out-of-position lines are located through the OUT directory which is searched in parallel with the cache tag array. Besides determining hit or miss, the OUT directory also provides the location of the out-of-position lines. On a hit to an out-of-position line, the data array is accessed again with the correct set-ID obtained from the OUT directory.

3.1 An Example Design

Figure 3 shows the block diagram of a straightforward implementation of the adaptive group-associative cache. The SHT contains the set-ID (SID) of the sets that have been referenced recently. Since each set contains a single line in a direct-mapped cache, the SHT tracks the more-recently-used lines that are located in their direct-mapped positions. The OUT directory contains the address tag and the set-ID of the lines that have been recently displaced from their direct-mapped locations. In other words, the SHT and the OUT directory together define the set of lines that are not disposable, i.e. the set of lines that have been referenced recently and therefore should not be evicted. By design, a line cannot be recorded in both the SHT and the OUT directory at the same time. The performance of various SHT and OUT directory topologies and replacement policies will be investigated in Section 5. To simplify cache management, a *disposable* or *d* bit is maintained for each line to indicate whether the line should be evicted when it is replaced. As is the case for the valid bits, the *d* bits can be kept in a separate physical array.

Hit in Direct-Mapped Location. When a memory request occurs, the tag array, the data array, and the OUT directory are accessed in parallel. If there is a match in the tag array, the requested data is accessed as in a regular direct-mapped cache. The SHT is updated after a reference to reflect the most-recently-used line.

Hit in Out-Of-Position Lines. If the line is found through the OUT directory, the data is accessed in the next cycle using the set-ID fetched from the OUT directory. Afterwards, the requested line is swapped with the line located in the direct-mapped location so as to increase the hits to the direct-mapped position. A multiple-bank data array provides the needed bandwidth. Meanwhile, the tag corresponding to the

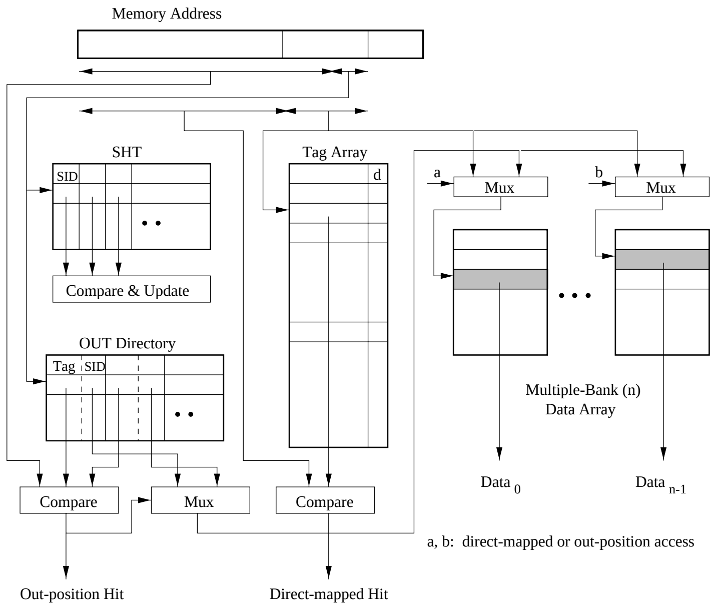

Figure 3: Block Diagram of a Group-Associative Cache.

requested line is dropped from the OUT directory and placed in the tag array at the direct-mapped position. The OUT directory is updated to record the new out-of-position line if it is not already there. Note that the cache tag array is used only to identify hits to the direct-mapped location. Out-of-position lines are located via the OUT directory. Therefore, there is no need to maintain the tags of the out-of-position lines in the tag array. Such entries are simply marked as invalid. This greatly simplifies the update of the tag array. In addition, unlike the fully-associative cache in which all the index bits are part of the address tag, none of the index bits is included in the direct-mapped tag array in the group-associative cache.

Cache Miss. There are two cases to consider when the requested line is not located anywhere in the cache. The first case happens when the line to be replaced in the direct-mapped location is disposable as indicated by the *d* bit. In this case, the line is simply evicted from the cache. The second case occurs when the line in the direct-mapped location is not disposable. In this case, a hole has to be identified to hold this line. The primary candidate is the LRU line in the OUT directory because the newly displaced line has to be entered into the OUT directory if it is not already there. When the OUT directory has empty slots, a nearby disposable line is selected for eviction. Such a selection can be implemented by searching a word of nearby *d* bits with a leading-1 detector. When this fails to find a disposable line, the LRU line in the OUT directory can be used as the backup candidate. The non-disposable line is then moved to the evicted line location in the data array to make room for the requested line. The tag of the requested line is placed at the direct-mapped location of the tag array and the tag corresponding to the evicted line is simply invalidated. In order to avoid extraneous searching of the SHT and the OUT directory when a cache miss occurs, the *d* bits should be accurately maintained. The *d* bit corresponding to a line is set when the line is dropped from the SHT or the OUT directory. It is reset when the line enters either directory.

Notice that in accessing the data array, an additional multiplexor is needed to select between the direct-mapped set and the set-ID retrieved from the OUT directory. However, this should have minimal impact on the direct-mapped hit time since the critical path is likely to be in the tag array access and the tag comparison logic [17], both of which remain unchanged from a direct-mapped design.

3.2 Performance Impact

The group-associative cache has a unique combination of features that enables it to more effectively utilize the available cache frames so as to reduce miss ratio. First of all, the lines recorded in both the SHT and the OUT directory have been referenced recently and therefore should not be evicted when a cache miss occurs. In other words, the SHT and the OUT directory help to more accurately maintain the global LRU information, thus improving the overall hit ratio.

Secondly, when a miss occurs, the line to be replaced in the direct-mapped location may be moved to another location in the cache instead of being evicted. This is similar to the victim cache approach where the replaced line or victim is always moved to the victim cache. However, unlike the victim cache which requires a separate physical cache to hold the victims, the group-associative cache is able to effectively use the large number of holes present in the direct-mapped cache to hold the victims. In a group-associative cache, only a separate directory is needed to record the tags and the locations of the out-of-position lines. Therefore, a much bigger “embedded victim cache” can be built at the same cost as the original victim cache. Moreover, the SHT and the OUT directory have a filtering effect, allowing only the more-recently-used lines to enter the embedded victim cache. This selective bypassing technique helps to reduce pollution in the embedded victim cache. In fact, compared to the fully-associative cache, the group-associative cache is less affected by cache pollution in that a line with poor locality can affect only a subset of the cache lines.

Thirdly, by recording the tag and location of the out-of-position lines in the OUT directory, the group-associative cache allows these lines to be placed anywhere in the cache, and not just in a fixed alternate location as is the case in the column-associative cache. This allows the out-of-position lines to share a common pool of potential holes, thus enabling a more efficient utilization of the cache then would occur under a static partitioning scheme. In addition, by dynamically allocating the holes in response to the reference pattern, the group-associative cache is able to minimize adverse impact on the hit ratio of the direct-mapped locations.

Finally, the ability to dynamically adjust the number of groups and the group associativity enables the group-

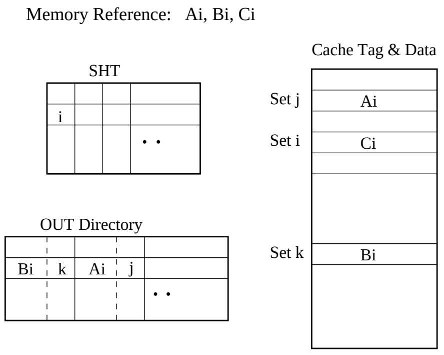

Figure 4: Accessing a Group-Associative Cache.

associative cache to approach a fully-associative cache in terms of miss ratio. For instance, in contrast to the column-associative cache which limits each set to only two lines, the group-associative cache can have groups containing anywhere from 0 to *s+1* lines, where *s* is the size of the OUT directory. This adaptive group size enables the group-associative cache to better capture program locality. For instance, suppose that three consecutive memory references A_i, B_i, C_i are mapped to the same cache set i. After the three references, only C_i will remain in a direct-mapped cache. In a column-associative cache, C_i will be kept in the primary location and B_i, in the alternate location. In comparison, all three most-recently-used lines will remain in the group-associative cache, as illustrated in Figure 4. In the figure, sets j and k have been identified as potential holes and are used to hold A_i and B_i respectively.

4 Extension to Handle Data Prefetch

The cache miss ratio can be further reduced by attempting to fetch data before they are actually needed. The major issue in such prefetching of data is that it may result in extraneous fetches and memory/bus traffic. Furthermore, if the prefetched data are entered into the cache, they may replace useful lines, thereby polluting the cache and generating more miss traffic. In order to prevent cache pollution, a separate stream or prefetch buffer [10, 9] has been used to hold the prefetch data. Since the group-associative cache can dynamically identify the cache frames that are not being effectively utilized, it may be worthwhile to explore the idea of prefetching data into these underutilized cache frames. This eliminates the need for a separate physical buffer and enables a bigger "embedded" prefetch buffer.

Several issues are involved in evaluating the cost and effectiveness of a prefetch scheme. In this paper, we present some preliminary results. We consider two simple prefetch methods. Whenever a miss occurs, the *sequential prefetch* method prefetches the next sequential line if it is not already in the cache. Upon the first access to a prefetched line, it triggers the prefetch of the following sequential line [22]. The *filtered* sequential prefetch method starts prefetching the next sequential line on a miss only when a previous sequential access pattern has been identified [18, 16]. In general, the filtered sequential scheme has a higher prefetch accuracy but results in a smaller cache hit ratio improvement.

Extending the group-associative cache to handle data prefetch is straightforward. When the direct-mapped location for a prefetched line consists of a disposable line, the disposable line is simply replaced and the d bit remains set. When the direct-mapped location is occupied by a non-disposable line, a hole must be identified to hold the prefetched line. This goes through the same mechanism that is used to identify a hole when a regular cache miss occurs and the direct-mapped location has a non-disposable line. To reduce any pollution of the OUT directory, the tag corresponding to the prefetched line can be inserted into the OUT directory in the middle of the LRU sequence.

To some extent, both the victim and column-associative caches also have the nice property of being able to accommodate prefetched data with limited adverse impact to the existing cache contents. For the victim cache, instead of placing the prefetched data directly in the direct-mapped cache, the prefetched lines can be entered into the fully-associative victim cache. For the column-associative cache, the prefetched data can be placed at the primary or secondary location that has the rehash bit set.

5 Performance Evaluation

In this section, we evaluate the performance impact of the group-associative cache using trace-driven simulations of workloads from both the commercial and engineering environments. Two basic metrics, miss ratio and average memory access time, are considered. Conventional and other recently proposed cache organizations, namely the victim cache and the column-associative cache, are evaluated and compared against the group-associative cache. In addition, the effectiveness of data prefetching on the selected cache organizations are also investigated.

5.1 Simulation Model

We simulate separate and identical instruction and data L_1 caches. The size of the L_1 cache ranges from 8 to 64KB with set-associativities of 1, 2, and 4. Fully-associative caches are also considered. These L_1 caches are backed up by a 512KB 4-way set-associative unified level 2 (L_2) cache. The line sizes of the L_1 and L_2 caches are 32 bytes and the LRU replacement policy is used in all the cases. Inclusion property is enforced between the L_1 and L_2 data caches.

For the group-associative cache, we vary the number of entries in the SHT from one-eighth to one-half the number of L_1 cache lines (i.e. 64 to 256 lines for 16KB cache), and the number of entries in the OUT directory from one-sixteenth to three-eighth the number of L_1 cache lines (i.e. 32 to 192 lines for 16KB cache). In addition, we vary the number of sets for both the SHT and the OUT directory from 1 to 16.

Within each set, we compare the true LRU replacement policy with a simpler partitioned LRU (PLRU) scheme [24]. Note that when the requested line is found in the OUT directory, it needs to be swapped with the line located in the direct-mapped location of the data array. To simplify this swap, the number of sets in the OUT directory must not exceed that in the SHT. In our simulations, we assume an equal number of sets in both directories and an identical design for both the instruction and data L_1 caches so as to confine the total design space.

In addition, we simulate fully-associative victim caches of one-sixteenth the L_1 cache size (i.e. 32 lines for 16KB cache). For the column-associative cache, we determine the secondary location by flipping the highest-order index bit. As described in [2], we include a *rehash* bit with each entry of the tag array to guide the search and replacement.

We consider two simple data prefetching techniques, sequential prefetch [22] and filtered sequential prefetch [18, 16]. For the filter, we use an 8-entry history table to identify sequential access patterns. Based on these techniques, we evaluate the hit ratio improvement as well as the extra memory traffic generated by data prefetching. For the direct-mapped cache, we assume that a separate prefetch buffer thirty-two times smaller than the L_1 cache (i.e. 16 lines for 16KB cache) is used to hold the prefetched lines. For the victim cache, we use the additional victim cache to hold both the recent victim lines and the prefetched lines. As in the group-associative cache, the prefetched line is placed in the middle of the LRU sequence. Finally, for the column-associative cache, we try to place the prefetched lines in the locations that contain rehashed lines.

In order to compute the average memory access time, the cache miss penalties at various cache levels are needed. Without going into a detailed timing analysis, we estimate these penalties based on a general trend of current microprocessors. We assume that a hit in a conventional direct-mapped L_1 cache requires a single cycle. If the memory request hits in the L_2 cache, it takes 8 cycles to satisfy the request. When the request misses both the L_1 and L_2 caches, the total access delay is 50 cycles. For set-associative caches, we assume that the cycle time is lengthened by up to 20% as suggested in [11], and adjust the miss penalties accordingly.

For the victim, column-associative, and group-associative caches, an extra delay is encountered when the requested data is present in an alternative location. Due to the fact that the processor pipeline is increasingly complex and difficult to turn around, we assume that the extra delay is 2 cycles. This is a conservative assumption since no other request is allowed to access the cache during this three cycle period. Note that the search of the alternative locations does not add to the L_1 cache miss penalty because the L_1 cache miss can be triggered once the requested line is not present in the primary location. In other words, the delay of searching the alternative locations can be overlapped with the L_1 cache miss penalty.

5.2 Workloads and Traces

We simulate workloads from both the commercial and engineering environments. For the commercial environment, we use the Transaction Processing Performance Council Benchmark C (TPC-C) [27]. The TPC-C benchmark is an industry standard benchmark for measuring the performance of on-line transaction processing systems. It is modeled after an order-entry environment and involves a mix of five distinct transaction types. Our trace captures the user, kernel and shared library activities of the server side of the workload. This was collected by a software tracing tool on an IBM RISC System/6000 system running AIX.

For the engineering environment, we use 11 applications from the SPEC95 benchmark suite [25]. Among these 11 applications are 5 integer intensive programs (*Compress, Gcc, Go, Li,* and *Vortex*), and 6 floating-point intensive programs (*Fpppp, Hydro2d, Su2cor, Tomcatv, Turb3d,* and *Wave5*). We used Sun's *Shade* tool [26] in a SPARC/Solaris environment to trace these SPEC95 applications. The standard SPEC95 input files were used. In order to avoid the initialization phase and capture the essential characteristics of these applications, the first 2 billion instructions were skipped. Our results are based on simulating 2 billion instructions after the caches are warmed-up.

5.3 Performance of Group-Associative Cache

We simulate different sizes and topologies of the SHT and OUT directory to establish reasonable design points. We use the notation *(a,b)* to denote a particular design point where the sizes of the SHT and OUT directory with respect to the number of L_1 cache lines are *a* and *b* respectively. There are several important results.

First, increasing the number of SHT entries beyond three-eighth the number of cache lines hardly improves the miss ratio. On the other hand, small SHTs with entries to track only one-eighth the number of cache lines do not perform well. In this case, increasing the size of the OUT directory may even hurt the miss ratio. This is because the SHT is responsible for identifying the more-recently-used lines that should be moved into the OUT directory. When the SHT is small, it is identifying too few more recently-used-lines. As a result, some lines that have not been referenced for a while will remain in the OUT directory. A balance between the capacity to store more-recently-used out-of-position lines and the ability to identify them is desirable.

Second, increasing the number of entries in the OUT directory usually improves the miss ratio. However, the improvement starts to diminish when more than one-quarter of the locations are allocated to the out-of-position lines. This is due to the fact that although increasing the number of OUT directory entries does improve the hit ratio to the out-of-position lines, it also hurts the hit ratio to the direct-mapped locations.

Third, increasing the number of sets in the SHT and OUT

Cache Size (SHT,OUT)	Sets	Replacement Algorithm	Direct-mapped Hits (%)	Total Hits (%)
16KB $(\frac{3}{8},\frac{4}{16})$	1 Set	LRU	85.817	90.846
		PLRU	85.778	90.737
	8 Set	LRU	85.920	90.892
		PLRU	85.903	90.823
	16 Set	LRU	85.890	90.891
		PLRU	85.923	90.782

Table 1: Data Cache Hit Ratio with Different SHT/OUT Directory Replacement Policies (TPC-C).

directory from 1 to 16 has limited effect on performance. Moreover, as shown in Table 1, the difference between the true LRU and the PLRU replacement schemes is very minor. Intuitively, we expect a fully-associative design (i.e. set=1) with true LRU replacement to outperform the set-associative design with PLRU replacement. However, the difference should be very small because the SHT and OUT directory only help to identify the lines that should be kept in the cache. They do not directly determine the lines that should be evicted. Therefore, even though the SHT and OUT directory become a little less accurate when they are organized into more sets, any adverse impact on the overall miss ratio is limited. In fact, the 8-set configuration shows a little better hit ratios than the 1-set configuration. A deeper analysis reveals that this unexpected behavior is a consequence of our algorithm for filling the OUT directory. Recall that the algorithm tries to keep the OUT directory full. In other words, it always tries to hoard cache locations for storing the out-of-position lines. Depending on the reference pattern, such an aggressive policy may adversely affect the hit ratio to the direct-mapped locations. When the number of sets is increased, the hoarding phenomenon is effectively reduced because the OUT directory has more sets each of which has to be separately filled.

Based on the simulation results, we select two SHT sizes – $\frac{2}{8}$ and $\frac{3}{8}$. When the number of entries in the SHT is two-eighth the number of cache lines, the OUT size is either $\frac{3}{16}$ or $\frac{4}{16}$. When the SHT size is $\frac{3}{8}$, the number of entries in the OUT directory is four-sixteenth or five-sixteenth the number of cache lines. Since the number of sets and the replacement schemes have little performance impact, we show only the results with 8 sets and true LRU replacement. While we evaluated the group-associative cache for both the instruction and data streams, due to space constraints, we only present the results for the data references.

Observe in Figure 5 that for TPC-C, the group-associative cache is able to achieve a miss ratio that is comparable to or better than that of the 4-way set-associative cache. In certain cases, the miss ratio approaches that of the fully-associative cache. The results suggest that the group-associative cache can indeed retain a majority of the more-recently-used lines. Further confirming this effect, we find that the average percentage of holes in the 8KB, 16KB,

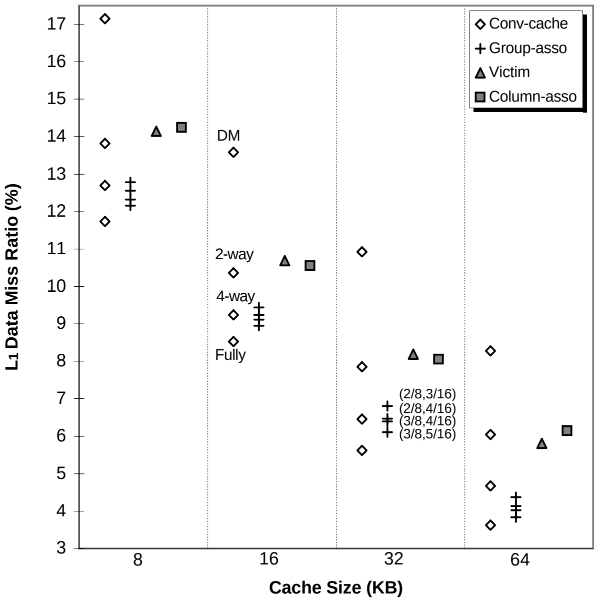

Figure 5: Miss Ratio Comparison (TPC-C).

32KB, and 64KB data caches has improved respectively from 37.5%, 39.9%, 42.6%, and 41.7% in the direct-mapped design to 18.2%, 18.7% 19.9%, and 19.0% in the $(\frac{3}{8},\frac{4}{16})$ group-associative caches. With a slightly bigger OUT directory $(\frac{5}{16})$, the percentage of holes is further reduced to 15.9%, 15.5%, 15.6%, and 14.5% respectively.

In addition, the selected group-associative caches achieve lower miss ratio than the victim and column-associative caches and the margin can be very sizable. For instance, for the 32KB caches, the miss ratio for the victim cache and the column-associative cache is about 28% and 26% higher than that for the $(\frac{3}{8},\frac{4}{16})$ group associative cache. Among the group-associative caches, those with bigger SHT and OUT directories perform better. For example, $(\frac{3}{8},\frac{5}{16})$ has the lowest miss ratio followed by $(\frac{3}{8},\frac{4}{16})$. The victim and the column associative caches show very similar miss ratio that is close to that of the 2-way set-associative cache.

There are two fundamental reasons as to why the group-associative cache is able to achieve a better overall miss ratio. First, the group-associative cache has the ability to capture extra hits to the out-of-position lines. Second, it is able to do this with minimal adverse impact on the hit ratio of the direct-mapped locations. This is illustrated in Figure 6 in which we plot the hit ratio to the direct-mapped and the alternative locations for the victim, column-associative and group-associative caches. As expected, the hit ratio to the direct-mapped locations remain unchanged for the victim cache. This hit ratio is reduced for both the column- and group-associative caches. However, due to the flexible locations for the out-of-position lines and the adaptive sharing of these locations, the reduction in hit ratio to the direct-mapped locations is very minimal for the group-associative cache. For instance, the hit ratio decreases from

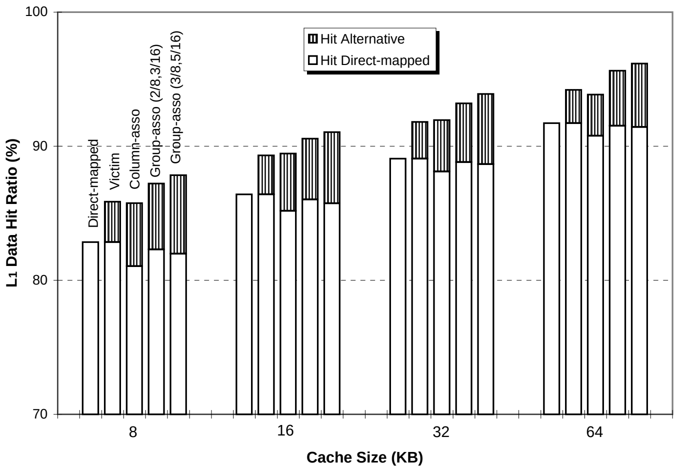

Figure 6: Impact on Direct-Mapped Hit Ratio (TPC-C).

89.1% to 88.8%, and 88.7% for the 32KB ($\frac{2}{8}$,$\frac{3}{16}$) and ($\frac{3}{8}$,$\frac{5}{16}$) group-associative caches while it decreases to 88.1% for the column-associative cache.

Furthermore, notice that the hit ratio of the alternative locations is higher for the group-associative cache. With a 32KB cache, the hit ratio of the out-of-position lines is 4.40% for the ($\frac{2}{8}$,$\frac{3}{16}$) group-associative cache and 5.24% for the ($\frac{3}{8}$,$\frac{5}{16}$) group-associative cache. The hit ratio of the alternative locations in the column-associative cache is 3.84% while the hit ratio of the victim cache is only 2.74%. In comparison with the victim cache, the higher hit ratio of the group-associative cache comes from the utilization of holes to build the bigger embedded victim cache plus the ability to selectively bypass this embedded victim cache. When compared to the column-associative cache, the group-associative cache prevails because of its ability to dynamically adjust the number of groups and the group associativity.

The different delays in accessing the direct-mapped and the alternative locations should be considered in evaluating the performance of the various cache organizations. Recall that in our simulation model, we assume that a hit to an alternative location in the victim, column- and group-associative caches takes 3 cycles. Based on results in [11], we also assume that the set-associative design lengthens the cycle time by up to 20%.

Figure 7 summarizes the average memory access time for the data references with various cache organizations. Note that all the results are normalized to the direct-mapped cycle time. Due to the longer cycle time with the set-associative cache, the conventional cache does not perform as well as the other cache organizations. Only the 4-way set associative design with an optimistic 10% cycle time degradation shows performance comparable to the victim and the column-associative caches. The group-associative cache has the shortest average memory access time. For instance, for the 32KB cache, the best average memory access time for the group-associative, victim and column-associative caches is 2.44, 2.54, and 2.55 respectively.

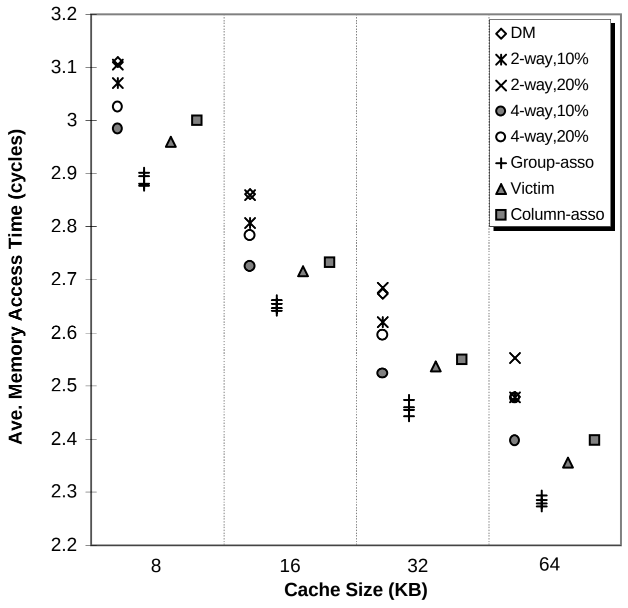

Figure 7: Ave. Memory Access Time Comparison (TPC-C).

Total Extra Bits	SHT	OUT	d-bit
13.7K	7 × 384	(32+7) × 256	1 × 1024

Table 2: Extra Space Calculation

The group-associative cache does require additional chip area to implement the SHT, the OUT directory and the *disposable* bits. As shown in Table 2, the extra space needed for the SHT and OUT directory of the 32KB ($\frac{3}{8}$,$\frac{4}{16}$) group-associative cache with 8 sets and a 40-bit address space is about 13.7 Kilo-bits (Kb). Without accounting for the peripheral logic, this additional area is less than 5% of that taken up by the 32KB direct-mapped cache. In comparison, the extra area needed for a victim cache with one-sixteenth the number of L_1 cache lines is over 6%.

5.4 Performance with Data Prefetch

Figure 8 summarizes the miss ratio improvement as well as the increase in memory traffic that results from applying the two simple prefetch mechanisms to the various cache designs. Each column in the figure is divided into 3 segments. The bottom-most segment depicts the cache miss ratio with data prefetch. The second segment represents the miss ratio improvement that comes from prefetching the data. The last segment reflects the net extra memory traffic that results when prefetching is performed. Note that in this figure, we consider only the ($\frac{3}{8}$,$\frac{4}{16}$) group-associative cache.

As expected, the two prefetching schemes both reduce the cache miss ratio markedly for all the cache designs. For instance, for the 32KB caches, sequential prefetch reduces the miss ratio for the direct-mapped, victim, column-associative and group-associative caches from 10.9%, 8.2%,

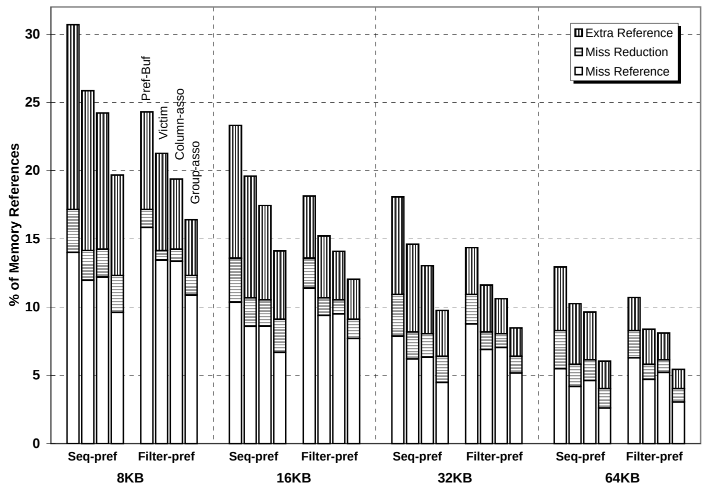

Figure 8: Miss Ratio and Memory Traffic with Data Prefetching (TPC-C)

8.1%, and 6.4% to 7.9%, 6.2%, 6.3%, and 4.4% respectively. However, memory traffic is also increased considerably. With sequential prefetch, memory traffic is increased by 65%, 78%, 62%, and 52% respectively. The corresponding figures with filtering are 31%, 41%, 32%, and 31%.

In comparison with the direct-mapped cache with separate prefetch buffer, the victim cache and the column-associative cache, the group-associative cache handles data prefetching better in terms of both the miss ratio improvement and the additional memory traffic. This is due to the fact that the group-associative cache is able to effectively control any cache pollution that may result from prefetching. In addition, the adaptive group associativity allows the prefetched lines to stay in the cache longer, thus increasing their chances of being used before replacement.

5.5 SPEC95 Applications

Figures 9 plots the data miss ratios for TPC-C and the selected programs from the SPEC95 benchmark suite. Observe that the selected SPEC95 programs, especially those that are floating-point intensive, show vastly different cache behavior. In these figures, we consider only two configurations, namely $(\frac{2}{8}, \frac{3}{16})$ and $(\frac{3}{8}, \frac{4}{16})$, for the group-associative cache. To reduce clutter, we only present the effect of applying filtered sequential prefetch.

The group-associative cache consistently achieves a miss ratio that is equal to or better than that of the 4-way set-associative cache for the selected SPEC95 programs. The miss ratio improvement is especially significant for applications such as Gcc, Go, Tomcatv, Turb3d, Vortex, and Wave5 which exhibit high conflict misses. For instance, Turb3d's miss ratio with a 32KB cache is reduced from 5.5% and 3.7% with the direct-mapped and 4-way set-associative designs respectively to 2.6% with the $(\frac{2}{8}, \frac{3}{16})$ group-associative cache. On the other hand, since most of the conflict misses for Compress, Hydro2d and Su2cor can be eliminated by the 2-way set-associative design, the benefit of the group-associative cache is not as dramatic. Observe also that the difference in miss ratio between the two group-associative caches is very minor for these SPEC95 applications.

Notice that the cache behavior of Tomcatv and Wave5 is unusual in that the direct-mapped, set-associative and group-associative designs may slightly out-perform the fully-associative cache in certain configurations. This is due to the fact that memory references with a constant stride are very common in these programs. Such a reference pattern results in heavy conflicts and pollutes the entire fully-associative cache. On the other hand, the group-associative cache is able to handle the conflicts by effectively using the holes while confining the pollution to a subset of the cache.

The filtered sequential prefetch scheme improves the miss ratio of the group-associative cache for all the programs, especially those that are floating-point intensive. For instance, the respective improvement for Hydro2d and Su2cor are about 60-70% and 30-60%. For Su2cor, prefetching is especially effective for the larger caches. In addition, the difference between the two group-associative configurations is much bigger when prefetching is performed. This is because Su2cor has a large amount of strided references and an uneven reference distribution across the cache sets. In this case, a larger cache and a bigger SHT/OUT directory will enable the prefetched lines to be kept longer so that they are more likely to be referenced. Gcc, Li, Vortex, Tomcatv, Turb3d, and Wave5 also show sizable improvements in group-associative miss ratio with prefetching.

6 Related Work

A general strategy to simultaneously attain a fast cache access time and a high hit ratio is to have two cache access paths. A fast path is used to achieve fast access time for the majority of memory references while a relatively slow path is used to boost the effective hit ratio. Two broad categories of such techniques can be distinguished.

The general idea in the first category is to decouple the tag and data paths in cache access so that, for the majority of memory references, the fast data array access and line selection can be carried out independently of the slow tag array access and comparison. Examples of techniques in this category include the MRU cache [4], the line-ID prediction scheme [15, 3], the partial-tag matching technique [14], the Direct-mapped Access Set-associative Check (DASC) cache [21], the difference-bit directory [12], and the alternative tag path method [17].

Techniques in the second category access a direct-mapped cache sequentially more than once in order to achieve a fast access time for the first access and a high hit ratio as a whole. Examples of such techniques include the hash-rehash cache [1] and the column-associative cache [2]. A way to extend the column-associative cache to include

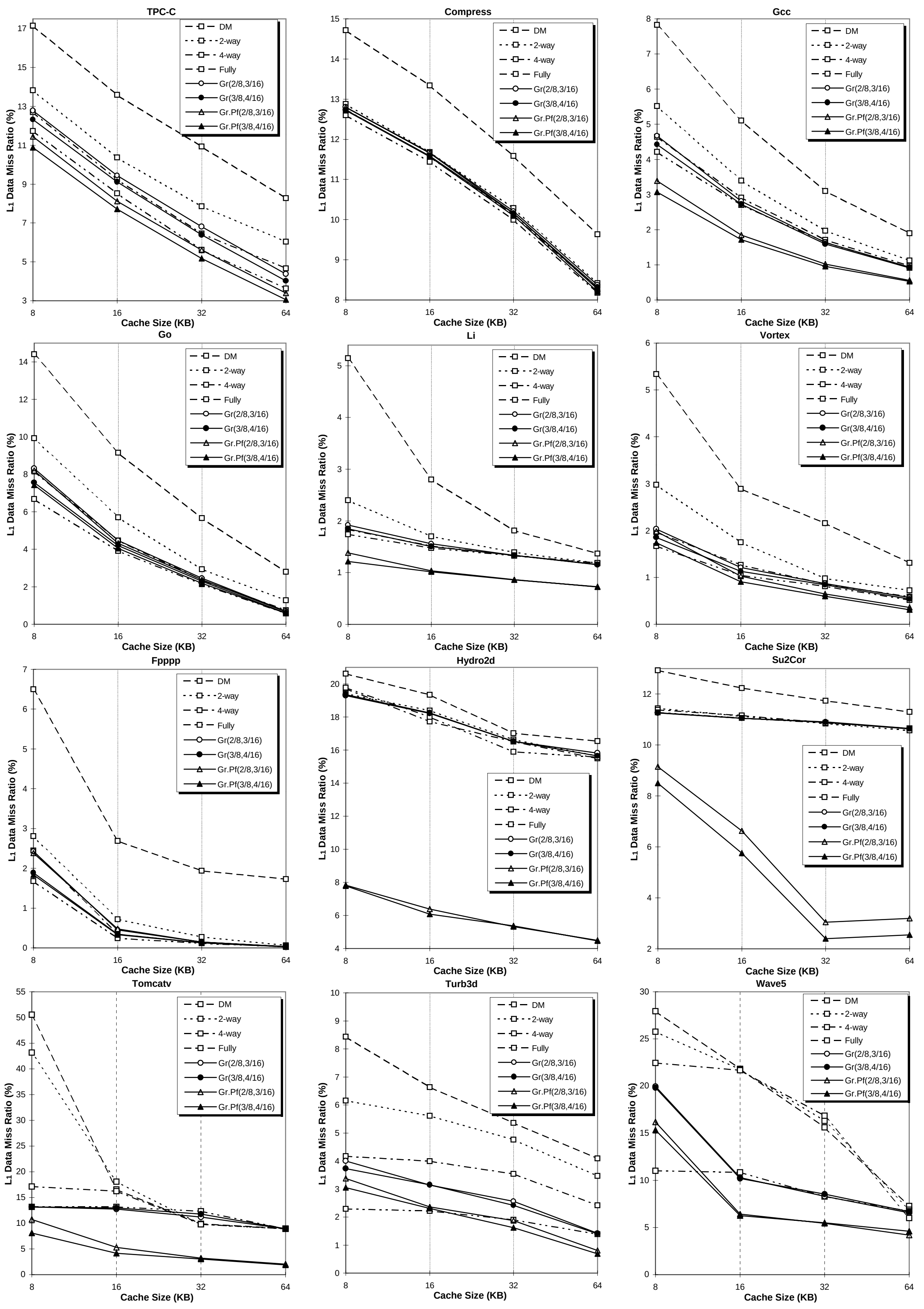

Figure 9: Cache Miss Ratio for Conventional and Group-Associative Caches (TPC-C and SPEC95).

multiple alternative locations is described in [28, 5].

A number of methods have been proposed to reduce cache conflict misses. One technique is to build a small buffer or victim cache to hold the lines that have been recently evicted from the cache [10]. The HP-PA7200 uses a small on-chip FIFO buffer called the *assist* cache, in addition to a direct-mapped L_1 cache, to ensure that the very recently used data will not be susceptible to conflict misses [13]. In [19, 8], a small fully-associative buffer is proposed for holding the lines that exhibit poor temporal locality so as to prevent them from entering and polluting the primary direct-mapped cache. Another approach to reducing conflict misses is to use better hashing or mapping functions [23, 20, 6].

7 Conclusions

In this paper, we observe that the direct-mapped cache, instead of faithfully maintaining the lines that have been referenced recently, retains a large number of less-recently-used lines that are not likely to be re-referenced before they are replaced. Based on this observation, we propose an adaptive group-associative cache that is able to dynamically identify the underutilized cache frames and to effectively use them to selectively retain some of the lines that are to be replaced. Performance evaluation using trace-driven simulations of both the TPC-C benchmark and selected programs from the SPEC95 benchmark suite show that the group-associative cache is able to decisively outperform the conventional and various performance-enhanced cache organizations. In particular, the miss ratio of the adaptive group-associative cache is consistently better than that of the 4-way set-associative cache and, in some cases, even approaches that of the fully-associative cache. As a result, the adaptive group-associative cache has the lowest average memory access time among the different cache organizations. Furthermore, our preliminary assessment indicates that the group-associative cache is able to handle data prefetching better than other cache organizations. In terms of cost, a first-cut estimate shows that the directories of the group-associative cache require about 5% of the area taken up by the cache.

8 Acknowledgment

The authors would like to thank Honesty Young of IBM Research and the anonymous referees for their valuable comments.

References

[1] A. Agarwal, J. Hennessy, and M. Horowitz, "Cache Performance of Operating Systems and Multiprogramming," *ACM Trans. Computer Systems*, Vol. 6(4), Nov. 1988, pp. 393–431.

[2] A. Agarwal, and S. Pudar, "Column-Associative Caches: A Technique for Reducing the Miss Rate of Direct-Mapped Caches," *Proc. 20th Int'l Symp. Comp. Arch.*, San Diego, CA, May 1993, pp. 179–190.

[3] B. Calder, D. Grunwald, and J. Emer, "Predictive Sequential Associative Cache," *Proc. 2nd Symp. High-Performance Comp. Arch.*, San Jose, CA, Jan. 1996, pp. 244–253.

[4] J. Chang, H. Chao, and K. So, "Cache Design of A Sub-Micron CMOS System/370," *Proc. 14th Int'l Symp. Comp. Arch.*, Pittsburgh, PA, June 1987, pp. 208–213.

[5] B. Chung, and J. Peir, "LRU-Based Column Associative Caches," *Comp. Arch. News*, Vol. 26(2) May 1998, pp. 9–17.

[6] A. Gonzalez, M. Valero, N. Topham and J.M. Parcerisa, "Eliminating Cache Conflict Misses Through XOR-Based Placement Functions," *Proc. 11th Int'l Conference Supercomputing*, Vienna, Austria, 1997, pp. 76–83.

[7] M. Hill "A Case for Direct-Mapped Caches," *IEEE Computer*, Vol. 21(12), Dec. 1988, pp. 25–40.

[8] T. Johnson, and W. Hwu, "Run-Time Adaptive Cache Hierarchy Management via Reference Analysis," *Proc. 24th Int'l Symp. Comp. Arch.*, Denver, CO, Jun. 1997, pp. 315–326.

[9] D. Joseph, and D. Grunwald, "Prefetching using Markov Predictors," *Proc. 24th Int'l Symp. Comp. Arch.*, Denver, CO, Jun 1997, pp. 252–263.

[10] N. Jouppi, "Improving Direct-Mapped Cache Performance by the Addition of A Small Fully-Associative Cache and Prefetch Buffers," *Proc. 17th Int'l Symp. Comp. Arch.*, Seattle, WA, May 1990, pp. 364–373.

[11] N. Jouppi and S. Wilton "Tradeoffs in Two-Level On-Chip Caching," *Proc. 21st Int'l Symp. Comp. Arch.*, Chicago, IL, April 1994, pp. 34–45.

[12] T. Juan, T. Lang, and J. Navarro, "The Difference-bit Cache," *Proc. 23rd Int'l Symp. Comp. Arch.*, Philadelphia, PA, May 1996, pp. 114–120.

[13] G. Kurpanek, et. al, "PA7200: A PA-RISC Processor with Integrated High Performance MP Bus Interface," *COMPCON Digest of Papers*, San Francisco, CA, Feb. 1994, pp. 375–382.

[14] L. Liu "Cache Design with Partial Address Matching," *MICRO'27*, San Jose, CA, Dec. 1994, pp. 128–136.

[15] L. Liu, "History Table for Set Prediction for Accessing a Set-Associative Cache," *United States Patent No. 5,418,922*, May 1995.

[16] S. Palacharla, and R. Kessler, "Evaluating Stream Buffers as a Secondary Cache Replacement," *Proc. 16th Int'l Symp. Comp. Arch.*, Chicago, IL, April 1994, pp. 24–33.

[17] J. Peir, W. Hsu, H. Young, and S. Ong, "Improving Cache Performance with Balanced Tag and Data Paths," *Proc. 7th Int'l Conf. Architectural Support for Programming Languages and Operating Systems*, Cambridge, MA, Oct. 1996, pp. 268–278.

[18] J. Pomerene, et. al, "Prefetching System for a Cache Having a Second Directory for Sequentially Accessed Blocks," US Patent 4807110, Feb. 1989.

[19] J. Rivers, and E. Davidson, "Reducing Conflicts in Direct-Mapped Caches with A Temporality-Based Design," *Proc. 1996 Int'l Conf. Parallel Processing*, Ithaca, NY, Aug. 1996, pp. 151–162.

[20] A. Seznec, "A Case for Two-Way Skewed-Associative Caches," *Proc. 20th Int'l Symp. Comp. Arch.*, San Diego, CA, May 1993, pp. 169–178.

[21] A. Seznec, "DASC Cache," *Proc. 1st Symp. High-Performance Comp. Arch.*, Raleigh, NC, Jan. 1995, pp. 134–143.

[22] A. Smith, "Sequential Program Prefetching in Memory Hierarchies," *IEEE Computer*, Vol. 11(12), Dec. 1978, pp. 7–21.

[23] A. Smith, "Cache Memories," *Computing Surveys*, Vol. 14(3), Sep. 1982, pp. 473–530.

[24] K. So and R. Rechtschaffen, "Cache Operations by MRU Change," *IEEE Trans. Computers*, Vol. 37(6), Jun. 1988, pp. 700–709.

[25] SPEC CPU95 Benchmark Suite, Version 1.10, Aug. 1995.

[26] Sun Microsystems, "Introduction to Shade," Revision C, April 1993.

[27] TPC Council, "TPC Benchmark C, Standard Specification, Rev. 3.6.2," Jun. 1997.

[28] C. Zhang, X. Zhang, and Y. Yan, "Two Fast and High-Associativity Cache Schemes," *IEEE Micro*, Vol. 17(5), Sep/Oct 1997, pp. 40–49.