McGill University
School of Computer Science
ACAPS Laboratory

Advanced Compilers, Architectures and Parallel Systems

A Unified Framework for Hybrid Access Cache Design and Its Applications

Kevin B. Theobald
Herbert H. J. Hum†
Guang R. Gao

ACAPS Technical Memo 65
December 1, 1993

theobald@cs.mcgill.ca, herbert@ece.concordia.ca, gao@cs.mcgill.ca

†Concordia University Dept. of Electrical and Computer Engineering, Montréal, Québec

Abstract

High-speed microprocessors for high-performance computing demand fast on-chip caches to keep the processor usefully busy. Direct-mapped caches have been utilized for their superior access times, but suffer from higher miss rates due to address conflicts. Recently, this has led to several proposals for hybrid access caches combining the superior speed of direct-mapped caches with the higher hit rates of associative caches. These include the *MRU cache* [3], the *victim cache* [6], and the *column-associative cache* [2].

Contributions of this paper include:

- A model which unifies these emerging hybrid designs for on-chip caches within a single framework called the *Hybrid Access Cache* model (HAC model). In this model, a hybrid access cache is conceptually viewed as two caches: a fast direct-mapped primary section, and a slower but associative secondary section, with an efficient mechanism for exchanging data between the two. Under this model, a rich design space is structured in a way which helps cache designers systematically to explore, experiment and compare various combinations of hybrid cache designs in order to meet area and performance goals.
- Existing hybrid cache designs are found to be near the extreme points in the design space as revealed by the framework, which suggests the possibility of other designs. In this paper, we develop a novel hybrid access cache scheme, the *half-and-half cache*, which has half of the cache lines reserved for direct-mapped accesses and the other half for associative accesses. Lying in the center of the design space, it tries to exploit equally the advantages of both direct-mapping and associativity.
- We compare a total of twelve cache designs using trace-driven simulation, and illustrate their performance differences under our model. Simulation results are presented for single-process SPEC traces, multiple-process SPEC traces, and ATUM traces. Results demonstrate that the performance of the half-and-half cache fares better than other hybrid cache schemes for certain types of benchmarks.

Keywords: cache, half-and-half cache, hybrid access cache framework, microprocessors, trace simulation.

Contents

List of Figures

List of Tables

1 Introduction

In the past decade, the operating frequencies of high-performance microprocessors have increased dramatically, but memory speeds have not kept pace. Processor designers have had to turn to on-chip caches to overcome this gap. Because of the great differential in access time between an on-chip cache and an off-chip cache or main memory, an effective on-chip cache design is becoming more and more crucial to maximizing processor performance. To get the most throughput from an on-chip cache, it is best to have a cache which has a *hit time* of one clock cycle, i.e., the cache can retrieve the data in one cycle provided it is stored in the cache. In summary, on-chip cache design is now becoming a central issue in high-performance microprocessor architectures.

Selecting an on-chip cache design is non-trivial because there is such a variety of cache configurations from which to choose. Conventional cache designs have at least four different design parameters: the line size (L); the number of sets (N); the number of lines per set, or *associativity* (K); and the *replacement policy* for choosing which line to remove from a set when a new line must be inserted. This parameter space can be divided into three general categories. A *fully-associative* cache has only one set, i.e., $N = 1$. Thus, there must be a tag comparator for every tag stored in memory, which generally makes this design impractical for large caches. A *direct-mapped* cache has only one line per set, i.e., $K = 1$. Thus, each memory address can only be stored in one unique place in the cache. A *set-associative* cache is any cache in which both N and K are greater than 1.

Tempering the varied choices of cache designs, studies have shown tradeoffs between the associativity of a cache, its access time, and its miss rate (percentage of accesses which are misses). Because direct-mapped caches can only store a memory block in one specific line, they are more prone to conflicts between addresses (memory blocks) that map to the same set, and thus tend to have more misses than set-associative caches. However, due to a shorter critical circuit path, the access time of a direct-mapped cache is lower than the access time for a set-associative cache storing the same amount of data [5, 7]. In a set-associative cache, the data line from the cache can not be written into the memory output buffer until the tag comparisons are complete, since the cache has to know *which* of the data lines to select. On the other hand, since an address can only be stored in one line in a direct-mapped cache, the direct-mapped cache can write the selected line from the cache to the output buffer *at the same time* as it is comparing the tags. The data can then be invalidated if the tags don't match.

For moderate-speed microprocessors, it is usually possible to build a cache with a small amount of associativity (such as $K \leq 4$) and still attain a one cycle hit time. However, for high-speed processors with shorter clock cycles, even a cache with 2-way associativity may be too slow to return its data in one cycle; the designer has to use a direct-mapped cache to meet the clock speed requirement.

1.1 Development of Hybrid Caches

Because direct-mapped caches tend to have higher miss rates, researchers have developed various hybrid schemes in an attempt to combine the fast access of a direct-mapped cache with the better hit rate of a set-associative cache. The simulation results of So and Rechtschaffen [10] showed that in a set-associative cache, the vast majority of hits within a given set involve the MRU (most recently used) member of that set. This observation led to the *MRU cache* [3] — a set-associative cache which is biased toward the MRU member of each set. When the cache is accessed, all tags in the selected set are compared simultaneously. However, even before the outcomes of the comparisons are known, the line containing the MRU of the set is immediately latched into the output buffer. If the tag check should subsequently show that the MRU tag does *not* match the address tag, then the data is invalidated. If one of the other tags matches, then that line is latched into the output buffer in the next cycle, and that line becomes the new MRU. Thus, an *MRU hit* has an access time of one cycle, and a *non-MRU hit* takes two cycles.

Another way to boost the hit rate of a direct-mapped cache is to add a *victim cache* [6]. A victim cache is a small fully-associative cache and is typically no more than sixteen lines. Whenever a line is replaced in the direct-mapped cache, the *victim* (the line that is replaced) is transferred to the victim cache. The direct-mapped cache is accessed in a single cycle, as before, and the output buffer invalidated if the tag mismatches. If the desired address is in the victim cache, then the line is fetched from the victim cache (in the second cycle), and that line is swapped with the appropriate line in the direct-mapped cache.

Since some accesses require two cycles to perform, a simple way to combine fast access with better associativity is to read a direct-mapped cache twice. In the *hash-rehash cache* [1], if a read misses on the first cycle, a second read is done using a different hashing function to select an alternate line. If the second try is a hit, the first and second lines are swapped. An improved version called the *column-associative cache* [2] keeps with each tag a *rehash bit* which indicates whether the corresponding line represents a first-cycle hit or a rehash (second-cycle) hit. This bit is used to limit the swapping in order to reduce thrashing.

1.2 A Unified Framework

It can be seen that these hybrid cache designs represent very different solutions to the same problem. It would be quite useful if the alternatives in designing a hybrid cache could be characterized in a way such that useful insights into their relations could be readily obtained, and new promising design points could be quickly identified and explored. In this paper, we show that most of these existing hybrid caches can be viewed as variations of a general cache design framework, which we call the Hybrid Access Cache (HAC) model. Under the HAC model, a hybrid cache can be viewed as consisting of a direct-mapped primary section, a slower secondary section (usually associative), and an efficient mechanism for exchanging data between the two. The direct-mapped section provides fast, single-cycle access to frequently-accessed data, while the slower section provides extra associativity in order to eliminate some of the conflict misses that would occur in a purely direct-mapped cache. Unlike conventional two level caches (a first-level and a second-level cache) the two sections in a hybrid access cache are organized as one level in the memory hierarchy. (A more detailed comparison of HAC's vs. two-level caches can be found in Section 2.3.1.)

The purpose of the HAC model is to allow the user to think about a whole spectrum of values for the cache design parameters, in order to balance the benefits of fast access and greater associativity. To illustrate, we will show that the existing hybrid cache designs, such as the MRU, victim and column-associative caches, are all near extreme points in the design space. We also demonstrate how the use of this framework allowed us to conceive of a novel cache organization, which we call the "half-and-half" cache. In this cache, half of the memory storage is allocated to a direct-mapped section, and the other half is allocated to a K-way set-associative section. In this manner, the associative section can take advantage of higher associativity without negatively impacting the performance of the direct-mapped portion.

1.3 Synopsis

In the next section, we present the HAC model in greater detail, discussing its design framework, typical operations of a HAC, and how to measure its costs and performance. That section also discusses implementation details for HAC's, and contrasts them with two-level caches. In Section 3, we show how the HAC model can be helpful in exploring the different schemes. We then use the model to design the half-and-half cache.

We have performed trace-driven cache simulation to compare various hybrid access cache designs and illustrate their performance differences under our model. Simulations have been performed on both single-process and multiple-process traces derived from SPEC benchmarks, as well as on ATUM traces [1]. In Section 4, we present the results of simulations and compare the performance of the half-and-half to those of the MRU cache, the victim cache, and the column-associative cache. The results demonstrate that the performance of the half-and-half cache is better than other hybrid cache schemes for many benchmark programs, particularly at intermediate cache sizes. Finally, we present our conclusions in Section 5. Experimental results are listed in greater detail in an appendix.

2 The Hybrid Access Cache Model

This section reviews basic cache design and terminology, and then develops the Hybrid Access Cache as a model for caches which combine features of direct-mapped and associative caches. We then discuss the operation, implementation, and performance and cost measurements for HAC's.

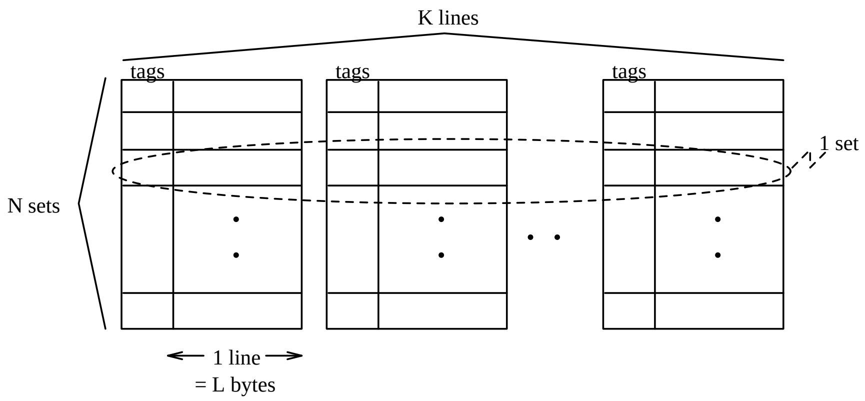

Figure 1: Typical Cache Structure

2.1 Basic Cache Design

A typical cache (on-chip or off-chip) is shown in Figure 1. A cache is divided into an $N \times K$ array of *lines*. Each line contains L bytes of data (L a power of 2), corresponding to L contiguous bytes in main memory, and a *tag* which indicates which block of main memory is being stored in that line.

The lowest $\log_2 L$ bits of a memory address are used as an offset into a particular line. A set-selection hashing function s is applied to the remaining bits of the address (call these bits a) in order to select one of the N sets. Some or all of these address bits form the *tag*. Upon receiving a request, the tags of the K member(s) of the selected set $s(a)$ are (simultaneously) compared with the tag of the requested address. If a match (termed a *hit*) occurs, the line corresponding to the tag that matched is read from the cache, and the portion of the line specified by the line offset is either sent to the processor (if read), or updated (if written). If none of the K tag(s) match (termed a *miss*), the cache must choose one line of the set to be replaced by the new line.

This general description covers many different conventional cache designs. A typical cache design requires five different design parameters and choices: the line size (L); the number of sets (N); the number of lines per set, or *associativity* (K); the *replacement policy* for choosing which line to remove from a set when a new line must be inserted; and the hashing function. The simplest and most common implementation is to make N a power of 2, and make $s(a)$ be the identity of the lowest $\log_2 N$ bits of a. Selecting a line in this manner is called *bit selection*. With this scheme, these bits do not need to be stored in the tag fields. In this paper, we generally assume bit selection is used, but our model is applicable to other implementations as well.[1]

[1] For instance, Seznec proposes different hashing functions for the two banks of a 2-way set-associative cache [8].

If only size is being considered, then for a given replacement policy, the size of the cache may be represented by the triple (N, K, L). The three main types of cache organizations are:

1. Fully-associative: $N = 1$;
2. Direct-mapped: $K = 1$;
3. Set-associative: $N > 1; K > 1$;

For an associative cache (full or set), studies [9] have shown that a Least Recently Used (LRU) replacement policy exhibits generally good performance. Thus, many associative caches employ some form of this policy.

2.2 Adding Associativity to Direct-Mapped Caches

Inherently, direct-mapped caches have a speed advantage over associative caches in that the access time of an associative cache must include time for the associative tag matches—an overhead not incurred in a direct-mapped cache [5, 7]. However, a direct-mapped cache generally suffers a higher miss rate as compared to an associative cache of the same size due to the restriction that a memory block can only map to a unique cache line. Interferences (conflicts) due to multiple blocks mapping to the same line results in increased miss rates, whereas an associative cache would offer multiple cache lines to accommodate those blocks. To get the speed advantage of direct-mapped caches, as well as the lower miss rates offered by associative caches, researchers have proposed caches incorporating associativity in direct-mapped designs. We term these new cache designs *Hybrid Access Caches* (HAC's) due to their hybrid nature: they are part direct-mapped and part associative, and they have dual accessing modes. In a HAC, first the direct-mapped part of the cache is probed. If that is not successful, a further probe is performed, this time in the associative section, thus the term 'hybrid access.' Cache designs within this category include the MRU cache [3], the victim cache [6], the hash-rehash cache [1], and the column-associative cache [2].

2.3 Viewing HAC's as Dual Caches

A HAC can be conceptually viewed as consisting of two (dual) caches: a fast *primary* cache which is direct-mapped, and a *secondary* cache, which may be direct-mapped, set-associative, or fully-associative. An example configuration is shown in Figure 2 where the secondary cache is a two-way set-associative cache. Any memory block may be cached in either the primary or secondary section, but not at the same time. Data in the primary cache may be accessed in a single cycle, while data in the secondary cache requires two or more cycles to access. The goal of the hybrid access cache organization is to try to keep most of the frequently-used data in the primary cache, while using the secondary cache to store data which would otherwise be

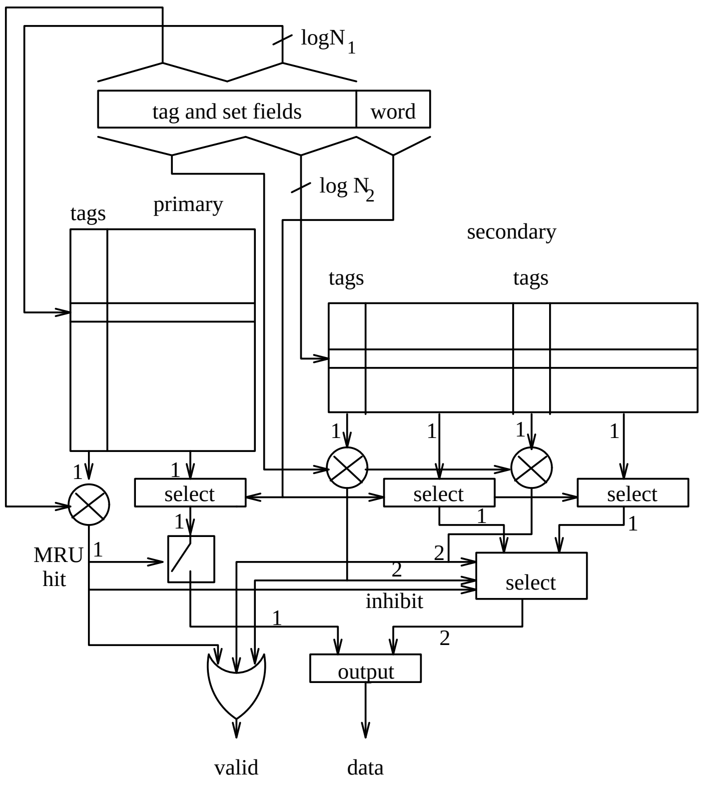

Figure 2: Structure of a HAC

swapped out of the cache due to address conflicts in the direct-mapped primary cache. Even though accessing the secondary cache may require two or more cycles, this is far better than the standard miss penalty.[2]

The primary and secondary caches may be different sizes, and may have different hashing functions s_1 and s_2 to access the sets in the two caches (s_1 is used to access the primary while s_2 is for the secondary). However, since data is to be exchanged between the primary and secondary caches, the line sizes must be the same. Therefore, the sizes of the two caches can be represented by the triples $(N_1, 1, L)$ and (N_2, K, L).

Conceptually, a HAC performs a read in the following manner (refer to Figure 2). Assume a is the address with the line-offset bits removed:

1. In the first cycle, the cache reads the tag and data in line $s_1(a)$ of the primary cache. (In the figure, a number beside an arc indicates which cycle the output will appear. Moreover, the figure assumes s_1 and s_2 uses bit selection.) The designated data is latched into the

[2]Considering that high-speed microprocessors can operate at more than ten times the access time of off-chip memory, the need to go off-chip can be very costly.

output buffer of the cache, becoming available at the end of the first cycle. In the same cycle, the HAC reads the tags and data of all members of set $s_2(a)$ of the secondary cache.

2. The single tag from the primary cache and the K tags from the secondary cache are compared against the appropriate bits of the address.

3. If the tag from the primary cache matches the address, then a *primary hit* occurs. The cache immediately cancels the probe of the secondary cache, so that it may be used immediately in the next cycle.

4. If the tag from the primary cache does not match the address, then the output buffer is invalidated, and the cache must operate for at least one more cycle.

5. In the second cycle (if there was no primary hit), the tag comparators from the secondary cache are examined. If one of them matches (a *secondary hit*), then the appropriate data line is latched into the output buffer. The cache must now move the selected data (and tag) from the secondary cache into the primary cache, while simultaneously moving the displaced line from the primary cache into the secondary cache.[3] (In Jouppi's terminology, this displaced primary-cache line is called the *victim* [6].) If $K > 1$ and the secondary cache uses an LRU replacement policy, then it would make sense for the LRU status bits in the set to be changed so that the victim becomes the new MRU element, since it just came from the primary cache.

6. If all tag comparators mismatch, then the cache must fetch the line from main memory. This data will go into the primary cache in the line that was just checked. The resulting primary-cache victim moves over to the secondary cache, which will cause the LRU element of the secondary-cache set to be flushed. When the data returns from main memory, the cache copies it into the primary cache.

The swapping done in steps 5 and 6 impose a restriction on the hashing functions:

$$s_1(a_1) = s_1(a_2) \Rightarrow s_2(a_1) = s_2(a_2) \qquad (1)$$

In other words, if two addresses map to the same line in the primary cache, then they should map to the same set in the secondary cache.

This need can be seen in the following example, in which the primary cache has size $(4, 1, 1)$, the secondary cache has size $(8, 1, 1)$, and bit selection is used (see Figure 3). Assume that the memory blocks at addresses 00001 and 00101 are in the secondary cache (in sets 001 and 101, respectively), and memory block at address 01001 is in the primary cache (in set 01). What happens if the processor reads address 00101? The data corresponding to address 00101 should be moved from set 101 of the secondary cache to set 01 of the primary cache, displacing address 01001. Yet this victim must go into set *001* of the secondary cache, *not* set 101. The former set

[3]This swapping is not shown in Figure 2 for clarity.

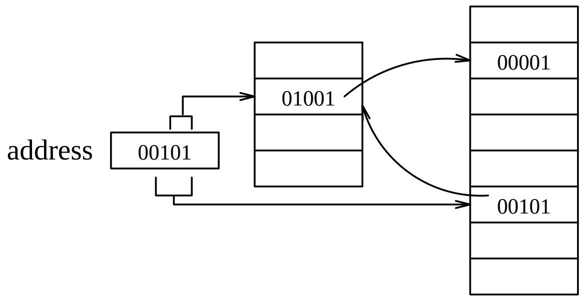

Figure 3: Problems with Bad Hashing Functions

must flush its other address, while set 101 is left without a victim, and must be marked invalid. This results in a complex accessing scheme and wastes precious cache space.

Assuming that the co-domain of s_2 is the entire set space in the secondary cache, this restriction means that $N_2 \leq N_1$. If bit selection is used, then N_1 and N_2 must be powers of 2; typically, the rightmost $\log_2 N_1$ bits of a are used to select the primary cache line, and the rightmost $\log_2 N_2$ bits of that field are used to select the secondary cache set.

2.3.1 HAC's and Two-Level Caches

At first glance, our conceptual view of a HAC superficially resembles a two-level cache, because both systems have a slower secondary cache which attempts to cache things which are missed by the primary cache. However, two-level cache systems usually have substantially different hardware implementations for the two caches (e.g., off-chip SRAM chips for the second level cache of a two-level cache system versus on-chip memory for the HAC's). Since HAC's would be implemented on the same chip with the CPU and the chip area is limited, there is a direct tradeoff between the sizes of the two caches. This distinction leads to the following important differences between HAC's and two-level caches:

- In a two-level cache system, the second-level cache is usually much larger than the first-level cache. In a HAC, on the other hand, the size of the secondary cache is limited by the requirement that it have no more sets than the primary cache. Moreover, in many cases, the secondary cache is smaller than the primary cache.
- As a result of this difference, the main purpose of a second-level cache is to alleviate *capacity misses* [4], those due to the small size of the first-level cache. In a HAC, the main purpose of the secondary cache is to alleviate *conflict misses*, those due to address interferences caused by direct mapping in the primary cache.
- Typically, two-level caches have the *multilevel inclusion property* [4]—meaning that a copy of everything in the first-level cache is also in the second-level cache—for increased

efficiency in consistency maintenance between I/O and the processor. In a HAC, no line can be in both caches at the same time.

- Lastly, there is no restriction as to where a line is mapped to in the second-level cache in relation to the first-level cache. In a HAC, two addresses mapping to the same line in the primary cache must map to the same set in the secondary cache (eq. 1).

2.3.2 Implementation Details

A few words should be said about implementation details. We assume that the swapping can occur within the second clock cycle. To do this, both subcaches need to write back during the second cycle. At the end of the first cycle, all data from both the primary line and the secondary set are latched in a buffer big enough to hold all $K + 1$ lines, plus tags and LRU status bits. (These must be read in one cycle, else the cache cannot perform one-cycle hits.) Since the results of tag matches are known at the end of one cycle (again, because it does one-cycle hits), these results can be used to control write-backs. The same mux that selects one of the secondary lines for the output can be used to route that output to the primary cache. In addition, each column of the secondary cache needs a multiplexor so that its input can come either from its own line in the buffer or from the line in the buffer linked to the primary cache. LRU status bits are updated in the usual way.

To minimize storage costs, tags can be limited to only the bits not used in set selection (assuming bit selection is used). This means that the tags in the primary cache will be shorter than the tags in the secondary if $N_1 > N_2$. When a primary-cache victim is moved to the secondary, the extra bits can simply be read from the upper bits of the primary line number.

For cache writes, various HAC's can implement different strategies. One such strategy can be as follows: a hit in the primary cache will simply alter the contents in the primary cache and a hit in the secondary cache will also result in the modification of the corresponding line in the secondary cache. If a write request misses in both, the write can be directed to the next level in the memory hierarchy. There are many other strategies a HAC can adopt such as swapping lines with the primary cache if there was a hit in the secondary, fetching the line from memory if there was a miss in both caches and storing it in either cache, etc.

2.4 Performance and Cost Measurements for HAC's

When comparing simple caches, the *miss rate* (ratio of misses to total accesses) is usually an adequate measure of performance. However, it is not sufficient to describe the performance of HAC's, because there are two caches, and hence two separate miss rates. The relation between the miss penalties must also be considered, because a HAC could have few main-memory accesses, yet still have relatively poor performance because many of the hits occur in the secondary cache and take longer to access.

If a single parameter is needed to compare different cache alternatives, then the average effective memory-cycle time (t_{eff}) can be used. This parameter averages the costs of all types of misses. If h_1 and h_2 are the fractions of accesses which hit in the primary cache and secondary cache, respectively, and if t_1 and t_2 are the access times of the two caches, then

$$t_{eff} = t_1 h_1 + t_2 h_2 + (1 - h_1 - h_2)(t_1 + t_{main}) \quad (2)$$

This assumes that the main memory access begins at the end of the first access, once all the comparator results are compared. This may be simplified to

$$t_{eff} = t_1 + (t_2 - t_1)h_2 + (1 - h_1 - h_2)t_{main} \quad (3)$$

If we assume t_1 is 1 cycle and t_2 is 2 cycles, then this becomes

$$t_{eff} = 1 + h_2 + (1 - h_1 - h_2)t_{main} \quad (4)$$

To compute the hardware overhead costs of a HAC, one should consider the number of tag comparisons and the extra hardware needed to perform the swapping. The number of comparators needed is $K + 1$, which is about what is needed if a conventional K-way set-associative cache is used.[4] All other hardware costs are roughly equivalent, except for the extra data paths needed to move a primary cache victim over to the secondary cache. These costs will probably be small since that operation is the inverse of the column-selection used to choose one secondary line (effectively a demultiplexer). The two can be laid out and routed in parallel, so the overall area will not increase by much.

3 Using the Hybrid Access Cache Model

The Hybrid Access Cache framework can be used to represent the design space of possible cache configurations and to gain insights into how well they might be expected to perform. In this section, we show how to do this in general, and show that specific hybrid schemes are equivalent to specific cases in the HAC model. Using the parameter space created by the HAC model, we develop a new cache configuration called the "half-and-half" cache. The half-and-half cache can often give better access times than the other cache configurations, especially for a middle range of cache sizes, as shown by our experiments in the next section.

3.1 The HAC Design Space

Section 2.3 presented a few of the restrictions of the HAC model:

[4] In the column-associative cache, only one tag comparator is required. However, it can be reorganized to take advantage of an extra comparator and data paths for faster access times.

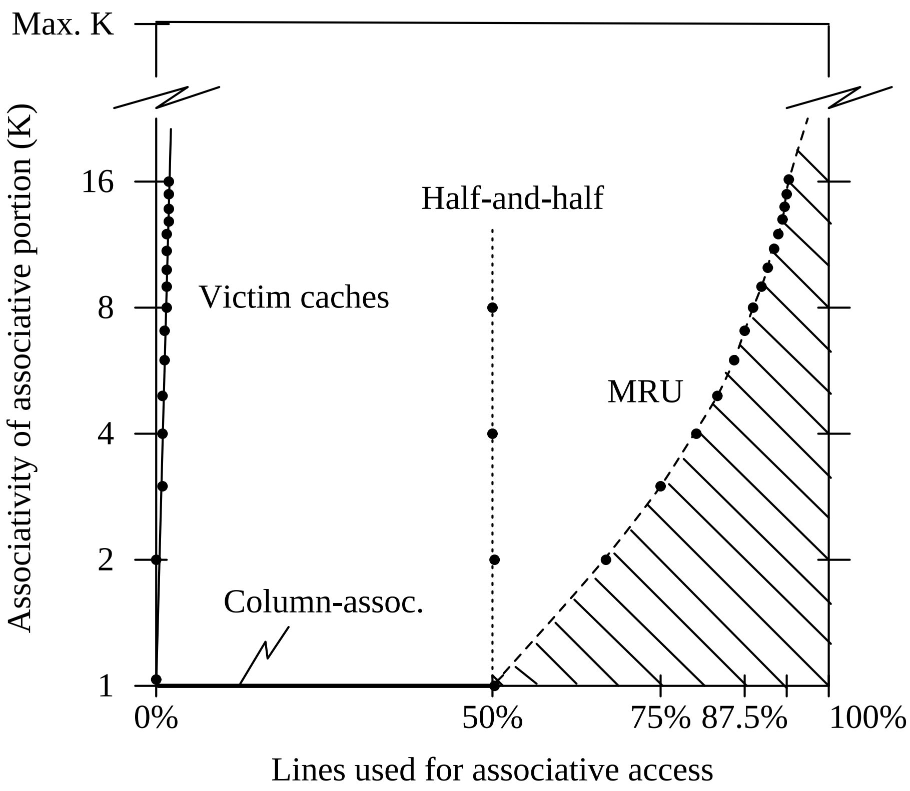

Figure 4: Design space of the HAC model

1. The line sizes are the same;
2. $N_2 \leq N_1$.

This still leaves us with a large design space to explore. Figure 4 shows the HAC design space. On the horizontal axis, we have the percentage of cache lines allocated to associative accesses, i.e., number of lines in the secondary cache. Obviously, as we increase the number of lines to one cache, the number of lines allocated to the other is decreased accordingly. On the vertical axis is the associativity of the secondary cache, K, where an associativity of one indicates that the secondary cache has a direct-mapped organization. Note that we do not have any mention of line size nor of the total cache size. They are omitted for clarity.

In the HAC design space, we assume that the total area of a HAC can be allocated to the primary cache which has a one CPU-cycle access time; thus allocating lines to the associative secondary cache only decreases the amount of HAC memory which can respond in one cycle, not adding to the size of the HAC. Moving to the right in the design space indicates that the HAC has more and more lines dedicated to the associative accesses so that conflicts from the direct-mapped accesses can be better resolved. Whereas moving to the left in the design space indicates that more and more lines are used in a direct-mapped manner for the increased speed.

Thus there is a trade-off when a cache design is moved left or right in the space. Moving towards the top of the design space indicates that the cache has more associativity in the associative portion to resolve more conflicts, but at the cost of more tag matching hardware. Again, there is another trade-off when moving a cache design towards the top or bottom of the design space.

In the above figure, design points for previously proposed HAC's are shown (the description of the "half-and-half" will be explained in a later section). The victim cache (i.e., a direct-mapped cache augmented by a victim cache) occupies design points on the far left. The reason is that almost all the cache lines are devoted to a direct-mapped access and only a very small portion is devoted to the victim cache. Jouppi suggests a victim cache which has 4 to 16 lines and is fully associative. Thus the design space for the DM & victim is a steeply-sloped straight line starting from a K value of 1 and percentage of DM lines at $\sim$100% and rising to the right.

The MRU cache occupies the design points which start at 50% associative lines and a K value of one and forming a hyperbolic curve towards the upper right-hand corner of the HAC design space. In the MRU cache, the structure is an overall set-associative cache with one line in each set dedicated to the most recently used. Therefore, a four-way set-associative cache converted to an MRU cache would have 25% of the total cache lines dedicated to direct-mapped access. As for the cache lines in the associative portion (75% of the total), the associativity of that secondary cache would be three. The MRU caches mark the other extreme of the design space; the region to the right of the MRU curve cannot correspond to any legal designs given that $N_2 \leq N_1$ (see Section 2.3).

The column-associative (CA) cache (and its hash-rehash predecessor) is a HAC which does not have a single point in the design space, but a horizontal line. In terms of the mapping of addresses to cache lines, the CA cache is equivalent to a two-way set-associative cache, because the bit-flipping rehash function of the CA cache effectively partitions the cache line space into 2-way sets. In terms of access, however, the CA sometimes acts like a direct-mapped cache, because the distribution of cache lines to direct-mapped accesses is dependent on the access patterns of the application. If accesses do not cause conflicts in the CA cache, then the CA cache functions as if it devoted 100% of its lines for direct-mapped access. However, as conflicts arise, lines begin to be used for associative (rehashed) accesses. Eventually, if all accesses cause conflicts, then up to half of the lines can be used for associative accesses. Thus the horizontal line extends from 100% DM access to 50%. The line representing the CA cache is placed on the design space of $K = 1$ because a line can only be rehashed to one other location.

3.2 Restricting the HAC Design Space

In the HAC design space as mentioned above, all design points within the space is possible since we can have a secondary cache which has x% of the total lines and use a multiple rehashing scheme to access it; thus increasing the associativity of the secondary cache.[5] However, such

[5]This can be seen by examining what the hash-rehash scheme does to a direct-mapped cache.

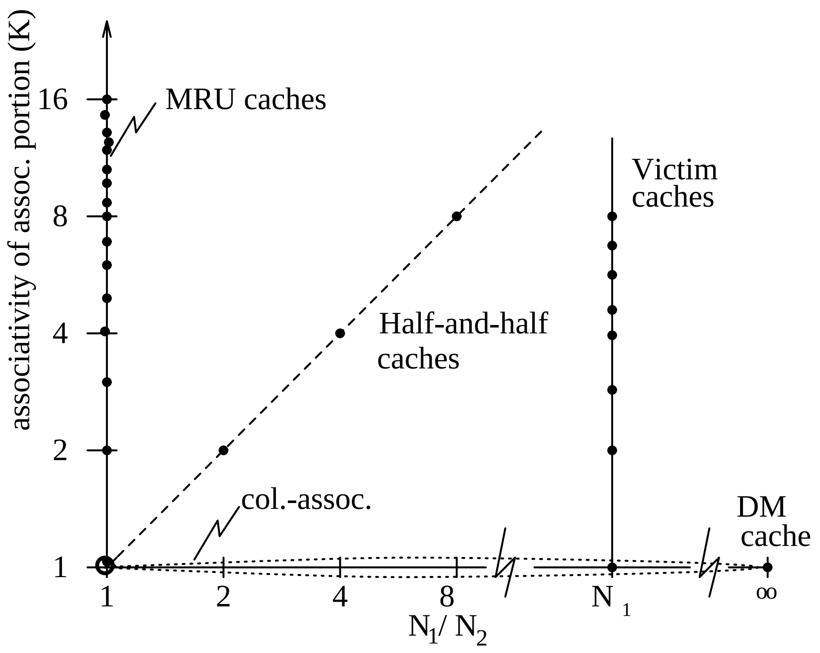

Figure 5: Design space of more efficient HAC's

odd line distributions along with cumbersome access mechanisms leads to inefficient designs. So for now, we only consider what happens when the hashing functions use bit-selection. Since this means that N_1 and N_2 are powers of 2, it also means that N_1/N_2 is also a power of 2.

Even with this restriction, we are still presented with a large range of sizes for the secondary cache. Given a primary cache of size N_1, we can set N_2 to any power of 2 up to N_1, and set K to any positive integer (although in practice, K would be limited by the high costs of high associativity). We may view this as another two-dimensional design space, but one for more efficient HAC designs (Figure 5). In this design space, the vertical axis remains the same, the only difference is that the horizontal axis is now the ratio of the number of sets for direct-mapped access to the number of sets for associative accesses. A drawback of this design space is the lack of an indication of how storage space is devoted to the primary and secondary caches.

To interpret the design points in this space, we can adopt these few observations. When moving towards the right, the number of sets for direct-mapped access is increased or the number of sets for associative accesses decreases. In a HAC with constant size, for every increase in the number of sets allocated to associative accesses (moving to the left), there is a corresponding drop to the number of lines for DM access. Increasing the number of lines for associative access alleviates the conflict misses arising from direct-mapped accesses. Moving up and down the design space has the same meaning as in the previous HAC design space.

The various HAC designs are shown in the space. The design region of the victim cache is a vertical line where N_2 is one (a victim cache is fully associative). The design region of the

MRU cache is a vertical line where $N_1 = N_2$. If the total cache size is fixed, then as the design point for a MRU cache moves upwards, there is a corresponding decrease in the number of sets in both the direct-mapped and associative-access portions. All values of K are possible for an MRU cache; however, if the total number of lines in the cache is to be a power of 2, then K is restricted to the set $\{1, 3, 7, 15, \ldots\}$.[6] The design point of a column-associative cache is again a horizontal line at $K = 1$ and running from $N_1 = N_2$ to $N_2 = 0$.

3.3 Exploring the Design Space

The preceding sections gave the characteristics of the HAC design space, and illustrated where existing designs fit into the space. For a primary cache of a given size $(N_1, 1, L)$, there are many choices for the size of the secondary cache (N_2, K, L). A designer may explore the design space of Figure 5 to look for a good hybrid design which combines the benefits of fast access and associativity, assuming that at least part of the cache must be direct-mapped to get the benefits of single-cycle access.

One can choose any powers of 2 for N_1 and N_2, provided $N_2 \leq N_1$, and any number for K. If we write

$$N_1 = 2^r(N_2) \tag{5}$$

where r is the log of the ratio between the number of sets in the primary and secondary caches, then the total number of lines N in the combined cache is given by

$$N = (2^r + K)N_2 \tag{6}$$

Even with a reasonable upper bound for K (e.g., 16), this gives the chip architect a lot of flexibility in choosing an appropriate cache size. For instance, suppose that a chip floorplan has enough room for roughly 1300–1400 cache lines. If simple direct-mapped caches were used, a 2K-line cache would not fit, while the next smaller size, a 1K-line cache, would not use all available space. One could build a 1K-line primary cache and a 256-line secondary cache, with enough space left over to hold the extra circuitry necessitated by the dual configuration. Of course, other configurations are possible.

This design space is still large. Does the HAC framework provide any insights into which regions of the space are most likely to yield the best caches?

We may answer this question by considering some basic properties of caches and the way they are used in real applications. One observation about real workloads is that as the cache size increases, the number of conflict misses goes down, so that associativity becomes less of a win. Larger caches can get by with lower rates of associativity. Therefore, as one increases the

[6] This is not an absolute requirement. One can have a 5-way MRU cache, just as one can have a 5-way set-associative cache.

cache size, the region of greatest interest would tend to move toward the bottom of the design space in Figure 4.

Also, if the cache size is small, then capacity misses are a problem, and space is precious. Therefore, one would want to take most of what little data can be cached and keep it in the fast cache, so that at least the hits that do occur can occur quickly. In other words, a designer would want to focus attention on the left side of Figure 4. As cache size increases relative to the program working set, capacity misses are less problematic, so one can afford to devote more memory to the secondary cache to alleviate conflict misses, and one's attention would move to the right.

Combining these two trends, one would expect that for a small on-chip cache size, the best candidates for cache configurations would be in the upper left corner of Figure 4. In this region, we see the victim caches. The victim cache is a small, fully-associative cache containing perhaps 16 lines. Slight variations are possible, such as replacing the fully-associative cache with a 4-way set-associative cache in order to reduce costs. We call these "modified" victim caches.

As cache size increases, the region of interest moves downward and to the right, so that one would expect something like a low-K MRU cache or a CA cache to best fit the bill.

For the intermediate range of cache sizes, the middle region of Figure 4 looks like the best region to explore. No previous cache designs have used this middle region of space. We can try keeping the overall number of lines constant, dividing the cache into primary (fast-access) and secondary (slower-access) sections, then checking if any design points fall within the region of interest.

If the cache size remains constant, then N in equation 6 must be a power of two. There are only three sets of valid solutions for K and r:

1. If $K = 0$, this is the degenerate case of no secondary cache;
2. If K is non-zero and odd, then $r = 0$ and $K = 2^i - 1$, $i \geq 1$;
3. If K is non-zero and even, then $r \geq 1$ and $K = 2^r$.

The solutions where K is odd correspond to a hybrid cache consisting of a primary cache, with $N/2^i$ lines, and a secondary cache, with $N/2^i$ lines and associativity $K = 2^i - 1$. Since the primary cache is accessed in one cycle and the secondary cache is accessed in two, this hybrid is functionally identical to the MRU cache of Chang, Chao, and So [3].

3.4 The Half-and-Half Cache

The solutions where K is odd have been implemented in real cache designs, but what about the other solutions where $K = 2^r$? This seems to represent a new design option. If the total number of lines in the cache is N, then a cache corresponding to the solution $K = 2^r$ has the following structure:

1. The primary cache contains $\frac{1}{2}N$ lines (half of the total storage);
2. The secondary cache is 2^r-way set-associative with $\frac{1}{2}(N/2^r)$ sets. The total number of lines is $\frac{1}{2}N$ lines.

Because both primary and secondary caches contain half of the lines, we call this the "half-and-half" cache.

The half-and-half cache superficially resembles the MRU cache (indeed, a half-and-half cache with $r = 0$ is identical to a 2-way MRU cache). However, there are several differences (when $r > 0$). A half-and-half cache may have more address conflicts than an MRU cache with the same associativity, because only half of the storage of the half-and-half is associative. However, the benefits of associativity in the MRU cache come at a price: as associativity increases, the size of the primary cache decreases, meaning that fewer and fewer lines can be accessed in one cycle. One would expect that increasing the associativity of the secondary cache portion in the MRU does not necessarily decrease the effective access time.

In the half-and-half, on the other hand, half of the cache is always fast regardless of the degree of associativity. Thus, while the two may have similar average access times for small values of K, larger K values should, in most cases, improve the access time of the half-and-half relative to the MRU cache. Given that the half-and-half cache occupies such points in the HAC design space, one would expect that increasing the associativity of the secondary cache would not penalize the performance. In fact, one would expect the half-and-half with a given associativity would perform better than one with a lower associativity.

In the next section, these performance expectations are confirmed by experimental data.

4 Simulation Experiments

In order to validate our hypothesis, and compare the half-and-half cache to the other cache designs, we ran a series of trace-driven simulation experiments. We programmed a cache simulator to emulate twelve different cache designs with three different sizes, feeding it sixty-two traces. This section describes the cache comparisons, the simulations, the performance metrics, and the results.

4.1 Caches Designs Simulated

We simulated eleven hybrid cache designs in addition to a baseline direct-mapped cache. For each HAC, we assume that the direct-mapped cache is to be improved through the addition of

Param.	DM	VCF	VC4	RVF	RV4	MC2	MC4	MC8	CA†	HH2	HH4	HH8
N_1	N	N	N	N	N	$N/2$	$N/4$	$N/8$	N to $N/2$	$N/2$	$N/2$	$N/2$
N_2	—	1	4	1	4	$N/2$	$N/4$	$N/8$	$N - N_1$	$N/4$	$N/8$	$N/16$
K	—	16	4	16	4	1	3	7	1	2	4	8

†The CA cache is somewhat different from the others, as explained in the list.

Table 1: Size Parameters for Simulated Caches

associativity. The baseline cache has one of three sizes: 8K, 32K or 128K. These represent a range from an average value for present-day on-chip caches to what is likely to be possible in the future. All line sizes are 32 bytes. Thus, if we let N be the number of lines in the baseline cache, then $N \in \{256, 1024, 4096\}$.

The caches are compared in Table 1. The table gives the parameters N_1 and N_2 in terms of N. The total number of lines in the cache is $N_1 + N_2 K$. All caches have a total of N, except for VCF and VC4 (acronyms explained below), which have a full-sized direct-mapped cache *plus* a 16-line victim cache. The following describes the cache configurations:

DM: This is the baseline direct-mapped cache.

VCF: This represents a baseline direct-mapped cache with an additional victim cache. The victim cache is fully-associative and has 16 lines, regardless of the size of the primary cache.

VC4: This is a variation of VCF in which the secondary cache is structured as a 4-way set-associative cache with four sets. The total number of lines (16) is the same as VCF, but the hardware costs will be lower.

RVF: This is a DM/victim cache combination in which the size of the direct-mapped section is *reduced* by the same number of lines as in the victim cache, so that the total number of lines is N. The set-selection function s is implemented using modulo arithmetic. This is not suggested as a practical cache design, but is included because the VCF and VC4 caches have more lines than the others. If these caches perform well, the improvement could be due merely to the increase in the storage capacity of the cache, rather than its organization, so the RVF makes the comparison more "fair" by modeling a cache with the same number of lines.

RV4: This is a 4-way reduced victim cache, the corresponding variant of VC4.

MC2,4,8: These are MRU caches with three different associativities. For example, an associativity of four implies that one line in each set is devoted to direct-mapped access (primary cache) and the other three lines are used for associative accesses in the secondary cache (thus $K = 3$).

CA: This is a column-associative cache. As explained at the end of Section 3.1, the CA cache does not fit into the HAC design space in an exact point. We simulated this cache using the state diagrams from the original paper [2], except that we assume that a secondary hit (e.g., a hit after a rehash) requires only two cycles.[7]

HH2,4,8: These are half-and-half caches. The primary cache takes half of the lines, and the secondary cache is a set-associative cache with 2, 4, or 8 sets.[8]

It is assumed that all hits in the primary cache take one cycle and all hits in the secondary cache take two cycles. For this study, we assumed a miss penalty of 20 in eq. 4.[9]

We have chosen to limit this study to data caches only. We have chosen not to simulate instruction caches at this time since we believe that associativity is less of an issue for instruction caches. Whether the half-and-half cache is beneficial for instruction or unified caches is a topic for further research. Moreover, we have limited ourselves to examining only the miss rates due to read accesses. The reason is that there are many write policies applicable in a HAC, and we decided not to examine these issues by not including misses on writes. Examining write policies in a HAC is another topic for research.

4.2 Trace Sources

In our simulations we used three groups of traces: single-process traces from SPEC89 benchmarks, synthetic multi-process traces from interleaved SPEC89 runs, and ATUM traces representing complete traces captured from a processor by special hardware [1].

The first set of traces (Table 2) were single-process traces generated from the SPEC89 benchmarks. These were run on standard input data as indicated in the table. Because gcc and eqntott create subprocesses, we only simulated the 19 calls to the C compiler in gcc, and the main process (after pre-processing) in eqntott. Two versions of eqntott were compiled: the standard version and a "fast" version in which the qsort routine supplied with the eqntott code was replaced with the qsort routine from the espresso benchmark. All programs were run to completion, in order to account for the variations in memory use patterns throughout a

[7] The published design for the column-associative cache has a secondary hit time of four cycles, but this is because it uses a low-cost implementation which assumes only one access path into the cache. Thus, extra cycles are needed whenever it needs to swap data. However, the column-associative cache can be viewed as a specialized 2-way set-associative cache; thus operations can complete in two cycles instead of four if the cache is augmented with hardware similar to that used by the half-and-half.

[8] There is no HH1 because that would be equivalent to MC2.

[9] Of course, this value is arbitrary and varying it can affect the differences between HAC's. For instance, the relative performance of a direct-mapped cache (with a higher miss rate but no secondary hits) would increase relative to associative caches (with more secondary hits but fewer misses) if the miss penalty were to drop. Nevertheless, this value is similar to that used in other cache comparisons, and represents a middle ground between high-end machines with fast secondary caches and low-end machines with on-chip caches only.

Program	Input	Ops $\times 10^6$	% Ld.	% St.	Comp. misses	Input	Ops $\times 10^6$	% Ld.	% St.	Comp. misses
gcc	cexp	18.2	14.9	8.4	6,051	jump	40.2	15.2	7.9	9,714
	dbxout	43.1	14.7	8.8	15,961	optabs	50.2	15.3	7.5	17,803
	emit-rtl	31.2	15.1	7.7	6,511	print-tree	41.3	14.7	8.9	14,068
	explow	54.8	14.2	8.8	22,224	recog	59.2	14.9	8.0	10,863
	gcc	42.7	15.2	8.0	9,019	regclass	27.9	15.2	7.8	8,165
	genoutput	22.1	15.2	7.8	8,434	stmt	113.7	15.1	7.9	18,670
	genrecog	30.7	15.1	8.0	9,863	toplev	49.8	15.1	8.0	16,204
	insn-emit	48.3	15.2	6.9	8,415	tree	49.3	15.1	7.5	9,246
	insn-recog	148.6	14.7	9.3	30,781	varasm	57.0	15.3	7.6	14,235
	integrate	60.2	15.1	8.0	17,217	TOTAL	988.5	15.0	8.2	253,444
espresso	bca	435.9	23.2	2.2	6,829	ti	696.1	21.1	5.7	30,750
	cps	570.5	21.3	4.6	6,579	tial	1,124.0	20.7	4.6	2,374
						TOTAL	2,826.5	21.3	4.5	46,532
xlisp						(queens 9)	5,155.5	20.9	9.0	1,710
eqntott						int3	1,196.7	16.9	1.3	52,100
eqn-fast						int3	161.5	18.8	9.1	52,127
SYMB						TOTAL	10,328.6	19.9	6.8	405,913
doduc	tiny	46.2	22.4	6.5	1,824	ref	1,348.6	22.5	6.4	1,886
	small	234.9	22.5	6.4	1,847	TOTAL	1,629.7	22.5	6.4	5,557
fpppp	4 atoms	147.1	37.0	7.8	2,536	8 atoms	1,689.4	37.0	7.3	2,845
						TOTAL	1,836.5	37.0	7.4	5,381
Good FP						TOTAL	3,466.2	30.2	6.9	10,938
spice2g6						short	2,368.4	29.1	4.2	16,429
nasa7	mxm	859.4	30.9	6.3	14,590	gmt	901.6	32.8	11.4	63,537
	fft	1,113.2	20.6	20.5	16,891	emi	569.1	21.9	6.8	65,946
	cho	979.5	32.3	13.6	46,012	vpe	825.4	30.0	8.8	53,506
	btr	994.7	36.3	6.7	85,731	TOTAL	6,243.0	29.5	11.2	346,213
matrix						300	1,704.3	25.8	12.7	67,964
tomcatv	N=33	20.4	30.2	7.7	2,178	N=129	322.4	31.1	7.6	29,151
	N=65	80.5	30.8	7.7	7,590	N=257	1,300.3	31.5	7.5	115,285
						TOTAL	1,723.7	31.4	7.6	154,204
Bad FP						TOTAL	12,039.4	29.1	9.5	584,810
SPEC89						TOTAL	25,834.2	25.6	8.1	1,001,661

Table 2: Symbolic SPEC89 Benchmarks

program's execution. All simulations were performed on Sparc computers, and the benchmarks were compiled with the standard Sparc Fortran compiler version 2.0.1 or with the GNU C Compiler version 2.4.5.

The benchmarks have been divided into three groups: the four symbolic benchmarks (SYMB), two numerical benchmarks (doduc and fpppp) which showed relatively good cache performance (GOOD), and the remaining numerical benchmarks (spice2g6, nasa7, matrix, and tomcatv)

Trace	Input	Ops $\times 10^6$	% Loads	% Stores	Comp. miss.
SYM4	gcc,espresso,li,eqntott	400.0	20.3	6.1	87,451
FP4	spice2g6,doduc,gmt,fpppp	400.0	24.7	7.6	15,287
MIX6	gcc,emi,eqntott,fpppp,espresso,tomcatv	600.0	24.1	5.9	169,495

Table 3: Multitasking SPEC89 Benchmarks

which had relatively poor cache performance (BAD). The total for each group includes every input run on that benchmark, and is shown in the table.

There were two major weaknesses with these traces. First, they only involved single processes, and did not measure the effects of multitasking. Second, our trace simulator was not able to trace within system calls, removing the effects of system calls from consideration.

To measure the effects of multitasking, we decided to generate a few extra traces by merging together several SPEC89 traces. Three multiprocess streams were generated (see Table 3). The first (SYM4) combined the four symbolic SPEC89 benchmarks: gcc (input insn-recog.i), espresso (input cps.in), li, and eqntott (original version). The second (FP4) interleaved four numerically-intensive SPEC89 benchmark programs: spice2g6, doduc (input ref), the GMT subsection of the nasa7 benchmark, and fpppp (NATOMS=4). The last (MIX6) is a mixture of numerical and symbolic code: gcc (input stmt.i), the EMI subsection of the nasa7 benchmark, eqntott (fast version), fpppp (NATOMS=8), espresso (input tial.in), and tomcatv (N=129). When forming these mixtures, we looked for benchmarks which did not substantially favor any particular cache organization in the single-process traces. We halted each trace after 100 million operations.

Each stream was generated by assuming round-robin scheduling. For each of the three streams, at each of the three cache sizes, one multitrace was run in which each trace was run to completion before switching to another, i.e., there was no interleaving. Then five additional traces were run, in which the number of operations executed before switching to another trace was one of $\{1{,}000{,}000, 100{,}000, 10{,}000, 1{,}000, 100\}$. It is important to consider different levels of granularity, since as processor speeds continue to increase, there can be a concomitant increase in the number of instructions executed before doing a process switch. Moreover, simulating different levels of granularity can provide insights to the cache performance for systems which are frequently or less frequently interrupted.

The multitasking traces still have the problem of lacking system calls, so we used ATUM traces [1] for our third series (see Table 4). These traces are based on actual VAX execution traces, and include system calls. Some of them are also multitasked. However, not as much weight should be placed on these traces, since they were generated from a CISC processor (VAX), and may not accurately mimic access patterns found on today's prevalent RISC processors. (Of course, a CISC designer may also wish to use the HAC framework to design an

Input	Total ops	% Loads	% Stores	Comp. misses	Input	Total ops	% Loads	% Stores	Comp. misses
dec0	178,959	58.2	40.5	2,356	memxx	225,799	55.5	43.9	1,430
fora	188,135	56.5	42.1	2,647	mul2	163,670	55.8	43.3	1,455
forf	177,297	58.9	39.1	3,561	mul8	229,977	60.1	38.6	3,009
fsxzz	116,105	65.9	32.6	1,726	pasc	229,065	53.5	46.0	1,222
ivex	138,458	67.6	29.7	3,806	savec	88,877	81.5	17.6	772
linp	202,426	89.8	9.5	1,528	spic	222,995	60.7	38.9	991
lisp	121,604	80.8	18.5	841	ue02	157,837	60.2	37.7	3,298
macr	154,126	60.8	37.1	3,121	TOTAL	2,595,330	63.1	35.6	31,763

Table 4: ATUM Traces

on-chip cache.) These traces were included primarily to show the possible effects of system calls.

4.3 Results

Results are shown in the graphs in Figures 6–12. We use the average cache access time as defined in equation 4 in Section 2.4 as our primary metric in comparing caches. The first three figures show the results for the SPEC89 benchmarks. Due to space limitations, we combined the results for all inputs on a particular benchmark into one set of cost values. The results were combined by adding the counts for each type of hit or miss (as if the different cases were run in sequence and the cache were flushed between jobs). The two different versions of eqntott were combined as well. Figures 6–8 also show the aggregate results for the three categories of SPEC89 benchmarks, the aggregate results for the entire SPEC89 suite, and the combined results for the ATUM traces.

A few observations can be made by comparing the caches for different sizes. Figure 6 shows the greatest performance variations between hybrid caches. These results demonstrate the benefits of keeping most of the cache lines in the fast-access section. For the SPEC89 traces, the MRU caches have the worst performance, especially those with high associativities (at the right side of the design space). Victim caches and half-and-half caches perform much better. For the ATUM traces, however, high associativity clearly improves cache performance; even the MC8 performs reasonably well.

Thus, it would seem that adding associativity can help to improve cache performance, even when it comes at a higher hit cost. It is apparent from the graphs that associativity is less of a benefit for large caches. This is a well-known observation about caches. For large caches, column-associativity often produces equal or better results, which shows the advantages of allowing the cache to become effectively direct-mapped when the cache is large enough to

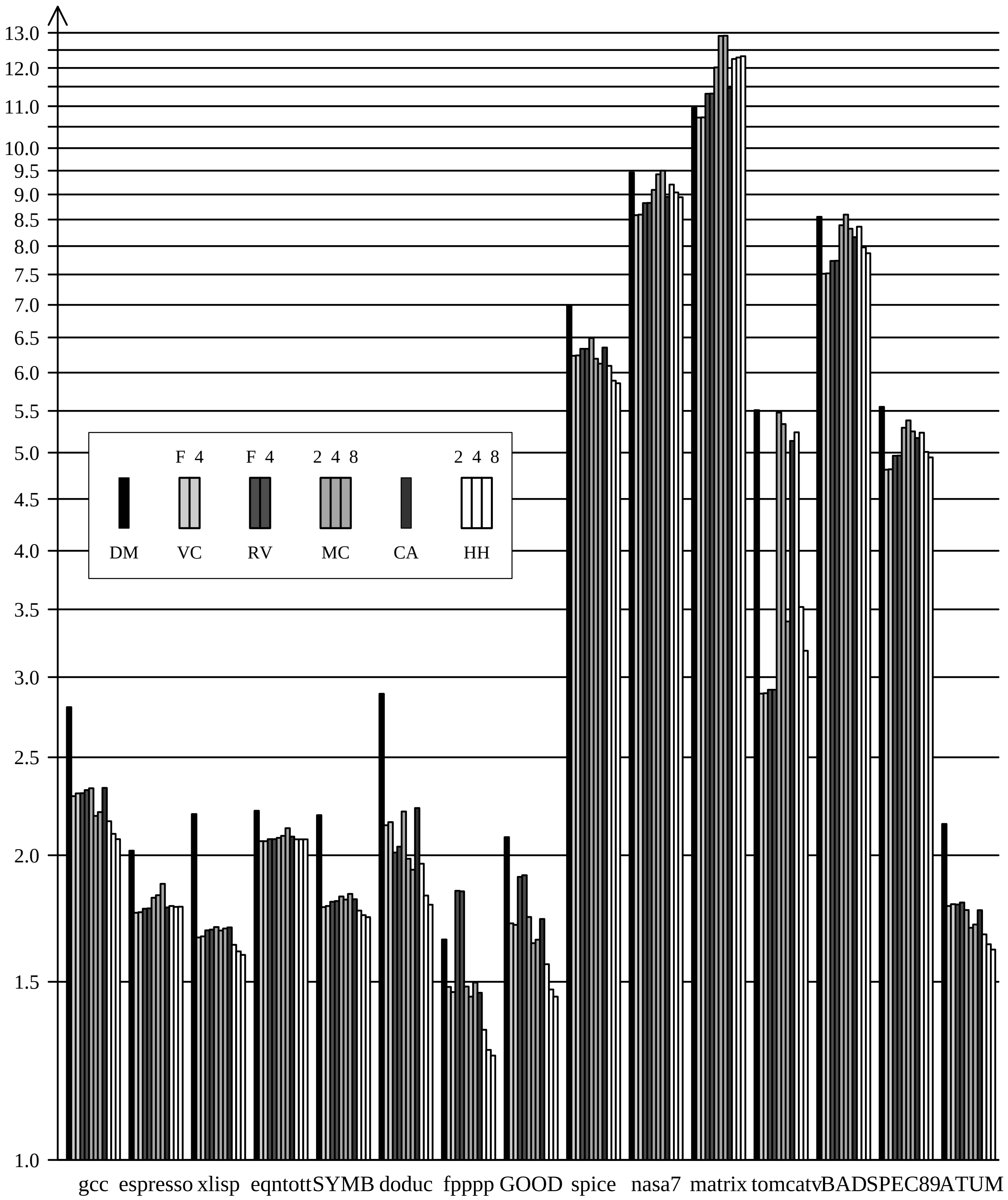

Figure 6: Combined Results for 8K Cache

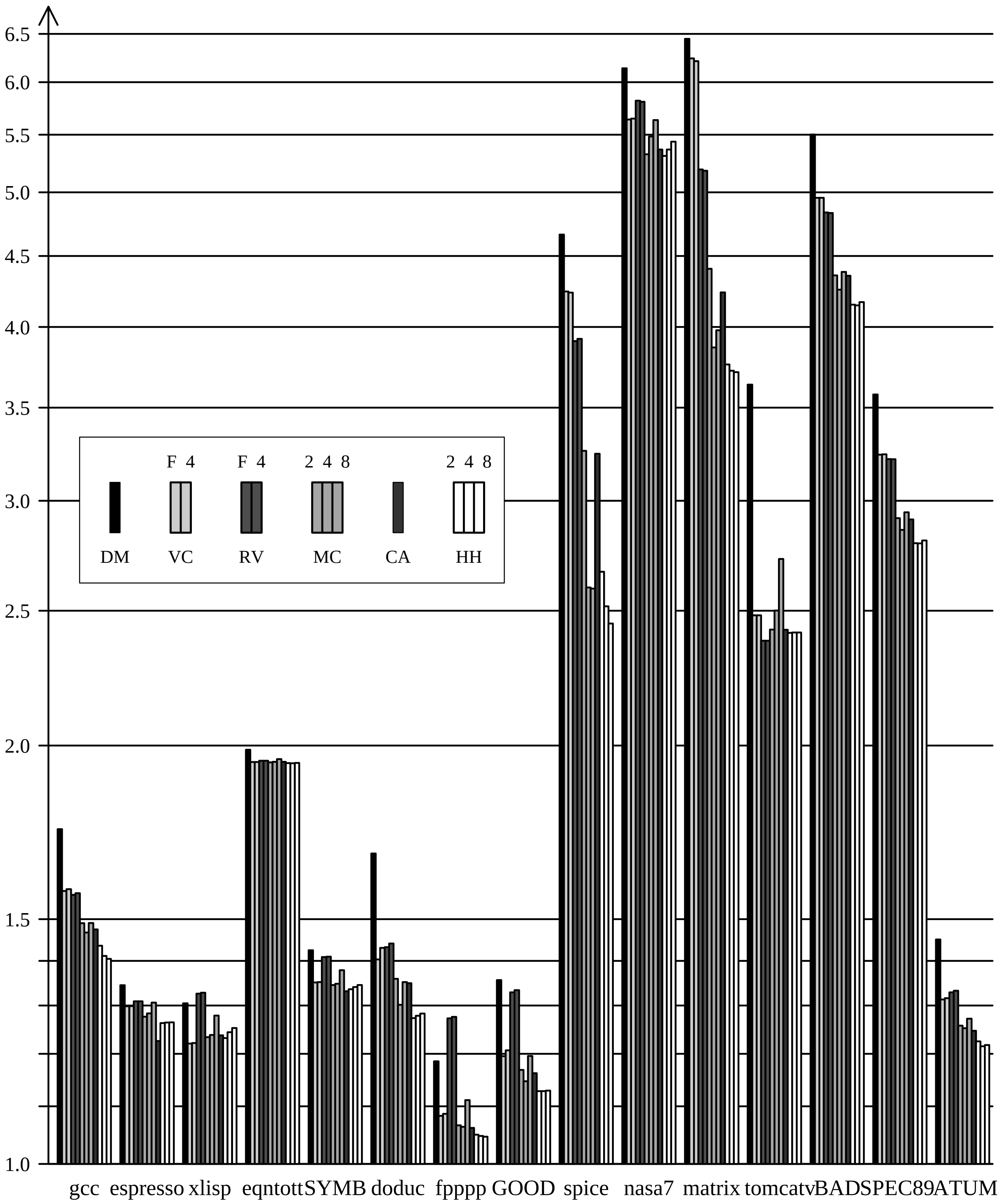

Figure 7: Combined Results for 32K Cache

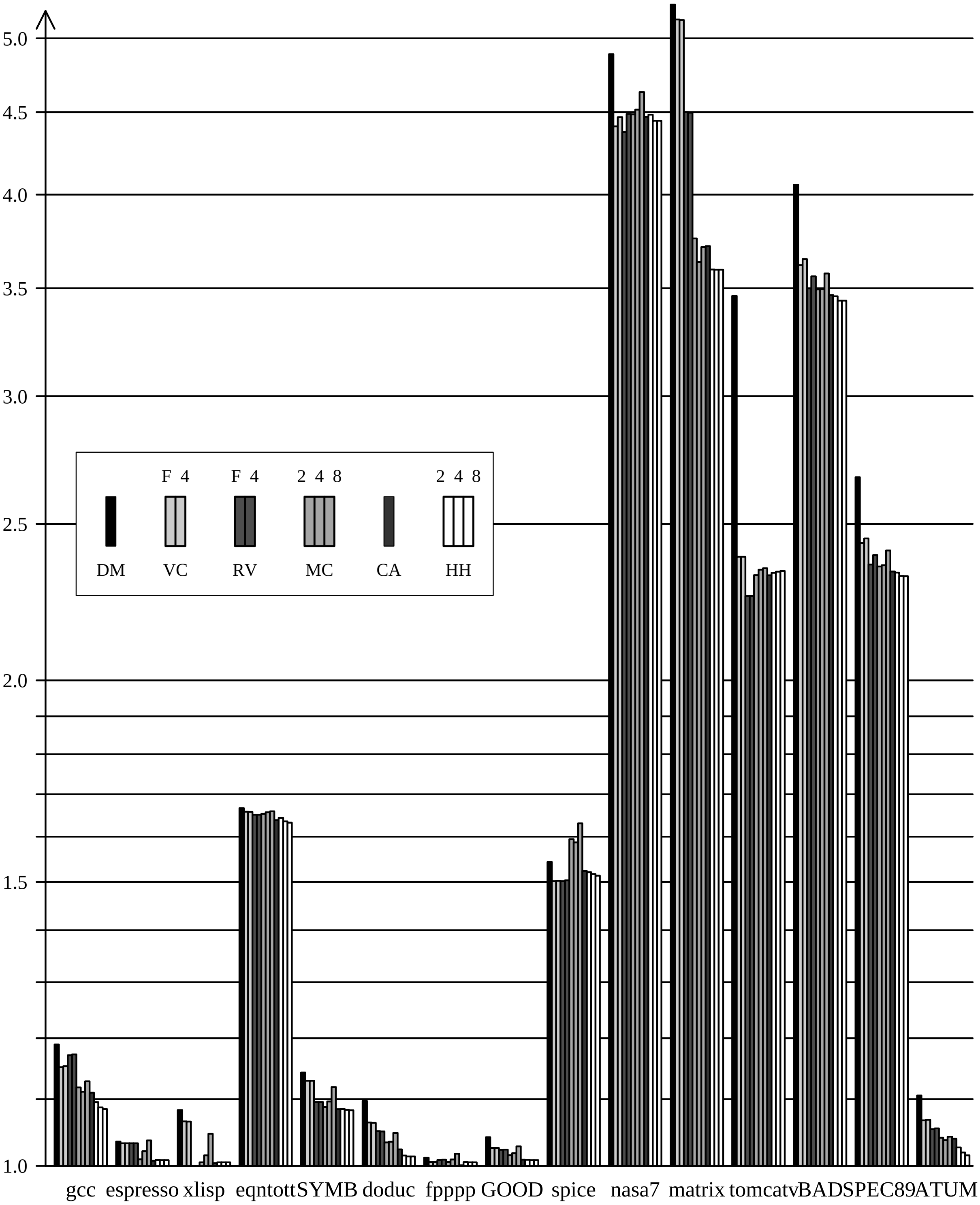

Figure 8: Combined Results for 128K Cache

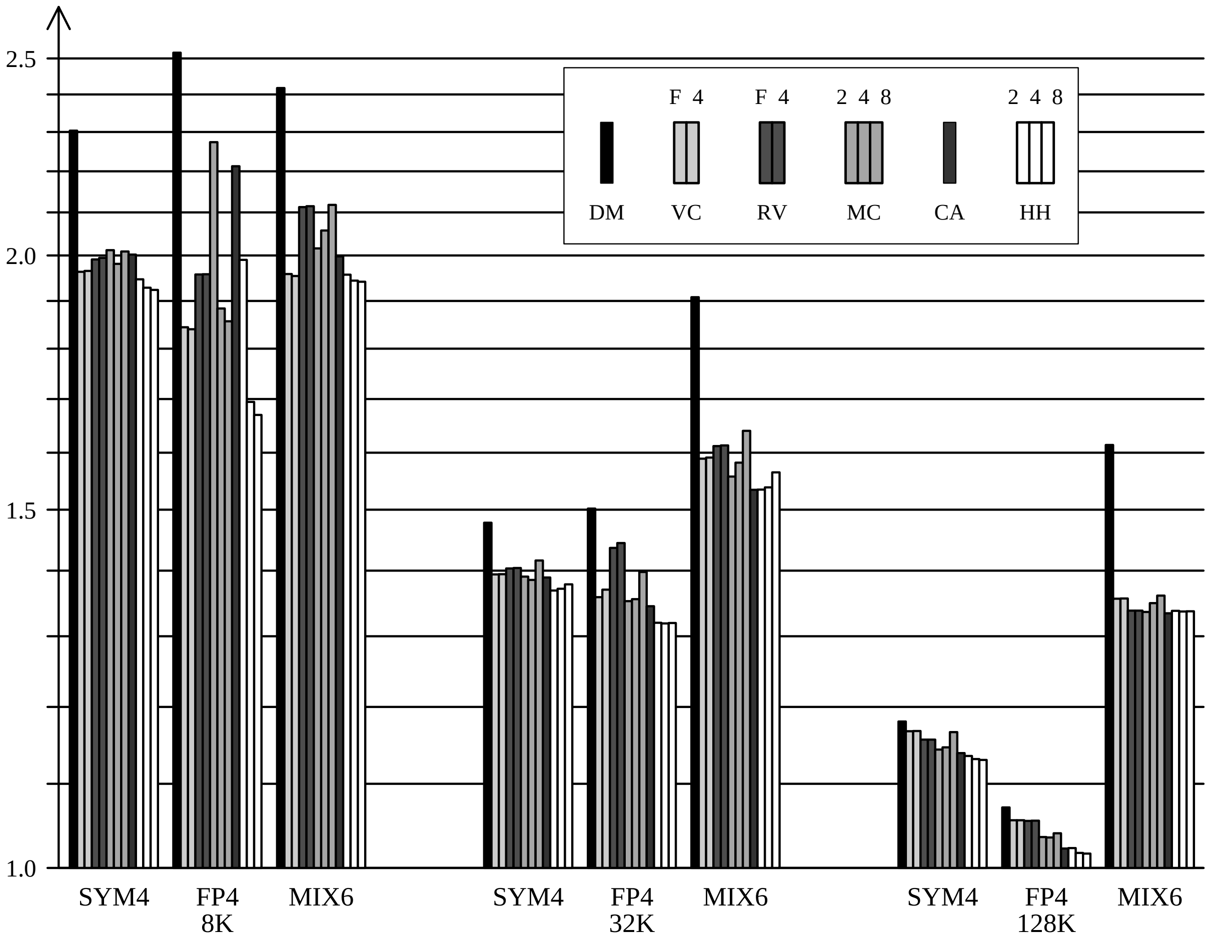

Figure 9: Baseline Results for Multitasking Traces

eliminate most conflicts. Other low-associativity caches such as the MC2 and the HH2 also perform well.

For intermediate cache sizes, how much associativity is needed? In most cases, adding more associativity seems to improve the performance, because HH8 usually has the best performance. However, in most cases, the improvement from $K = 2$ to $K = 8$ is slight, and in the traces with large differences, most of the improvement comes with increasing K from 2 to 4. Unlike the half-and-half cache, the MRU cache usually does not benefit from greater associativity, even though the tables in the appendix show miss rates declining with increasing K. This is because the benefits of greater associativity are usually canceled by the extra costs of secondary hits, since as K increases, the amount of the cache dedicated to fast accesses decreases.

Reducing the size of the direct-mapped cache, as in the reduced victim caches (RVF and RV4), generally increases the average access costs as expected, especially for smaller caches. There are some anomalies, particularly in the larger caches, in which the *reduced* caches have

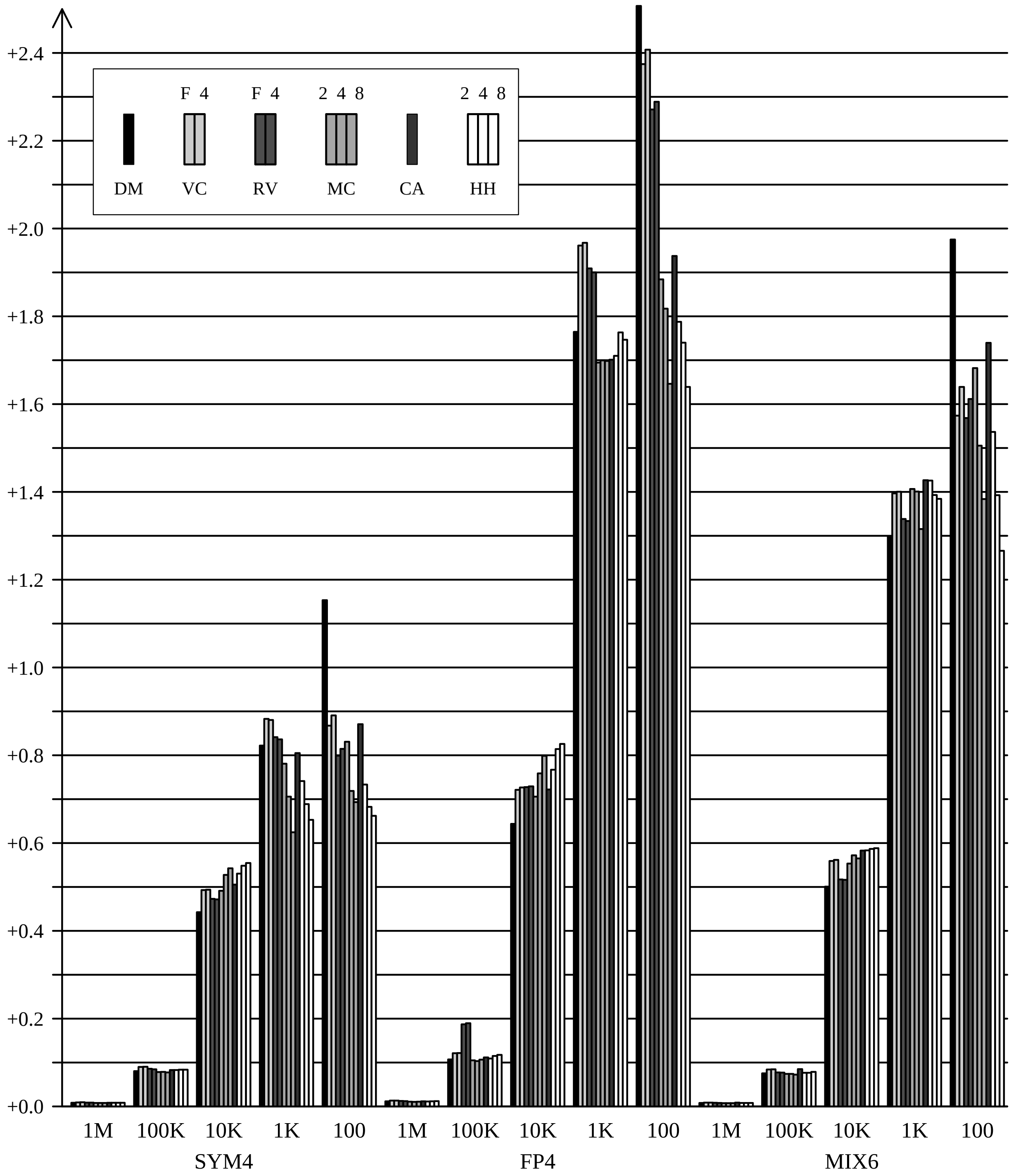

Figure 10: Multitasking Results for 8K Cache

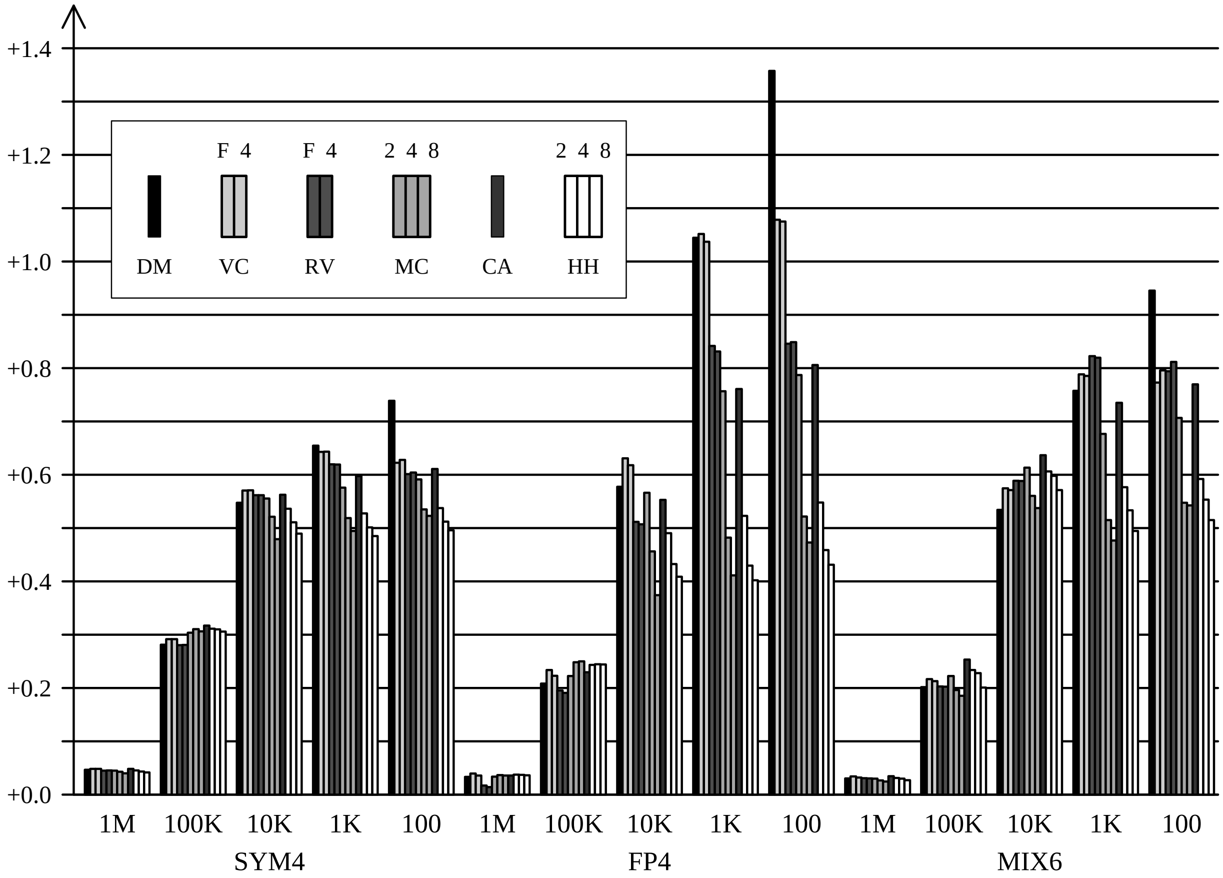

Figure 11: Multitasking Results for 32K Cache

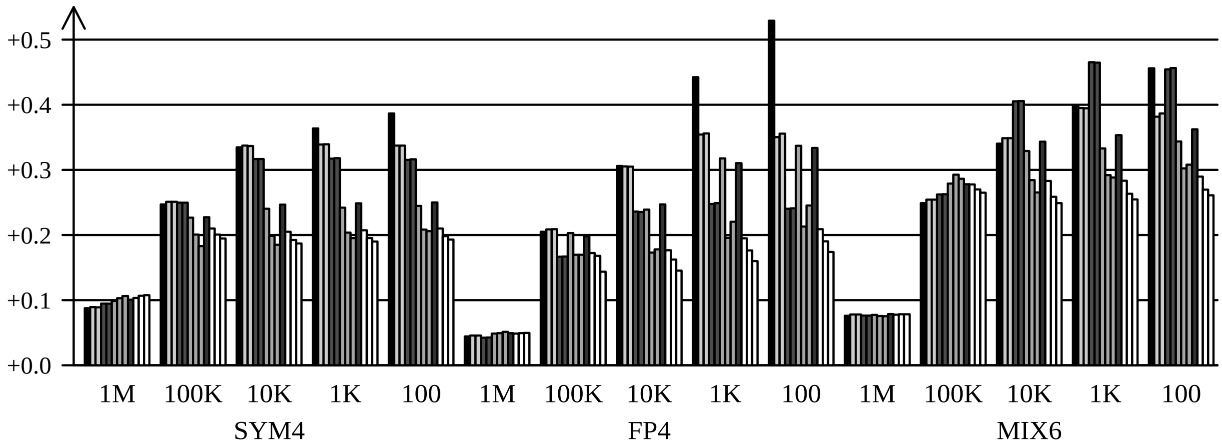

Figure 12: Multitasking Results for 128K Cache

better performance. This shows that the data access patterns of a particular program have an important impact on cache performance. In each of the VC and RV cache sets, there is very little difference in performance between the fully-associative victim cache and the set-associative victim cache, suggesting that a fully-associative victim cache is probably not worth the extra hardware cost.

Figure 9 shows the results of the simulations of the non-interleaved versions of the multi-tasking traces. Figures 10–12 show what happens when context-switching is introduced. These figures show the *relative* performance of each cache under multitasking, i.e., each bar shows the *difference* between the average read cost for the interleaved trace and the non-interleaved trace for the same cache.

5 Conclusions

In this paper, we have proposed a unified framework for the design of Hybrid Access Caches (HACs) as on-chip caches. Under our model, a rich design space is structured in a way which helps the cache designers systematically to explore, experiment and compare various combinations of hybrid cache designs in order to meet performance and cost requirements. Using this framework, existing hybrid access caches can be viewed as points within this design space, and much insight can be obtained by comparing their tradeoffs in the space.

By exploring this framework, we conceived a novel HAC organization called the Half-and-Half cache. It achieves a good balance of the advantages from both the speed of direct-mapped caches and the toleration of access interference of set associative caches. We presented simulation results which have illustrated the use of this framework, and the results confirm that the half-and-half cache has better performance than other hybrid access cache designs for many applications.

A Experimental Data

This appendix gives tabular results for the cache simulations. Tables are divided into groups of three, each group containing one table for each cache size ({8K, 32K, 128K}). Each table lists, for each cache design, the percentages of reads which are primary hits and misses. (A dash means there were no misses; an entry of 0.000 means there were misses, but fewer than 0.0005%.) The percentage of secondary hits can be found by subtracting these two percentages from 100%. Each table also lists the actual cost values, and underlines the lowest value found.

These tables show one additional cache design which was left out of Section 4 for space reasons. The simulations included a fully associative (FA) cache. To be consistent with our model, we represented the FA cache as a 1-line primary cache and an $(N-1)$-line secondary

Input	Rate	FA	DM	VCF	VC4	RVF	RV4	MC2	MC4	MC8	CA	HH2	HH4	HH8
gcc: insn-emit	Prim.	26.802	93.014	93.014	93.014	92.626	92.626	88.784	82.922	75.028	92.015	88.784	88.784	88.784
	Misses	2.910	6.986	4.453	4.547	4.582	4.632	4.521	3.442	3.154	4.652	3.613	3.295	3.156
	Cost	2.285	2.397	1.916	1.934	1.944	1.954	1.971	1.825	1.849	1.964	1.799	1.738	1.712
gcc: insn-recog	Prim.	29.688	90.570	90.570	90.570	90.686	90.686	86.695	81.622	75.435	89.414	86.695	86.695	86.695
	Misses	5.768	9.430	7.070	7.130	6.996	7.050	6.949	6.120	5.891	7.111	6.221	5.940	5.855
	Cost	2.799	2.886	2.438	2.449	2.422	2.433	2.453	2.347	2.365	2.457	2.315	2.262	2.245
gcc: TOTAL	Prim.	27.589	90.991	90.991	90.991	90.937	90.937	86.499	81.068	73.960	89.772	86.499	86.499	86.499
	Misses	4.694	9.009	6.310	6.382	6.383	6.472	6.288	5.257	4.981	6.470	5.402	5.082	4.949
	Cost	2.616	2.802	2.289	2.303	2.303	2.320	2.330	2.188	2.207	2.332	2.161	2.101	2.075
espresso: bca	Prim.	8.065	91.710	91.710	91.710	91.591	91.591	89.597	86.429	81.836	91.334	89.597	89.597	89.597
	Misses	7.659	8.290	7.376	7.379	7.499	7.500	7.550	7.447	7.493	7.444	7.371	7.351	7.378
	Cost	3.375	2.658	2.484	2.485	2.509	2.509	2.538	2.551	2.605	2.501	2.505	2.501	2.506
espresso: cps	Prim.	18.946	94.841	94.841	94.841	94.898	94.898	93.099	90.242	86.309	94.547	93.099	93.099	93.099
	Misses	3.632	5.159	3.715	3.720	3.903	3.908	3.795	3.647	3.592	3.761	3.616	3.533	3.514
	Cost	2.501	2.032	1.757	1.758	1.793	1.793	1.790	1.790	1.819	1.769	1.756	1.740	1.737
espresso: ti	Prim.	27.171	95.199	95.199	95.199	94.297	94.297	93.365	90.418	84.340	94.750	93.365	93.365	93.365
	Misses	3.551	4.801	3.495	3.512	3.584	3.596	3.700	3.504	3.490	3.609	3.461	3.405	3.404
	Cost	2.403	1.960	1.712	1.715	1.738	1.740	1.769	1.762	1.820	1.738	1.724	1.713	1.713
espresso: tial	Prim.	17.818	96.105	96.105	96.105	96.526	96.526	93.734	90.937	86.002	95.473	93.734	93.734	93.734
	Misses	2.521	3.895	2.239	2.254	2.241	2.248	2.536	2.532	2.528	2.342	2.405	2.444	2.453
	Cost	2.301	1.779	1.464	1.467	1.461	1.462	1.545	1.572	1.620	1.490	1.520	1.527	1.529
xlisp: (queens 9)	Prim.	34.598	94.014	94.014	94.014	95.324	95.324	91.320	85.779	82.084	93.407	91.320	91.320	91.320
	Misses	2.563	5.986	3.152	3.176	3.366	3.379	3.221	2.859	2.707	3.324	2.867	2.739	2.673
	Cost	2.141	2.197	1.659	1.663	1.686	1.689	1.699	1.685	1.694	1.698	1.631	1.607	1.595
eqntott: int3	Prim.	3.626	93.674	93.674	93.674	93.500	93.500	92.787	91.211	87.727	93.522	92.787	92.787	92.787
	Misses	5.557	6.326	5.567	5.567	5.596	5.597	5.598	5.564	5.562	5.674	5.571	5.572	5.573
	Cost	3.020	2.265	2.121	2.121	2.128	2.128	2.136	2.145	2.180	2.143	2.131	2.131	2.131
eqn-fast: int3	Prim.	12.821	95.664	95.664	95.664	95.397	95.397	94.381	90.373	85.039	95.400	94.381	94.381	94.381
	Misses	3.313	4.336	3.417	3.421	3.550	3.555	3.470	3.343	3.334	3.510	3.367	3.355	3.352
	Cost	2.501	1.867	1.693	1.693	1.721	1.722	1.715	1.731	1.783	1.713	1.696	1.694	1.693

Table 5: Data for Symbolic Benchmarks on 8K Caches

Input	Rate	FA	DM	VCF	VC4	RVF	RV4	MC2	MC4	MC8	CA	HH2	HH4	HH8
gcc: insn-emit	Prim.	27.242	97.702	97.702	97.702	97.442	97.442	95.484	92.959	88.753	97.353	95.484	95.484	95.484
	Misses	0.426	2.298	1.491	1.519	1.607	1.627	1.049	0.858	0.707	1.080	0.836	0.690	0.592
	Cost	1.808	1.460	1.306	1.312	1.331	1.335	1.244	1.234	1.247	1.232	1.204	1.176	1.158
gcc: insn-recog	Prim.	29.605	95.118	95.118	95.118	95.464	95.464	93.113	90.573	86.716	94.714	93.113	93.113	93.113
	Misses	2.838	4.882	4.094	4.104	3.731	3.751	3.255	2.997	2.820	3.274	2.978	2.807	2.769
	Cost	2.243	1.976	1.827	1.829	1.754	1.758	1.687	1.664	1.669	1.675	1.635	1.602	1.595
gcc: TOTAL	Prim.	27.427	96.295	96.295	96.295	96.249	96.249	93.880	91.007	86.530	95.832	93.880	93.880	93.880
	Misses	1.837	3.705	2.815	2.840	2.758	2.780	2.256	1.985	1.873	2.278	1.970	1.844	1.806
	Cost	2.075	1.741	1.572	1.577	1.562	1.566	1.490	1.467	1.491	1.475	1.436	1.411	1.404
espresso: bca	Prim.	8.065	94.904	94.904	94.904	94.746	94.746	92.981	91.710	89.597	93.656	92.981	92.981	92.981
	Misses	5.929	5.096	4.873	4.876	4.997	4.997	5.970	5.949	5.941	4.875	5.948	5.940	5.933
	Cost	3.046	2.019	1.977	1.977	2.002	2.002	2.204	2.213	2.233	1.990	2.200	2.199	2.197
espresso: cps	Prim.	18.946	99.155	99.155	99.155	99.292	99.292	97.227	94.841	93.099	99.064	97.227	97.227	97.227
	Misses	0.674	0.845	0.423	0.427	0.416	0.419	0.307	0.275	0.308	0.314	0.271	0.286	0.316
	Cost	1.939	1.169	1.089	1.090	1.086	1.087	1.086	1.104	1.128	1.069	1.079	1.082	1.088
espresso: ti	Prim.	27.171	98.615	98.615	98.615	98.558	98.558	97.416	95.199	93.365	98.391	97.416	97.416	97.416
	Misses	0.686	1.385	1.125	1.127	1.172	1.172	0.690	0.667	0.700	0.659	0.617	0.635	0.623
	Cost	1.859	1.277	1.228	1.228	1.237	1.237	1.157	1.175	1.199	1.141	1.143	1.146	1.144
espresso: tial	Prim.	17.818	99.069	99.069	99.069	98.927	98.927	97.563	96.105	93.734	99.023	97.563	97.563	97.563
	Misses	0.013	0.931	0.781	0.783	0.841	0.840	0.123	0.015	0.013	0.106	0.016	0.014	0.014
	Cost	1.824	1.186	1.158	1.158	1.170	1.170	1.048	1.042	1.065	1.030	1.027	1.027	1.027
xlisp: (queens 9)	Prim.	34.598	98.476	98.476	98.476	97.983	97.983	95.307	94.014	91.320	97.998	95.307	95.307	95.307
	Misses	1.071	1.524	1.082	1.087	1.610	1.619	0.984	0.938	1.008	1.145	0.973	1.035	1.082
	Cost	1.858	1.305	1.221	1.222	1.326	1.328	1.234	1.238	1.278	1.238	1.232	1.244	1.253
eqntott: int3	Prim.	3.626	94.754	94.754	94.754	94.715	94.715	94.232	93.674	92.787	94.548	94.232	94.232	94.232
	Misses	5.027	5.246	5.055	5.055	5.076	5.075	5.037	5.024	5.022	5.065	5.031	5.031	5.037
	Cost	2.919	2.049	2.013	2.013	2.017	2.017	2.015	2.018	2.026	2.017	2.014	2.014	2.015
eqn-fast: int3	Prim.	12.821	97.195	97.195	97.195	97.253	97.253	96.516	95.664	94.381	96.977	96.516	96.516	96.516
	Misses	2.231	2.805	2.469	2.469	2.487	2.487	2.322	2.248	2.235	2.341	2.265	2.257	2.255
	Cost	2.296	1.561	1.497	1.497	1.500	1.500	1.476	1.470	1.481	1.475	1.465	1.464	1.463

Table 6: Data for Symbolic Benchmarks on 32K Caches

Input	Rate	FA	DM	VCF	VC4	RVF	RV4	MC2	MC4	MC8	CA	HH2	HH4	HH8
gcc:	Prim.	27.242	99.618	99.618	99.618	99.292	99.292	98.881	97.702	95.484	99.571	98.881	98.881	98.881
insn-emit	Misses	0.018	0.382	0.251	0.251	0.380	0.388	0.105	0.028	0.019	0.112	0.031	0.021	0.019
	Cost	1.731	1.076	1.051	1.051	1.079	1.081	1.031	1.028	1.049	1.025	1.017	1.015	1.015
gcc:	Prim.	29.605	98.061	98.061	98.061	97.894	97.894	96.709	95.118	93.113	97.774	96.709	96.709	96.709
insn-recog	Misses	1.348	1.939	1.760	1.773	1.715	1.714	1.418	1.335	1.300	1.383	1.276	1.230	1.221
	Cost	1.960	1.388	1.354	1.356	1.347	1.347	1.302	1.302	1.316	1.285	1.275	1.267	1.265
gcc:	Prim.	27.427	99.053	99.053	99.053	98.739	98.739	97.897	96.295	93.880	98.895	97.897	97.897	97.897
TOTAL	Misses	0.347	0.947	0.747	0.755	0.835	0.842	0.513	0.393	0.353	0.523	0.391	0.348	0.335
	Cost	1.792	1.189	1.151	1.153	1.171	1.173	1.119	1.112	1.128	1.110	1.095	1.087	1.085
espresso:	Prim.	8.065	99.182	99.182	99.182	99.169	99.169	99.006	94.904	92.981	99.174	99.006	99.006	99.006
bca	Misses	0.004	0.818	0.787	0.787	0.791	0.791	0.015	0.004	0.004	0.015	0.004	0.003	0.003
	Cost	1.920	1.164	1.158	1.158	1.159	1.159	1.013	1.052	1.071	1.011	1.011	1.011	1.011
espresso:	Prim.	18.946	99.972	99.972	99.972	99.973	99.973	99.541	99.155	97.227	99.962	99.541	99.541	99.541
cps	Misses	0.009	0.028	0.020	0.020	0.021	0.021	0.014	0.008	0.008	0.019	0.007	0.008	0.008
	Cost	1.812	1.006	1.004	1.004	1.004	1.004	1.007	1.010	1.029	1.004	1.006	1.006	1.006
espresso:	Prim.	27.172	99.861	99.861	99.861	99.869	99.869	99.317	98.641	97.431	99.842	99.317	99.317	99.317
ti	Misses	0.078	0.139	0.110	0.110	0.109	0.109	0.088	0.079	0.078	0.093	0.080	0.078	0.077
	Cost	1.743	1.028	1.022	1.022	1.022	1.022	1.024	1.029	1.040	1.019	1.022	1.022	1.022
espresso:	Prim.	17.818	100.000	100.000	100.000	99.999	99.999	99.987	99.069	97.563	100.000	99.987	99.987	99.987
tial	Misses	—	0.000	0.000	0.000	0.000	0.000	0.000	—	—	0.000	—	—	—
	Cost	1.822	1.000	1.000	1.000	1.000	1.000	1.000	1.009	1.024	1.000	1.000	1.000	1.000
xlisp:	Prim.	34.598	99.586	99.586	99.586	100.000	100.000	99.480	98.476	95.307	99.586	99.480	99.480	99.480
(queens 9)	Misses	—	0.414	0.323	0.323	—	—	0.000	0.000	—	0.000	0.000	—	—
	Cost	1.654	1.083	1.066	1.065	1.000	1.000	1.005	1.015	1.047	1.004	1.005	1.005	1.005
eqntott:	Prim.	3.626	96.376	96.376	96.376	96.417	96.417	95.395	94.754	94.232	95.906	95.395	95.395	95.395
int3	Misses	3.493	3.624	3.580	3.579	3.540	3.540	3.513	3.506	3.492	3.451	3.464	3.419	3.402
	Cost	2.627	1.725	1.716	1.716	1.709	1.708	1.714	1.719	1.721	1.697	1.704	1.696	1.692
eqn-fast:	Prim.	12.821	98.609	98.609	98.609	98.612	98.612	97.977	97.195	96.516	98.422	97.977	97.977	97.977
int3	Misses	1.100	1.391	1.336	1.336	1.330	1.330	1.205	1.139	1.100	1.225	1.147	1.110	1.103
	Cost	2.081	1.278	1.268	1.268	1.267	1.267	1.249	1.244	1.244	1.249	1.238	1.231	1.230

Table 7: Data for Symbolic Benchmarks on 128K Caches

cache. Because so little of the FA cache is primary, the primary hit rates tend to be low, and the access costs correspondingly high, since most *successful* reads cost two cycles.

We included this simulation because when comparing caches with different levels of associativity for a given benchmark, it is useful to see how many hits occur when there are no conflict misses. In a few cases, the FA cache performs worse than the DM, generally in problems with large working sets that have poor cache performance due to capacity misses.

Because there were so many runs of gcc, we decided to leave out most of the data for the individual input cases, and simply present the aggregate figure. However, we show the individual results for two of the cases, insn-emit and insn-recog, because these produced, for all cache sizes, the lowest and highest best-cache access cost, respectively.

In the final three tables, we list the results for the multitasking simulations for all six context-switching times. In each group, the first row of data (100,000,000) is the baseline non-interleaved case (see Figure 9).

Input	Rate	FA	DM	VCF	VC4	RVF	RV4	MC2	MC4	MC8	CA	HH2	HH4	HH8
doduc:	Prim.	27.784	90.655	90.655	90.655	91.909	91.909	86.019	79.941	71.712	89.457	86.019	86.019	86.019
tiny	Misses	3.263	9.345	5.376	5.456	4.979	5.012	5.502	3.980	3.334	5.780	4.186	3.485	3.295
	Cost	2.342	2.869	2.115	2.130	2.027	2.033	2.185	1.957	1.916	2.204	1.935	1.802	1.766
doduc:	Prim.	27.978	90.560	90.560	90.560	91.747	91.747	85.900	79.773	71.481	89.357	85.900	85.900	85.900
small	Misses	3.335	9.440	5.508	5.564	5.097	5.142	5.653	4.099	3.451	5.921	4.308	3.591	3.375
	Cost	2.354	2.888	2.141	2.152	2.051	2.059	2.215	1.981	1.941	2.231	1.960	1.823	1.782
doduc:	Prim.	28.020	90.554	90.554	90.554	91.889	91.889	85.868	79.764	71.436	89.347	85.868	85.868	85.868
ref	Misses	3.358	9.446	5.517	5.600	4.868	5.032	5.622	4.127	3.418	5.900	4.328	3.604	3.408
	Cost	2.358	2.889	2.143	2.158	2.006	2.037	2.210	1.987	1.935	2.228	1.964	1.826	1.789
fpppp:	Prim.	7.283	96.134	96.134	96.134	90.966	90.966	90.967	80.816	70.368	95.276	90.967	90.967	90.967
4 atoms	Misses	0.912	3.866	2.834	2.727	5.246	5.162	2.421	1.669	1.351	2.530	1.655	1.297	1.199
	Cost	2.100	1.773	1.577	1.557	2.087	2.071	1.550	1.509	1.553	1.528	1.405	1.337	1.318
fpppp:	Prim.	6.769	96.797	96.797	96.797	92.241	92.241	91.765	81.849	71.540	96.035	91.765	91.765	91.765
8 atoms	Misses	0.714	3.203	2.327	2.239	3.930	3.921	2.082	1.386	1.094	2.197	1.354	1.042	0.956
	Cost	2.068	1.641	1.474	1.458	1.824	1.823	1.478	1.445	1.492	1.457	1.340	1.280	1.264
spice2g6:	Prim.	10.311	70.064	70.064	70.064	69.629	69.629	58.960	50.479	44.042	65.833	58.960	58.960	58.960
short	Misses	24.086	29.936	25.974	25.992	26.475	26.467	26.739	24.718	24.020	26.367	24.650	23.582	23.404
	Cost	6.473	6.987	6.234	6.238	6.334	6.332	6.491	6.192	6.123	6.351	6.094	5.891	5.857
nasa7:	Prim.	10.497	88.953	88.953	88.953	86.882	86.882	15.603	15.565	15.564	81.477	15.603	15.603	15.603
mxm	Misses	25.240	11.047	6.208	6.212	11.971	11.990	16.033	25.237	25.240	16.061	20.596	15.609	15.590
	Cost	6.691	3.209	2.290	2.291	3.406	3.409	4.890	6.639	6.640	4.237	5.757	4.810	4.806
nasa7:	Prim.	23.831	37.019	37.019	37.019	54.249	54.249	33.481	29.253	28.865	36.883	33.481	33.481	33.481
fft	Misses	45.556	62.981	45.469	45.469	45.545	45.545	45.523	45.459	45.441	45.597	45.425	45.419	45.419
	Cost	10.417	13.596	10.269	10.269	10.111	10.111	10.315	10.345	10.345	10.295	10.296	10.295	10.295
nasa7:	Prim.	12.658	45.475	45.475	45.475	45.021	45.021	42.235	41.016	40.346	42.973	42.235	42.235	42.235
cho	Misses	56.382	54.525	51.728	51.837	54.790	54.780	55.378	56.181	56.317	55.152	55.518	55.773	55.845
	Cost	12.586	11.905	11.374	11.394	11.960	11.958	12.099	12.264	12.297	12.049	12.126	12.174	12.188
nasa7:	Prim.	7.659	66.109	66.109	66.109	68.076	68.076	50.671	44.450	41.459	59.849	50.671	50.671	50.671
btr	Misses	31.984	33.891	32.442	32.409	30.636	30.658	29.441	30.318	32.745	29.150	28.970	29.325	26.878
	Cost	8.000	7.778	7.503	7.497	7.140	7.144	7.087	7.316	7.807	6.940	6.998	7.065	6.600
nasa7:	Prim.	6.641	45.300	45.300	45.300	41.916	41.916	42.975	40.971	37.966	44.158	42.975	42.975	42.975
gmt	Misses	55.423	54.700	53.946	53.942	56.255	56.255	54.527	55.092	55.265	54.155	54.720	54.807	54.836
	Cost	12.464	11.940	11.797	11.796	12.269	12.269	11.930	12.058	12.121	11.848	11.967	11.984	11.989
nasa7:	Prim.	42.361	93.122	93.122	93.122	93.022	93.022	91.838	83.646	80.959	92.776	91.838	91.838	91.838
emi	Misses	5.633	6.878	5.980	5.972	6.006	5.992	5.706	5.679	5.645	6.013	5.738	5.716	5.720
	Cost	2.647	2.376	2.205	2.203	2.211	2.208	2.166	2.242	2.263	2.215	2.172	2.168	2.168
nasa7:	Prim.	27.361	43.649	43.649	43.649	49.216	49.216	43.541	43.355	43.167	43.556	43.541	43.541	43.541
vpe	Misses	50.608	56.351	50.531	50.910	50.535	50.644	53.118	52.126	50.653	53.151	52.937	50.969	50.562
	Cost	11.342	12.270	11.164	11.236	11.109	11.130	11.657	11.470	11.192	11.663	11.623	11.249	11.171
matrix:	Prim.	3.239	50.211	50.211	50.211	48.026	48.026	38.222	36.649	35.994	43.140	38.222	38.222	38.222
300	Misses	59.329	49.789	48.541	48.561	51.573	51.580	54.727	59.336	59.329	52.075	55.948	56.161	56.350
	Cost	13.240	10.958	10.721	10.725	11.319	11.320	12.016	12.907	12.913	11.463	12.248	12.288	12.324
tomcatv:	Prim.	11.224	86.358	86.358	86.358	92.229	92.229	85.126	83.329	70.175	85.906	85.126	85.126	85.126
N=33	Misses	6.777	13.642	7.039	7.035	6.829	6.830	7.075	6.767	6.764	7.099	6.831	6.831	6.828
	Cost	3.175	3.728	2.474	2.473	2.375	2.375	2.493	2.453	2.583	2.490	2.447	2.447	2.446
tomcatv:	Prim.	10.777	85.884	85.884	85.884	91.591	91.591	84.988	72.486	46.281	85.747	84.988	84.988	84.988
N=65	Misses	6.927	14.116	7.398	7.399	7.398	7.397	7.247	6.929	6.952	7.270	6.924	6.936	6.928
	Cost	3.208	3.823	2.547	2.547	2.490	2.489	2.527	2.592	2.858	2.524	2.466	2.468	2.466
tomcatv:	Prim.	10.584	85.835	85.835	85.835	87.691	87.691	73.710	47.267	44.566	85.156	73.710	73.710	73.710
N=129	Misses	8.119	14.165	7.410	7.410	7.789	7.790	7.535	7.577	8.314	7.973	7.600	7.911	8.037
	Cost	3.437	3.833	2.549	2.550	2.603	2.603	2.695	2.967	3.134	2.663	2.707	2.766	2.790
tomcatv:	Prim.	11.933	74.770	74.770	74.770	89.012	89.012	48.651	46.116	45.342	71.343	48.651	48.651	48.651
N=257	Misses	10.376	25.230	9.202	9.217	10.079	10.079	25.622	24.143	10.375	24.486	23.962	11.952	9.604
	Cost	3.852	6.046	3.001	3.004	3.025	3.025	6.382	6.126	3.518	5.939	6.066	3.784	3.338
SYMB:	Prim.	26.082	94.044	94.044	94.044	94.685	94.685	91.601	87.250	82.938	93.481	91.601	91.601	91.601
	Misses	3.400	5.956	3.782	3.803	3.928	3.943	3.882	3.585	3.483	3.921	3.577	3.482	3.439
	Cost	2.385	2.191	1.778	1.782	1.799	1.802	1.821	1.809	1.832	1.810	1.764	1.746	1.737
GOOD:	Prim.	14.238	94.576	94.576	94.576	92.044	92.044	89.660	81.067	71.448	93.653	89.660	89.660	89.660
	Misses	1.649	5.424	3.469	3.439	4.340	4.380	3.340	2.358	1.923	3.512	2.409	1.951	1.825
	Cost	2.171	2.085	1.713	1.708	1.904	1.912	1.738	1.637	1.651	1.731	1.561	1.474	1.450
BAD:	Prim.	12.498	62.236	62.236	62.236	64.880	64.880	47.240	42.733	40.257	58.539	47.240	47.240	47.240
	Misses	35.356	37.764	32.288	32.329	33.605	33.614	36.121	36.959	35.404	35.535	35.977	33.949	33.394
	Cost	8.593	8.553	7.512	7.520	7.736	7.738	8.391	8.595	8.324	8.166	8.363	7.978	7.872
SPEC89:	Prim.	17.004	77.254	77.254	77.254	78.457	78.457	67.762	62.657	58.480	74.972	67.762	67.762	67.762
	Misses	20.075	22.746	18.854	18.878	19.736	19.752	20.899	21.095	20.170	20.627	20.580	19.402	19.075
	Cost	5.644	5.549	4.810	4.814	4.965	4.968	5.293	5.381	5.247	5.169	5.233	5.009	4.947

Table 8: Data for Floating-Point Benchmarks and SPEC89 Aggregates on 8K Caches

Input	Rate	FA	DM	VCF	VC4	RVF	RV4	MC2	MC4	MC8	CA	HH2	HH4	HH8
doduc:	Prim.	27.784	96.688	96.688	96.688	96.136	96.136	95.225	90.655	86.019	96.326	95.225	95.225	95.225
tiny	Misses	1.266	3.312	1.888	2.031	1.966	2.009	1.545	1.035	1.044	1.564	1.126	1.139	1.158
	Cost	1.963	1.662	1.392	1.419	1.412	1.420	1.341	1.290	1.338	1.334	1.262	1.264	1.268
doduc:	Prim.	27.978	96.647	96.647	96.647	95.997	95.997	95.185	90.560	85.900	96.271	95.185	95.185	95.185
small	Misses	1.279	3.353	1.937	2.077	2.050	2.095	1.615	1.082	1.097	1.625	1.168	1.209	1.219
	Cost	1.963	1.671	1.402	1.428	1.429	1.438	1.355	1.300	1.349	1.346	1.270	1.278	1.280
doduc:	Prim.	28.020	96.634	96.634	96.634	95.987	95.987	95.153	90.554	85.868	96.252	95.153	95.153	95.153
ref	Misses	1.283	3.366	1.950	2.090	2.069	2.113	1.639	1.094	1.109	1.645	1.187	1.212	1.238
	Cost	1.964	1.673	1.404	1.431	1.433	1.442	1.360	1.302	1.352	1.350	1.274	1.279	1.284
fpppp:	Prim.	7.283	98.755	98.755	98.755	97.154	97.154	97.722	96.134	90.967	98.632	97.722	97.722	97.722
4 atoms	Misses	0.152	1.245	0.442	0.510	1.169	1.194	0.311	0.181	0.161	0.326	0.189	0.172	0.159
	Cost	1.956	1.249	1.096	1.109	1.251	1.255	1.082	1.073	1.121	1.076	1.059	1.055	1.053
fpppp:	Prim.	6.769	99.101	99.101	99.101	96.752	96.752	98.294	96.797	91.765	99.008	98.294	98.294	98.294
8 atoms	Misses	0.142	0.899	0.383	0.398	1.274	1.291	0.251	0.160	0.150	0.265	0.168	0.156	0.151
	Cost	1.959	1.180	1.082	1.085	1.275	1.278	1.065	1.062	1.111	1.060	1.049	1.047	1.046
spice2g6:	Prim.	10.311	81.694	81.694	81.694	83.146	83.146	77.008	70.064	58.960	80.233	77.008	77.008	77.008
short	Misses	5.863	18.306	16.105	16.061	14.418	14.491	10.678	6.834	6.226	10.764	7.565	6.781	6.408
	Cost	3.011	4.661	4.243	4.235	3.908	3.922	3.259	2.598	2.593	3.243	2.667	2.518	2.448
nasa7:	Prim.	10.497	93.178	93.178	93.178	93.155	93.155	91.783	88.953	15.603	93.125	91.783	91.783	91.783
mxm	Misses	5.402	6.822	5.600	5.601	6.505	6.510	5.450	5.402	5.402	5.481	5.402	5.402	5.402
	Cost	2.921	2.364	2.132	2.132	2.304	2.305	2.118	2.137	2.870	2.110	2.109	2.109	2.109
nasa7:	Prim.	23.831	43.229	43.229	43.229	54.494	54.494	40.215	37.019	33.481	43.186	40.215	40.215	40.215
fft	Misses	41.959	56.771	45.402	45.401	45.420	45.420	45.412	45.379	45.355	45.440	45.391	45.364	45.362
	Cost	9.734	12.354	10.194	10.194	10.085	10.085	10.226	10.252	10.283	10.202	10.222	10.217	10.217
nasa7:	Prim.	12.658	76.223	76.223	76.223	75.887	75.887	56.114	45.475	42.235	66.929	56.114	56.114	56.114
cho	Misses	32.499	23.777	23.571	23.548	21.739	21.721	26.831	31.336	32.484	29.372	27.650	30.695	32.501
	Cost	8.048	5.755	5.716	5.712	5.372	5.368	6.537	7.499	7.750	6.911	6.692	7.271	7.614
nasa7:	Prim.	7.659	90.944	90.944	90.944	85.832	85.832	82.078	66.109	50.671	90.093	82.078	82.078	82.078
btr	Misses	5.441	9.056	8.050	8.075	13.230	12.960	5.982	5.382	5.225	6.407	5.582	5.478	5.510
	Cost	2.957	2.811	2.620	2.625	3.655	3.604	2.316	2.361	2.486	2.316	2.240	2.220	2.226
nasa7:	Prim.	6.641	75.691	75.691	75.691	74.809	74.809	58.045	45.300	42.975	74.495	58.045	58.045	58.045
gmt	Misses	7.051	24.309	24.163	24.156	25.034	25.028	8.216	7.129	7.166	8.812	7.667	7.944	8.091
	Cost	3.273	5.862	5.834	5.833	6.008	6.007	2.981	2.902	2.932	2.929	2.876	2.929	2.957
nasa7:	Prim.	42.361	95.616	95.616	95.616	95.369	95.369	93.886	93.122	91.838	94.411	93.886	93.886	93.886
emi	Misses	5.619	4.384	4.317	4.316	4.519	4.515	3.850	5.165	5.619	3.120	3.589	3.908	5.105
	Cost	2.644	1.877	1.864	1.864	1.905	1.904	1.793	2.050	2.149	1.649	1.743	1.804	2.031
nasa7:	Prim.	27.361	43.731	43.731	43.731	49.390	49.390	43.704	43.649	43.541	43.707	43.704	43.704	43.704
vpe	Misses	41.850	56.269	50.531	50.866	50.535	50.560	53.029	52.013	50.530	53.037	52.863	50.866	50.530
	Cost	9.678	12.254	11.164	11.227	11.108	11.113	11.638	11.446	11.165	11.640	11.607	11.227	11.164
matrix:	Prim.	3.239	72.767	72.767	72.767	77.721	77.721	64.913	50.211	38.222	72.111	64.913	64.913	64.913
300	Misses	12.427	27.233	26.152	26.006	20.896	20.834	16.074	12.468	12.427	15.564	12.675	12.471	12.428
	Cost	4.329	6.447	6.241	6.213	5.193	5.181	4.405	3.867	3.979	4.236	3.759	3.720	3.712
tomcatv:	Prim.	11.224	96.607	96.607	96.607	96.645	96.645	94.686	86.358	85.126	95.579	94.686	94.686	94.686
N=33	Misses	4.107	3.393	3.104	3.104	3.278	3.280	3.115	3.283	3.718	3.246	3.196	3.563	3.694
	Cost	2.668	1.679	1.624	1.624	1.656	1.657	1.645	1.760	1.855	1.661	1.660	1.730	1.755
tomcatv:	Prim.	10.777	87.092	87.092	87.092	93.356	93.356	86.488	85.884	84.988	86.688	86.488	86.488	86.488
N=65	Misses	6.586	12.908	6.735	6.735	6.373	6.372	6.477	6.588	6.580	6.479	6.550	6.584	6.580
	Cost	3.144	3.582	2.409	2.409	2.277	2.277	2.366	2.393	2.400	2.364	2.380	2.386	2.385
tomcatv:	Prim.	10.584	86.587	86.587	86.587	92.939	92.939	86.286	85.835	73.710	86.476	86.286	86.286	86.286
N=129	Misses	6.831	13.413	7.203	7.203	6.836	6.836	6.903	6.825	6.825	6.906	6.841	6.841	6.838
	Cost	3.192	3.683	2.503	2.503	2.370	2.370	2.449	2.438	2.560	2.447	2.437	2.437	2.436
tomcatv:	Prim.	11.933	86.714	86.714	86.714	92.799	92.799	86.494	74.770	48.651	86.681	86.494	86.494	86.494
N=257	Misses	6.746	13.286	7.162	7.162	6.981	6.981	6.826	6.747	6.753	6.831	6.745	6.748	6.747
	Cost	3.162	3.657	2.494	2.494	2.398	2.398	2.432	2.534	2.796	2.431	2.417	2.417	2.417
SYMB:	Prim.	26.070	97.876	97.876	97.876	97.593	97.593	95.521	94.045	91.603	97.482	95.521	95.521	95.521
	Misses	1.600	2.124	1.734	1.738	2.024	2.030	1.580	1.517	1.549	1.611	1.532	1.557	1.580
	Cost	2.043	1.425	1.351	1.352	1.409	1.410	1.345	1.348	1.378	1.331	1.336	1.341	1.345
GOOD:	Prim.	14.238	98.220	98.220	98.220	96.507	96.507	97.166	94.576	89.660	98.025	97.166	97.166	97.166
	Misses	0.542	1.780	0.934	0.995	1.545	1.572	0.738	0.487	0.486	0.750	0.525	0.526	0.531
	Cost	1.961	1.356	1.195	1.207	1.329	1.334	1.169	1.147	1.196	1.162	1.128	1.128	1.129
BAD:	Prim.	12.498	77.489	77.489	77.489	79.826	79.826	70.969	62.236	47.240	76.032	70.969	70.969	70.969
	Misses	14.131	22.511	19.629	19.625	19.125	19.104	16.140	15.146	15.023	16.396	15.054	15.031	15.149
	Cost	4.560	5.502	4.955	4.954	4.835	4.831	4.357	4.255	4.382	4.355	4.151	4.146	4.169
SPEC89:	Prim.	17.000	87.116	87.116	87.116	87.996	87.996	82.757	77.255	67.762	86.189	82.757	82.757	82.757
	Misses	8.080	12.884	11.100	11.110	11.020	11.015	9.171	8.584	8.529	9.318	8.546	8.542	8.612
	Cost	3.365	3.577	3.238	3.240	3.214	3.213	2.915	2.858	2.943	2.909	2.796	2.795	2.809

Table 9: Data for Floating-Point Benchmarks and SPEC89 Aggregates on 32K Caches

Input	Rate	FA	DM	VCF	VC4	RVF	RV4	MC2	MC4	MC8	CA	HH2	HH4	HH8
doduc: tiny	Prim.	27.784	99.540	99.540	99.540	99.629	99.629	98.658	96.688	95.225	99.493	98.658	98.658	98.658
	Misses	—	0.460	0.292	0.291	0.234	0.230	0.105	0.009	—	0.094	0.007	—	—
	Cost	1.722	1.092	1.060	1.060	1.048	1.047	1.033	1.035	1.048	1.023	1.015	1.013	1.013
doduc: small	Prim.	27.978	99.515	99.515	99.515	99.605	99.605	98.628	96.647	95.185	99.466	98.628	98.628	98.628
	Misses	—	0.485	0.310	0.308	0.246	0.243	0.108	0.009	—	0.097	0.007	—	—
	Cost	1.720	1.097	1.064	1.063	1.051	1.050	1.034	1.035	1.048	1.024	1.015	1.014	1.014
doduc: ref	Prim.	28.020	99.512	99.512	99.512	99.600	99.600	98.628	96.634	95.153	99.463	98.628	98.628	98.628
	Misses	—	0.488	0.311	0.310	0.249	0.246	0.108	0.009	—	0.097	0.007	—	—
	Cost	1.720	1.098	1.064	1.064	1.051	1.051	1.034	1.035	1.048	1.024	1.015	1.014	1.014
fpppp: 4 atoms	Prim.	7.283	99.886	99.886	99.886	98.700	98.700	99.251	98.755	97.722	99.885	99.251	99.251	99.251
	Misses	—	0.114	0.030	0.032	0.075	0.101	0.001	0.001	—	0.002	0.001	—	—
	Cost	1.927	1.023	1.007	1.007	1.027	1.032	1.008	1.013	1.023	1.001	1.008	1.007	1.007
fpppp: 8 atoms	Prim.	6.769	99.945	99.945	99.945	99.939	99.939	99.497	99.101	98.294	99.943	99.497	99.497	99.497
	Misses	—	0.055	0.025	0.027	0.033	0.034	0.002	0.001	—	0.003	0.001	—	—
	Cost	1.932	1.011	1.005	1.006	1.007	1.007	1.005	1.009	1.017	1.001	1.005	1.005	1.005
spice2g6: short	Prim.	10.311	97.284	97.284	97.284	97.280	97.280	88.417	81.694	77.008	96.671	88.417	88.417	88.417
	Misses	2.153	2.716	2.495	2.500	2.497	2.505	2.517	2.126	2.109	2.580	2.131	2.111	2.092
	Cost	2.306	1.543	1.501	1.502	1.502	1.503	1.594	1.587	1.631	1.523	1.521	1.517	1.513
nasa7: mxm	Prim.	10.497	98.422	98.422	98.422	98.385	98.385	93.885	93.178	91.783	97.550	93.885	93.885	93.885
	Misses	5.402	1.578	1.218	1.223	1.559	1.564	1.940	3.130	5.367	1.832	2.253	2.832	3.670
	Cost	2.921	1.316	1.247	1.248	1.312	1.313	1.430	1.663	2.102	1.373	1.489	1.599	1.758
nasa7: fft	Prim.	23.831	49.282	49.282	49.282	55.814	55.814	46.077	43.229	40.215	48.914	46.077	46.077	46.077
	Misses	35.864	50.718	45.020	45.020	44.183	44.181	45.386	45.361	45.338	45.388	45.209	45.184	44.621
	Cost	8.576	11.144	10.061	10.061	9.837	9.836	10.162	10.186	10.212	10.135	10.129	10.124	10.017
nasa7: cho	Prim.	12.658	86.079	86.079	86.079	85.997	85.997	83.304	76.223	56.114	85.899	83.304	83.304	83.304
	Misses	12.001	13.921	13.846	13.836	13.917	13.904	11.859	11.835	11.934	11.888	12.160	12.047	12.015
	Cost	4.154	3.784	3.770	3.768	3.784	3.782	3.420	3.486	3.706	3.400	3.477	3.456	3.450
nasa7: btr	Prim.	7.659	94.372	94.372	94.372	94.414	94.414	93.120	90.944	82.078	94.271	93.120	93.120	93.120
	Misses	4.422	5.628	5.195	5.194	5.145	5.141	4.561	4.413	4.423	4.583	4.453	4.452	4.458
	Cost	2.764	2.126	2.043	2.043	2.033	2.033	1.935	1.929	2.020	1.928	1.915	1.915	1.916
nasa7: gmt	Prim.	6.641	88.909	88.909	88.909	88.858	88.858	84.510	75.691	58.045	88.486	84.510	84.510	84.510
	Misses	6.952	11.091	11.076	11.075	11.129	11.128	7.191	6.932	6.941	7.388	6.920	6.915	6.919
	Cost	3.254	3.218	3.215	3.215	3.226	3.226	2.521	2.560	2.738	2.519	2.470	2.469	2.470
nasa7: emi	Prim.	42.361	99.444	99.444	99.444	99.097	99.097	99.429	95.616	93.886	99.441	99.429	99.429	99.429
	Misses	0.481	0.556	0.501	0.501	0.710	0.716	0.481	0.481	0.481	0.484	0.481	0.481	0.481
	Cost	1.668	1.111	1.101	1.101	1.144	1.145	1.097	1.135	1.153	1.098	1.097	1.097	1.097
nasa7: vpe	Prim.	27.361	43.751	43.751	43.751	50.726	50.726	43.744	43.731	43.704	43.745	43.744	43.744	43.744
	Misses	11.837	56.249	44.009	46.272	43.473	47.971	53.006	51.982	50.530	53.009	52.845	50.854	50.530
	Cost	3.975	12.250	9.924	10.354	9.753	10.607	11.634	11.439	11.164	11.634	11.603	11.225	11.163
matrix: 300	Prim.	3.239	78.764	78.764	78.764	81.929	81.929	76.753	72.767	64.913	78.616	76.753	76.753	76.753
	Misses	12.427	21.236	20.653	20.636	17.476	17.454	13.289	12.427	12.427	13.173	12.432	12.427	12.427
	Cost	4.329	5.247	5.136	5.133	4.501	4.497	3.757	3.633	3.712	3.717	3.595	3.594	3.594
tomcatv: N=33	Prim.	14.338	99.978	99.978	99.978	99.900	99.900	99.757	96.642	94.729	99.978	99.757	99.757	99.757
	Misses	—	0.022	0.000	0.000	0.070	0.062	0.000	—	—	0.000	—	—	—
	Cost	1.857	1.004	1.000	1.000	1.014	1.013	1.002	1.034	1.053	1.000	1.002	1.002	1.002
tomcatv: N=65	Prim.	10.777	97.282	97.282	97.282	97.168	97.168	95.700	87.092	86.488	96.371	95.700	95.700	95.700
	Misses	3.861	2.718	2.712	2.712	2.753	2.753	2.712	2.741	2.903	2.771	2.715	2.883	3.050
	Cost	2.626	1.544	1.543	1.543	1.551	1.551	1.558	1.650	1.687	1.563	1.559	1.591	1.623
tomcatv: N=129	Prim.	10.584	87.346	87.346	87.346	93.766	93.766	86.887	86.587	86.286	86.947	86.887	86.887	86.887
	Misses	6.483	12.654	6.577	6.577	6.179	6.179	6.230	6.458	6.477	6.231	6.388	6.417	6.433
	Cost	3.126	3.531	2.376	2.376	2.236	2.236	2.315	2.361	2.368	2.314	2.345	2.350	2.353
tomcatv: N=257	Prim.	11.933	87.010	87.010	87.010	93.339	93.339	86.861	86.714	86.494	86.912	86.861	86.861	86.861
	Misses	6.660	12.990	6.999	6.999	6.609	6.609	6.637	6.664	6.662	6.638	6.655	6.662	6.662
	Cost	3.146	3.598	2.460	2.460	2.322	2.322	2.392	2.399	2.401	2.392	2.396	2.397	2.397
SYMB:	Prim.	26.070	99.287	99.287	99.287	99.485	99.485	98.968	97.878	95.522	99.224	98.968	98.968	98.968
	Misses	0.391	0.713	0.642	0.642	0.475	0.476	0.408	0.396	0.391	0.403	0.392	0.384	0.381
	Cost	1.814	1.143	1.129	1.129	1.095	1.096	1.088	1.096	1.119	1.084	1.085	1.083	1.083
GOOD:	Prim.	14.238	99.791	99.791	99.791	99.756	99.756	99.180	98.220	97.166	99.773	99.180	99.180	99.180
	Misses	—	0.209	0.126	0.126	0.110	0.111	0.039	0.004	—	0.036	0.003	—	—
	Cost	1.858	1.042	1.026	1.026	1.023	1.024	1.016	1.018	1.028	1.009	1.009	1.008	1.008
BAD:	Prim.	12.504	84.716	84.716	84.716	86.930	86.930	81.379	77.489	70.969	84.395	81.379	81.379	81.379
	Misses	8.696	15.284	12.976	13.134	12.468	12.783	12.148	11.950	12.024	12.160	11.966	11.855	11.854
	Cost	3.527	4.057	3.618	3.648	3.500	3.560	3.494	3.496	3.575	3.466	3.460	3.439	3.438
SPEC89:	Prim.	17.003	91.637	91.637	91.637	92.868	92.868	89.671	87.116	82.757	91.444	89.671	89.671	89.671
	Misses	4.735	8.363	7.103	7.187	6.779	6.947	6.577	6.463	6.500	6.582	6.470	6.408	6.407
	Cost	2.730	2.673	2.433	2.449	2.359	2.391	2.353	2.357	2.407	2.336	2.333	2.321	2.321

Table 10: Data for Floating-Point Benchmarks and SPEC89 Aggregates on 128K Caches

Input	Rate	FA	DM	VCF	VC4	RVF	RV4	MC2	MC4	MC8	CA	HH2	HH4	HH8
dec0:	Prim.	43.307	92.903	92.903	92.903	92.147	92.147	89.321	83.967	77.475	91.637	89.321	89.321	89.321
	Misses	3.008	7.097	5.159	5.160	5.415	5.474	4.894	4.066	3.741	5.143	4.184	3.954	3.551
	Cost	2.138	2.419	2.051	2.051	2.107	2.119	2.037	1.933	1.936	2.061	1.902	1.858	1.782
fora:	Prim.	36.042	93.906	93.906	93.906	93.592	93.592	90.968	85.922	79.322	92.950	90.968	90.968	90.968
	Misses	3.086	6.094	4.468	4.499	4.560	4.548	4.161	3.468	3.277	4.444	3.581	3.416	3.306
	Cost	2.226	2.219	1.910	1.916	1.931	1.928	1.881	1.800	1.829	1.915	1.771	1.739	1.718
forf:	Prim.	35.754	94.930	94.930	94.930	94.980	94.980	92.596	88.085	80.932	94.177	92.596	92.596	92.596
	Misses	2.718	5.070	3.579	3.595	3.523	3.557	3.403	2.949	2.815	3.613	3.028	2.915	2.856
	Cost	2.159	2.014	1.731	1.734	1.720	1.726	1.721	1.679	1.726	1.745	1.649	1.628	1.617
fsxzz:	Prim.	54.278	96.894	96.894	96.894	96.939	96.939	95.281	93.716	91.382	96.438	95.281	95.281	95.281
	Misses	1.018	3.106	1.986	1.986	1.925	1.925	1.951	1.278	1.097	2.020	1.463	1.207	1.113
	Cost	1.651	1.621	1.408	1.408	1.396	1.396	1.418	1.306	1.295	1.419	1.325	1.277	1.259
ivex:	Prim.	27.195	93.168	93.168	93.168	93.770	93.770	89.494	86.010	80.626	92.384	89.494	89.494	89.494
	Misses	3.099	6.832	4.303	4.343	4.162	4.195	4.300	3.546	3.267	4.463	3.712	3.441	3.379
	Cost	2.317	2.366	1.886	1.893	1.853	1.859	1.922	1.814	1.815	1.924	1.810	1.759	1.747
linp:	Prim.	4.220	93.387	93.387	93.387	92.903	92.903	92.270	89.108	85.142	93.226	92.270	92.270	92.270
	Misses	5.244	6.613	5.511	5.547	5.786	5.841	5.426	5.260	5.254	5.508	5.266	5.257	5.254
	Cost	2.954	2.323	2.113	2.120	2.170	2.181	2.108	2.108	2.147	2.114	2.078	2.076	2.076
lisp:	Prim.	25.486	94.813	94.813	94.813	94.675	94.675	91.581	86.543	75.601	94.272	91.581	91.581	91.581
	Misses	1.352	5.187	2.002	2.107	1.900	1.945	1.845	1.431	1.369	1.927	1.453	1.321	1.329
	Cost	2.002	2.037	1.432	1.452	1.414	1.423	1.435	1.406	1.504	1.423	1.360	1.335	1.337
macr:	Prim.	29.856	93.541	93.541	93.541	94.030	94.030	91.496	85.474	78.703	92.843	91.496	91.496	91.496
	Misses	3.242	6.459	3.753	3.786	3.773	3.809	3.618	3.278	3.233	3.843	3.351	3.295	3.243
	Cost	2.317	2.292	1.778	1.784	1.776	1.783	1.772	1.768	1.827	1.802	1.722	1.711	1.701
memxx:	Prim.	50.302	96.142	96.142	96.142	96.978	96.978	94.885	92.291	89.405	95.884	94.885	94.885	94.885
	Misses	1.284	3.858	1.956	1.964	1.750	1.755	1.995	1.546	1.345	2.009	1.593	1.397	1.334
	Cost	1.741	1.772	1.410	1.412	1.363	1.364	1.430	1.371	1.361	1.423	1.354	1.317	1.305
mul2:	Prim.	52.570	95.240	95.240	95.240	95.010	95.010	93.120	88.595	86.021	94.639	93.120	93.120	93.120
	Misses	2.538	4.760	3.249	3.213	3.530	3.560	3.118	2.827	2.654	3.181	2.898	2.716	2.672
	Cost	1.956	1.952	1.665	1.658	1.721	1.726	1.661	1.651	1.644	1.658	1.619	1.585	1.577
mul8:	Prim.	44.638	95.547	95.547	95.547	95.114	95.114	93.246	89.450	85.349	95.137	93.246	93.246	93.246
	Misses	1.046	4.453	2.166	2.260	2.186	2.322	2.089	1.362	1.188	2.132	1.531	1.288	1.224
	Cost	1.752	1.891	1.456	1.474	1.464	1.490	1.464	1.364	1.372	1.454	1.358	1.312	1.300
pasc:	Prim.	43.261	88.690	88.690	88.690	91.055	91.055	83.601	77.916	70.507	87.058	83.601	83.601	83.601
	Misses	2.534	11.310	7.718	7.799	6.919	6.909	6.135	4.354	3.352	5.911	4.417	3.729	3.259
	Cost	2.049	3.262	2.579	2.595	2.404	2.402	2.330	2.048	1.932	2.252	2.003	1.872	1.783
savec:	Prim.	21.328	98.135	98.135	98.135	98.345	98.345	97.554	96.897	96.105	97.912	97.554	97.554	97.554
	Misses	0.701	1.865	1.071	1.086	1.030	1.017	1.155	0.892	0.698	1.157	0.951	0.814	0.740
	Cost	1.920	1.373	1.222	1.225	1.212	1.210	1.244	1.200	1.172	1.241	1.205	1.179	1.165
spic:	Prim.	46.432	96.368	96.368	96.368	93.984	93.984	94.023	91.185	86.263	95.679	94.023	94.023	94.023
	Misses	2.580	3.632	2.674	2.682	3.450	3.521	2.776	2.514	2.498	2.740	2.471	2.401	2.366
	Cost	2.026	1.726	1.544	1.546	1.716	1.729	1.587	1.566	1.612	1.564	1.529	1.516	1.509
ue02:	Prim.	37.277	92.063	92.063	92.063	92.351	92.351	89.350	83.993	77.005	91.230	89.350	89.350	89.350
	Misses	4.167	7.937	5.660	5.747	5.477	5.586	5.271	4.662	4.355	5.542	4.755	4.507	4.451
	Cost	2.419	2.587	2.155	2.171	2.117	2.138	2.108	2.046	2.057	2.141	2.010	1.963	1.952
total:	Prim.	35.870	94.258	94.258	94.258	94.216	94.216	91.806	87.812	82.506	93.586	91.806	91.806	91.806
	Misses	2.640	5.742	3.817	3.853	3.843	3.886	3.601	3.024	2.808	3.692	3.099	2.903	2.799
	Cost	2.143	2.148	1.783	1.790	1.788	1.796	1.766	1.696	1.709	1.766	1.671	1.634	1.614

Table 11: Data for ATUM Traces on 8K Caches

References

[1] 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.

[2] Anant Agarwal and Steven D. Pudar, "Column-Associative Caches: A Technique for Reducing the Miss Rate of Direct-Mapped Caches," in *Proceedings of the 20th Annual International Symposium on Computer Architecture*, San Diego, California, pp. 179–190, ACM SIGARCH and IEEE Computer Society, May 17–19, 1993. *Computer Architecture News,* 21(2), May 1993.

Input	Rate	FA	DM	VCF	VC4	RVF	RV4	MC2	MC4	MC8	CA	HH2	HH4	HH8
dec0:	Prim.	43.307	97.686	97.686	97.686	97.488	97.488	95.833	92.903	89.321	97.236	95.833	95.833	95.833
	Misses	0.578	2.314	1.477	1.489	1.723	1.739	1.050	0.773	0.637	1.206	0.813	0.695	0.676
	Cost	1.677	1.463	1.304	1.306	1.353	1.355	1.241	1.218	1.228	1.257	1.196	1.174	1.170
fora:	Prim.	36.042	97.849	97.849	97.849	97.446	97.446	96.254	93.906	90.968	97.430	96.254	96.254	96.254
	Misses	0.497	2.151	1.414	1.451	1.721	1.722	0.927	0.782	0.674	1.065	0.813	0.656	0.577
	Cost	1.734	1.430	1.290	1.297	1.353	1.353	1.214	1.210	1.218	1.228	1.192	1.162	1.147
forf:	Prim.	35.754	97.954	97.954	97.954	95.767	95.767	96.890	94.930	92.596	97.624	96.890	96.890	96.890
	Misses	0.597	2.046	1.570	1.571	1.632	1.655	1.073	0.859	0.728	1.174	0.917	0.796	0.733
	Cost	1.756	1.409	1.319	1.319	1.352	1.357	1.235	1.214	1.212	1.247	1.205	1.182	1.170
fsxzz:	Prim.	54.278	99.063	99.063	99.063	99.183	99.183	98.381	96.894	95.281	98.895	98.381	98.381	98.381
	Misses	0.161	0.937	0.687	0.696	0.530	0.549	0.474	0.384	0.280	0.511	0.366	0.282	0.243
	Cost	1.488	1.187	1.140	1.142	1.109	1.112	1.106	1.104	1.100	1.108	1.086	1.070	1.062
ivex:	Prim.	27.195	97.541	97.541	97.541	97.455	97.455	96.185	93.168	89.494	97.116	96.185	96.185	96.185
	Misses	1.114	2.459	1.857	1.876	1.942	1.962	1.504	1.258	1.149	1.628	1.316	1.258	1.211
	Cost	1.940	1.492	1.377	1.381	1.394	1.398	1.324	1.307	1.323	1.338	1.288	1.277	1.268
linp:	Prim.	4.220	96.523	96.523	96.523	95.304	95.304	94.461	93.387	92.270	95.534	94.461	94.461	94.461
	Misses	5.156	3.477	3.227	3.225	4.237	4.250	3.138	3.854	4.425	2.389	3.409	3.655	4.147
	Cost	2.937	1.695	1.648	1.647	1.852	1.855	1.652	1.798	1.918	1.499	1.703	1.750	1.843
lisp:	Prim.	25.486	97.764	97.764	97.764	99.197	99.197	96.837	94.813	91.581	97.675	96.837	96.837	96.837
	Misses	—	2.236	0.423	0.460	0.356	0.350	0.188	0.040	—	0.204	0.051	0.001	—
	Cost	1.745	1.447	1.103	1.110	1.076	1.075	1.067	1.059	1.084	1.062	1.041	1.032	1.032
macr:	Prim.	29.856	97.944	97.944	97.944	97.528	97.528	96.916	93.541	91.496	97.550	96.916	96.916	96.916
	Misses	1.219	2.056	1.827	1.829	1.896	1.888	1.471	1.296	1.258	1.603	1.345	1.309	1.253
	Cost	1.933	1.411	1.368	1.368	1.385	1.384	1.310	1.311	1.324	1.329	1.286	1.280	1.269
memxx:	Prim.	50.302	98.753	98.753	98.753	99.039	99.039	97.561	96.142	94.885	98.583	97.561	97.561	97.561
	Misses	0.273	1.247	0.846	0.850	0.650	0.652	0.707	0.657	0.531	0.763	0.645	0.557	0.487
	Cost	1.549	1.249	1.173	1.174	1.133	1.134	1.159	1.163	1.152	1.159	1.147	1.130	1.117
mul2:	Prim.	52.570	98.324	98.324	98.324	98.158	98.158	97.257	95.240	93.120	98.064	97.257	97.257	97.257
	Misses	0.098	1.676	1.388	1.383	1.233	1.230	0.799	0.420	0.287	0.874	0.483	0.353	0.288
	Cost	1.493	1.335	1.281	1.279	1.253	1.252	1.179	1.127	1.123	1.185	1.119	1.094	1.082
mul8:	Prim.	44.638	98.526	98.526	98.526	98.519	98.519	97.785	95.547	93.246	98.372	97.785	97.785	97.785
	Misses	0.812	1.474	1.013	1.015	0.977	0.979	0.867	0.801	0.792	0.889	0.775	0.772	0.770
	Cost	1.708	1.295	1.207	1.208	1.201	1.201	1.187	1.197	1.218	1.185	1.169	1.169	1.168
pasc:	Prim.	43.261	94.590	94.590	94.590	97.281	97.281	92.792	88.690	83.601	94.325	92.792	92.792	92.792
	Misses	0.001	5.410	2.267	2.351	1.545	1.722	1.221	0.183	0.042	1.323	0.244	0.091	0.017
	Cost	1.568	2.082	1.485	1.501	1.321	1.354	1.304	1.148	1.172	1.308	1.118	1.089	1.075
savec:	Prim.	21.328	99.550	99.550	99.550	99.687	99.687	99.194	98.135	97.554	99.478	99.194	99.194	99.194
	Misses	—	0.450	0.264	0.262	0.221	0.226	0.166	0.104	0.057	0.182	0.101	0.057	0.019
	Cost	1.787	1.090	1.055	1.054	1.045	1.046	1.040	1.038	1.035	1.040	1.027	1.019	1.012
spic:	Prim.	46.432	98.890	98.890	98.890	98.744	98.744	98.035	96.368	94.023	98.722	98.035	98.035	98.035
	Misses	—	1.110	0.694	0.704	0.905	0.903	0.344	0.140	0.011	0.389	0.156	0.023	0.006
	Cost	1.536	1.222	1.143	1.145	1.185	1.184	1.085	1.063	1.062	1.087	1.049	1.024	1.021
ue02:	Prim.	37.277	96.817	96.817	96.817	96.851	96.851	94.966	92.063	89.350	96.196	94.966	94.966	94.966
	Misses	1.589	3.183	2.637	2.631	2.545	2.559	2.123	1.808	1.699	2.315	1.881	1.793	1.736
	Cost	1.929	1.637	1.533	1.532	1.515	1.518	1.454	1.423	1.429	1.478	1.408	1.391	1.380
total:	Prim.	35.870	97.748	97.748	97.748	97.707	97.707	96.466	94.258	91.806	97.388	96.466	96.466	96.466
	Misses	1.007	2.252	1.530	1.544	1.609	1.628	1.169	1.024	1.001	1.162	0.999	0.945	0.961
	Cost	1.833	1.450	1.313	1.316	1.329	1.332	1.257	1.252	1.272	1.247	1.225	1.215	1.218

Table 12: Data for ATUM Traces on 32K Caches

[3] J. H. Chang, H. Chao, and K. So, "Cache Design of A Sub-Micron CMOS System/370," in *Proceedings of the 14th Annual International Symposium on Computer Architecture*, Pittsburgh, Pennsylvania, pp. 208–213, IEEE Computer Society and ACM SIGARCH, June 2–5, 1987. *Computer Architecture News,* 15(2), June 1987.

[4] John L. Hennessy and David A. Patterson, *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, Inc., 1990.

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

Input	Rate	FA	DM	VCF	VC4	RVF	RV4	MC2	MC4	MC8	CA	HH2	HH4	HH8
dec0:	Prim.	43.307	99.345	99.345	99.345	99.588	99.588	98.754	97.686	95.833	99.285	98.754	98.754	98.754
	Misses	—	0.655	0.346	0.364	0.282	0.281	0.129	0.047	0.011	0.151	0.045	0.016	—
	Cost	1.567	1.131	1.072	1.076	1.058	1.058	1.037	1.032	1.044	1.036	1.021	1.016	1.012
fora:	Prim.	36.042	99.270	99.270	99.270	99.671	99.671	98.843	97.849	96.254	99.216	98.843	98.843	98.843
	Misses	—	0.730	0.334	0.345	0.206	0.207	0.098	0.011	0.002	0.123	0.020	0.006	0.002
	Cost	1.640	1.146	1.071	1.073	1.042	1.043	1.030	1.024	1.038	1.031	1.015	1.013	1.012
forf:	Prim.	35.754	99.394	99.394	99.394	99.392	99.392	98.837	97.954	96.890	99.305	98.837	98.837	98.837
	Misses	—	0.606	0.461	0.468	0.419	0.415	0.182	0.079	0.024	0.205	0.085	0.026	0.012
	Cost	1.642	1.121	1.094	1.095	1.086	1.085	1.046	1.036	1.036	1.046	1.028	1.017	1.014
fsxzz:	Prim.	54.278	99.728	99.728	99.728	99.779	99.779	99.425	99.063	98.381	99.684	99.425	99.425	99.425
	Misses	—	0.272	0.183	0.183	0.133	0.135	0.106	0.047	0.026	0.124	0.051	0.035	0.018
	Cost	1.457	1.054	1.037	1.037	1.028	1.028	1.026	1.018	1.021	1.027	1.015	1.012	1.009
ivex:	Prim.	27.195	99.338	99.338	99.338	99.167	99.167	98.658	97.541	96.185	99.186	98.658	98.658	98.658
	Misses	—	0.662	0.528	0.531	0.669	0.668	0.281	0.119	0.053	0.329	0.123	0.069	0.018
	Cost	1.728	1.132	1.107	1.108	1.135	1.135	1.067	1.047	1.048	1.071	1.037	1.027	1.017
linp:	Prim.	4.220	99.873	99.873	99.873	99.827	99.827	99.623	96.523	94.461	99.870	99.623	99.623	99.623
	Misses	—	0.127	0.043	0.052	0.072	0.079	0.004	0.001	—	0.006	—	—	—
	Cost	1.958	1.025	1.010	1.011	1.015	1.017	1.005	1.035	1.055	1.002	1.004	1.004	1.004
lisp:	Prim.	25.486	99.660	99.660	99.660	99.850	99.850	99.108	97.764	96.837	99.654	99.108	99.108	99.108
	Misses	—	0.340	—	0.005	0.014	0.013	0.005	—	—	0.009	—	—	—
	Cost	1.745	1.068	1.003	1.004	1.004	1.004	1.010	1.022	1.032	1.005	1.009	1.009	1.009
macr:	Prim.	29.856	99.277	99.277	99.277	99.368	99.368	98.720	97.944	96.916	99.164	98.720	98.720	98.720
	Misses	—	0.723	0.568	0.574	0.518	0.517	0.272	0.124	0.045	0.299	0.134	0.064	0.025
	Cost	1.701	1.145	1.115	1.116	1.105	1.105	1.064	1.044	1.039	1.065	1.038	1.025	1.017
memxx:	Prim.	50.302	99.380	99.380	99.380	99.691	99.691	98.939	98.753	97.561	99.347	98.939	98.939	98.939
	Misses	—	0.620	0.541	0.540	0.216	0.220	0.449	0.343	0.245	0.453	0.392	0.303	0.208
	Cost	1.497	1.124	1.109	1.109	1.044	1.045	1.096	1.078	1.071	1.093	1.085	1.068	1.050
mul2:	Prim.	52.570	99.417	99.417	99.417	99.637	99.637	99.071	98.324	97.257	99.374	99.071	99.071	99.071
	Misses	—	0.583	0.447	0.438	0.242	0.240	0.116	0.009	—	0.138	0.016	0.001	—
	Cost	1.474	1.117	1.091	1.089	1.050	1.049	1.031	1.018	1.027	1.032	1.012	1.009	1.009
mul8:	Prim.	44.638	99.520	99.520	99.520	99.127	99.127	99.134	98.526	97.785	99.457	99.134	99.134	99.134
	Misses	—	0.480	0.403	0.405	0.332	0.337	0.298	0.191	0.095	0.300	0.192	0.119	0.065
	Cost	1.554	1.096	1.081	1.082	1.072	1.073	1.065	1.051	1.040	1.062	1.045	1.031	1.021
pasc:	Prim.	43.261	99.097	99.097	99.097	99.137	99.137	98.446	94.590	92.792	99.075	98.446	98.446	98.446
	Misses	—	0.903	0.176	0.185	0.127	0.148	0.039	0.020	—	0.044	0.017	—	—
	Cost	1.567	1.181	1.042	1.044	1.033	1.037	1.023	1.058	1.072	1.018	1.019	1.016	1.016
savec:	Prim.	21.328	99.786	99.786	99.786	99.833	99.833	99.615	99.550	99.194	99.769	99.615	99.615	99.615
	Misses	—	0.214	0.119	0.119	0.079	0.083	0.052	0.003	—	0.051	0.010	—	—
	Cost	1.787	1.043	1.025	1.025	1.017	1.017	1.014	1.005	1.008	1.012	1.006	1.004	1.004
spic:	Prim.	46.432	99.883	99.883	99.883	99.769	99.769	99.467	98.890	98.035	99.869	99.467	99.467	99.467
	Misses	—	0.117	0.011	0.014	0.048	0.084	0.024	—	—	0.025	0.001	—	—
	Cost	1.536	1.023	1.003	1.004	1.011	1.018	1.010	1.011	1.020	1.006	1.005	1.005	1.005
ue02:	Prim.	37.277	98.823	98.823	98.823	99.110	99.110	98.025	96.817	94.966	98.654	98.025	98.025	98.025
	Misses	—	1.177	0.990	0.997	0.748	0.750	0.473	0.178	0.052	0.509	0.220	0.080	0.047
	Cost	1.627	1.235	1.200	1.201	1.151	1.151	1.110	1.066	1.060	1.110	1.062	1.035	1.029
total:	Prim.	35.870	99.471	99.471	99.471	99.531	99.531	99.002	97.748	96.466	99.416	99.002	99.002	99.002
	Misses	—	0.529	0.326	0.331	0.259	0.265	0.164	0.080	0.039	0.178	0.088	0.050	0.028
	Cost	1.641	1.106	1.067	1.068	1.054	1.055	1.041	1.038	1.043	1.040	1.027	1.020	1.015

Table 13: Data for ATUM Traces on 128K Caches

[6] 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 International Symposium on Computer Architecture*, Seattle, Washington, pp. 364–373, IEEE Computer Society and ACM SIGARCH, May 28–31, 1990. *Computer Architecture News,* 18(2), June 1990.

[7] S. Przybylski, M. Horowitz, and J. Hennessy, "Performance Tradeoffs in Cache Design," in *Proceedings of the 15th Annual International Symposium on Computer Architecture*, Honolulu, Hawaii, pp. 290–298, IEEE Computer Society and ACM SIGARCH, May 30–June 2, 1988. *Computer Architecture News,* 16(2), May 1988.

Input	Rate	FA	DM	VCF	VC4	RVF	RV4	MC2	MC4	MC8	CA	HH2	HH4	HH8
SYM4: 100,000,000	Prim.	19.066	93.483	93.483	93.483	93.757	93.757	91.044	87.422	83.432	92.919	91.044	91.044	91.044
	Misses	4.474	6.517	4.727	4.738	4.888	4.905	4.857	4.502	4.440	4.901	4.511	4.417	4.391
	Cost	2.659	2.303	1.963	1.965	1.991	1.994	2.012	1.981	2.009	2.002	1.947	1.929	1.924
SYM4: 1,000,000	Prim.	19.066	93.441	93.441	93.441	93.718	93.718	91.025	87.413	83.427	92.888	91.025	91.025	91.025
	Misses	4.514	6.559	4.774	4.785	4.932	4.950	4.898	4.544	4.481	4.944	4.555	4.460	4.435
	Cost	2.667	2.312	1.973	1.975	2.000	2.003	2.020	1.989	2.017	2.011	1.955	1.937	1.932
SYM4: 100,000	Prim.	19.065	93.080	93.080	93.080	93.313	93.313	90.848	87.329	83.382	92.609	90.848	90.848	90.848
	Misses	4.869	6.920	5.180	5.192	5.317	5.327	5.260	4.910	4.845	5.321	4.936	4.848	4.822
	Cost	2.734	2.384	2.053	2.056	2.077	2.079	2.091	2.060	2.087	2.085	2.029	2.013	2.008
SYM4: 10,000	Prim.	19.055	91.270	91.270	91.270	91.629	91.629	89.513	86.548	82.940	90.799	89.513	89.513	89.513
	Misses	7.368	8.730	7.205	7.221	7.265	7.276	7.362	7.233	7.270	7.449	7.222	7.221	7.230
	Cost	3.209	2.746	2.456	2.459	2.464	2.466	2.504	2.509	2.552	2.507	2.477	2.477	2.479
SYM4: 1,000	Prim.	18.959	89.373	89.373	89.373	89.758	89.758	86.595	82.946	79.552	88.294	86.595	86.595	86.595
	Misses	7.295	10.627	9.158	9.155	9.105	9.096	8.733	7.980	7.522	8.895	8.177	7.807	7.594
	Cost	3.196	3.125	2.846	2.846	2.832	2.831	2.793	2.687	2.634	2.807	2.688	2.617	2.577
SYM4: 100	Prim.	18.024	87.715	87.715	87.715	88.416	88.416	83.815	78.098	71.339	86.474	83.815	83.815	83.815
	Misses	7.278	12.285	8.989	9.123	8.813	8.913	8.850	7.794	7.452	9.146	7.991	7.629	7.495
	Cost	3.203	3.457	2.831	2.856	2.790	2.809	2.843	2.700	2.702	2.873	2.680	2.611	2.586
FP4: 100,000,000	Prim.	25.871	92.418	92.418	92.418	92.720	92.720	87.965	80.482	71.134	91.295	87.965	87.965	87.965
	Misses	2.726	7.582	4.044	4.022	4.656	4.660	6.072	3.623	2.988	5.925	4.580	3.023	2.892
	Cost	2.259	2.516	1.844	1.840	1.957	1.958	2.274	1.884	1.856	2.213	1.990	1.695	1.670
FP4: 1,000,000	Prim.	25.871	92.360	92.360	92.360	92.667	92.667	87.940	80.471	71.129	91.247	87.940	87.940	87.940
	Misses	2.784	7.640	4.111	4.089	4.719	4.721	6.129	3.678	3.044	5.984	4.638	3.084	2.953
	Cost	2.270	2.528	1.857	1.853	1.970	1.970	2.285	1.894	1.867	2.225	2.002	1.707	1.682
FP4: 100,000	Prim.	25.870	91.884	91.884	91.884	91.766	91.766	87.707	80.368	71.087	90.843	87.707	87.707	87.707
	Misses	3.308	8.116	4.653	4.634	5.591	5.607	6.612	4.161	3.547	6.490	5.140	3.615	3.496
	Cost	2.370	2.623	1.965	1.962	2.145	2.148	2.379	1.987	1.963	2.325	2.100	1.810	1.787
FP4: 10,000	Prim.	25.860	89.199	89.199	89.199	89.389	89.389	85.761	79.389	70.666	88.152	85.761	85.761	85.761
	Misses	7.081	10.801	7.670	7.676	8.310	8.323	9.670	7.560	7.167	9.559	8.500	7.192	7.123
	Cost	3.087	3.160	2.565	2.566	2.685	2.687	2.980	2.643	2.655	2.935	2.757	2.509	2.496
FP4: 1,000	Prim.	25.760	83.595	83.595	83.595	84.081	84.081	78.676	73.161	66.970	81.383	78.676	78.676	78.676
	Misses	10.378	16.405	13.903	13.914	14.249	14.206	14.501	12.183	11.708	14.358	13.091	11.814	11.595
	Cost	3.714	4.281	3.806	3.808	3.866	3.858	3.968	3.583	3.555	3.914	3.701	3.458	3.416
FP4: 100	Prim.	24.783	79.883	79.883	79.883	80.125	80.125	72.953	64.146	55.815	77.235	72.953	72.953	72.953
	Misses	10.325	20.117	15.883	16.034	15.947	16.042	15.199	12.328	10.846	15.382	13.197	11.391	10.728
	Cost	3.714	5.023	4.219	4.248	4.229	4.247	4.158	3.701	3.503	4.150	3.778	3.435	3.309
MIX6: 100,000,000	Prim.	14.739	92.912	92.912	92.912	91.915	91.915	87.799	77.378	71.915	92.223	87.799	87.799	87.799
	Misses	4.225	7.088	4.673	4.648	5.431	5.444	4.706	4.376	4.406	4.845	4.395	4.327	4.313
	Cost	2.655	2.418	1.959	1.954	2.113	2.115	2.016	2.058	2.118	1.998	1.957	1.944	1.942
MIX6: 1,000,000	Prim.	14.739	92.873	92.873	92.873	91.879	91.879	87.782	77.371	71.911	92.192	87.782	87.782	87.782
	Misses	4.265	7.127	4.717	4.693	5.473	5.484	4.745	4.415	4.445	4.889	4.435	4.367	4.355
	Cost	2.663	2.425	1.968	1.963	2.121	2.123	2.024	2.065	2.125	2.007	1.965	1.952	1.950
MIX6: 100,000	Prim.	14.738	92.536	92.536	92.536	91.566	91.566	87.635	77.307	71.878	91.915	87.635	87.635	87.635
	Misses	4.615	7.464	5.096	5.074	5.822	5.830	5.087	4.762	4.786	5.276	4.790	4.722	4.719
	Cost	2.729	2.493	2.043	2.039	2.190	2.192	2.090	2.132	2.191	2.083	2.034	2.021	2.020
MIX6: 10,000	Prim.	14.733	90.409	90.409	90.409	89.595	89.595	86.370	76.696	71.548	89.902	86.370	86.370	86.370
	Misses	7.310	9.591	7.485	7.473	8.030	8.039	7.544	7.352	7.361	7.791	7.389	7.338	7.334
	Cost	3.242	2.918	2.518	2.516	2.630	2.631	2.570	2.630	2.683	2.581	2.540	2.531	2.530
MIX6: 1,000	Prim.	14.677	86.420	86.420	86.420	85.828	85.828	81.398	72.762	68.674	84.818	81.398	81.398	81.398
	Misses	10.420	13.580	11.682	11.677	12.154	12.143	11.774	11.505	11.159	11.963	11.562	11.322	11.262
	Cost	3.833	3.716	3.355	3.354	3.451	3.449	3.423	3.458	3.433	3.425	3.383	3.337	3.326
MIX6: 100	Prim.	14.137	83.037	83.037	83.037	82.097	82.097	74.896	63.077	55.436	80.608	74.896	74.896	74.896
	Misses	10.052	16.963	12.437	12.755	13.168	13.410	12.879	11.546	10.822	13.389	11.804	10.976	10.298
	Cost	3.769	4.393	3.533	3.593	3.681	3.727	3.698	3.563	3.502	3.738	3.494	3.336	3.208

Table 14: Data for Multitasking Traces on 8K Caches

[8] André Seznec, "A case for two-way skewed-associative caches," in *Proceedings of the 20th Annual International Symposium on Computer Architecture*, San Diego, California, pp. 169–178, ACM SIGARCH and IEEE Computer Society, May 17–19, 1993. *Computer Architecture News,* 21(2), May 1993.

[9] Alan Jay Smith, "Cache Memories," *ACM Computing Surveys*, 14(3):473–530, September

Input	Rate	FA	DM	VCF	VC4	RVF	RV4	MC2	MC4	MC8	CA	HH2	HH4	HH8
SYM4:	Prim.	19.066	97.609	97.609	97.609	97.627	97.627	95.243	93.483	91.044	97.264	95.243	95.243	95.243
100,000,000	Misses	2.095	2.391	1.949	1.951	1.999	2.003	1.805	1.685	1.719	1.904	1.692	1.705	1.741
	Cost	2.207	1.478	1.394	1.395	1.404	1.404	1.391	1.385	1.416	1.389	1.369	1.371	1.378
SYM4:	Prim.	19.066	97.375	97.375	97.375	97.423	97.423	95.146	93.441	91.025	97.076	95.146	95.146	95.146
1,000,000	Misses	2.263	2.625	2.191	2.193	2.226	2.230	2.037	1.908	1.928	2.149	1.925	1.928	1.955
	Cost	2.239	1.525	1.443	1.443	1.449	1.450	1.436	1.428	1.456	1.438	1.414	1.415	1.420
SYM4:	Prim.	19.065	96.203	96.203	96.203	96.287	96.287	94.448	93.080	90.847	95.867	94.448	94.448	94.448
100,000	Misses	3.443	3.797	3.409	3.412	3.405	3.410	3.362	3.297	3.321	3.499	3.288	3.294	3.309
	Cost	2.464	1.759	1.686	1.686	1.684	1.685	1.694	1.696	1.722	1.706	1.680	1.681	1.684
SYM4:	Prim.	19.055	94.872	94.872	94.872	94.940	94.940	92.665	91.270	89.513	94.117	92.665	92.665	92.665
10,000	Misses	4.043	5.128	4.807	4.810	4.815	4.818	4.592	4.312	4.159	4.699	4.380	4.258	4.181
	Cost	2.578	2.026	1.965	1.965	1.965	1.966	1.946	1.907	1.895	1.952	1.905	1.882	1.868
SYM4:	Prim.	18.959	94.337	94.337	94.337	94.464	94.464	91.605	89.373	86.596	93.463	91.605	91.605	91.605
1,000	Misses	4.002	5.663	5.159	5.164	5.093	5.096	4.644	4.198	4.087	4.850	4.278	4.151	4.103
	Cost	2.571	2.133	2.037	2.038	2.023	2.024	1.966	1.904	1.911	1.987	1.897	1.873	1.863
SYM4:	Prim.	18.024	93.915	93.915	93.915	94.154	94.154	90.572	87.717	83.817	93.013	90.572	90.572	90.572
100	Misses	4.005	6.085	5.031	5.061	4.980	5.000	4.671	4.198	4.090	4.896	4.274	4.154	4.106
	Cost	2.581	2.217	2.017	2.022	2.005	2.008	1.982	1.920	1.939	2.000	1.906	1.883	1.874
FP4:	Prim.	25.871	97.490	97.490	97.490	96.407	96.407	96.285	92.418	87.965	97.234	96.285	96.285	96.285
100,000,000	Misses	1.505	2.510	1.755	1.816	2.110	2.150	1.661	1.473	1.462	1.670	1.488	1.483	1.486
	Cost	2.027	1.502	1.359	1.370	1.437	1.444	1.353	1.356	1.398	1.345	1.320	1.319	1.319
FP4:	Prim.	25.871	97.322	97.322	97.322	96.725	96.725	96.190	92.360	87.940	97.091	96.190	96.190	96.190
1,000,000	Misses	1.673	2.678	1.954	1.997	2.215	2.241	1.834	1.662	1.648	1.851	1.680	1.674	1.672
	Cost	2.059	1.536	1.398	1.406	1.454	1.458	1.387	1.392	1.434	1.381	1.357	1.356	1.356
FP4:	Prim.	25.870	96.449	96.449	96.449	95.896	95.896	95.527	91.884	87.707	96.248	95.527	95.527	95.527
100,000	Misses	2.775	3.551	2.930	2.934	3.111	3.127	2.792	2.752	2.764	2.827	2.730	2.732	2.731
	Cost	2.269	1.710	1.592	1.593	1.632	1.635	1.575	1.604	1.648	1.575	1.563	1.564	1.564
FP4:	Prim.	25.860	94.603	94.603	94.603	94.428	94.428	92.958	89.199	85.761	93.942	92.958	92.958	92.958
10,000	Misses	2.974	5.397	4.923	4.917	4.698	4.715	4.466	3.705	3.315	4.407	3.894	3.584	3.463
	Cost	2.307	2.079	1.989	1.988	1.948	1.952	1.919	1.812	1.772	1.898	1.810	1.751	1.728
FP4:	Prim.	25.760	92.266	92.266	92.266	92.612	92.612	88.822	83.595	78.676	91.231	88.822	88.822	88.822
1,000	Misses	2.957	7.734	7.015	7.000	6.339	6.326	5.251	3.547	3.138	5.359	3.847	3.351	3.208
	Cost	2.304	2.547	2.410	2.407	2.278	2.276	2.109	1.838	1.809	2.106	1.843	1.748	1.721
FP4:	Prim.	24.783	90.702	90.702	90.702	91.463	91.463	86.461	79.883	72.953	89.596	86.461	86.461	86.461
100	Misses	2.978	9.298	7.074	7.116	6.300	6.357	5.287	3.559	3.162	5.510	3.855	3.380	3.239
	Cost	2.318	2.860	2.437	2.445	2.282	2.293	2.140	1.877	1.871	2.151	1.868	1.778	1.751
MIX6:	Prim.	14.739	95.460	95.460	95.460	96.274	96.274	94.178	92.912	87.799	95.155	94.178	94.178	94.178
100,000,000	Misses	2.719	4.540	2.862	2.873	3.024	3.031	2.628	2.690	2.727	2.556	2.508	2.528	2.665
	Cost	2.369	1.908	1.589	1.591	1.612	1.613	1.558	1.582	1.640	1.534	1.535	1.538	1.565
MIX6:	Prim.	14.739	95.308	95.308	95.308	96.129	96.129	94.110	92.873	87.782	95.041	94.110	94.110	94.110
1,000,000	Misses	2.845	4.692	3.033	3.035	3.179	3.183	2.783	2.828	2.856	2.731	2.668	2.683	2.805
	Cost	2.393	1.938	1.623	1.623	1.643	1.643	1.588	1.609	1.665	1.568	1.566	1.569	1.592
MIX6:	Prim.	14.738	94.452	94.452	94.452	95.299	95.299	93.615	92.536	87.636	94.300	93.615	93.615	93.615
100,000	Misses	3.690	5.548	3.949	3.941	4.041	4.046	3.769	3.704	3.694	3.845	3.708	3.697	3.694
	Cost	2.554	2.110	1.806	1.804	1.815	1.816	1.780	1.778	1.825	1.788	1.768	1.766	1.766
MIX6:	Prim.	14.733	92.788	92.788	92.788	93.443	93.443	91.469	90.409	86.370	92.146	91.469	91.469	91.469
10,000	Misses	5.334	7.212	5.746	5.739	5.975	5.979	5.714	5.509	5.481	5.748	5.557	5.531	5.529
	Cost	2.866	2.442	2.164	2.163	2.201	2.202	2.171	2.143	2.178	2.171	2.141	2.136	2.136
MIX6:	Prim.	14.677	91.673	91.673	91.673	92.233	92.233	89.114	86.419	81.397	90.682	89.114	89.114	89.114
1,000	Misses	4.803	8.327	6.811	6.808	7.142	7.133	5.922	5.059	4.898	6.190	5.276	5.069	5.003
	Cost	2.766	2.665	2.377	2.377	2.435	2.433	2.234	2.097	2.117	2.269	2.111	2.072	2.059
MIX6:	Prim.	14.137	90.733	90.733	90.733	90.925	90.925	87.288	83.038	74.897	89.685	87.288	87.288	87.288
100	Misses	4.809	9.267	6.681	6.814	6.922	7.021	5.985	5.052	4.904	6.318	5.261	5.077	5.013
	Cost	2.772	2.853	2.362	2.387	2.406	2.425	2.264	2.130	2.183	2.303	2.127	2.092	2.080

Table 15: Data for Multitasking Traces on 32K Caches

1982.

[10] Kimming So and Rudolph N. Rechtschaffen, "Cache Operations by MRU-Change," in *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers*, Port Chester, New York, pp. 584–586, October 6–9, 1986.

Input	Rate	FA	DM	VCF	VC4	RVF	RV4	MC2	MC4	MC8	CA	HH2	HH4	HH8
SYM4:	Prim.	19.066	99.099	99.099	99.099	99.126	99.126	98.531	97.609	95.243	98.978	98.531	98.531	98.531
100,000,000	Misses	0.629	0.901	0.834	0.835	0.777	0.777	0.678	0.643	0.624	0.677	0.633	0.612	0.607
	Cost	1.929	1.180	1.167	1.168	1.156	1.156	1.143	1.146	1.166	1.139	1.135	1.131	1.130
SYM4:	Prim.	19.066	98.661	98.661	98.661	98.659	98.659	98.158	97.375	95.146	98.522	98.158	98.158	98.158
1,000,000	Misses	1.196	1.339	1.280	1.282	1.251	1.251	1.177	1.173	1.178	1.184	1.158	1.154	1.155
	Cost	2.037	1.268	1.257	1.257	1.251	1.251	1.242	1.249	1.272	1.240	1.239	1.238	1.238
SYM4:	Prim.	19.065	97.863	97.863	97.863	97.897	97.897	97.097	96.203	94.448	97.571	97.097	97.097	97.097
100,000	Misses	1.507	2.137	2.089	2.091	2.026	2.026	1.795	1.626	1.545	1.800	1.663	1.593	1.556
	Cost	2.096	1.427	1.418	1.419	1.406	1.406	1.370	1.347	1.349	1.366	1.345	1.332	1.325
SYM4:	Prim.	19.055	97.426	97.426	97.426	97.555	97.555	96.344	94.872	92.665	97.073	96.344	96.344	96.344
10,000	Misses	1.434	2.574	2.520	2.519	2.362	2.362	1.828	1.545	1.462	1.875	1.597	1.510	1.476
	Cost	2.082	1.515	1.505	1.504	1.473	1.473	1.384	1.345	1.351	1.385	1.340	1.323	1.317
SYM4:	Prim.	18.959	97.280	97.280	97.280	97.460	97.460	96.037	94.337	91.605	96.923	96.037	96.037	96.037
1,000	Misses	1.433	2.720	2.522	2.526	2.359	2.363	1.820	1.544	1.461	1.878	1.593	1.510	1.476
	Cost	2.083	1.544	1.506	1.507	1.474	1.474	1.386	1.350	1.362	1.388	1.342	1.326	1.320
SYM4:	Prim.	18.025	97.165	97.165	97.165	97.394	97.394	95.781	93.916	90.572	96.808	95.781	95.781	95.781
100	Misses	1.434	2.835	2.507	2.510	2.346	2.350	1.821	1.545	1.463	1.879	1.595	1.511	1.478
	Cost	2.092	1.567	1.505	1.505	1.472	1.472	1.388	1.354	1.372	1.389	1.345	1.329	1.323
FP4:	Prim.	25.871	99.645	99.645	99.645	99.643	99.643	98.621	97.490	96.285	99.599	98.621	98.621	98.621
100,000,000	Misses	0.012	0.355	0.273	0.273	0.270	0.271	0.114	0.053	0.015	0.096	0.047	0.018	0.014
	Cost	1.744	1.071	1.055	1.055	1.055	1.055	1.036	1.035	1.040	1.022	1.023	1.017	1.016
FP4:	Prim.	25.871	99.424	99.424	99.424	99.440	99.440	98.379	97.322	96.190	99.321	98.379	98.379	98.379
1,000,000	Misses	0.311	0.576	0.501	0.501	0.482	0.484	0.357	0.304	0.281	0.342	0.291	0.265	0.262
	Cost	1.800	1.115	1.101	1.101	1.097	1.098	1.084	1.084	1.091	1.072	1.072	1.067	1.066
FP4:	Prim.	25.870	98.621	98.621	98.621	98.821	98.821	97.436	96.449	95.528	98.314	97.436	97.436	97.436
100,000	Misses	0.625	1.379	1.318	1.319	1.103	1.107	1.121	0.892	0.869	1.071	0.891	0.840	0.708
	Cost	1.860	1.276	1.264	1.264	1.221	1.222	1.239	1.205	1.210	1.220	1.195	1.185	1.160
FP4:	Prim.	25.860	98.115	98.115	98.115	98.456	98.456	96.540	94.603	92.958	97.686	96.540	96.540	96.540
10,000	Misses	0.682	1.885	1.800	1.797	1.449	1.448	1.262	0.811	0.777	1.295	0.867	0.762	0.669
	Cost	1.871	1.377	1.361	1.360	1.291	1.290	1.274	1.208	1.218	1.269	1.199	1.179	1.162
FP4:	Prim.	25.760	97.433	97.433	97.433	98.116	98.116	95.144	92.266	88.822	96.873	95.144	95.144	95.144
1,000	Misses	0.689	2.567	2.021	2.031	1.492	1.501	1.603	0.807	0.781	1.587	0.891	0.764	0.673
	Cost	1.873	1.513	1.410	1.412	1.302	1.304	1.353	1.231	1.260	1.333	1.218	1.194	1.176
FP4:	Prim.	24.783	97.000	97.000	97.000	97.975	97.975	93.895	90.702	86.461	96.393	93.895	93.895	93.895
100	Misses	0.701	3.000	1.978	2.006	1.447	1.452	1.640	0.816	0.789	1.684	0.899	0.772	0.681
	Cost	1.885	1.600	1.406	1.411	1.295	1.296	1.373	1.248	1.285	1.356	1.232	1.208	1.190
MIX6:	Prim.	14.735	96.930	96.930	96.930	98.217	98.217	96.436	95.461	94.184	96.794	96.436	96.436	96.436
100,000,000	Misses	1.594	3.070	1.713	1.715	1.685	1.685	1.581	1.601	1.595	1.589	1.590	1.585	1.586
	Cost	2.156	1.614	1.356	1.356	1.338	1.338	1.336	1.350	1.361	1.334	1.338	1.337	1.337
MIX6:	Prim.	14.735	96.550	96.550	96.550	97.845	97.845	96.149	95.309	94.117	96.452	96.149	96.149	96.149
1,000,000	Misses	1.987	3.450	2.103	2.105	2.067	2.068	1.973	1.991	1.987	1.986	1.983	1.982	1.983
	Cost	2.230	1.690	1.434	1.434	1.414	1.415	1.413	1.425	1.436	1.413	1.415	1.415	1.415
MIX6:	Prim.	14.738	95.684	95.684	95.684	96.926	96.926	94.954	94.452	93.615	95.338	94.954	94.954	94.954
100,000	Misses	3.025	4.316	2.986	2.988	2.997	3.000	2.972	3.089	3.071	2.976	2.973	2.929	2.903
	Cost	2.427	1.863	1.611	1.611	1.600	1.601	1.615	1.642	1.647	1.612	1.615	1.607	1.602
MIX6:	Prim.	14.733	95.229	95.229	95.229	96.215	96.215	94.014	92.788	91.470	94.686	94.014	94.014	94.014
10,000	Misses	2.803	4.771	3.458	3.459	3.714	3.715	3.185	2.957	2.848	3.286	2.953	2.819	2.770
	Cost	2.385	1.954	1.705	1.705	1.743	1.744	1.665	1.634	1.626	1.677	1.621	1.596	1.586
MIX6:	Prim.	14.677	94.936	94.936	94.936	95.826	95.826	93.447	91.672	89.113	94.355	93.447	93.447	93.447
1,000	Misses	2.801	5.064	3.686	3.688	4.009	4.006	3.177	2.939	2.846	3.320	2.924	2.813	2.768
	Cost	2.385	2.013	1.751	1.751	1.803	1.803	1.669	1.642	1.650	1.687	1.621	1.600	1.592
MIX6:	Prim.	14.137	94.651	94.651	94.651	95.629	95.629	92.926	90.733	87.288	94.059	92.926	92.926	92.926
100	Misses	2.809	5.349	3.602	3.630	3.940	3.951	3.205	2.944	2.852	3.352	2.930	2.820	2.776
	Cost	2.392	2.070	1.738	1.743	1.792	1.794	1.680	1.652	1.669	1.696	1.628	1.607	1.598

Table 16: Data for Multitasking Traces on 128K Caches