LABORATORY FOR COMPUTER SCIENCE

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

MIT/LCS/TM-489

COLUMN-ASSOCIATIVE CACHES: A TECHNIQUE FOR REDUCING THE MISS RATE OF DIRECT-MAPPED CACHES

Anant Agarwal
Steven D. Pudar

November 1993

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

Column-Associative Caches:
A Technique for Reducing the Miss Rate of Direct-Mapped Caches

Anant Agarwal and Steven D. Pudar
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139

Abstract

Direct-mapped caches are a popular design choice for high-performance processors; unfortunately, direct-mapped caches suffer systematic interference misses when more than one address map into the same cache set. This paper describes the design of *column-associative* caches, which minimize the conflicts that arise in direct-mapped accesses by allowing conflicting addresses to dynamically choose alternate hashing functions, so that most of the conflicting data can reside in the cache. At the same time, however, the critical hit access path is unchanged. The key to implementing this scheme efficiently is the addition to each cache set of a *rehash bit*, which indicates whether that set stores data that is referenced by an alternate hashing function. When multiple addresses map into the same location, these *rehashed locations* are preferentially replaced. We demonstrate using trace-driven simulations and an analytical model that a column-associative cache removes virtually all interference misses for large caches, without altering the critical hit access time.

1 Introduction

The cache is an important component of the memory system of workstations and mainframe computers, and its performance is often a critical factor in the overall performance of the system. The advent of RISC processors and VLSI technology have driven down processor cycle times and made frequent references to main memory unacceptable.

Caches are characterized by several parameters, such as their size, their replacement algorithm, their block size, and their degree of associativity [1]. For cache accesses, a typical address a is divided into at least two fields, the tag field (typically the high-order bits) and the index field (the low-order bits), as shown in Figure 1. The index field is used to reference one of the sets, and the tag field is compared to the tags of the data blocks within that set. If the tag field of the address matches one of tag fields of the referenced set, then we have a *hit*, and the data can be obtained from the block that exhibited the hit.[1] In a *d-way set-associative cache*, each set contains d distinct blocks of data accessed by addresses with common index fields but different tags. When the degree of associativity is reduced to one, each set can then hold no more than one block of data. This configuration is called a *direct-mapped cache*.

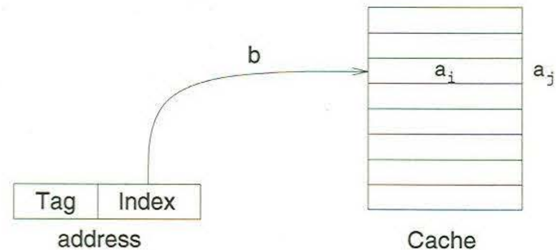

Figure 1: Indexing into a direct-mapped cache using bit-selection hashing.

For a cache of given size, the choice of its degree of associativity influences many performance parameters such as the silicon area (or, alternatively, the number of chips) required to implement the cache, the cache access time, and the miss rate. Because a direct-mapped cache allows only one data block to reside in the cache set that is directly specified by the address index field, its *miss rate* (the ratio of misses to total references) tends to be worse than that of a set-associative cache of the same total size. However, the higher miss rate of direct-mapped caches is mitigated by their smaller *hit access time* [2, 3]. A set-associative cache of the same total size always displays a higher hit access time because an associative search of a set is required during each reference, followed by a multiplexing of the appropriate data word to the processor. Furthermore, direct-mapped caches are simpler and easier to design, and they require less area. Overall, direct-mapped caches are often the most economical choice for use in workstations, where cost-performance is the most important criterion.

1.1 The Problem

Unfortunately, the large number of interference misses that occur in direct-mapped caches are still a major problem. An *interference miss* (also known as a *conflict miss*) occurs when two addresses map into the same cache set in a direct-mapped cache, as shown

[1] In most caches, more than one data word can reside in a data block. In this case, an *offset* is the third and lowest-order field in the address, and it is used to select the appropriate data word.

in Figure 1. Consider referencing a cache with two addresses, a_i and a_j, that differ only in some of the higher-order bits (which often occurs in multiprogramming environments). In this case, the addresses will have different tags but identical index fields; therefore, they will reference the same set. If we denote the set that is selected by choosing the low-order bits of an address a as $b[a]$, then we have $b[a_i] = b[a_j]$ for conflicting addresses. The name b comes from the bit-selection operation performed on the bits to obtain the index.

Assume the following reference pattern: $a_i\ a_j\ a_i\ a_j\ a_i\ a_j\ \cdots$. A set-associative cache will not suffer a miss if the program issues the above sequence of references because the data referenced by a_i and a_j can co-reside in a set. In a direct-mapped cache, however, the reference to a_j will result in an interference miss because the data from a_i occupies the selected cache block. The percentage of misses that are due to conflicts varies widely among different applications, but it is often a substantial portion of the overall miss rate.

We believe these interference misses can be largely eliminated by implementing control logic which makes better use of cache area. The challenge, then, is determining a simple, area-efficient cache control algorithm to reduce the number of interference misses and to boost the performance without increasing the degree of associativity.

1.2 Contributions of This Paper

This paper presents the design of a column-associative cache that resolves conflicts by allowing alternate hashing functions, which results in significantly better use of cache area. Using trace-driven simulation, we demonstrate that its miss rate is much better than that of Jouppi's victim cache [4] and the hash-rehash cache of Agarwal, Horowitz, and Hennessy [5], and virtually the same as that of a two-way set-associative cache. Furthermore, its hit access time is the same as that of a direct-mapped cache. To help explain the behavior of the column-associative cache, we also develop and validate an analytical model for this cache.

The rest of this paper is organized as follows. The next section discusses other efforts with similar goals. Section 3 presents the column-associative cache, and Section 4 develops an analytical model for this cache. Section 5 presents the results of trace-driven simulations comparing the performance of several cache designs, and Section 6 concludes the paper.

2 Previous Work

Several schemes have been proposed for reducing the number of interference misses. A general approach to improving direct-mapped cache access is Jouppi's victim cache [4]. A victim cache is a small, fully-associative cache that provides some extra cache lines for data removed from the direct-mapped cache due to misses. Thus, for a reference stream of conflicting addresses, such as $a_i\ a_j\ a_i\ a_j\ \ldots$, the second reference, a_j, will miss and force the data indexed by a_i out of the set. The data that is forced out is placed in the victim cache. Consequently, the third reference, a_i, will not require accessing main memory because the data can be found in the victim cache.

However, this scheme requires a sizable victim cache for adequate performance because it must store *all* conflicting data blocks. Like the column-associative cache, it requires two or more access times to fetch a conflicting datum. (One cycle is needed to check the primary cache, the second to check the victim cache, and a possible third to store the datum into the primary cache.) Because of its fixed size relative to the primary direct-mapped cache, both our results and those presented by Jouppi (see Figure 3-6 in [4]) show that it is not very effective at resolving conflicts for large primary caches. On the other hand, because the area available to resolve conflicts in the column-associative cache increases with primary cache size, it resolves virtually all conflicts in large caches.

The scheme in [6] is proposed for instruction caches and uses two instruction buffers (of size equal to a cache line) between the instruction cache and the instruction register, and an instruction encoding that makes it easy to detect the presence of branch instructions in the buffers.

Kessler et al. [7] propose inexpensive implementations of set-associative caches by placing the multiple blocks in a set in sequential locations of cache memory. Tag checks, done serially, avoid the wide datapath requirements of conventional set-associative caches. The principle focus of this study was a reduction in implementation cost. The performance (measured in terms of average access time) of this scheme could often be worse than a direct-mapped cache for long strings of consecutive addresses, which occur commonly. For example, a long sequential reference stream of length equal to the cache size would fit into a direct-mapped cache, and subsequent references to any of these locations would result in a first-time hit. However, in a d-way set-associative implementation of this scheme, only $1/d$ of the references would succeed in the first access.

A similar problem exists in the MRU scheme proposed by So et al. [8]. The MRU scheme is a means for speeding up set-associative cache accesses. It maintains a few bits with each cache set indicating the most recently used block in the set. An access to a given set immediately reads out its MRU block, betting on the likelihood that it is the desired block. If it isn't, then an associative search accompanies a second access. Clearly, a two-way set-associative cache does not require an associate search, but does require a second access. Unfortunately, only $1/d$ of the references in a long sequential address stream would result in first-time hits into a d-way set-associative cache using this scheme.

A more desirable cache design would reduce the interference miss rate to the same extent as a set-associative cache, but at the same time, it would maintain the critical hit access path of the direct-mapped cache. The hash-rehash cache [5] had similar goals, but in Section 3.1 we demonstrate that it has one serious drawback. The technique introduced in Section 3 removes this drawback and largely eliminates interference misses by implementing slightly more complex control logic to make better use of the cache area. By maintaining direct-mapped cache access, these schemes do not affect the critical hit access time. With proper design, the few additional cycles required to execute the algorithms in case of a miss are balanced by the decrease in the miss rate due to fewer conflicts. This decrease in the interference miss rate is achieved not by set associativity but by exploiting temporal locality to make more efficient use of the given cache area—a notion called *column associativity*.

3 Column-Associative Caches

The fundamental idea behind a column-associative cache is to resolve conflicts by dynamically choosing different locations (accessed by different hashing functions) in which conflicting data

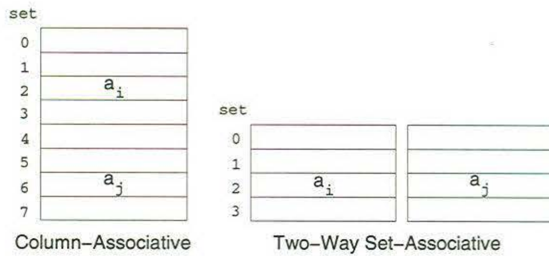

Figure 2: Comparison of column-associative and two-way set-associative caches of equal size. The conflict $b[a_i] = b[a_j]$ is resolved by both schemes.

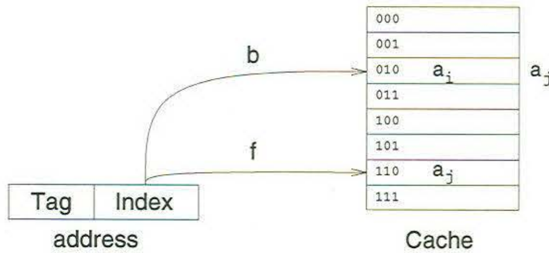

Figure 3: Indexing into a cache by bit selection and by bit flipping. The conflict $b[a_i] = b[a_j]$ is resolved by the bit-flipping rehash.

can reside. Figure 2 compares the column-associative cache with a two-way set-associative cache of equal size. When presented with conflicting addresses ($b[a_i] = b[a_j]$), the set-associative cache resolves the conflict statically by referencing another location within the same set. On the other hand, the column-associative cache is direct-mapped, and when presented with conflicting addresses, a different hashing function is dynamically applied in order to place or locate the data in a *different* set. One simple choice for this other hashing function is bit selection with the highest-order bit inverted, which we term *bit flipping*. If $b[a] = 010$, then $f[a] = 110$, as illustrated in Figure 3. Therefore, conflicts are resolved not within a set but within the entire cache, which can be thought of as a column of sets—thus the name *column associativity*.

Column associativity can obviously improve upon direct-mapped caching by resolving a large number of the conflicts encountered in an address stream. In addition, as long as the control logic used to implement column associativity is simple and fast, then the benefits of direct-mapped caches over set-associative caches (as discussed in Section 1) are maintained, especially the lower hit access time. Because hits are much more frequent than misses, the extra cycles required to implement the column-associative algorithm on a miss can be easily balanced by the small improvement in hit access time on every hit, resulting in a smaller *average* memory access time when compared to a two-way set-associative cache. Of course, column associativity could be extended to emulate degrees of associativity higher than two, but it is likely that the complexity of implementing such an algorithm would add little to the performance and might even degrade it.

Additionally, the column-associative implementation uses sets within the cache itself to store conflicting data; only a simple rehash of the address is required to access this data. By comparison, a victim-cache implementation requires an entirely separate, fully-associative cache to store the conflicting data. Not only does the victim cache consume extra area, but it can also be quite slow due

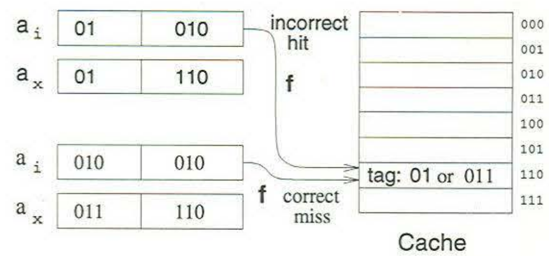

Figure 4: Appending the high-order bit of the index to the tag. This technique is necessary when bit flipping is implemented.

to the need for an associative search and for the logic to maintain a least-recently-used replacement policy. Of course, storing conflicting data within the cache—instead of in a separate victim cache—very likely results in the loss of useful data, but this effect (henceforth referred to as *clobbering*) can be minimized as discussed in Section 3.2.

The remainder of our discussion proceeds in two steps. First, we describe a basic system that uses multiple hashing functions and discuss its drawbacks. Then, we add rehash bits to this design to alleviate its problems.

3.1 Multiple Hashing Functions

Like the hash-rehash cache in [5], column-associative caches use two (or possibly more) distinct hashing functions, h_1 and h_2, to access the cache, where $h_1[a]$ denotes the index obtained by applying hashing function h_1 to the address a. If $h_1[a_i]$ indexes to valid data, a *first-time* hit occurs; if it misses, $h_2[a_i]$ is then used to access the cache. If a *second-time* hit occurs, the data is retrieved. The data in the two cache lines are then swapped so that the next access will likely result in a first-time hit. However, if the second access also misses, then the data is retrieved from main memory, placed in the cache line indexed by $h_2[a_i]$, and swapped with the data in the first location.

Using two or more hashing functions mimics set associativity, because for conflicting addresses (that is, a_i and a_j for which $h_1[a_i] = h_1[a_j]$), rehashing a_j with h_2 resolves the conflict with a high probability (that is, $h_1[a_i] \neq h_2[a_j]$). However, notice that the hit access time of a first-time hit remains unchanged. For simplicity and for speed, the first-time access is performed with bit selection (that is, $h_1 = b$), and bit flipping is often used for h_2 (that is, $h_2 = f$).

The use of bit flipping as a second hashing function results in a potential problem. Consider two addresses, a_i and a_x, which differ only in the high-order bit of the index field (that is, $f[a_i] = b[a_x]$). These two addresses are distinct; however, the tag fields are identical, thus a rehash access with $f[a_i]$ results in a hit with a data block that should only be accessed by $b[a_x]$. This is unacceptable, because a data block must have a one-to-one correspondence with a unique address. For addresses whose indexes are the same and which thus reference the same set, the tags are compared in order to determine whether an address should access the data block. This suggests a simple solution to the situation, appending the high-order bit of the index field to the tag, as illustrated in Figure 4. The rehash with $f[a_i]$ will correctly fail because the data block is once again referenced by a unique address, a_x. This scheme is assumed to be in place whenever bit flipping is used.

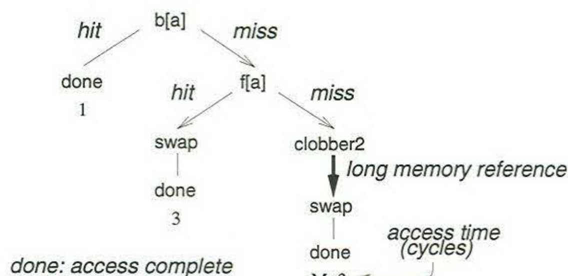

Figure 5: Decision tree for the hash-rehash algorithm.

mnemonic	*action*	*cycles*
b[a]	bit-selection access	1
f[a]	bit-flipping access	1
swap	swap data in sets accessed by $b[a]$ and $f[a]$	2
clobber2	get data from memory, place in set $f[a]$	M
clobber1	get data from memory, place in set $b[a]$	M
Rbit=1?	check if set $b[a]$ is a rehashed location	0

Table 1: Decision tree mnemonics and cycle times for each action.

To illustrate the operations more clearly, the hash-rehash algorithm has been expressed as the *decision tree* in Figure 5, simply a translation of the verbal description of the hash-rehash algorithm into a tree structure. Table 1 explains the mnemonics used in this decision tree and in the others which are introduced in this paper. The table also includes the number of cycles required to complete an action, which is necessary for the calculation of average access time.

In the decision tree, note that after a first-time miss and a second-time hit, which require two cycles to complete, a swap is performed. According to Table 1, the swap requires an additional two cycles to complete. The design requirements for accomplishing a swap in two cycles is discussed in Section A of the appendix. However, given an extra buffer for the cache, this swap need not involve the processor, which may be able to do other useful work while waiting for the cache to become available again. If this is the case half of the time, then the time wasted by a swap is one cycle. Therefore, for all decision trees in this paper, we assume that a swap adds only one cycle to the execution time. (However, we provide access time results for both one and two cycle swaps.) Thus, the three cycles indicated in the swap branch of Figure 5 results from one cycle for the initial cache access, one cycle for the rehash access, and one cycle wasted during the swap.

Unfortunately, the hash-rehash cache has a serious drawback, which often reduces its performance to that of a direct-mapped cache, as can be seen in Section 5.3. The source of its problems is that a rehash is attempted after *every* first-time miss, which can replace potentially useful data in the rehashed location, even when the primary location had an inactive block. Consider the following reference pattern: $a_i\ a_j\ a_x\ a_j\ a_x\ a_j\ a_x\ \cdots$, where the addresses a_i and a_j map into the same cache location with bit selection, and a_x is an address which maps into the same location with bit flipping (that is, where $b[a_i] = b[a_j]$, and $f[a_i] = b[a_x]$). This situation is illustrated in Figure 6. After the first two references, both the hash-rehash and the column-associative algorithms will have the data referenced by a_j (which will be called j for brevity) and the data i

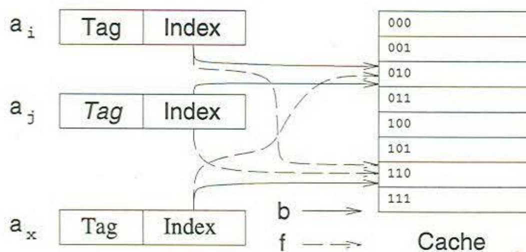

Figure 6: The potential for secondary thrashing in a reference stream of the form $a_i\ a_j\ a_x\ a_j\ a_x\ a_j\ a_x\ \cdots$. Different fonts are used to indicate different index fields and tags. In this case, $b[a_i] = b[a_j]$ and $f[a_i] = b[a_x]$.

in the non-rehashed and rehashed locations, respectively. When the next address, a_x, is encountered, both algorithms attempt to access the set $b[a_x]$, which contains the rehashed data i. But when this first-time miss occurs, the hash-rehash algorithm next tries to access $f[a_x]$, which results in a second-time miss and the clobbering of the data j. This pattern continues as long as a_j and a_x alternate; the data referenced by one of them is clobbered as the inactive data block i is swapped back and forth but never replaced. We will refer to this negative effect as *secondary thrashing* in the future.

The following section describes how the use of a rehash bit can lessen the effects of these limitations.

3.2 Rehash Bits

The key to implementing column associativity effectively is inhibiting a rehash access if the location reached by the first-time access itself contains a rehashed data block. This idea can be implemented as follows. Every cache set contains an extra bit which indicates whether the set is a *rehashed location*, that is, whether the data in this set is indexed by $f[a]$. This algorithm, which is illustrated as a decision tree in Figure 7, is similar to that of the hash-rehash cache; however, the key difference lies in the fact that when a cache set must be replaced, a rehashed location is always chosen—immediately if possible. Thus, if the first-time access is a miss, then the rehashed-location bit (or *rehash bit* for short) of that set is checked (Rbit=1?, as listed in Table 1). If it has been set to one, then no rehash access will be attempted, and the data retrieved from memory is placed in that location. Then the rehash bit is reset to zero to indicate that the data in this set is to be indexed by $b[a]$ in the future. On the other hand, if the rehash bit is already a zero, then upon a first-time miss the rehash access will continue as described in Section 3.1. Note that if a second-time miss occurs, then the set whose data will be replaced is again a rehashed location, as desired.

Of course, at start-up (or after a cache flush), all of the empty cache locations should have their rehash bits set to one. The reason that this algorithm can correctly replace a location with a set rehash bit immediately after a first-time miss is based on the fact that bit flipping is used as the second hashing function. Given two addresses a_i and a_x, if $f[a_i] = b[a_x]$, then it must be true that $f[a_x] = b[a_i]$. Therefore, if a_i accesses a location using $b[a_i]$ whose rehash bit is set to one, then there are only two possibilities.

1. The accessed location is an empty location from start-up, or
2. there exists a non-rehashed location at $f[a_i]$ (or $b[a_x]$) which previously encountered a conflict and placed the data in *its*

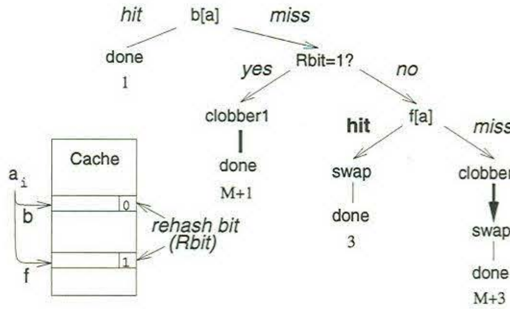

Figure 7: Decision tree for a column-associative cache.

rehashed location, $f[a_x]$.

In both cases, it makes sense to replace the location reached during the first-time access that had its rehash bit set to one.

However, it must be proven that a third possibility does not exist, namely, the location $b[a_i]$ has its rehash bit set to one, but the data referenced by a_i actually resides in $f[a_i]$ simultaneously. Consider the actions taken by the algorithm when one of the conditions precedes the other. First, if $b[a_i]$ is a rehashed location, then any first-time miss results in the immediate clobbering of that location and the resetting of the rehash bit to zero. Therefore, it is not possible for the placement of the data into $f[a_i]$ to follow this condition.

On the other hand, if the data referenced by a_i already resides in $f[a_i]$ due to a conflict, then the rehash bit of $b[a_i]$ must be a zero, because it contains the most recently accessed data. The only way to change this bit is if $b[a_i]$ were to be used as a rehashed location in order to resolve a different conflict. However, because bit flipping is the rehashing function, the only location for which this situation can occur is $f[a_i]$ itself. A first-time access to this location, though, would automatically clobber the rehashed data. Therefore, it is clear that the two conditions for this third possibility can never occur simultaneously. This important property could not be utilized in the column-associative algorithm if bit flipping was not the second hashing function or if more than two hashing functions were included.

Like the hash-rehash cache, the column-associative algorithm attempts to exploit temporal locality by swapping the most recently accessed data into the non-rehashed location, if a rehash is indeed attempted. The use of the rehash bit helps utilize cache area more efficiently because it immediately indicates whether a location is rehashed and should be replaced in preference over a non-rehashed location.

In addition to limiting rehash accesses and clobbering, the rehash bits in the column-associative cache eliminate secondary thrashing. Referring to the reference stream, $a_i\ a_j\ a_x\ a_j\ a_x\ a_j\ a_x\ \cdots$, in Figure 6, the third reference accesses $b[a_x]$, but it finds the rehash bit set to one. Thus, the data i is replaced immediately by x, the desired action. Of course, this column-associative cache suffers thrashing if three or more conflicting addresses alternate, as in $a_i\ a_j\ a_x\ a_i\ a_j\ a_x\ a_i\ \cdots$, but this case is much less probable than two alternating addresses.

4 A Simple Analytical Model for Column-Associative Caches

We have developed a simple analytical model for the column-associative cache that predicts the percentage of interference misses removed from a direct-mapped cache using only one measured parameter—the size of the program's working set—from an address trace. Our model builds on the self-interference component of the direct-mapped cache model of Agarwal, Horowitz, and Hennessy [9], and it estimates the percentage of interference misses removed by computing the percentage of cache block conflicts removed by the rehash algorithm. Because the behavior is captured in a simple, closed-form expression, our model yields valuable insights into the behavior of the column-associative cache. Validations against empirically derived cache miss rates suggest that the model's predictions are fairly accurate as well.

Like the self-interference model in [9], the percentage reduction in cache block conflicts in the column-associative cache is captured by two parameters: S and u. The parameter S represents the number of cache sets; in direct-mapped caches, the product of S and the block size yields the cache size. The parameter u denotes the working-set size of the program, and must be measured from an address trace of a program. The *working set* of a program is the set of distinct blocks a program accesses within some interval of time.

The model makes the assumption that blocks have a uniform probability of mapping to any cache set, and that the mappings for different blocks are independent of each other. The same assumption is also made for the rehash accesses. This assumption is commonly made in cache modeling studies [10, 11, 9]. Although this assumption makes the models generally overestimate miss rates, its effect is less severe when we are interested in the *ratios* of the number of conflicting blocks in direct-mapped caches and column-associative caches.

A detailed derivation of the model appears in Section B in the appendix, and this section summarizes the major results. Let c_d denote the number of conflicting blocks in a direct-mapped cache, and c_{cac} the corresponding number of conflicting blocks in a column-associative cache. Blocks are said to conflict when multiple blocks from the working set of a program map to a given cache set. In a column-associative cache, conflicting blocks are blocks that conflict even after a rehash is attempted. Section 5.1 provides further discussion on the notion of conflicts.

Section B in the appendix derives expressions for the number of conflicting blocks in direct-mapped and column-associative caches in terms of $P(d)$, which is the probability that d program blocks (out of a total of u) map to a given cache set. Because blocks are assumed to map with equal likelihood to any cache set, the distribution of the number of blocks in a cache set is binomial, which yields

$$P(d) = \binom{u}{d} \left(\frac{1}{S}\right)^d \left(1 - \frac{1}{S}\right)^{u-d} \tag{1}$$

The following are expressions for the number of conflicting blocks.

$$c_d = u - SP(1)$$

$$c_{cac} = u - SP(1) - SP(2)(1 + P(0) - P(1) - P(2))$$

We estimate the percentage of interference misses removed by the percentage reduction in the number of conflicting blocks. Our validation experiments indicate that this is a good approximation. Thus, the percentage of interference misses removed,

$$\frac{c_d - c_{cac}}{c_d} = \frac{SP(2)\left(1 + P(0) - P(1) - P(2)\right)}{u - SP(1)} \qquad (2)$$

It is instructive to take the first-order approximations of the expression in Equation 2 after substituting for $P(d)$ from Equation 1 and simplifying the resulting expression. The first-order approximation is valid when $S >> u$ and $u >> 1$, which allow us to use $\left(1 - 1/S\right)^{u-1} \approx \left(1 - u/S\right)$. Proceeding along these lines, we obtain

$$\frac{c_d - c_{cac}}{c_d} \approx \left(1 - \frac{2u}{S}\right) \qquad (3)$$

It is easy to see from the above equation that the percentage of conflicts removed by rehashing will approach unity as the cache size is increased. Similarly, roughly 50% of the conflicts are removed when the cache is four times larger than the working set of the program.

To demonstrate the accuracy of the model, we plot in Figure 13 the measured values of the average percentages of interference misses removed and the values obtained using Equation 2 for our traces. The predictions for each of the individual traces is also fairly accurate, as displayed in Figures 8 and 9. Both the model and the simulations use a block size of 16 bytes. The analytical model uses only one parameter—the working-set size, u—measured from each trace. Table 3 shows the working set sizes for each of our traces.

5 Results

This section presents the data obtained through simulation of the various caches and an analysis of these results. First, the metrics which have been used to evaluate the performance of the caches must be described.

5.1 Cache Performance Metrics

We use three cache performance metrics in our results: the cache miss rate, the percentage of interference misses removed, and the average memory access time.

The *miss rate* is the ratio of the number of misses to the total number of references.

The *percentage of interference misses removed* is the percentage by which the number of interference misses in the cache under consideration is reduced over those in a direct-mapped cache. An interference miss is defined as a miss that results when a block that was previously displaced from the cache is subsequently referenced. In a single processor environment, the total number of misses minus the misses due to first-time references is the number of interference misses.[2]

[2] A similar parameter was used by Jouppi [4] as a useful measure of the performance of victim caches. We note that our interference metric measures the sum of the intrinsic interference misses and the extrinsic interference misses in the classification of Agarwal, Horowitz, and Hennessy [9], and the sum of the capacity, conflict, and context-switching misses in the terminology of Hill and Smith [12].

This metric is particularly useful for determining the success of a particular scheme because all cache implementations must share the same compulsory or first-time miss rate for a given reference stream, but they may have different interference miss rates. The percentage of interference misses removed is calculated by the equation

$$\frac{direct\ miss\ rate - miss\ rate}{direct\ miss\ rate - compulsory\ miss\ rate} \times 100\%$$

where, for a given address trace and cache size, the miss rate is that of the particular cache design, and the direct miss rate is that of a direct-mapped cache of equal size. The compulsory miss rate is the ratio of unique references to total references for that trace.

Finally, the *average memory access time* is defined as the average number of cycles required to complete one reference in a particular address stream. This metric is useful in assessing the performance of a specific caching scheme because although a particular cache design may demonstrate a lower miss rate than a direct-mapped cache, it may do so at the expense of the hit access time. As mentioned earlier, our graphs include access time results for both one-cycle and two-cycle swaps.

Let the cache access time for a hit be one cycle, and let M represent the number of cycles required to service a miss from the main memory (in our simulations, $M = 20$). If R is the total number of references in the trace, H_1 is the total number of hits on a first-time access, and H_2 is the total number of hits on a second-time access, then the average memory access time for the various schemes can be computed from the decision trees of Section 3 as shown below.

For direct-mapped caches, the access time is one for hits, and one plus M for misses. Thus,

$$t_{ave} = \frac{1}{R}\left[H_1 + (M+1)(R - H_1)\right]$$

For hash-rehash caches, the access time is one for first-time hits, 3 for rehash hits (Every first-time miss is followed by a rehash.), and $(M+3)$ otherwise.

$$t_{ave} = \frac{1}{R}\left[H_1 + 3H_2 + (M+3)(R - H_1 - H_2)\right]$$

For column-associative caches, we need an additional parameter R_2, which is the total number of second time accesses. (Recall that second-time accesses are attempted only when the rehash bit is zero.) Thus the access time is one for first-time hits, and three for the H_2 hits during a rehash attempt. If a rehash is not attempted, then $(M+1)$ cycles are spent. Rehash attempts that miss suffer a penalty of $(M+3)$ cycles. Therefore,[3]

$$t_{ave} = \frac{1}{R}\left[H_1 + 3H_2 + (M+1)(R - H_1 - R_2) + (M+3)(R_2 - H_2)\right]$$

The simulator described in the next section measures R, R_2, H_1, and H_2 for each of the cache types, and it derives average memory access times from the above equations.

[3] The cycles per instruction (or CPI) assuming single-cycle instruction execution can be calculated easily from the average access time. For a unified instruction and data cache with a single cycle access time, the CPI with a 100% hit rate is $(1 + l)$, where l is the fraction of instructions that are loads or stores. In the presence of cache misses, however, the average access time becomes t_{ave}, and the CPI becomes $(1 + l)t_{ave}$.

name	trace description
LISP0	LISP runs of BOYER (a theorem prover)
DEC0.1	Behavioral simulator of cache hardware, DECSIM
SPIC0	SPICE simulating a 2-input tristate NAND buffer
IVEX0	Interconnect verify, a DEC program checking net lists in a VLSI chip
FORL0	FORTRAN compile of LINPACK

Table 2: Description of uniprocessor traces used during simulation.

trace	no. of references u	unique	total	compulsory miss rate (%)
LISP0	392	1,789	262,760	0.6808
DEC0.1	463	2,418	334,775	0.7223
SPIC0	740	2,834	358,168	0.7912
IVEX0	774	11,087	307,172	3.6097
FORL0	826	6,787	314,110	2.1607
MUL6.0		5,267	400,698	1.3145

Table 3: Number of references (both instructions and data) and compulsory miss rate for each of the address traces simulated. The block size for measuring u and $unique$ is set to 16 bytes (four words).

5.2 Simulator and Trace Descriptions

We wrote trace-driven simulators for direct-mapped, set-associative, victim, hash-rehash, and column-associative caches. Multiprogrammed simulations assume that a process identifier is associated with each reference to distinguish between the data of different processes. All caches are assumed to be combined instruction and data caches.

The traces used in this study come from the ATUM experiment of Sites and Agarwal [13]. The ATUM traces comprise realistic workloads and include both operating system and multiprogramming activity. The five uniprocessor traces, derived from large programs running under VMS, are described in Table 2. We also use a multiprogramming trace called MUL6.0, which includes activity from six processes including a FORTRAN compile, a directory search, and a microcode address allocator. Each trace length is on the order of a half million references. We believe these lengths are adequate for our purposes, since we explicitly subtract the number of first-time misses and present the percentage of interference misses removed, and because it is possible to differentiate the performance of the various caching methods without resorting to measurement methods that yield cache miss rates with a degree of accuracy exceeding the first or second decimal place. The compulsory miss rates and other parameters for these traces are listed in Table 3. In the table, u is the average number of unique blocks in 10,000 reference windows, while *unique* is the total number of unique blocks in the entire trace.

5.3 Measurements and Analysis

In this section, the results of the trace-driven simulations are plotted and interpreted. Before introducing the plots, a few of their features must be explained. If the miss rate of a cache happens to be worse than that of a direct-mapped cache for the particular cache size, as is occasionally the case for a hash-rehash cache, then the percentage of interference misses removed becomes a negative quantity. On the graph, this is instead indicated by a point at zero percent.[4]

The victim cache size has been set to 16 entries. This is based on simulation data which suggests that the removal of conflicts quickly saturates beyond this size. In addition, remember that each victim-cache entry is a complete cache line, storing the tag, status bits, and the data block, which contains four words in these simulations.

5.3.1 Miss Rates and Interference Misses Removed

LISP0 and DEC0.1 The results for the LISP0 and DEC0.1 traces are very similar, so only LISP0 results are plotted in Figure 8. It is evident that all of the cache designs exhibit much lower miss rates than the direct-mapped cache. The lowest miss rates are achieved by the two-way set-associative and the column-associative caches. The victim and hash-rehash caches have higher miss rates.

A striking feature of the miss rate plots is the relationship between the direct-mapped and hash-rehash caches. Whenever doubling the cache size results in a sharp decrease in the direct-mapped miss rate, the same change in cache size yields a sharp and similarly sized *increase* of the hash-rehash miss rate. This effect makes sense intuitively—a hash-rehash cache is designed to resolve conflicts through the use of alternate cache locations. It is successful as long as the the number of conflicts decreases only slightly as the cache size increases. However, if an increase in cache size itself suddenly removes a large portion of the conflicts, then the hash-rehash algorithm clobbers many locations and suffers a sharp drop in the second-time hit rate because it is attempting to resolve conflicts which no longer exist.[5] Notice that the column-associative cache does not suffer from this degradation because its access algorithm is designed specifically to alleviate the problems of clobbering and low second-time hit rates.

Referring to the percentages of interference misses removed in Figure 8, notice that the dashed curve corresponding to the predictions of the model is very close to the curve obtained from simulations. The LISP0 trace has a small working set compared to the other traces (see Table 3), and therefore the percentage of interference misses removed quickly approaches 100% for all but the victim cache, which is a phenomenon readily explained by the approximate analytical expression for this metric: $(1 - 2u/S)$.

SPIC0, IVEX0, and FORL0 The results for these traces are also similar enough to be grouped together. The data for the SPIC0 trace has been plotted in Figure 9. Nearly all of the results from the previous section apply to the simulations with these three traces, but there are several important differences. Because the working-set sizes of these traces are larger than the LISP0 trace, the percentages of interference misses removed by column associativity start at much lower values and approach 100% more slowly. Because the victim cache is much more sensitive to working set size, it does not attain the same percentages found for LISP0 and DEC0.1; for these traces, the victim cache lies around 25% or less for this metric. Recall that the victim cache size remains constant, while the column-associative and the set-associative caches can devote larger areas to resolve conflicts as cache size increases.

[4] This is why the points for the hash-rehash cache are not connected in the graphs showing percentage of interference misses removed.

[5] The addition of one, high-order bit to the index could separate two groups of addresses which conflict often because they differ for the first time in that bit.

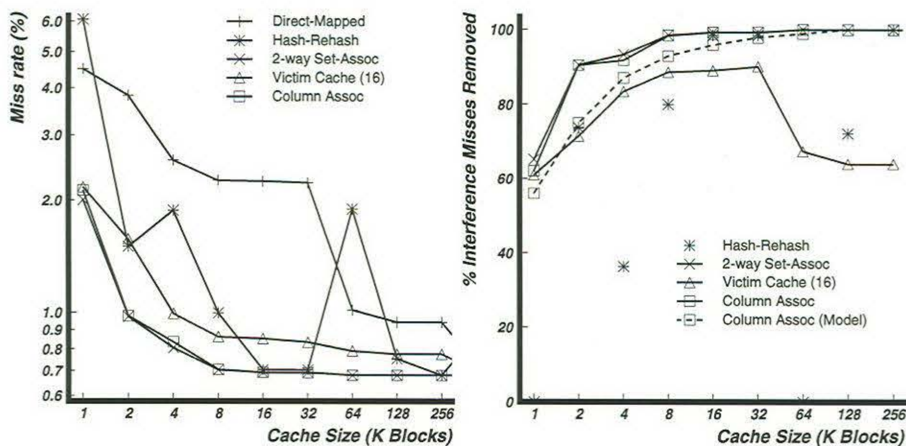

Figure 8: Miss rates and percentages of interference misses removed versus cache size for LISP0. Block size is 16 bytes.

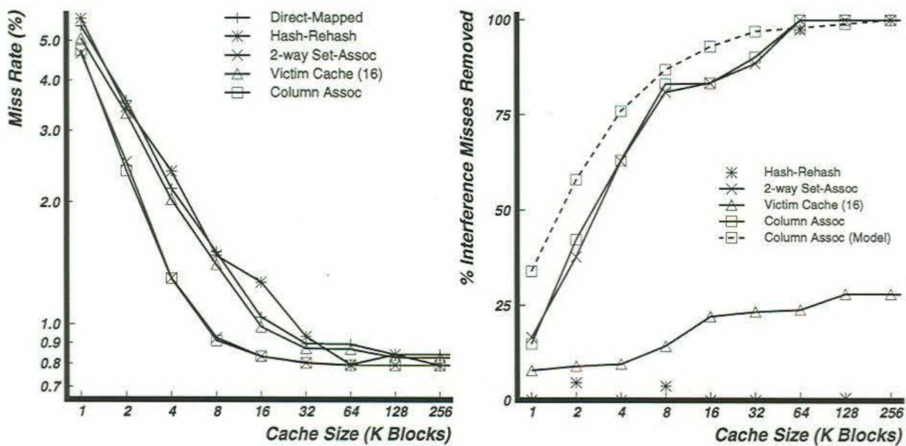

Figure 9: Miss rates and percentages of interference misses removed versus cache size for SPIC0. Block size is 16 bytes.

The plots for SPIC0 in Figure 9 reveal another interesting fact: the column-associative cache outperforms the two-way set-associative cache for some of the cache sizes. A hypothesis that explains this behavior is based on the fact that when comparing the two caches at an equal cache size, the set-associative cache has only half that number of cache sets. As seen before, doubling the cache size and thus adding a high-order bit to the index may eliminate a large number of conflicts that have been occurring because many addresses differ for the first time in that bit. For example, consider the addresses 0001111, 0101111, and 1011111. All three result in multiple conflicts (thrashing) if only the four, low-order bits are used as the index. This is a cache size of 2^4 or 16 for the column-associative cache, but the total cache size is 32 for the two-way set-associative cache, and it still exhibits thrashing. Note that both caches have 16 sets. A 32-set column-associative cache, however, uses five bits for the index. In this case, the conflicts between 0101111 and 1011111 are automatically eliminated because of the different fifth bits.

MUL6.0 The miss rates and percentages of interference misses removed for the multiprogramming trace are plotted in Figure 10. Once again, many of the observations made for the other trace results apply to MUL6.0. Perhaps the most telling result is the relatively poor performance of the victim cache. Its miss rate is virtually the same as that of the direct-mapped cache (for cache sizes greater than 2K blocks). The large working sets of multiprogramming workloads make the fixed size of the victim cache a serious liability. The larger available area for storing conflicts in the column-associative cache is clearly a big win in this situation.

5.3.2 Average Memory Access Times

Two key factors must be considered when interpreting the access time data. First, although the average memory access times of set-associative caches are in reality increased due to their higher hit access times, the graphs in this paper assume their hit access times are the same as that of direct-mapped caches. If realistic access times of two-way set-associative caches are considered, their average memory access times might well become greater that those of column associative caches. (This is why the corresponding curves are labeled "Ideal").

Second, the average memory access time is very sensitive to the time required to service a miss (M). The results assume $M = 20$ cycles. For larger (and still reasonable) miss penalties, the designs such as column-associative caches which reduce the number of accesses to main memory ($R - H_1 - H_2$) will look even more impressive than indicated by our results.

The results for the LISP0 and SPIC0 traces are presented together in Figure 11. As before, DEC0.1 is similar to LISP0, while IVEX0 and FORL0 are similar to SPIC0. All the average memory access time plots are largely similar in shape to the miss rate plots, which is expected, because t_{ave} is a linear function of the miss rate.

The graph for LISP0 shows that column associativity achieves much lower average memory access times than a direct-mapped cache. The improvement is about 0.3 cycles for most cache sizes. For SPIC0, the column-associative cache exceeds 0.2 cycle improvements only for small caches. This fact is confirmed when the miss rate plot is considered—the direct-mapped interference miss rate is not much higher than the compulsory miss rate, unlike the case for LISP0. The results for MUL6.0 are largely similar: the column-associative cache saves about 0.2 cycles over direct-mapped caches, and the two-way set associative cache saves a further 0.1 cycle, when the caches are less than 4K blocks. (With a 16-byte block, the cache size is 64K bytes.) The savings are smaller for larger caches.

Perhaps most important, however, is the fact that the column-associative cache achieves an average access time close to the two-way set-associative cache, even though the hit access time of the set-associative cache was (unrealistically) kept the same as that of a direct-mapped cache.

5.4 Summary

This section presents data for each of the metrics averaged over all of the traces. The resulting plots serve as excellent examples for reviewing the major points made in this section.

When the miss rates of all six traces are averaged for each cache size, the plot in Figure 12 is the result. The direct-mapped miss rate is the baseline for comparison and falls quickly from 6.0% to 2.0%, before settling toward the average compulsory miss rate of about 1.5%.

The other cache designs can be split into two groups, based not only on their similar miss rate curves but also on the relationships among their access algorithms. The first group contains the hash-rehash cache and the victim cache, which have similar control algorithms. The hash-rehash cache is usually an improvement upon direct-mapped caching; the miss rate drops more quickly from 6.0% to about 1.7%. However, at the transition point, the hash-rehash miss rate increases about as much as the direct-mapped miss rate decreases. This is due to the fact that once the cache size exceeds the working-set size, the interference miss rate drops markedly. The many rehash accesses performed by the hash-rehash algorithm now are more likely to clobber live data than to resolve conflicts. The victim cache does not suffer from this effect, because it is designed to alleviate the main problems with the hash-rehash algorithm: clobbering and low second-time hit rates.

The second group consists of the two-way set-associative and the column-associative caches. The miss rates of these caches are almost 2.0% lower than direct-mapped miss rates for small caches, just under 1.0% near the transition, and right at the compulsory miss rate for large caches. As predicted in Section 3, the column-associative cache achieves two-way set-associative miss rates.

The plot in Figure 13 shows the average percentages of interference misses removed. (This average does not include the MUL6.0 numbers, so that we could compare the simulation averages with the model.) The curves for set-associative and column-associative caches are almost identical, starting at about 40% and climbing to 100% when the cache size reaches 256 K blocks. As predicted in Section 2, the performance of the victim cache relative to the column-associative cache degrades with cache size. Finally, the dashed curve for the model is seen to be surprisingly close to simulation results when the individual trace anomalies are averaged out.

The average memory access time (t_{ave}) data for the six traces have been averaged and plotted in Figure 14. Based on this average plot and on most of the other data, the column-associative cache appears to be good choice under most operating conditions. In this example, t_{ave} is reduced by over 0.2 cycles for small to moderate caches, and by about 0.1 cycles for moderate to large caches.

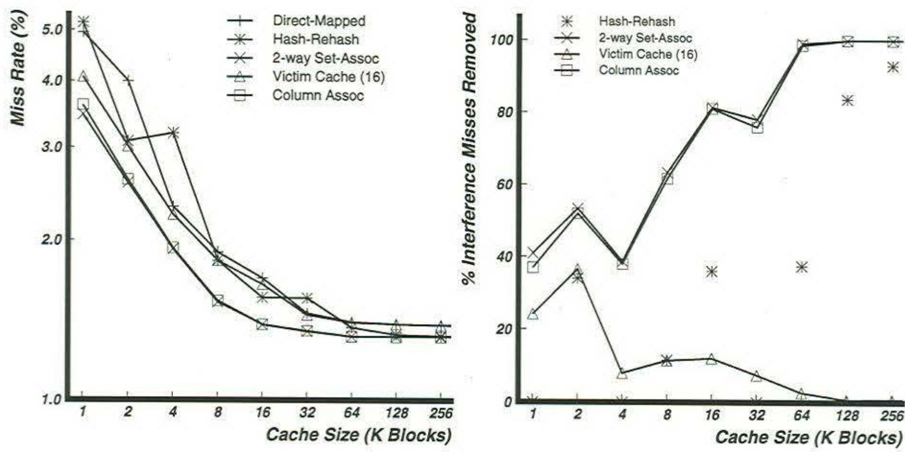

Figure 10: Miss rates and percentages of interference misses removed versus cache size for MUL6.0. Block size is 16 bytes.

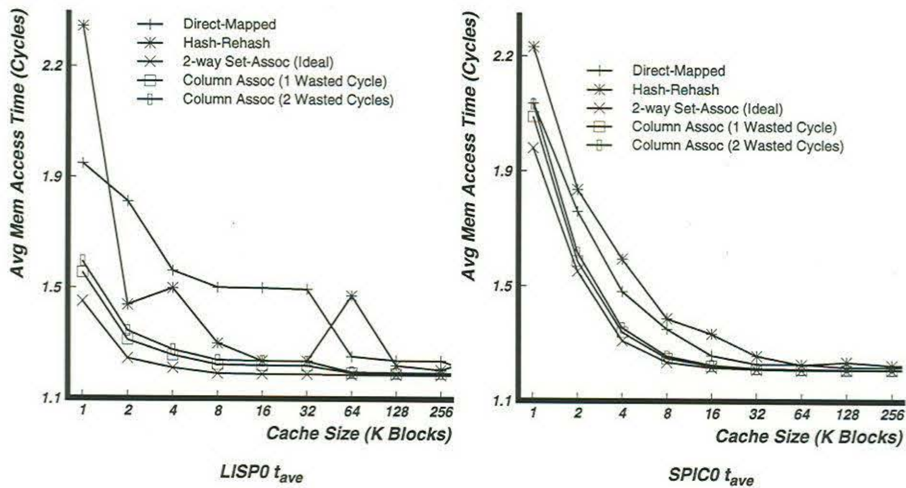

Figure 11: Average memory access times (in cycles) versus cache size for LISP0 and SPIC0. Block size is 16 bytes. The hit access time of two-way set-associative caches is assumed to be the same as that of a direct-mapped cache.

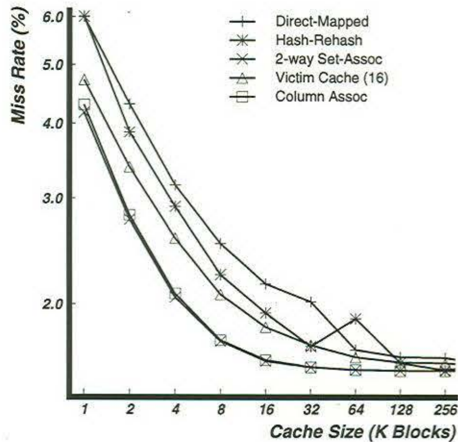

Figure 12: Miss rates versus cache size, averaged over all six traces. Block size is 16 bytes.

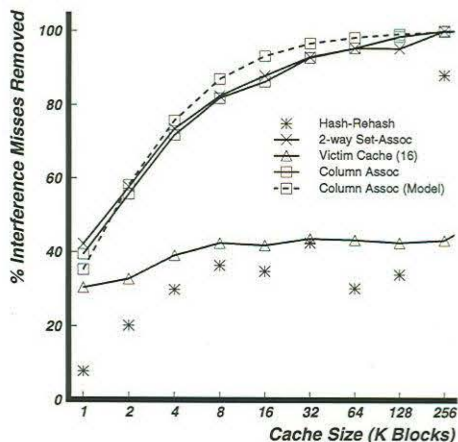

Figure 13: Percentages of interference misses removed versus cache size, averaged over the single process traces. Block size is 16 bytes.

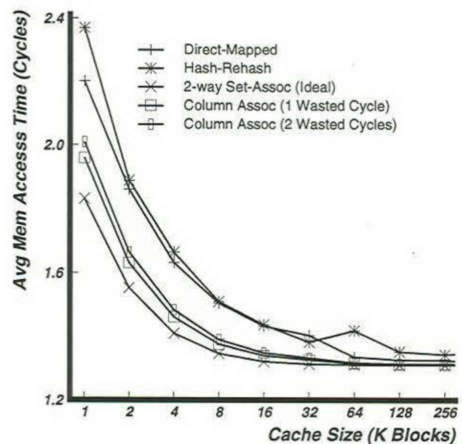

Figure 14: Average memory access times versus cache size, averaged over all six traces. Block size is 16 bytes. The hit access time of two-way set-associative caches is assumed to be the same as that of a direct-mapped cache.

6 Conclusions

The goal of this research has been to develop area-efficient cache control algorithms for improved cache performance. The main metrics used to evaluate cache performance have been the miss rate and average memory access time; unfortunately, minimizing one of them usually affects the other adversely. The optimal cache design would remove interference misses as well as a two-way set-associative cache but would maintain the fast hit access times of a direct-mapped cache.

Two previous solutions which attempted to achieve this are the hash-rehash cache and the victim cache. Although some performance gain is achieved by both these schemes, the success of the hash-rehash cache is very erratic and is hampered by clobbering and low second-time hit rates. The drawbacks of the victim cache include the need for a large, fully-associative buffer and its lack of robust performance (in terms of its miss rate) as the size of the primary cache increases.

This paper proposed the design of a column-associative cache that has the good hit access time of a direct-mapped cache and the high hit rate of a set-associative cache. The fundamental idea behind column associativity is to resolve conflicts by dynamically choosing different locations in which the conflicting data can reside. The key aspect which distinguishes the column-associative cache is the use of a rehash bit to indicate whether a cache set is a rehashed location.

Trace-driven simulations confirm that the column-associative cache removes almost as many interference misses as does the two-way set-associative cache. In addition, the average memory access times for this cache are close to that of an ideal two-way set-associative cache, even when access time of the two-way set-associative cache is assumed to be the same as that of a direct-mapped cache. Finally, the hardware costs of implementing this scheme are minor, and almost negligible if the state represented by the rehash bit could be encoded into the existing status bits of many practical cache designs.

7 Acknowledgments

The research reported in this paper is funded by NSF grant # MIP-9012773 and DARPA contract # N00014-87-K-0825.

References

[1] Alan Jay Smith. Cache Memories. *Computing Surveys*, 14(4):473–530, September 1982.

[2] Steven Przybylski, Mark Horowitz, and John Hennessy. Performance Tradeoffs in Cache Design. In *Proceedings of the 15th Annual Symposium on Computer Architecture*, pages 290–298. IEEE Computer Society Press, June 1988.

[3] Mark D. Hill. A Case for Direct-Mapped Caches. *IEEE Computer*, 21(12):25–40, December 1988.

[4] Norman P. Jouppi. Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers. In *Proceedings of the 17th Annual Symposium on Computer Architecture*, pages 364–373. IEEE Computer Society Press, August 1990.

[5] Anant Agarwal, John Hennessy, and Mark Horowitz. Cache Performance of Operating Systems and Multiprogramming. *ACM Transactions on Computer Systems*, 6(4):393–431, November 1988.

[6] Matthew K. Farrens and Andrew R. Pleszkun. Improving Performance of Small On-Chip Instruction Caches. In *Proceedings of the 16th Annual Symposium on Computer Architecture*, pages 234–241. IEEE Computer Society Press, May 1989.

[7] R.E. Kessler, Richard Jooss, Alvin Lebeck, and Mark D. Hill. Inexpensive Implementations of Set-Associativity. In *Proceedings of the 16th Annual Symposium on Computer Architecture*, pages 131–139. IEEE Computer Society Press, May 1989.

[8] Kimming So and Rudolph N. Rechtschaffen. Cache Operations by MRU Change. Technical Report RC 11613, IBM T.J. Watson Research Center, November 1985.

[9] Anant Agarwal, Mark Horowitz, and John Hennessy. An Analytical Cache Model. *ACM Transactions on Computer Systems*, 7(2):184–215, May 1989.

[10] Alan Jay Smith. A Comparative Study of Set Associative Memory Mapping Algorithms And Their Use for Cache and Main Memory. *IEEE Transactions on Software Engineering*, SE-4(2):121–130, March 1978.

[11] Dominique Thiebaut and Harold S. Stone. Footprints in the Cache. *ACM Transactions on Computer Systems*, 5(4):305–329, November 1987.

[12] M. D. Hill and A. J. Smith. Evaluating Associativity in CPU Caches. *IEEE Transactions on Computers*, 38(12):1612–1630, December 1989.

[13] Richard L. Sites and Anant Agarwal. Multiprocessor Cache Analysis using ATUM. In *Proceedings of the 15th International Symposium on Computer Architecture*, pages 186–195, New York, June 1988. IEEE.

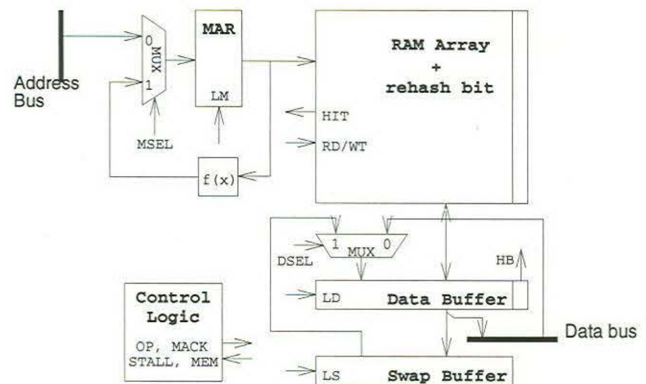

Figure 15: Column-associative cache implementation. Every cache set must have a rehash bit appended to it.

A Cache Implementation Example

The datapaths required for a column-associative cache are displayed in Figure 15. Since the rehashing function used is bit flipping, the functional block $f(x)$ is simply an inverter. In order to accomplish the swap of conflicting data, a data buffer is required. All buffers are assumed to be edge triggered. An n-bit multiplexor can be used to switch the current contents of the memory address register (MAR) between the two conflicting addresses. A MUX is also needed at the input of the data buffer, so that it may read data from either the swap buffer or the data bus. Finally, a rehash bit is added to each cache set; when this bit is read out into the data buffer, it then serves as a control signal. In some implementations the rehash state can be encoded using the existing state bits associated with each cache line, thus eliminating the need for an extra bit.

First-time hits proceed as in direct-mapped caches; however, if there is a first-time miss and the rehash bit of this location is a one, then the column-associative algorithm requires that this location be replaced by the data from memory, which is accomplished in the XWAIT state. When the memory acknowledges completion (MACK), the data is taken off the data bus (LD) and written back into the cache (WT). On the other hand, if the first location is not rehashed ($!HB$), then a rehash is to be performed. The processor is stalled (STALL), MSEL and LM are asserted to load the MAR with $f[a]$, the second-time access is begun (RD), LS is asserted to move the first datum into the swap buffer, and the state changes to f1[a].

If there is a second-time hit, then the correct datum resides in the data buffer. In order to perform the swap, state f1[a] loads the MAR with the original address, $f(f(a))$, and issues a write (WT). State f1[a] also moves the datum accessed the first time from the swap buffer to the data buffer (by asserting DSEL and LD), where it can be written back into the rehashed location in the next state, f2[a]. A second time miss is handled similarly by states WAIT1 and WAIT2, except that the correct datum to be swapped into the non-rehashed location comes from the memory.

B Modeling Column-Associative Caches

The model for column-associative caches uses two parameters: S and u. The parameter S represents the number of cache sets; in direct-mapped caches, the product of S and the block size yields the cache size. The parameter u denotes the working-set size in

state	input	output	next state
IDLE	OP	LM, RD	b[a]
b[a]	HIT		IDLE
	$!HIT$, $!HB$	STALL,MSEL,LM,RD,LS	f1[a]
	$!HIT$, HB	MEM, STALL	XWAIT
f1[a]	HIT	MSEL, LM, WT DSEL, LD	f2[a]
	$!HIT$	MEM	WAIT1
f2[a]		MSEL, LM, WT	IDLE
WAIT1	MACK	MSEL, LM, WT DSEL, LD	WAIT2
WAIT2		MSEL, LM, WT	IDLE
XWAIT	MACK	LD, WT	IDLE

Table 4: State flow table for the control logic of a column-associative cache. In constructing the state flow table all cache accesses are assumed to be reads.

blocks of the program, and must be measured from an address trace of a program. The working set of a program is the set of distinct blocks a program accesses within some interval of time. The notion of the working set of a process used in this paper is the same as that used by Agarwal et al. [9]. For the purpose of computing the cache miss rate, Agarwal et al. suggest measuring the size of the working set over some time interval (typically represented by about 10,000 references in an address trace) which is long enough that the rate of acquiring new blocks drops significantly below the initial start-up rate. Accordingly, we measure the working sets for our traces using 10,000 as the time interval.

The model builds on the self-interference component of the direct-mapped cache model of Agarwal et al. [9] and estimates the percentage of interference misses removed by computing the percentage of cache block interferences removed by the rehash algorithm. As mentioned earlier, the model makes the assumption that blocks have a uniform probability of mapping to any cache set, and that the mappings for different blocks are independent of each other.

Let us first compute the number of conflicting blocks (c_d) in a direct-mapped cache, and then obtain the corresponding number of blocks (c_{cac}) in a column-associative cache. Blocks in a direct-mapped cache are said to conflict when multiple blocks from the working set of the program map to a given cache set.

Let $P(d)$ denote the probability that d program blocks (out of a total of u) map to a given cache set. Because blocks are assumed to map with equal likelihood to any cache set, the distribution of the number of blocks in a cache set is binomial; accordingly we have,

$$P(d) = \binom{u}{d} \left(\frac{1}{S}\right)^d \left(1 - \frac{1}{S}\right)^{u-d} \qquad (4)$$

In the above equation, the probability that a given block maps into a specific cache set if $1/S$, and the probability that d specific blocks map into a cache set is $(1/S)^d$. The corresponding probability that none of remaining $u - d$ blocks from the program's working set map into that set is $(1 - 1/S)^{u-d}$. Finally, we can choose d blocks from the working set of size u in $\binom{u}{d}$ ways.

We can now compute the number of conflicting blocks in a direct-mapped cache by subtracting from u the number of cache

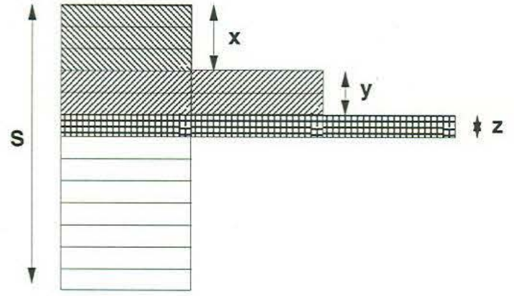

Figure 16: Mapping of cache blocks to cache sets.

sets with exactly one block. Since the cache has S sets, the number of cache sets with exactly one block is $SP(1)$. The number of conflicting blocks in a direct-mapped cache is, therefore,

$$c_d = u - SP(1)$$

We now compute the number of blocks (c_{cac}) that suffer conflicts after a rehash phase in a column-associative cache. The rehash algorithm works by attempting to place a conflicting block into some other cache set. The number of conflicting blocks can now be computed by estimating the probability a block suffers a conflict on a rehash.

Figure 16 shows a mapping of program blocks (represented by shaded rectangles) to cache sets in a direct-mapped cache of size S sets. The mapping in the figure indicates that x cache sets contain exactly one block, y cache sets contain exactly two blocks, and z cache sets contain three or more blocks. Because the blocks are binomially distributed, we know that

$$x = SP(1) \qquad (5)$$

$$y = SP(2) \qquad (6)$$

$$z = \Sigma_{d=3}^{u} SP(d) = S - SP(0) - SP(1) - SP(2) \qquad (7)$$

To compute the number of conflicts eliminated, we need to focus on the cache sets with exactly two blocks, because, as discussed earlier, the rehash algorithm with bit flipping does not eliminate any conflicts in cache set with three or more blocks. When exactly two blocks are mapped to a given cache set, our rehash algorithm with bit flipping allows one of these blocks to be mapped elsewhere. Therefore, to compute the number of conflicts that are eliminated, let us detach y blocks as depicted in Figure 17, and then randomly reassign them to cache sets.

When the detached blocks are reassigned to cache sets, they may introduce additional conflicts. The number of conflicting blocks introduced depends on the number of blocks in the cache sets to which the detached blocks are assigned.

1. If a detached block falls into an empty cache set, then no additional conflicts are introduced. Note that because the bit-flipping rehash algorithm produces a one-to-one mapping, it does not place multiple detached blocks into the same cache set.

2. If a detached block falls into a cache set with more than two blocks already mapped to it, then one additional conflict is introduced. With random placement, the fraction of detached blocks that are placed in such sets is z/S. Thus, the number of resulting conflicts is yz/S.

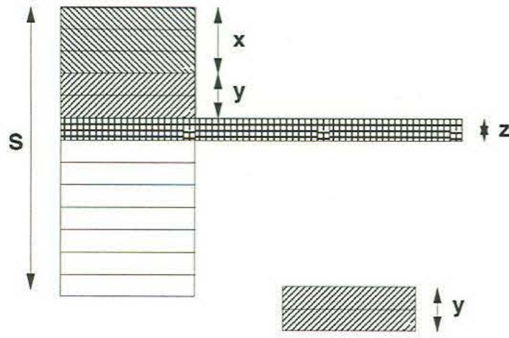

Figure 17: Mapping of cache blocks to cache sets with y blocks detached.

3. Finally, if the detached block falls into a cache set with one block, then two additional conflicts are introduced: one corresponding to the previously mapped block, and the other corresponding to the detached block. The probability of this placement is $(x+y)/S$, and the number of additional conflicts is $2y(x+y)/S$.

Adding the conflicts introduced by the reassignment of y blocks $(y\frac{z}{S} + 2y\frac{x+y}{S})$ to the initial number of conflicts before the introduction of the detached blocks $(u - x - 2y)$, we obtain the total number of conflicting blocks in the column-associative cache.

$$c_{cac} = u - x - 2y + y\frac{z}{S} + 2y\frac{x+y}{S}$$

Substituting for x, y, and z, from Equations 5, 6, 7, and simplifying, we get

$$c_{cac} = u - SP(1) - SP(2)(1 + P(0) - P(1) - P(2))$$

We estimate the percentage of interference misses removed by the percentage reduction in the number of conflicting blocks. Our validation experiments indicate that this is a good approximation. Thus, the percentage of interference misses removed is

$$\frac{c_d - c_{cac}}{c_d} = \frac{SP(2)\left(1 + P(0) - P(1) - P(2)\right)}{u - SP(1)} \tag{8}$$