Article Information

Journal Title: IEEE Transactions on Computers

Volume: 38 **Issue:** 12
Month/Year: 1989**Pages:** 1612-1630

Article Author: Hill, M.D., M.D.

Article Title: Evaluating associativity in CPU caches

Evaluating Associativity in CPU Caches

MARK D. HILL, MEMBER, IEEE, AND ALAN JAY SMITH, FELLOW, IEEE

Abstract—**Because of the infeasibility or expense of large fully-associative caches, cache memories are usually designed to be set-associative or direct-mapped. This paper presents 1) new and efficient algorithms for simulating alternative direct-mapped and set associative caches, and 2) uses those algorithms to quantify the effect of limited associativity on the cache miss ratio.**

We introduce a new algorithm, *forest simulation*, for simulating alternative direct-mapped caches and generalize one, which we call *all-associativity simulation*, for simulating alternative direct-mapped, set-associative, and fully-associative caches. We find that while all-associativity simulation is theoretically less efficient than forest simulation or stack simulation (a commonly used simulation algorithm); in practice, it is not much slower and allows the simulation of many more caches with a single pass through an address trace.

We also provide data and insight into how varying associativity affects the miss ratio. We show: 1) how to use simulations of alternative caches to isolate the cause of misses; 2) that the principal reason why set-associative miss ratios are larger than fully-associative ones is (as one might expect) that too many active blocks map to a fraction of the sets even when blocks map to sets in a uniform random manner; and 3) that reducing associativity from eight-way to four-way, from four-way to two-way, and from two-way to direct-mapped causes relative miss ratio increases in our data of respectively about 5, 10, and 30 percent, consistently over a wide range of cache sizes and a range of line sizes.

Index Terms—**Associativity, buffer, cache memory, computer architecture, direct-mapped, memory systems, performance evalulation, set-associative and trace-driven simulation algorithms.**

I. Introduction

THREE important CPU cache parameters are cache size, block (line) size, and associativity [27]. Cache size (buffer size, capacity) is so important that it is a part of almost all cache studies (for a partial bibliography see [29]). Block size (line size) has recently been examined in detail in [30]. Here we concentrate on associativity (degree of associativity, set size) which is the number of places in a cache where a block can reside.

Selecting optimal associativity is important, because changing associativity has a significant impact on cache performance and cost. Increasing associativity improves the likelihood that a block is resident by decreasing the probability that too many recently-referenced blocks map to the same place and by allowing more blocks to be considered for replacement. The effect of associativity on cache miss ratio has never been isolated and quantified, and that is one of the major goals of this paper. Conversely, increasing associativity often increases cache cost and access time, since more blocks (frames) must be searched in parallel to find a reference [16].

Fig. 1 illustrates set-associativity. A set-associative cache uses a *set-mapping function f* to partition all *blocks* (data in an aligned, fixed-sized region of memory) into a number of equivalence classes. Some number of *block frames* in the cache are assigned to hold recently-referenced blocks from each equivalence class. Each group of block frames is called a *set*. The number of such groups, equal to the number of equivalence classes, is called the *number of sets (s)*. The number of block frames in each set is called the *associativity* (degree of associativity, set size, n). The number of block frames in the cache (c) always equals the associativity times the number of sets ($c = n \cdot s$). A cache is *fully-associative* if it contains only one set ($n = c, s = 1$), is *direct-mapped* if each set contains one block frame ($n = 1, s = c$), and is *n-way set-associative* otherwise (where n is the associativity, $s = c/n$).

On a reference to block x, the set-mapping function f feeds the "set decoder" with $f(x)$ to select one set (one row), and then each block frame in the set is searched until x is found (a cache hit) or the set is exhausted (a cache miss). On a cache miss, one block in set $f(x)$ is replaced with the block x obtained from memory. Finally, the word requested from block x is returned to the processor. Here for conceptual simplicity we show the word within the block selected last (in the box "compare block number with tags and select data word"). Many implementations, however, select the word within the block while selecting the set to reduce the number of bits that must be read; i.e., only words are gated into the multiplexer, not full lines. The most commonly used set-mapping function is the block number modulo the number of sets, where the number of sets is a power of two. This set mapping function is called *bit selection* since the set number is just the number given by the low-order bits of the block address. For 256 sets, for example, $f(x) = x \bmod 256$ or $f(x) = x$ AND 0*xff*, where mod is remainder and AND is bitwise AND.

The method we use for examining associativity in CPU caches is *trace-driven simulation*. It uses one or more (ad-

Manuscript received February 14, 1989; revised July 10, 1989. The work presented here is based on research supported in part by the National Science Foundation under Grants CCR-8202591 and MIP-8713274, by the State of California under the MICRO program, by the Defense Advanced Research Projects Agency monitored by Naval Electronics Systems Command under Contract N00039-85-C-0269, the graduate school at the University of Wisconsin-Madison, and by IBM Corporation, Digital Equipment Corporation, Philips Research Laboratories/Signetics Corporation, Apple Computer Corporation, and the Hewlett Packard Corporation.

M. D. Hill is with the Department of Computer Sciences, University of Wisconsin, Madison, WI, 53706.

A. J. Smith is with the Computer Science Division, Department of Electrical Engineering and Computer Science, University of California, Berkely, CA 94720.

IEEE Log Number 8931174.

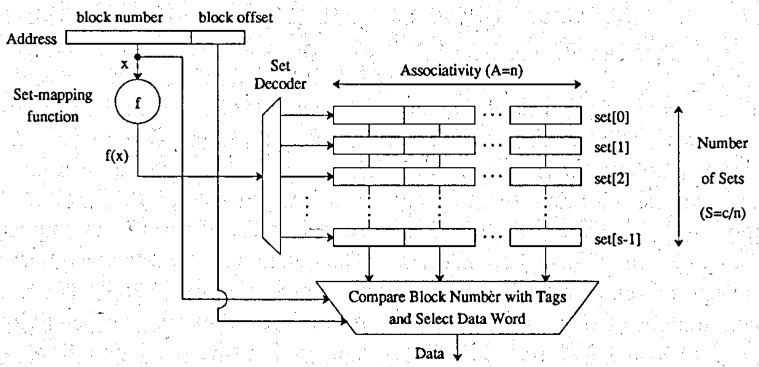


Fig. 1. Set-associative mapping.

dress) *traces* and a (cache) *simulator*. A trace is the log of a dynamic series of memory references, recorded during the execution of a program or workload. The information recorded for each reference must include the address of the reference and may include the reference's type (instruction fetch, data read, or data write), length, and other information. A *simulator* is a program that accepts a trace and parameters that describe one or more caches, mimics the behavior of those caches in response to the trace, and computes performance metrics (e.g., miss ratio) for each cache.

We analyze associativity in caches with trace-driven simulation for the same reasons as are discussed in [28]. The principal advantage of trace-driven simulation over random number driven simulation or analytical modeling is that there exists no generally-accepted model for program behavior (at the cache level) with demonstrated validity and predictive power. The major disadvantage is that workload samples must be relatively short, due to disk space and simulation time limits.

The CPU time required to simulate many alternative caches with many traces can be enormous. Mattson *et al.* [19] addressed a similar problem for virtual memory simulation by developing a technique we call *stack* simulation, which allows miss ratios for all memory sizes to be computed simultaneously, during one pass through the address trace, subject to several constraints including a fixed page size. While stack simulation can be applied to caches, each cache configuration with a different number of sets requires a separate simulation. For this reason, this paper first examines better algorithms for simulating alternative direct-mapped and set-associative caches, and then uses those algorithms to study associativity in caches.

The rest of this paper is organized as follows. Section II reviews previous work on cache simulation algorithms and associativity in caches. In Section III, we explain our methods in more detail and describe our traces. Section IV discusses cache simulation algorithms, including properties that facilitate rapid simulation, a new algorithm for simulating alternative direct-mapped caches, and an extension to an algorithm for simulating alternative caches with arbitrary set-mapping functions. Section V examines the effect of associativity on miss ratio, including categorizing the cause of misses in set-associative caches, relating set-associative miss ratios to fully-associative ones, comparing miss ratios from similar set-associative caches, and extending the *design target miss ratios* from [28] and [30] to caches with reduced associativity.

Readers interested in the effect of associativity on miss ratio but not in cache simulation algorithms may skip Section IV, as Section V is written to stand alone.

II. Related Work

A. Simulation Algorithms

The original paper on memory hierarchy simulation is by Mattson *et al.* [19]. They introduce *inclusion*, show when inclusion holds, and develop *stack simulation*, which uses inclusion to rapidly simulate alternative caches. *Inclusion* is the property that after any series of references, larger alternative caches always contain a superset of the blocks in smaller alternative caches.[1] Mattson *et al.* show inclusion holds between alternative caches that have the same block size, do no prefetching, use the same set-mapping function (and therefore have the same number of sets), and use replacement algorithms that before each reference induce a total priority ordering on all previously referenced blocks (that map to each set) and use only this priority ordering to make the next replacement decision. Replacement algorithms which meet the above condition, called *stack algorithms*, include LRU, OPTIMUM, and (if properly defined) RANDOM [6]. FIFO does not qualify since cache capacity affects a block's replacement priority. In Section IV-A, we will prove when inclusion holds for caches that use arbitrary set-mapping functions and LRU replacement.

Mattson *et al.* develop *stack simulation* to simulate alternative caches that have the same block size, do no prefetching, use the same set-mapping function, and use a stack replacement algorithm. Since inclusion holds, a single list per set, called a *stack*, can be used to represent caches of all associatives, with the first n elements of each stack representing the blocks in an n-way set-associative cache. For each reference, stack simulation performs three operations: 1) locate the reference in the stack, 2) update one or more metrics to indicate which caches contained the reference, and 3) update the stack to reflect the contents of the caches after the reference. We

[1] *Inclusion* is different from *multilevel inclusion* defined by Baer and Wang [5]. While inclusion is a property relating alternative caches, multilevel inclusion relates caches in the same cache hierarchy.

call these three operations FIND, METRIC, and UPDATE, and will show that the algorithms discussed in later in Sections IV-B and IV-C use the same steps.

The most straightforward implementation of stack simulation is to implement each stack with a linked list and record hits to position n by incrementing a counter *distance[n]*. After N references have been processed, the miss ratio of an n-way set-associative cache is simply $1 - \sum_{i=1}^{n} distance[i]/N$. Since performance with a linked list will be poor if many elements of a stack must for searched on each reference, other researchers have developed more complex implementations of stack simulation, using hash tables, m-ary trees, and AVL trees [8], [21], [33]. While these algorithms are useful for some memory hierarchy simulations, Thompson [33] concludes that linked list stack simulation is near optimal for most CPU cache simulations. Linked list stack simulation is fast when few links are traversed to find a reference. On average, this is the case in CPU cache simulations since 1) CPU references exhibit a high degree of locality, and 2) CPU caches usually have a large number of sets and limited associativity, dividing active blocks among many stacks and bounding maximum stack size; different results are found for file system and database traces. For this reason, we consider only linked list stack simulation further, and use *stack simulation* to refer to linked list stack simulation.

Mattson *et al.* also briefly mention a way of simulating caches with different numbers of sets (and therefore different set-mapping functions). In two technical reports, Traiger and Slutz extend the algorithms to simulate alternative caches with different numbers of sets and block sizes [34], and with different numbers of sets, block sizes, and subblock sizes (sector and block sizes, address and transfer block sizes) [24]. They require that all alternative caches use LRU replacement, bit-selection for set mapping, and have block and subblock sizes that are powers of two. (Bit selection uses some of the bits of the block address as a binary number to specify the set.) In Section IV-C, we generalize to arbitrary set-mapping functions their algorithm for simulating alternative caches that use bit selection.

The speed of stack simulation can also be improved by deleting references (trace entries) that will hit and not affect replacement decisions in the caches to be simulated [25]. Puzak [23] shows that if all caches simulated use bit selection and LRU replacement, references that hit the most recently used element of a set can be deleted without affecting the total number of misses. We will show that this result trivially follows from properties we define in Section IV-A, allowing such references to be deleted from traces before using any of our simulation algorithms. (The total number of memory references in the original trace must be retained, in order to compute the miss ratio.)

B. Associativity

Previous work on associativity can be broken into the following three categories: 1) papers that discuss associativity as part of a more general analysis of 32 kbyte and smaller caches, among the more notable of which are [18], [17], [7], [32], [27],[2] and [11], and [13]; 2) papers that discuss associativity and other aspects of cache design for larger caches ([4], [2], and [22]); and 3) those that discuss only associativity ([26] and [16]). Since caches have been getting larger, papers in category 1) can also be characterized as older, while those in category 2) are more recent.

Papers in category 1) provide varying quantities of data regarding the effect of changing associativity in small caches. The qualitative trend they support is that changing associativity from direct-mapped to two-way set-associative improves miss ratio, doubling associativity to four-way produces a smaller improvement, doubling again to eight-way yields an even smaller improvement, and subsequent doublings yield no significant improvement. Our quantitative results are consistent with results in these papers. We extend their results by examining relative miss ratio changes to isolate the effect of associativity from other cache aspects, and by examining some larger caches.

Alexander *et al.* use trace-driven simulation to study small and large caches [4]. Unfortunately, the miss ratios they give are much lower than those that have been measured with hardware monitors and real workloads; see [28] for reports of real measurements.

Agarwal *et al.* use traces gathered by modifying the microcode of the VAX 8200 to study large caches and to try to separate operating system and multiprogramming effects [2]. They briefly examine associativity, where they find that associativity in large caches impacts multiprogramming workloads more strongly than uniprocessor workloads. They find for one workload that decreasing associativity from two-way to direct-mapped increases the multiprogramming miss ratio by 100 percent and the uniprogramming miss ratio by 43 percent. These numbers are much larger than the average miss ratio change we find (30 percent).

Przybylski *et al.* [22] examine cache implementation tradeoffs. They find that reducing associativity from two-way to direct-mapped increases miss ratio 25 percent, regardless of cache size, which is consistent with our results. One contribution of that paper is a method of translating the architectural impact of a proposed design change into time by computing the cache hit time increase that will exactly offset the benefit of the proposed change. A change improves performance only if the additional delay required to implement the change is less than the above increase. Przybylski *et al.* find that the architectural impact times for increasing associativity are often small, especially for large caches, calling into question the benefit of wide associativity.

The first paper to concentrate exclusively on associativity is [26]. That paper presents a model that allows miss ratios for set associative caches to be accurately derived from the fully associative miss ratio. In Section V-B, we further validate those results by showing that the model accurately relates the miss ratios of many caches, including large direct-mapped caches, to LRU distance probabilities.

The second paper to concentrate on associativity is [16], based on parts of [15]. It shows that many large single-level

[2] This survey includes results for some large caches with wide associativity (e.g., 32-way set-associative 64 kbyte caches).

caches in uniprocessors should be direct-mapped, since the drawbacks of direct-mapped caches (e.g., worse miss ratios and more-common worst case behavior) have small significance for large caches with small miss ratios, while the benefits of direct-mapped caches (lower cost and faster access time) do not diminish with increasing cache size. Here we examine miss ratio in more detail, but do not discuss implementation considerations.

TABLE I
DATA ON TRACES

Five-Trace Group			
Trace Sample Name	Instruction References (%)	Length (1000's of references)	Dynamic Size (K-bytes)
mul2_2nd500K	53	500	218
mul8_2nd500K	51	500	292
ue_2nd500K	55	500	277
mvs1_2nd500K	52	500	163
mvs2_2nd500K	55	500	201

23-Trace Group			
Trace Name	Instruction References (%)	Length (1000's of references)	Dynamic Size (K-bytes)
dec0	50	362	120
	50	353	125
fora	52	388	144
forf	52	401	128
	53	387	152
	53	414	105
	52	368	205
fsxzz	51	239	104
ivex	60	342	210
macr	55	343	199
memxx	49	445	139
mul2	52	386	204
	53	383	169
	56	367	165
mul8	51	408	218
	54	390	196
	46	429	194
null	58	170	55
savec	50	432	94
	61	228	54
ue	56	358	205
	57	372	191
	55	364	221

III. Methods and Traces

In this section, we discuss the use of the miss ratio as a suitable metric (among others), describe the traces that we use, show how we estimate average steady-state miss ratios, and show that our traces yield results consistent with those observed from running systems.

To first order, the effective access time of a cache can be modeled as $t_{\text{cache}} + \text{miss\_ratio} \cdot t_{\text{memory}}$. (Additional factors which affect access time including the overhead of write backs, extra time for line crossers, page crossers, and TLB misses, and the fact that writes may be slower than reads. These latter delays are much less significant than those given in the expression.) The *miss ratio* is the number of cache misses divided by the number of memory references, t_{memory} is the time for a cache miss, and t_{cache} is the time to access the cache on a hit. The two latter parameters are implementation dependent, and in [15] there is a discussion of their effect on cache performance. As noted earlier, increases in associativity, while generally improving the miss ratio, can increase access time, and thus degrade overall performance. Here, we concentrate on miss ratio because it is easy to define, interpret, compute, and is implementation independent. This independence facilitates cache performance comparisons between caches not yet implemented and those implemented with different technologies and in different kinds of systems.

Results in this paper are based on two partially overlapping groups of traces, called the *five-trace* and *23-trace* groups, respectively. Table I presents data on the traces. The first column gives the name of each trace sample. The second gives the fraction of all references that are instruction references. In these simulations, we do not distinguish between data reads and writes. The third column gives the length of the address traces in 1000's of references. The final column gives the number of distinct bytes referenced by the trace, where any reference in an aligned 32-byte block is considered to have touched each byte in the block.

Each of the trace samples in the five-trace group comes from the second 500 000 references of a longer trace. The first three samples are user and system VAX-11 traces gathered with ATUM [1]. Trace *mul2_2nd500k* contains a circuit simulator and a microcode address allocator running concurrently under VMS. Trace *mul8_2nd500k* is an eight-job multiprogrammed workload under VMS: spice, alloc, a Fortran compile, a Pascal compile, an assembler, a string search in a file, jacobi (a numerical benchmark) and an octal dump. Trace *ue_2nd500k* consists of several copies of a program that simulates interactive users running under Ultrix. The other two samples in the trace group, *mvs1_2nd500k* and *mvs2_2nd500k*, are collections of IBM 370 references from system calls invoked in two Amdahl standard MVS workloads [28].

The second trace group contains 23 samples of various workloads gathered on a VAX-11 with ATUM [1]. Trace samples that exhibit unstable behavior (e.g., a particular doubling of cache size or associativity alters the miss ratio observed by many factors of two) have been excluded from both groups.

We estimate the steady-state miss ratios for a trace sample using the miss ratio for a trace after the cache is *warm* (the *warm-start miss ratio*). A cache is *warm* if its future miss ratio is not significantly affected by the cache recently being empty [2]. We compute warm-start miss ratios using the second 250K references of each 500K-reference trace sample. We found that most caches with our traces are warm by 250K references by locating the knee in the graph of the cumulative misses to empty block frames versus references, a method of determining when caches are warm proposed in Agarwal *et al.* [2]. Furthermore, results for these multiprogrammed traces properly include cold-start effects whenever a process resumes execution.

Fig. 2(a) and (b) displays miss ratio data for unified caches (mixed, i.e., cache data and instructions together) with 32-byte blocks. Solid lines show the average warm-start miss ratios with different associativities (1, 2, 4, and 8). The average warm-start miss ratio is the arithmetic average of warm-start miss ratios for each of the five traces in the five-trace group. The arithmetic mean is used because it represents the miss ratio of a workload consisting of an equal number of references from each of the traces. Previous experiments (as were done for [31] and [15]) showed that little difference was observed when other averaging methods were used. The dashed line (labeled "inf") gives the warm-start miss ratio of an infi-

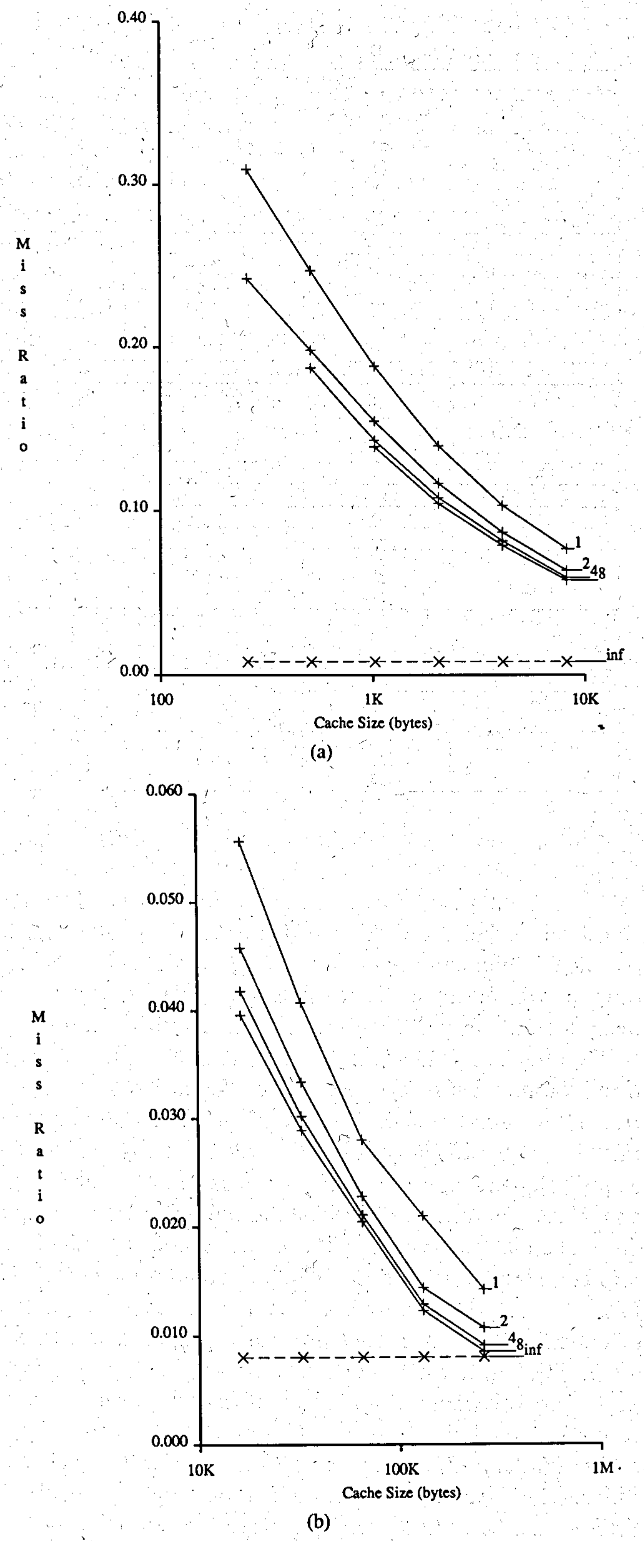


Fig. 2. Miss ratios for five-trace workload with caches of associativities of 1, 2, 4, and 8. The dashed line shows the miss ratio for an infinite cache. (a) Smaller caches. (b) Larger caches.

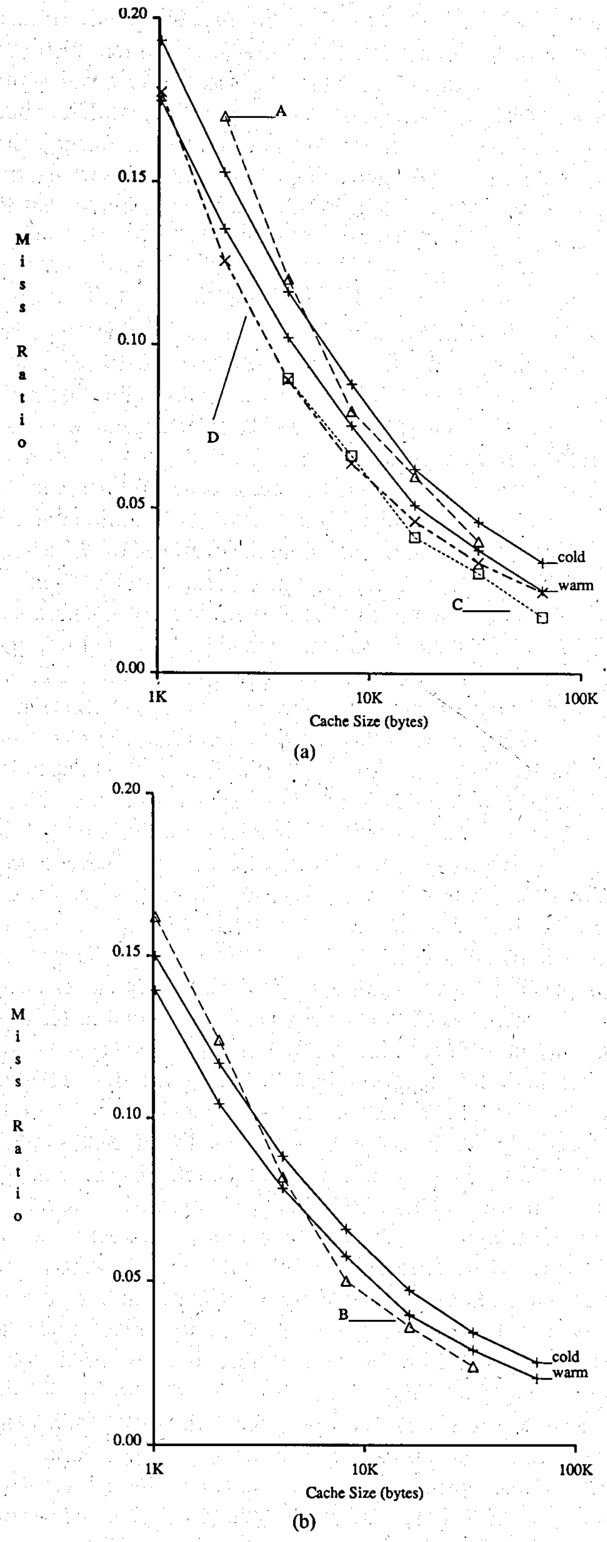


Fig. 3. Comparison of our miss ratio data (solid lines) with other published data (*A*, *B*, *C*, *D*). (a) 16-byte blocks. (b) 32-byte blocks.

nite cache, a cache so large that it never replaces any blocks. Measurements for the 23-trace group are similar.

Fig. 3 compares miss ratios for the five-trace group in eight-way set-associative unified caches, having 16-byte and 32-byte blocks, to miss ratios from other sources. Line "cold" measures miss ratios from an empty cache, while line "warm" does not count misses until after 250K references. Since the trace samples include multiprogramming effect, both contain some cold-start misses [12]. Lines labeled *A* and *B* show the design target miss ratios for fully-associative caches from [28] and [30]. The line labeled *C* from [2] shows four-way set-associative miss ratio results from Fig. 17 in that paper. Finally, the line labeled D from [27] shows four-, six- and eight-way set-associative miss ratios taken from hardware monitor measurements on an Amdahl 470 (Fig. 33 of that paper, assuming 50 percent supervisor execution). Fig. 3 demonstrates that the miss ratios of the five-trace group are consistent with those measured and/or proposed for actual operating environments.

Despite the similarities with previously published data, miss ratio data for large caches (greater than 64K bytes) are subject to greater error, since only a few thousand misses may occur during a trace sample. To reduce sensitivity to such error, results in Section V concentrate on the relationship between the miss ratios of alternative caches rather than on the miss ratio values themselves.

IV. Simulation Techniques for Alternative Direct-Mapped and Set-Associative Caches

In this section we first discuss two properties, *set refinement* and *inclusion*, that facilitate the rapid simulation of alternative caches. We then develop a new algorithm that uses both set-refinement and inclusion to rapidly simulate alternative direct-mapped caches. Next we generalize an algorithm that simulates alternative set-associative caches using bit selection [34] to one that allows arbitrary set-mapping functions. Finally we compare implementations of the algorithms.

A. Properties that Facilitate Rapid Simulation

Two properties useful for simulating alternative direct-mapped and set-associative caches are *set-refinement*[3] (introduced below) and *inclusion* (introduced in Mattson *et al.* [19]). Here we discuss these properties with respect to caches that have the same block size, do no prefetching, use LRU replacement, have arbitrary associativities, and can use arbitrary set-mapping functions. Let $C_1(A = n_1, F = f_1)$ and $C_2(A = n_2, F = f_2)$ be two such caches, where cache C_i has associativity n_i and set-mapping function f_i, $i = 1, 2$.

Definition 1: Set-refinement: Set-mapping function f_2 *refines* set-mapping function f_1 if $f_2(x) = f_2(y)$ implies $f_1(x) = f_1(y)$, for all blocks x and y.

Furthermore, cache $C_2(A = n_2, F = f_2)$ is said to *refine* an alternative cache $C_1(A = n_1, F = f_1)$ if set-mapping function f_2 *refines* set-mapping function f_1. *Refines* is so named becauses f_2 *refines* f_1 implies set-mapping function f_2 induces a *finer* partition on all blocks than does f_1. Since set refinement is clearly transitive, if f_{i+1} refines f_i for each $i = 1, L - 1$ then f_j refines f_i for all $j > i$, implying a hierarchy of sets. We will use set refinement to facilitate the rapid simulation of alternative direct-mapped caches (Section IV-B) and set-associative caches (Section IV-C).

Definition 2: Inclusion: Cache $C_2(A = n_2, F = f_2)$ *includes* an alternative cache $C_1(A = n_1, F = f_1)$ if, for any block x after any series of references, x is resident in C_1 implies x is resident in C_2.

Thus, when cache C_2 includes cache C_1, C_2 always contains a superset of the blocks in C_1. Inclusion facilitates rapid simulation of alternative caches by allowing hits in larger caches to be inferred from hits detected in smaller ones. Mattson *et al.* [19] show when inclusion holds for alternative caches that use the same set-mapping function (and hence the same number of sets). Next we show when it holds with LRU replacement and arbitrary set-mapping functions.

Theorem 1: Given the same block size, no prefetching and LRU replacement, cache $C_2(A = n_2, F = f_2)$ includes cache $C_1(A = n_1, F = f_1)$ if and only if set-mapping function f_2 refines f_1 (set-refinement) and associativity $n_2 \geq n_1$ (nondecreasing associativity).

Proof: Suppose cache C_2 includes cache C_1. Suppose further that a large number of blocks map to each set in both caches, as is trivially true for practical set-mapping functions (e.g., bit selection). To demonstrate that inclusion implies both set-refinement and nondecreasing associativity, we show that a block can be replaced in cache C_1 and still remain in cache C_2, violating inclusion, if either 1) set-refinement does not hold or 2) set-refinement holds but the larger cache has the smaller associativity.

1) If cache C_2 does not refine cache C_1, then there exists at least one pair of blocks x and y such that $f_2(x) = f_2(y)$ and $f_1(x) \neq f_1(y)$. Since we assume many blocks map to each set, there exist many blocks z_i for which $f_2(z_i) = f_2(x) = f_2(y)$. Since $f_1(x) \neq f_1(y)$, either $f_1(z_i) \neq f_1(x)$ or $f_1(z_i) \neq f_1(y)$ (or both), implying set-refinement is violated many times. Without loss of generality, assume that many z_i's map to different f_1 sets than x (otherwise, many map to a different f_1 sets than y). Let n_2 of these be denoted by $w_1, \cdots, w_{n_2}$.[4] Consider references to $x, w_1, \cdots, w_{n_2}$. Inclusion is now violated since x is in cache C_1, but not in cache C_2. It is in cache C_1, because blocks $w_1, \cdots, w_{n_2}$ mapped to other sets than x and could not force its replacement; x is replaced in n_2-way set-associative cache C_2, since LRU replacement is used and the n_2 other blocks mapped to its set are more recently referenced.

2) Let $x_0, \cdots, x_{n_2}$ be a collection of blocks that map to the same f_2 set. Since we are assuming f_2 refines f_1, they also map the same f_1 set. Consider references to $x_0, x_1, \cdots, x_{n_2}$. Inclusion is now violated since x_0 is in n_1-way set-associative cache C_1, but not in n_2-way set-associative cache C_2($n_1 > n_2$ implies $n_1 \geq n_2 + 1$).

Suppose cache C_2 refines cache C_1 and $n_2 \geq n_1$. Initially both caches are empty and inclusion holds, because everything (nothing) in cache C_1 is also in cache C_2. Consider the first time inclusion is violated, i.e., some block is in cache C_1 that is not in cache C_2. This can only occur when some block x_0 is replaced from cache C_2, but not from cache C_1. A block x_0 can only be replaced from cache C_2 if n_2 blocks, x_1 through x_{n_2}, all mapping to $f_2(x_0)$, are referenced after it. By set-refinement, $f_1(x_0) = f_1(x_1) = \cdots = f_1(x_{n_2})$. Since $n_2 \geq n_1$, x_0 must also be replaced in cache C_1. □

Several corollaries, used to develop the cache simulation algorithms in the next two sections, follow directly from the above definitions and theorem.

1) If cache C_2 refines cache C_1 and their set-mapping functions f_2 and f_1 are different (partition blocks differently), then cache C_2 has more sets than cache C_1. The number of sets

[3] *Set-refinement* is called *set-hierarchy* in [15].

[4] Blocks $w_1, \cdots, w_{n_2}$ exist if at least $2n_2$ blocks map to set $f_2(x)$.

in a cache is equal to the number of classes in the partition induced by its set-mapping function. If f_2 has fewer classes than f_1 and at least one block maps to every f_1 class, set-refinement is violated since some pair of those blocks must map to the same f_2 class. If f_2 has the same number of classes as f_1 and at least one block maps to every f_1 class, then there exists a one-to-one correspondence between f_2 classes and f_1 classes, implying both functions induce the same partition.

2) If bit selection is used, a cache with 2^i sets refines one with 2^j ones, for all $i \geq j$. That is, set-mapping function $x \bmod 2^i$ refines $x \bmod 2^j$, $i \geq j$. For all blocks x and y ($x \bmod 2^i = y \bmod 2^i$) implies ($x \bmod 2^j = y \bmod 2^j$), because 2^i can be factored into positive integers 2^{i-j} and 2^j, and ($x \bmod ab = y \bmod ab$) implies ($x \bmod b = y \bmod b$), for all positive integers a and b.

3) Cache C_2 must be strictly larger than a *different* cache C_1 to include it. Two caches are different if they can contain different blocks (after some series of references). If cache C_2 is smaller than cache C_1, inclusion is violated whenever C_1 is full. If C_2 and C_1 are the same size, different, and both full, then inclusion will be violated whenever they hold different blocks.

4) Set refinement implies inclusion in direct-mapped caches. By Theorem 1, inclusion requires set-refinement and nondecreasing associativity. Since all direct-mapped caches have associativity one, only set-refinement is necessary.

5) Inclusion holds between direct-mapped caches using bit selection. Implied by corollaries 2) and 4).

6) Inclusion does not hold between many pairs of different set-associative caches. It does not hold a) between two different set-associative caches of the same size [by corollary 3)], b) if the larger cache has smaller associativity (Theorem 1), and c) if set-refinement is violated (also Theorem 1). Set-refinement can be violated even when bit selection is used (e.g., the larger cache is twice as big but has four times the associativity of the smaller cache).

7) The *includes* relation is a partial ordering of the set of caches. The proof of this, omitted here, need only show that *includes* is reflexive, antisymmetric, and transitive; see [15].

8) Similarly, the *refines* relation is a partial ordering of the set of caches.

9) The *refines* relation can speed the simulation of alternative caches that use LRU replacement. Let these caches be denoted by C_i, $i = 1, 2, \cdots$. Construct a direct-mapped cache $C_0(A = 1, F = f_0)$ such that all caches C_i refine C_0. For arbitrary set-mapping functions, $f_0(x) = 0$ can be used; if all caches C_i use bit selection and have 2^m or more sets, $f_0(x) = x \bmod 2^m$ should be used. In any case, simulation speed can be improved by deleting all references (trace entries) that hit in cache C_0 and recording the deleted references as hits in all caches simulated. Such deletion is possible when caches C_i include cache C_0 and the deleted references would not have affected any replacement decisions [25]. Since each cache C_i refines cache C_0 and C_0 is direct-mapped, all caches C_i include cache C_0 by Theorem 1. All deleted references do not affect LRU replacement decisions since they are all to the most-recently-referenced (MRU) block in each set. To see why this is true for a cache $C_i(A = n_i, F = f_i)$, consider the direct-mapped cache $C_i'(A = 1, F = f_i)$ that always contains the MRU blocks from cache C_i. Cache C_i' refines cache C_0, since cache C_i' has the same set-mapping function as cache C_i and cache C_i refines cache C_0. Since *refines* implies *includes* in direct-mapped caches, all deleted references are in cache C_i' (and therefore to cache C_i's MRU blocks). Puzak shows this result for bit-selection [23].

B. Simulating Direct-Mapped Caches

This section develops a new algorithm, called *forest simulation*, for simulating alternative direct-mapped caches. Forest simulation requires that the set-mapping functions of all caches obey set-refinement. Since typical alternative designs for direct-mapped caches use numbers of sets which are powers of two, with the set selected via bit selection, this algorithm is applicable to the common case.

In the last section, we showed set-refinement implies inclusion in direct-mapped caches. Forest simulation takes advantage of inclusion, as does stack simulation, by searching for a block from the smallest to largest cache. When a block is found, a hit is implicitly recorded for all larger caches.

The data structure used by forest simulation to store cache blocks is a forest (a set of disjoint trees) where the number of levels equals the number of caches simulated, and the number of nodes in level i equals the number of blocks frames in the ith smallest cache. If bit selection is used by all caches, the forest can be stored in an array that contains twice as many elements as the largest cache, since the $i - 1$st smallest cache is at most half the size of the ith smallest cache.

Fig. 4(a) displays a forest for direct-mapped caches of size 1, 2, 4, and 8 block frames. The forest contains only one tree, because the smallest cache has only one block frame, and is binary, because each cache in this example is twice as large as the next smaller cache. We assume here that blocks are mapped to block frames with bit selection. Each node holds the information for one block frame in a direct-mapped cache. Nodes are labeled with the tag values which they could contain if bit selection is used for all caches. The node at the root of the tree has no block number bits constrained, because a one-block direct-mapped cache can hold any block. This is illustrated with a t representing arbitrary high-order bits of the block number and three x's representing DON'T CARES for the three low-order bits. The tags *txx0* and *txx1* in the nodes of level two indicate that the blocks can reside in these nodes are constrained to have even and odd block numbers, respectively. Similar rules with more bits constrained apply to the rest of the levels.

For each reference, the key idea in forest simulation is to begin at level 1 and proceed downward in the forest until the reference is found or the forest exhausted. At each level, the location of the search is guided by the set-mapping function for that level. At each level traversed, the node examined is changed to contain the reference. If the node is found at level i, *distance*[i] is incremented. After N references have been processed, the miss ratio of the ith smallest direct-mapped cache is $1 - \sum_{j=1}^{i} distance[j]/N$.

Consider the example shown in Fig. 4(b) and (c). Fig. 4(b) depicts the forest of Fig. 4(a) after a series of references.

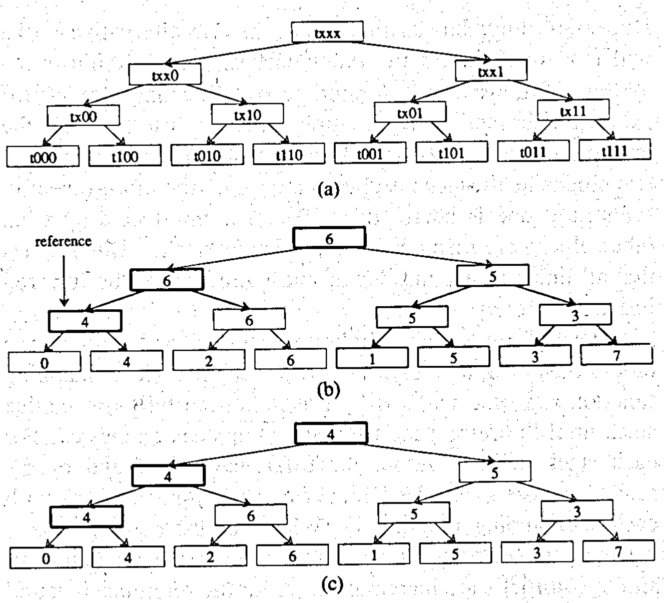


Fig. 4. Forest simulation example: the effect of referencing block 4 on directed-mapped caches of 1, 2, 4, and 8 block frames. (a) A forest with bit selection. (b) Before reference to block 4. (c) After the reference.

Information in the tree tells us that block 6 is in a cache of size one block frame; blocks 6 and 5 are in a direct-mapped cache of size two; blocks 4, 6, 5, and 3 are in a direct-mapped cache of size four; and blocks 0 through 7 are in a direct-mapped cache of size eight. Let the next reference be to block 4. A path from the root to a leaf is determined using the set-mapping function for each cache. A search begins at the root and stops when block 4 is found. All nodes encountered in the search that do not contain block 4 are modified to do so. The nodes in bold are examined to find block 4. Since block 4 is located at level 3, caches at levels 1 and 2 miss and caches at levels 3 and 4 hit. Fig. 4(c) shows the tree after this reference has been processed. The nodes in bold now contain the referenced block.

Fig. 5 shows pseudocode for the algorithm. We will analyze the performance of forest simulation in Section IV-D.

The principal limitation of forest simulation is that it only works for direct-mapped caches. Extending the algorithm to set-associative caches is possible, but complex, since a forest gives only a partial ordering of recently-referenced blocks and set-refinement does not imply inclusion in set-associative caches. Consider using the forest of Fig. 4(b) to simulate a two-block fully-associative cache that uses LRU replacement. It is not possible to tell whether the reference to block 4 hits in such a cache, since any of blocks 2, 4, or 5 could be second-most-recently referenced.

Forest simulation can be extended to simulate n-way set associativity by replacing each node in the forest by an n-element LRU stack. At each reference, rather than just replacing the element at a node with the newest reference, the stack at that node is updated in the normal LRU manner; the descent in the tree stops as soon as the target block is found at level one in the stack at the current node. This is because, by reasoning similar to that used to show corollary 9), the reference will also be at distance one in all further levels. As should be evident, forest simulation (for direct-mapped caches) is a special case of this general algorithm, with the "n-element" stack consisting of only one element.

```
integer L /* number of direct-mapped caches */
/* set-mapping functions that obey set-refinement */
/* i.e., $f_{i+1}$ refines $f_i$ for i=1, ..., L-1. */
function $f_1(x), ..., f_L(x)$
integer $c_1, ..., c_L$ /* cache sizes (in blocks); let $C_i$ be $\sum_{j=1}^{i} c_j$ and $C_0 = 0$ */
integer N /* counts the number of references */
/* distance counts so that miss_ratio(A=1, F=$f_i$) = $1 - \sum_{j=1}^{i}$distance[i]/N */
integer distance[1:L]
integer forest[1:$C_L$] /* the forest */
define map(x, i) = ( $f_i(x) + C_{i-1}$ ) /* maps the forest into an array */

For each reference x {
        read(var x)
        N++
        /* FIND */
        found = FALSE
        for i=1 to L or found {
                y = forest[map(x, i)]
                if (x==y)
                        found = TRUE
                        /* METRIC */
                        distance[i]++
                else
                        /* UPDATE */
                        forest[map(x, i)] = x
        }
}
```

Fig. 5. Forest simulation.

We do not develop this algorithm further, because the discussion of the next section presents two forms of an algorithm for simulating alternative set-associative caches that is more general (set-refinement is not required) or faster.

C. Simulating Set-Associative Caches

This section develops an algorithm, called *all-associativity simulation*, for simulating alternative direct-mapped and set-associative caches that have the same block size, do no prefetching, and use LRU replacement. All-associativity works for caches with arbitrary set-mapping functions, but works more efficiently if set-refinement holds. All-associativity simulation does not try to take advantage of inclusion, since inclusion does not hold between many pairs of set-associative caches (see Section IV-A). This work generalizes to arbitrary set-mapping functions an algorithm developed for caches using bit selection only [19], [34]. The algorithms discussed in this section can also be extended to handle multiple-block sizes and sector sizes [24], [34].

In theory, the storage required for all-associativity simulation is $O(N_{\text{unique}})$, where N_{unique} is the number of unique blocks referenced in an address trace. Our experience is that the storage required in practice, however, is usually much smaller than the size of modern main memories. Simulation of a one-million-address trace having an infinite cache miss ratio of one percent, for example, requires storage for 10 000 blocks. Since blocks can be stored in two words (a tag plus a pointer), less than 100K bytes are needed. Future simula-

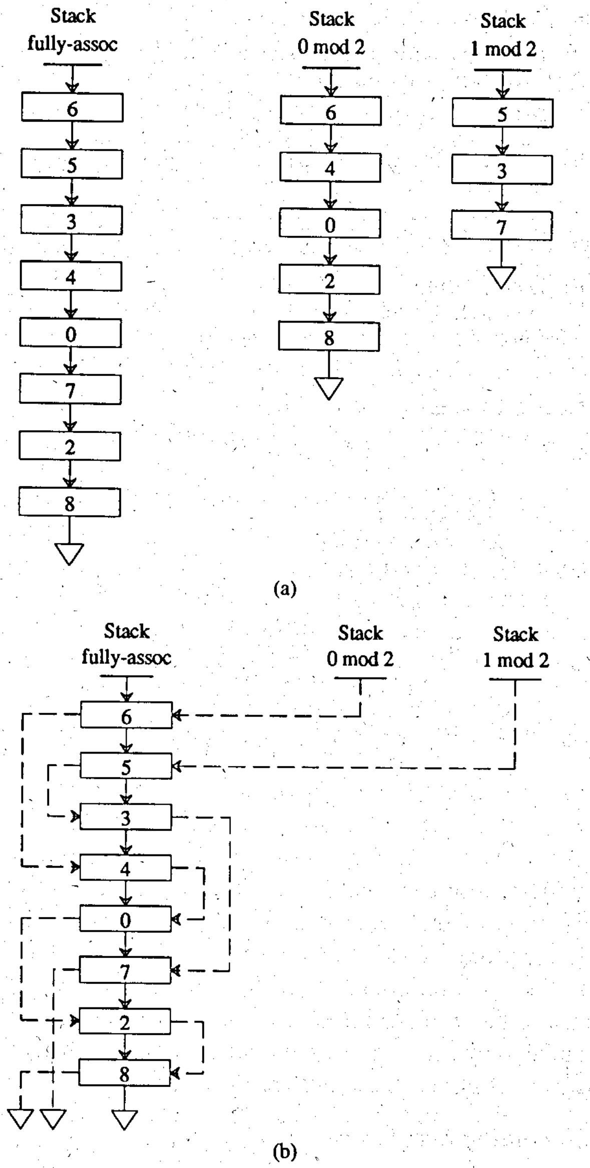


Fig. 6. Concurrent stack simulation with one (fully-associative) and two sets (even and odd blocks partitioned). (a) Separate storage. (b) Shared storage.

tions of multiple-megabyte caches may require tens of billions of references to be processed, potentially resulting in excess storage use. Storage for simulations of finite caches can be periodically (e.g., every 100 million references) reclaimed by discarding blocks not in the superset of the caches of interest; this latter approach is used in most other simulation algorithms as well. The algorithms below neglect storage reclamation.

Figs. 9 and 10 at the end of this section present pseudocode for all-associativity simulation not using and using set-refinement. The rest of this section provides insight into how all-associativity simulation works by developing it from stack simulation. A reader who understands the operation of the algorithms from Figs. 9 and 10 may skip to the next section.

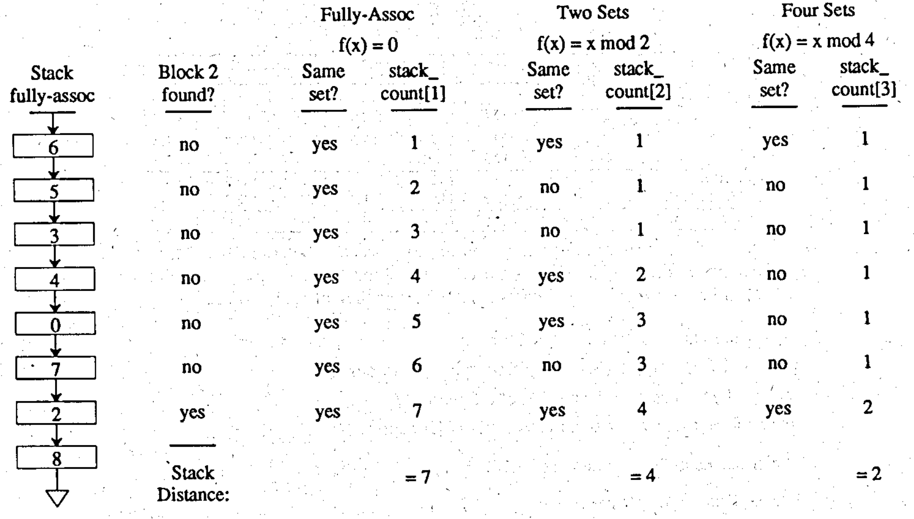
If we wish to simulate caches that have one, two, and four sets selected by bit selection (set-mapping functions x mod 1, x mod 2, and x mod 4) we can run three concurrent stack simulations (one with one stack, another with two and a third with four.) Fig. 6(a) illustrates the first two stack simulations. Due to locality, blocks that reside in one alternative cache will tend to reside in the other caches. Thus, as illustrated in Fig. 6(b), we can save storage by allocating storage for a block once and using multiple links to insert it into the multiple stacks. For LRU replacement, however, the order of two blocks in all stacks is always the same (the more-recently-referenced one is nearer the top) and is unaffected by what other blocks are members of a particular stack.[5] This implies that all links must point down, and therefore can be inferred instead of stored.

Instead of following the links of each stack and counting the blocks traversed, a block's stack distance for each set-mapping function can be calculated by traversing the fully-associative stack until the reference is found or the stack exhausted. For each stack node y before the reference x is found or the stack exhausted, we determine whether $f_i(y) = f_i(x)$ with each set-mapping function f_i. Whenever the equality holds, we increment *stack_count*[*i*]. If the reference is found, all *stack_count*[*i*]'s are incremented. After the reference is found or the stack exhausted, each *distance*[*i*, *stack_count*[*i*]] is incremented to indicate a hit to distance *stack_count*[*i*] with set-mapping function f_i. Fig. 7 illustrates that this method, which we call *all-associativity* simulation, on a reference to block 2.

The above method works for arbitrary set-mapping functions. A faster algorithm is possible if $f_{i+1}(x)$ refines $f_i(x)$, for $i = 1$ to $L-1$. All-associativity simulation can take advantage of set-refinement two ways. First, if f_1 implies multiple sets (not fully-associative), the algorithm can operate on the number of stacks induced by f_1 instead of simulating with one long fully-associative stack. The information lost by not maintaining one stack is the relative order of blocks in different f_1 sets. This information is not needed since the contrapositive of the implication used to define *refines* is $f_i(x) \neq f_i(y)$ implies $f_{i+1}(x) \neq f_{i+1}(y)$. Thus, two blocks in different f_1 sets will never be compared. Simulating with multiple stacks is faster than simulating with one, because the average number of active blocks the algorithm must look through to find a block is smaller, since active blocks are spread across many stacks (e.g., 512 stacks for simulating the VAX-11/780's cache [11]).

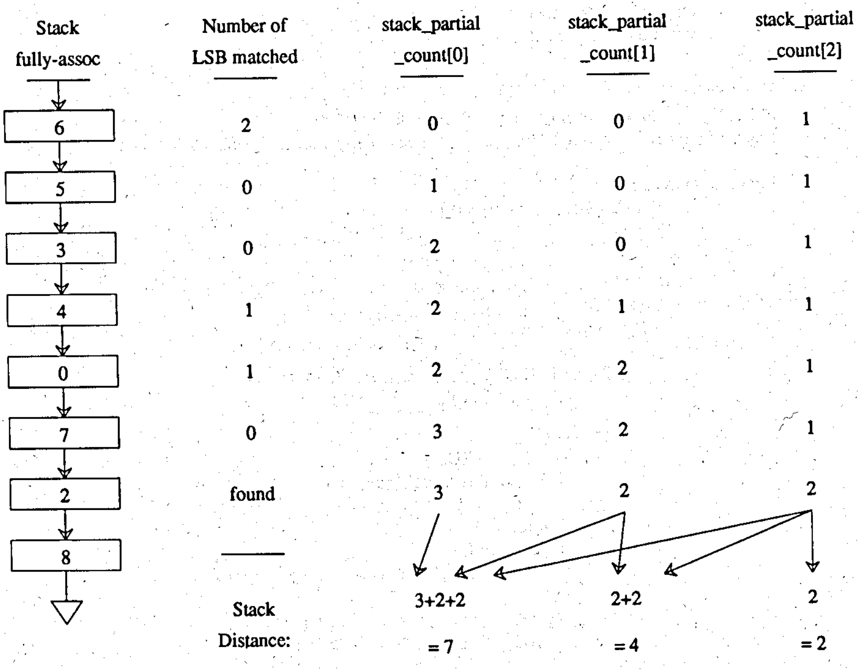
Second, the examination of "$f_i(x) = f_i(y)$ for $i = L$ down to 1" can be terminated the first time $f_i(x)$ equals $f_i(y)$, since the set-refinement forces the equality to hold for all smaller i. Furthermore, instead of incrementing *stack_count*[*i*] for each i where the equality holds, we need only increment *stack_partial_count*[*i*] for the maximum i for which it holds. When the processing for a reference terminates, we can compute *stack_count*[*i*] as $\sum_{j=i}^{L}$ *stack_partial_count*[*j*] and increment *distance*[*i*, *stack_count*[*i*]], for $i = 1, L$. Thus, using

[5] In RANDOM replacement, on the other hand, two blocks can be reordered in one group of stacks and not another if the current reference maps below them in one set of stacks and to another stack in another group of stacks. Consider blocks 0, 1, and 2 and a fully-associative stack and a pair of stacks for even and odd blocks. Reference 1, 0, and 2. The fully-associative stack holds (2 0 1), while the even and odd stacks hold (2 0) and (1). Now rereference block 1. RANDOM replacement requires that there is a 50 percent chance that the fully-associative stack changes to (1 0 2). Since the even stack is unaffected by a reference to an odd block, it remains as (2 0) and blocks 0 and 2 are now in a different order in different stacks.



		Fully-Assoc f(x) = 0		Two Sets f(x) = x mod 2		Four Sets f(x) = x mod 4	
Stack fully-assoc	Block 2 found?	Same set?	stack_ count[1]	Same set?	stack_ count[2]	Same set?	stack_ count[3]
6	no	yes	1	yes	1	yes	1
5	no	yes	2	no	1	no	1
3	no	yes	3	no	1	no	1
4	no	yes	4	yes	2	no	1
0	no	yes	5	yes	3	no	1
7	no	yes	6	no	3	no	1
2	yes	yes	7	yes	4	yes	2
8							
	Stack Distance:		= 7		= 4		= 2

Fig. 7. All-associativity simulation example: referencing block 2 in caches with 1, 2, and 4 sets.



Stack fully-assoc	Number of LSB matched	stack_partial _count[0]	stack_partial _count[1]	stack_partial _count[2]
6	2	0	0	1
5	0	1	0	1
3	0	2	0	1
4	1	2	1	1
0	1	2	2	1
7	0	3	2	1
2	found	3	2	2
8				
	Stack Distance:	3+2+2 = 7	2+2 = 4	2 = 2

Fig. 8. All-associativity simulation with set-refinement example: referencing block 2 in caches with 1, 2, and 4 sets.

set-refinement reduces the inner loop of all-associativity simulation with L set-mapping functions from L compares and 0 to L increments, to 1 to L compares and 0 or 1 increments. Since the expected number of compares in the improved algorithm can be as small as two,[6] this can result in nontrivial savings if L is large. Fig. 8 illustrates this optimization on reference to block 2.

D. Implementation and Comparison of Simulation Algorithms

To study the performance of stack, forest, and all-associativity simulation and to study CPU caches per se, we implemented these algorithms in C under UNIX 4.3 BSD. Stack and forest simulation were added to a general cache simulator that originally contained 1250 C statements[7] [14]. Adding stack simulation increased total code size by 150 statements, and adding forest simulation, 220 statements. Stack simulation is implemented using linked lists. The forest sim-

[6] Assume sets are selected with bit selection and the least-significant address bits of nodes in a stack are uniformly distributed. The probability that exactly i least significant bits match is $1/2^{i+1}$. The number of iterations given an i-bit match is $i+1$, with the final iteration used to detect the first mismatch. The expected number of iterations does not exceed two, since $\sum_{i=1}^{\infty}(i+1)/2^{i+1} = 2$.

[7] Measured by the number of source lines containing a semicolon or closing brace.

TABLE II
SIMULATION TIMES

Cache Size (bytes)	Associativity	Run-time in sec/1M-references (normalized)					
		Stack		Forest		All-Associativity	
<trivial trace>		304.3	(0.984)	304.7	(0.985)	294.6	(0.952)
16K	1-way	309.3	(1.000)	307.6	(0.994)	300.8	(0.972)
16K	4-way	312.5	(1.010)	--	--	309.2	(1.000)
1K to 8K	1-way	1234.4[8]	(4.0)	326.1	(1.054)	402.9	(1.303)
16K to 128K	1-way	1234.4[8]	(4.0)	321.0	(1.038)	332.3	(1.074)
16K to 128K	1-, 2- & 4-way	1806.6[8]	(6.0)	--	--	366.6	(1.185)

[8] Instead of determining the time for each stack simulation, we optimistically approximate the time required as the time for a fast stack simulation (128 kbyte direct-mapped cache) times the number of runs required.

```
integer L /* number of set-mapping functions */
function $f_1(x)$, ..., $f_L(x)$ /* arbitrary set-mapping functions */
integer N /* counter for the number of references */
integer max_assoc /* maximum associativity for metrics */
/* distance counts so that miss_ratio(A=k, F=$f_i$) = 1 - $\sum_{j=1}^{k}$distance[i,j]/N */
integer distance[1:L, 1:max_assoc]
integer stack_count[1:L] /*stack distance counters; reset for each reference. */

define stacknode_type {
          integer block_number
          stacknode_type *next
}
stacknode_type *stack /* top of stack pointer */
/* Let $N_{unique}$ be the number of unique blocks referenced. */
stacknode_type stacknodes[1:O($N_{unique}$)] /* dynamically allocated pool of stacknodes. */

For each reference x {
        for i=1 to L { stack_count[i] = 0 }
        read(var x)
        N++
        /* FIND */
        found = FALSE
        previous_node_pointer = NULL
        node_pointer = stack
        while ((NOT found) AND (node_pointer!=NULL)) {
                  y = node_pointer->block_number
                  if (x==y) {
                            found = TRUE
                            for i=1 to L { stack_count[i]++ }
                  }
                  else {
                            for i=1 to L {
                                      if ($f_i(x)$==$f_i(y)$) stack_count[i]++
                            }
                            previous_node_pointer = node_pointer
                            node_pointer = node_pointer->next
                  }
        }
        /* METRIC */
        if (found) {
                  for i=1 to L {
                            /*Record hits to distances ≤ max_assoc. */
                            if (stack_count[i] ≤ max_assoc) distance[i, stack_count[i]]++
                  }
        }
        /* If found, move the stack node of x to the top of the stack. */
        /* Otherwise, store x in a new stacknode and move it to the top of the stack. */
        UPDATE(x, found, previous_node_pointer, node_pointer)
}
```

Fig. 9. All-associativity simulation.

ulation implementation restricts the set-mapping functions to be the block number modulo the cache size in block frames, a slight generalization of bit selection. We implemented all-associativity simulation in a separate program containing 800 C statements and having far fewer options than the simulator above, and with the set-mapping function restricted to bit selection.

Table II lists simulation times for C language implementations of stack, forest, and all-associativity simulation. All caches simulated have 32-byte blocks, do no prefetching, use

```
integer L /* number of set-mapping functions */
/* set-mapping functions that obey set-refinement, */
/* i.e., $f_{i+1}$ refines $f_i$ for i=1, ..., L-1. */
function $f_1(x)$, ..., $f_L(x)$
integer number_of_stacks /* number of sets induced by $f_1(x)$ */
integer N /* number of references */
integer max_assoc /* maximum associativity for metrics */
/* distance counts so that miss_ratio(C(A=k, F=$f_i$)) = $1 - \sum_{j=1}^{k}$distance[i,j]/N */
integer distance[1:L, 1:max_assoc]
integer stack_partial_count[1:L] /* stack distance counters; reset for each reference. */

define stacknode_type {
        integer block_number
        stacknode_type *next
}
stacknode_type *stack[0:number of stacks-1] /* top of stack pointers */
/* Let $N_{unique}$ be the number of unique blocks referenced. */
stacknode_type stacknodes[1:O($N_{unique}$)] /* dynamically allocated pool of stacknodes. */



For each reference x {
        for i=1 to L { stack_partial_count[i] = 0 }
        read(var x)
        N++
        stack_number = $f_1(x)$
        /* FIND */
        found = FALSE
        previous_node_pointer = NULL
        node_pointer = stack[stack_number]
        while ((NOT found) AND (node_pointer!=NULL)) {
                y = node_pointer->block_number
                if (x==y) {
                        found = TRUE
                        stack_partial_count[L]++
                }
                else {
                        match = FALSE
                        for i=L down to 1 OR match {
                                if ($f_i(x)$==$f_i(y)$) {
                                        match = TRUE
                                        stack_partial_count[i]++
                                }
                        }
                        previous_node_pointer = node_pointer
                        node_pointer = node_pointer->next
                }
        }
        /* METRIC */
        if (found) {
                stack_count = 0
                for i=L down to 1 {
                        stack_count = stack_count + stack_partial_count[i]
                        /* Record hits to distances ≤ max_assoc. */
                        if (stack_count ≤ max_assoc) distance[i, stack_count]++
                }
        }
        /* If found, move the stack node of x to the top of its stack. */
        /* Otherwise, store x in a new stacknode and move it to the top of the stack. */
        UPDATE(x, stack_number, found, previous_node_pointer, node_pointer)
```

Fig. 10. All-associativity simulation with set-refinement.

LRU replacement, are unified (data and instructions cached together) and use bit selection. Results in the first row ("trivial trace") are for a trace consisting of one million copies of the same address, yielding one miss and 999 999 hits. All other results presented here are for a trace of one million memory references from system calls generated by an Amdahl standard MVS workload [28]. We also examined traces from three other architectures [15]. We omit these results here, since they are similar to those with the MVS trace. Results not in parentheses are the elapsed virtual times in seconds for simulation runs on an otherwise unloaded Sun-3/75 with 8M of memory, no local disk, and trace data read from a file server via an ethernet. Results in parentheses are normalized to the time for stack simulation to simulate a single 16 kbyte direct-mapped (1-way) cache with the MVS trace.

We compare these algorithms using only memory trace data, as opposed to data from other caching systems, because set-associativity is rarely used outside of CPU caches. Readers interested in simulation performance times for fully-associative caches, driven by traces of memory and disk references, should consult [33].

The simulation times in Table II allow us to answer the following three questions regarding how these implementations perform.

1) Are the implementations comparable?

Yes. We determine that implementations are comparable by simulating single caches, which, in theory, require the same simulation time. For a synthetic trace and a real trace and for two associativities, we found the virtual times (CPU times) for implementations of stack and forest simulation differed by less than 0.5 percent, while the implementation of all-associativity simulation is 1–3 percent faster (see Table II). That all-associative simulation is slightly faster is not surprising, since it was implemented in a separate program, while stack and forest simulation are part of a more powerful cache simulator.

2) What algorithm is fastest for simulating a collection of direct-mapped caches of similar size?

Forest simulation. However, forest simulation is not significantly faster than all-associativity simulation if caches are large. Both forest and all-associativity simulation are much faster than stack simulation since they require only one run, whereas stack simulation needs one run per cache size.

3) What algorithm is fastest for simulating a collection of direct-mapped and set-associative caches of similar size?

All-associativity simulation. All-associativity simulation requires only one run, which is not much slower than a single, simple simulation run. Forest simulation is not able to simulate nondirect-mapped caches. Stack simulation requires one run per unique number of sets. Simulating caches of c, $2c$, $4c$ through $2^s c$ block frames with associativities 1, 2, 4 through 2^a requires $s + a - 1$ stack simulations. One with $c/2^a$ sets, a second with $c/2^{a-1}$ sets, $\cdots$, another with c sets, another with $2c$ sets, $\cdots$, and finally one with $2^s c$ sets. The simulation in the final row of Table II, for example, required six stack simulations, using 128, 256, $\cdots$ and 4K stacks, respectively.

The speedups illustrated here for trace lengths of one million references (30 min down to 6 min) are impressive, but not critical. Traces to exercise multiple-megabyte caches, however, will be much longer. All-associativity simulation will allow billion-reference traces to be processed in a few days rather than a few weeks. Furthermore, simulating a wide variety of caches in one pass as a trace is generated facilitates simulations with traces too large to store.

V. The Relationship Between Associativity and Miss Ratio

In this section, we analyze how changes in associativity alter cache miss ratio. We find empirically that some simple relationships exist between the miss ratios of direct-mapped, set-associative, and fully-associative caches, largely independently of cache size. We concentrate on the relationship between miss ratios of alternative caches, rather than the absolute size of miss ratio, because our traces samples are short, never exceeding 500K references. We assume throughout that caches have a fixed block size, use LRU replacement, do no prefetching and pick the set of a reference with bit selection.

A. Categorizing Set-Associative Misses

The simulation algorithms described earlier facilitate computing the miss ratios for many alternative cache sizes and associativities. These data can be used to increase our understanding of a single cache's miss ratio. We do this by subdividing the observed misses into three categories: (*set-*)*conflict* misses (due to too many active blocks mapping to a fraction of the sets), *capacity* misses (due to fixed cache size), and *compulsory* misses (necessary in any case[9]).

The size of these components can be calculated as follows. First, the conflict miss ratio is the cache's miss ratio less the miss ratio for a fully-associative cache of the same size. Second, the capacity miss ratio is the fully-associative cache's miss ratio less the miss ratio for an infinite cache (one so large it never replaces a block). Finally, the compulsory miss ratio is the infinite cache's miss ratio, which is not zero since initial references to blocks still miss. This categorization is easy to compute, since it can be derived from average miss ratios and does not require a detailed manipulation of simulation programs (as does the model in [3]).

Table III illustrates this miss ratio categorization "ue," a trace of VAX-11 interactive users under Ultrix (see Table I). All miss ratios are warm-start and for a unified cache with 32-byte blocks. Under each miss ratio component, the first number is the component's absolute size, while the second is its relative contribution to the overall miss ratio. The reader should concentrate on trends rather than miss ratio values, since this table only gives results for three short trace samples of one workload. Compulsory miss ratios and results for larger caches are subject to more error. (That one conflict miss ratio is negative (eight-way set-associative 1 kbyte cache) is unimportant, since 1) the magnitude is very small (−0.0006), indicating that cache has approximately the same miss ratio as fully-associative cache, and 2) the behavior is possible [31].)

For this trace, we see 1) the absolute size of the conflict miss ratios for set-associative caches (not direct-mapped) are small,

[9] That is, necessary without violating our assumptions of a fixed block size, LRU replacement, no prefetching, and bit selection.

TABLE III
Three Miss Ratio Components

Cache Size (bytes)	Degree of Associativity	Miss Ratio	Miss Ratio Components (Relative Percent)					
			Conflict		Capacity		Compulsory	
1K	1-way	0.1913	0.0419	22%	0.1405	73%	0.0090	5%
1K	2-way	0.1609	0.0115	7%	0.1405	87%	0.0090	6%
1K	4-way	0.1523	0.0029	2%	0.1405	92%	0.0090	6%
1K	8-way	0.1488	-0.0006	-0%	0.1405	94%	0.0090	6%
2K	1-way	0.1482	0.0361	24%	0.1032	70%	0.0090	6%
2K	2-way	0.1223	0.0102	8%	0.1032	84%	0.0090	7%
2K	4-way	0.1148	0.0027	2%	0.1032	90%	0.0090	8%
2K	8-way	0.1128	0.0006	1%	0.1032	91%	0.0090	8%
4K	1-way	0.1089	0.0270	25%	0.0730	67%	0.0090	8%
4K	2-way	0.0948	0.0129	14%	0.0730	77%	0.0090	9%
4K	4-way	0.0868	0.0049	6%	0.0730	84%	0.0090	10%
4K	8-way	0.0842	0.0022	3%	0.0730	87%	0.0090	11%
8K	1-way	0.0868	0.0257	30%	0.0521	60%	0.0090	10%
8K	2-way	0.0693	0.0082	12%	0.0521	75%	0.0090	13%
8K	4-way	0.0650	0.0040	6%	0.0521	80%	0.0090	14%
8K	8-way	0.0629	0.0018	3%	0.0521	83%	0.0090	14%
16K	1-way	0.0658	0.0194	29%	0.0375	57%	0.0090	14%
16K	2-way	0.0535	0.0070	13%	0.0375	70%	0.0090	17%
16K	4-way	0.0494	0.0029	6%	0.0375	76%	0.0090	18%
16K	8-way	0.0478	0.0014	3%	0.0375	78%	0.0090	19%
32K	1-way	0.0503	0.0134	27%	0.0279	55%	0.0090	18%
32K	2-way	0.0412	0.0043	11%	0.0279	68%	0.0090	22%
32K	4-way	0.0383	0.0014	4%	0.0279	73%	0.0090	23%
32K	8-way	0.0377	0.0008	2%	0.0279	74%	0.0090	24%

making further increases in associativity of limited benefit, 2) the absolute size of conflict miss ratios for direct-mapped caches gets smaller with increasing cache size, making increasing associativity less important, and 3) the compulsory miss ratio is fixed but gets relatively more important with increasing cache size, limiting the potential benefit of further cache size increases. One deficiency of this categorization is that the magnitude of the capacity miss ratio does not bound the miss ratio reduction that increasing cache size can yield. This is because increasing cache size also increases the number of sets, reducing the conflict miss ratio.

B. How Set-Associative Miss Ratios Relate to Fully-Associative Ones

It has been previously shown [26] that set-associative miss ratios can be closely estimated from fully-associative ones; this observation was validated for several traces for 16 and 64 sets. We review that calculation in this section, and validate the results over a larger range of cache sizes and number of sets.

The model derives LRU distance probabilities with s sets, $p_i(s)$, from fully-associative LRU distance probabilities, q_i. $p_i(s)$ is the probability a reference is made to the ith most-recently-referenced block in one of s sets, while q_i is the probability a reference is made to the ith most-recently-referenced block in any set. Consequently, $q_i = p_i(1)$. LRU distance probabilities are equivalent to the miss ratios of caches using LRU replacement. The miss ratio for an n-way set-associative cache with s sets is $1 - \sum_{i=1}^{n} p_i(s)$, while the miss ratio for an n-block fully-associative cache is $1 - \sum_{i=1}^{n} q_i$.

Bayes rule[10] allows us to express a set-associative LRU distance probability in terms of fully-associative LRU distance probabilities:

$$p_n(s) = \sum_{i=1}^{\infty} \text{Prob(LRU distance } n \text{ with } s \text{ sets}$$
$$|\, \text{LRU distance } i \text{ with 1 set}) \cdot q_i.$$

The above equation can be used to estimate set-associative LRU distance probabilities from fully-associative LRU distance probabilities, or equivalently set-associative miss ratios from fully-associative miss ratios, using a simple approximation for Prob(LRU distance n with s sets|LRU distance i with 1 set). The approximation is based on the assumption that the probability that two blocks map the same set is $1/s$ and independent of where other blocks map. A reference to set-associative distance n occurs if exactly $n-1$ more-recently-referenced blocks map to the reference's set, while a reference to fully-associative distance i implies $i-1$ blocks are more-recently-referenced. By the above assumption, the probability that exactly $n-1$ of the $i-1$ more-recently-referenced blocks map to the set of the reference is 0 for $n > i$ and approximately

$$\binom{i-1}{n-1} \left[\frac{1}{s}\right]^{n-1} \left[\frac{s-1}{s}\right]^{i-n}, \qquad \text{for } n \le i.$$

Substitution yields

$$p_n(s) \approx \sum_{i=n}^{\infty} \binom{i-1}{n-1} \left[\frac{1}{s}\right]^{n-1} \left[\frac{s-1}{s}\right]^{i-n} \cdot q_i.$$

Fig. 11 shows actual miss ratios (solid lines) and miss ratios predicted with the above equation (dashed lines) for associativities 1, 2, 4, and 8. Data are based on using trace "mul2" to drive a unified cache with 32-byte blocks. Results here and for several other traces [15] yield three conclusions.

1) The predictions are quite accurate. In most cases, the relative error is less than 5 percent; only rarely is it greater than 10 percent.

2) Predictions are usually more pessimistic than the actual miss ratios. The cause of this phenomenon is that blocks selected with bit selection collide slightly less often than blocks whose set is selected at random (as the above approximation assumes), due to spatial locality [26].

3) The relative error gets smaller with increasing associativity, which is expected since many-way set-associative caches have miss ratios nearly identical to fully-associative caches.

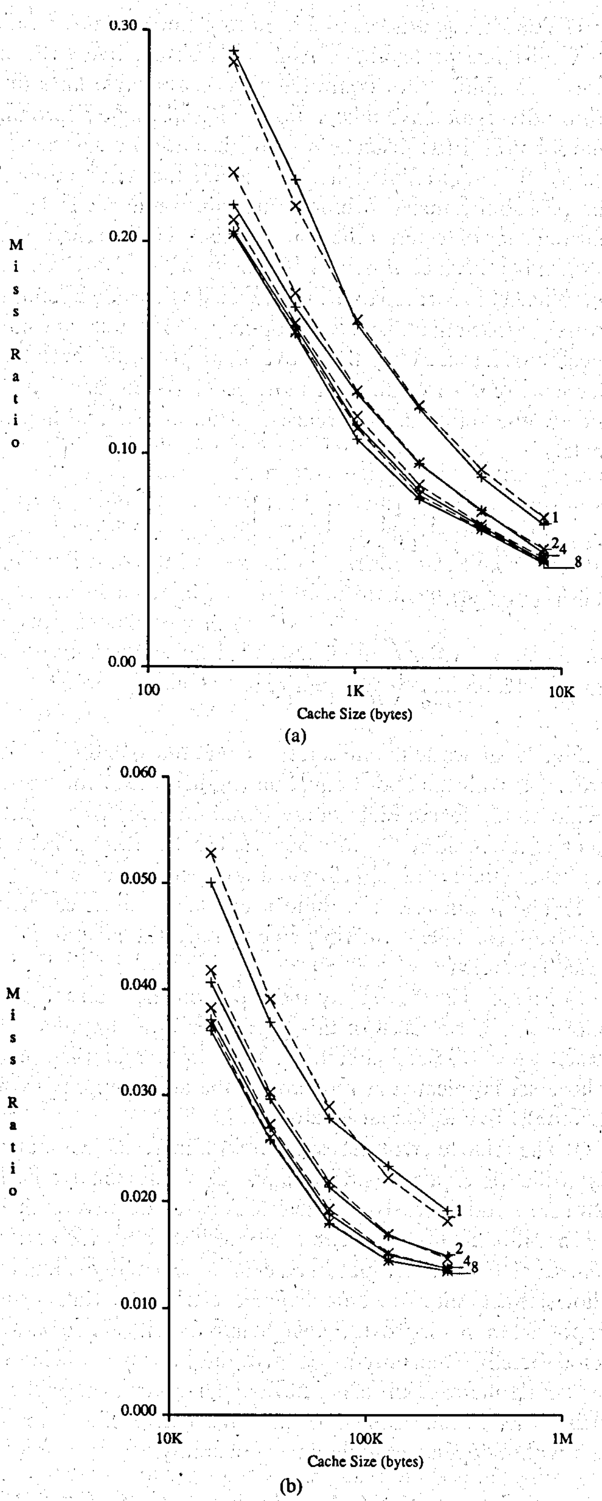
That this method is accurate is not important for deriving set-associative miss ratios, since all-associativity simulation allows exact values to be calculated efficiently. Rather, it is important in that it provides insight into the difference between set-associative and fully-associative miss ratios, showing that the actual increase in miss ratio is nearly identical to the increase that results from assuming that active blocks map to sets with independent and equal probability.

C. How Set-Associative Miss Ratios Relate to Each Other

Empirically we see that miss ratio is affected by changes in cache size, block size, and associativity. We would like to find some simple rules that can be used to quantify changes in associativity on cache miss ratios; we do that in this section.

We find that by examining relative miss ratio differences rather than absolute miss ratio differences one can almost eliminate the effect of cache size. Consider an n-way set-associative cache and a $2n$-way set-associative cache, hav-

[10] For some event A and a set of mutually exclusive and exhaustive events B_i, Bayes' rule states that $\text{Prob}(A) = \Sigma \text{Prob}(A|B_i) \cdot \text{Prob}(B_i)$.



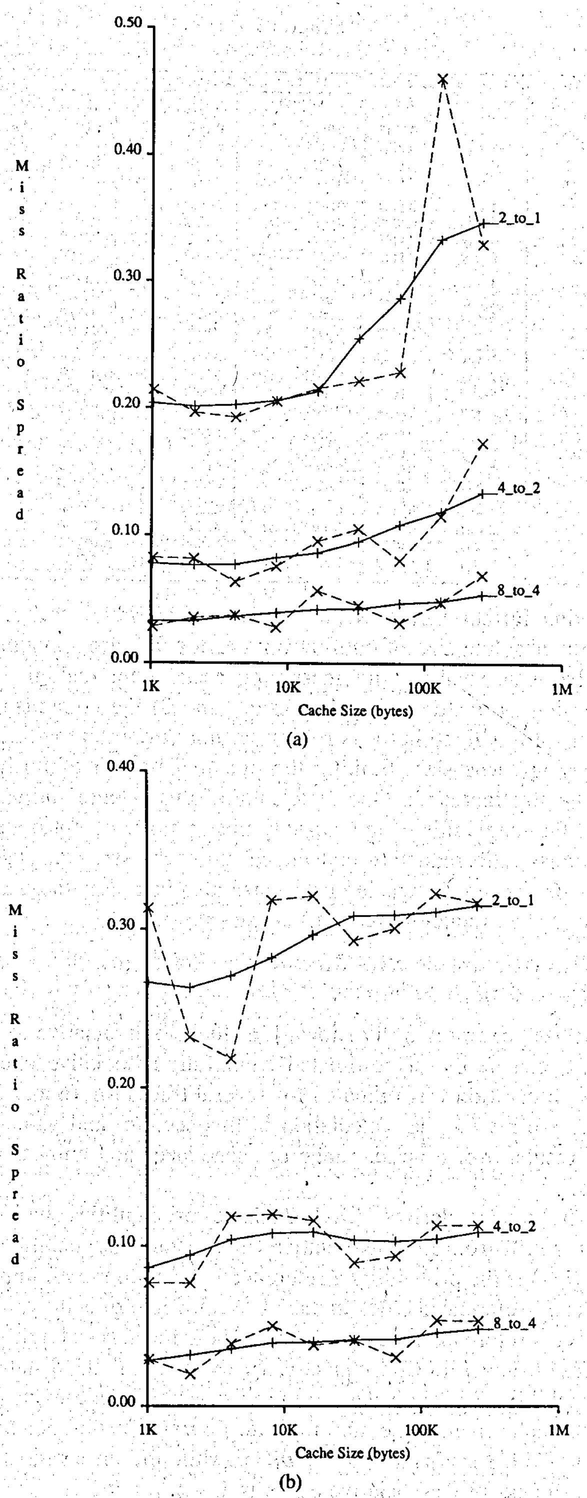
(a)

(b)

Fig. 11. Predicted (dashed) and actual (solid) miss ratios for trace "mul2" with caches of associativity 1, 2, 4, and 8. (a) Smaller caches. (b) Larger caches.

ing the same capacity, the same block size, and miss ratios $m(A = n)$ and $m(A = 2n)$. Let the *miss ratio spread* be the ratio of the miss ratios, less one:

$$\frac{m(A = n)}{m(A = 2n)} - 1 = \frac{m(A = n) - m(A = 2n)}{m(A = 2n)}.$$

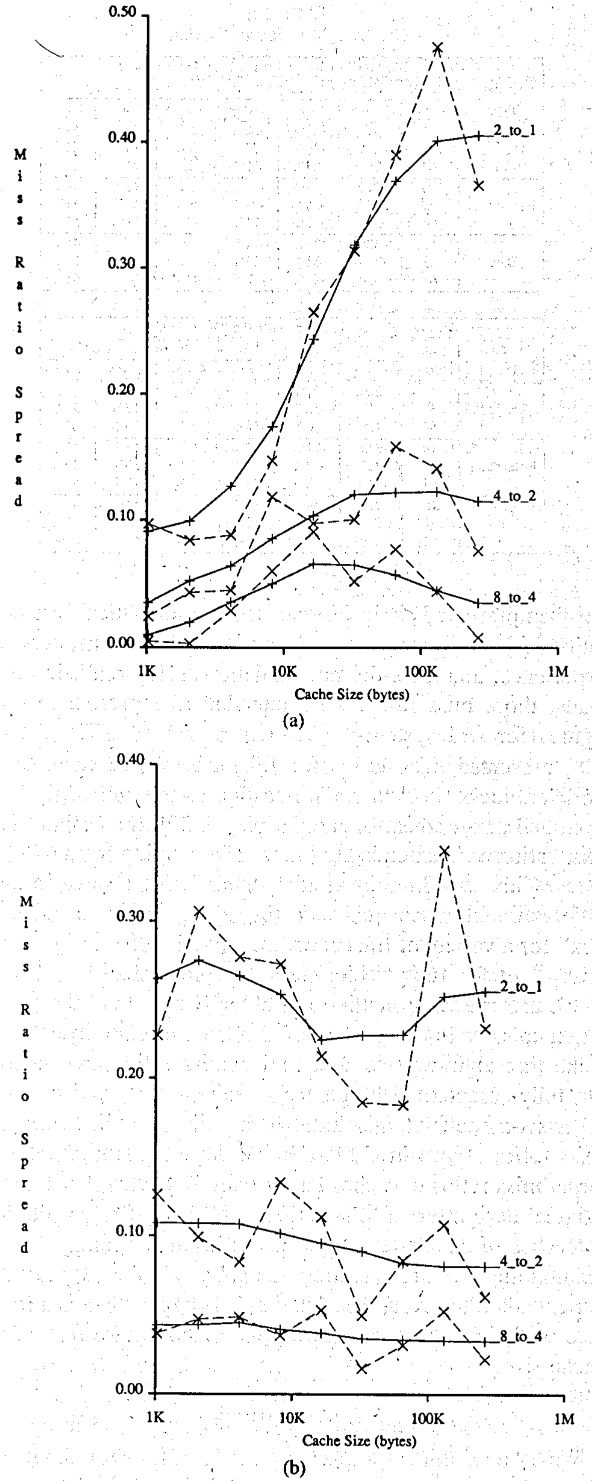


(a)

(b)

Fig. 12. Unified cache miss ratio spreads (solid lines are smoothed data). A line labeled "$2n$_to_n" displays $[m(A = n) - m(A = 2n)]/m(A = 2n)$ where $m(A = n)$ is the miss ratio of an n-way set-associative cache. (a) Five-trace group. (b) 23-trace group.

Figs. 12 and 13 and Table IV present data from trace-driven simulation. As discussed in Section III, data for larger caches are subject to more error than data for smaller caches, and measurements for caches larger than 64K should be treated with considerable caution. Fig. 12 shows some miss ratio



(a)

(b)

Fig. 13. More miss ratio spreads for the five-trace group (solid lines are smoothed data). (a) Instruction caches. (b) Data caches.

spreads of unified caches with 32-byte blocks for the five- and 23-trace groups. Fig. 13 examines miss ratio spreads for instruction and data cache with the five-trace group. The average miss ratio spread is computed using the ratio of the average miss ratios. Dashed lines present raw data, while solid lines are smoothed using a weighted average of adjacent spreads (recommended in [9]). We selected the weights to reduce variation between adjacent spreads, without suppressing larger trends. We assigned a weight of 0.20 to both adjacent spreads and 0.15 to spreads two sizes away, leaving a weight of 0.30 for the spread being smoothed.

TABLE IV
Smoothed Miss Ratio Spreads

Smoothed Miss Ratio Spreads for Unified Caches

Cache	Block Size 16 Bytes			Block Size 32 Bytes			Block Size 64 Bytes		
Size	8-to-4	4-to-2	2-to-1	8-to-4	4-to-2	2-to-1	8-to-4	4-to-2	2-to-1
1K	4%	9%	20%	5%	10%	30%	5%	12%	41%
2K	5%	10%	22%	5%	12%	29%	6%	13%	38%
4K	5%	11%	23%	6%	12%	29%	7%	14%	38%
8K	5%	10%	25%	6%	12%	29%	7%	14%	37%
16K	5%	10%	26%	5%	12%	31%	7%	13%	38%
32K	5%	10%	28%	5%	11%	32%	6%	13%	38%
64K	4%	10%	28%	5%	11%	33%	5%	12%	39%
128K	5%	10%	28%	5%	11%	33%	5%	12%	40%
256K	4%	10%	28%	5%	12%	34%	6%	13%	40%
AVG	5%	10%	25%	5%	11%	31%	6%	13%	39%

Smoothed Miss Ratio Spreads for Instruction Caches

Cache	Block Size 16 Bytes			Block Size 32 Bytes			Block Size 64 Bytes		
Size	8-to-4	4-to-2	2-to-1	8-to-4	4-to-2	2-to-1	8-to-4	4-to-2	2-to-1
1K	5%	11%	16%	4%	11%	16%	6%	10%	16%
2K	6%	13%	18%	5%	14%	17%	6%	13%	18%
4K	6%	13%	20%	6%	15%	20%	7%	15%	20%
8K	7%	13%	22%	7%	15%	23%	7%	15%	24%
16K	7%	13%	26%	7%	14%	28%	7%	15%	29%
32K	6%	12%	28%	7%	14%	30%	7%	15%	32%
64K	5%	11%	30%	6%	12%	32%	6%	13%	35%
128K	4%	11%	29%	5%	12%	32%	5%	14%	35%
256K	3%	8%	28%	4%	10%	31%	4%	12%	36%
AVG	6%	12%	24%	6%	13%	25%	6%	14%	27%

Smoothed Miss Ratio Spreads for Data Caches

Cache	Block Size 16 Bytes			Block Size 32 Bytes			Block Size 64 Bytes		
Size	8-to-4	4-to-2	2-to-1	8-to-4	4-to-2	2-to-1	8-to-4	4-to-2	2-to-1
1K	6%	13%	27%	6%	14%	30%	7%	14%	33%
2K	6%	12%	28%	7%	13%	31%	8%	14%	35%
4K	6%	11%	26%	7%	13%	29%	8%	14%	34%
8K	5%	10%	26%	6%	11%	30%	7%	13%	36%
16K	4%	9%	24%	5%	10%	28%	6%	12%	35%
32K	3%	8%	24%	4%	9%	29%	5%	11%	36%
64K	3%	8%	23%	3%	9%	28%	4%	11%	35%
128K	3%	7%	22%	4%	9%	29%	4%	11%	36%
256K	3%	7%	20%	4%	9%	27%	5%	12%	35%
AVG	4%	9%	24%	5%	11%	29%	6%	12%	35%

Table IV shows similar results from an alternative computation, taking the geometric average of the miss ratio spreads of individual traces. This method yields slightly larger spreads than those calculated using the ratio of average miss ratios (as in Fig. 12). Miss ratio spreads in rows labeled "AVG" are calculated by taking the geometric mean of the ratio of miss ratios for cache sizes from 1K to 256K bytes.

These results together with more data in [15] exhibit the following trends.

1) Miss ratio spreads for caches with more restricted associativity are larger, implying, for example, that direct-mapped and two-way set-associative miss ratios are further apart than two-way and four-way set-associative miss ratios. This result corroborates the previous work of many others.

2) Except for small instruction caches, miss ratio spreads do not vary rapidly with changing cache size, even though the miss ratios in their numerators and denominators vary by over an order of magnitude. The miss ratio spreads between small direct-mapped and two-way set-associative instruction caches are smaller than many other spreads due to the sequential behavior of instruction reference streams, which minimizes the usefulness of increasing associativity in small instruction

caches [31]. This sequentiality is much less of a factor for large instruction caches, and for such large instruction caches, the miss ratio spreads are similar to those for data and unified caches. The only major exception to these observations is the miss ratio spread between direct-mapped and two-way set-associative 128 kbyte caches with the five-trace group. We believe that the cause of this aberration lies in the particular traces and trace lengths used, not in some property of 128 kbyte caches.

3) Miss ratio spreads are positively correlated with block size. While the difference is not important with wide associativity, the miss ratio spread between direct-mapped and two-way set-associative unified caches with the 23-trace group increases from 25 to 31 to 39 percent as block size goes from 16 to 32 to 64 bytes. The reason for this is that for a given cache size, as the blocks become larger, the number of sets decreases, and the probability that two active blocks map into the same set increases (i.e., bigger blocks are more likely to "bump into each other".)

4) Miss ratio spreads between unified and data caches are similar. Instruction cache spreads are similar or smaller (see also [10]). Miss ratio spreads between direct-mapped and two-way set-associative instruction caches are significantly smaller than other spreads, as has been observed elsewhere [31].

Since the miss ratio spreads do not vary greatly with cache size, we can provide insight into the relationship between miss ratio and associativity by computing miss ratio spreads averaged over many cache sizes, as is done in Table IV. To one significant figure, halving associativity with these traces from eight-way to four-way to two-way to direct-mapped causes miss ratio spreads of 5, 10, and 30 percent regardless of cache size, cache type, or block size. Equivalently, one can look at set-associative miss ratios relative to direct-mapped or fully-associative ones, as depicted in Table V. Relative to direct-mapped, the miss ratios for eight-, four- and two-way set-associative are, respectively, about 34, 30, and 22 percent lower. Assuming that eight-way set-associative is effectively fully-associative, the miss ratio increases by 5 percent for four-way, 17 percent for two-way, and 52 percent for direct-mapped.

Our examination of miss ratios for caches with different associativities has shown that the miss ratio spread does not change significantly over a wide range of cache sizes, with exception of small instruction caches, for which the spread is unusually small. Consequently, the absolute miss ratio difference decreases as caches get larger, since absolute miss ratios get smaller. When the absolute miss ratio difference becomes sufficiently small, an interesting change occurs: the effective access time of a direct-mapped cache can be smaller than that of a set-associative cache of the same size, even though the direct-mapped cache has the larger miss ratio. This change occurs when implementation differences, that have previously been ignored, become more important than absolute miss ratio differences. This topic is considered in some detail in [16] and [22].

D. Extending Design Target Miss Ratios

In [28], it was noted that absolute miss ratios computed from trace-driven simulations were often optimistic. That paper then presented *design target miss ratios* which were miss ratios derived from hardware monitor measurements, personal experience, and trace-driven simulations using realistic workloads; those miss ratios were intended to represent realistic figures for real systems under real workloads. The data in [28] presented miss ratios for fully associative caches with 16-byte blocks, broken down into figures for unified, instruction, and data caches. In another paper [30], the design target miss ratios were extended to block sizes ranging from 4 to 128 bytes. This was done by finding the relative change in miss ratio as the block size changed (by taking "ratios of miss ratios" for a variety of traces) and propagating the design target miss ratios for 16-byte block to other block sizes.

TABLE V
RELATIVE MISS RATIO CHANGE

Relative Miss Ratio Change for the Five-Trace Group							
Cache	Block	From Direct-Mapped To			From Eight-Way To		
Type	Size	8-way	4-way	2-way	4-way	2-way	1-way
Unified	16	-31%	-27%	-20%	5%	17%	47%
	32	-33%	-30%	-22%	5%	18%	52%
	64	-38%	-34%	-26%	6%	21%	63%
Instruction	16	-31%	-27%	-20%	5%	17%	48%
	32	-32%	-28%	-21%	6%	18%	51%
	64	-33%	-30%	-22%	6%	18%	54%
Data	16	-32%	-29%	-21%	5%	16%	48%
	32	-34%	-31%	-23%	5%	17%	52%
	64	-39%	-35%	-26%	6%	20%	64%

Relative Miss Ratio Change for the 23-Trace Group							
Cache	Block	From Direct-Mapped To			From Eight-Way To		
Type	Size	8-way	4-way	2-way	4-way	2-way	1-way
Unified	16	-30%	-27%	-20%	5%	15%	44%
	32	-35%	-32%	-24%	5%	17%	54%
	64	-40%	-36%	-28%	6%	20%	67%
Instruction	16	-31%	-27%	-19%	6%	17%	45%
	32	-32%	-28%	-20%	6%	19%	49%
	64	-34%	-30%	-21%	6%	20%	53%
Data	16	-29%	-26%	-19%	4%	14%	42%
	32	-33%	-30%	-22%	5%	16%	50%
	64	-38%	-34%	-26%	6%	19%	61%

We use the same method in Table VI to extend the design target miss ratios to caches of limited associativity. We assume that eight-way set-associative miss ratios are equal to the fully-associative design target miss ratios, and compute other set-associative miss ratios using the smoothed *ratios of miss ratios* shown in Table IV. We do not extend the design target miss ratios to caches larger than 32 kbytes, because the original design target miss ratios in [28] and [30] are limited to caches of 32 kbytes or less, and the methodology for extending them to larger cache sizes is beyond the scope of this paper; note, however, that data in [27] suggest that as a rough rule of thumb, the miss ratio drops as the square root of the cache size.

VI. CONCLUSIONS

We have examined properties and algorithms for simulating alternative caches and have examined the relationship between associativity and miss ratio. We find that both *inclusion* (that larger caches contain a superset of the blocks in smaller caches [19]) and *set-refinement* (that blocks mapping to the same set in larger caches map to the same set in smaller caches) can be used by *forest simulation*, a new algorithm for rapidly simulating alternative direct-mapped caches. We show that inclusion is not useful, but set-refinement can be useful for *all-associativity simulation*, an algorithm for rapidly simulating alternative direct-mapped, set-associative, and fully-

TABLE VI
DESIGN TARGET MISS RATIOS

Design Target Miss Ratios for Unified Caches

Cache	Block Size 16 Bytes				Block Size 32 Bytes				Block Size 64 Bytes			
Size	8-way	4-way	2-way	1-way	8-way	4-way	2-way	1-way	8-way	4-way	2-way	1-way
1K	0.210	0.219	0.239	0.288	0.162	0.170	0.188	0.244	0.137	0.144	0.162	0.229
2K	0.170	0.179	0.197	0.240	0.124	0.130	0.146	0.188	0.098	0.104	0.118	0.163
4K	0.120	0.126	0.140	0.172	0.082	0.087	0.097	0.126	0.059	0.063	0.072	0.099
8K	0.080	0.084	0.093	0.116	0.050	0.053	0.059	0.077	0.033	0.035	0.040	0.055
16K	0.060	0.063	0.069	0.088	0.036	0.038	0.042	0.055	0.023	0.025	0.028	0.038
32K	0.040	0.042	0.046	0.059	0.024	0.025	0.028	0.037	0.014	0.015	0.017	0.023

Design Target Miss Ratios for Instruction Caches

Cache	Block Size 16 Bytes				Block Size 32 Bytes				Block Size 64 Bytes			
Size	8-way	4-way	2-way	1-way	8-way	4-way	2-way	1-way	8-way	4-way	2-way	1-way
1K	0.200	0.211	0.234	0.271	0.134	0.140	0.155	0.179	0.098	0.104	0.115	0.133
2K	0.150	0.159	0.179	0.210	0.098	0.103	0.117	0.138	0.068	0.072	0.082	0.097
4K	0.100	0.106	0.120	0.143	0.063	0.067	0.076	0.091	0.043	0.046	0.053	0.063
8K	0.060	0.064	0.072	0.089	0.037	0.039	0.045	0.056	0.023	0.025	0.028	0.035
16K	0.050	0.053	0.060	0.076	0.029	0.031	0.035	0.045	0.018	0.019	0.022	0.029
32K	0.030	0.032	0.036	0.046	0.017	0.018	0.021	0.027	0.010	0.011	0.012	0.016

Design Target Miss Ratios for Data Caches

Cache	Block Size 16 Bytes				Block Size 32 Bytes				Block Size 64 Bytes			
Size	8-way	4-way	2-way	1-way	8-way	4-way	2-way	1-way	8-way	4-way	2-way	1-way
1K	0.160	0.170	0.192	0.244	0.138	0.146	0.166	0.216	0.140	0.150	0.170	0.227
2K	0.120	0.127	0.143	0.183	0.094	0.101	0.114	0.149	0.083	0.089	0.102	0.138
4K	0.100	0.106	0.117	0.148	0.070	0.075	0.084	0.109	0.054	0.058	0.067	0.090
8K	0.080	0.084	0.092	0.116	0.053	0.056	0.062	0.081	0.039	0.042	0.047	0.064
16K	0.060	0.062	0.068	0.084	0.039	0.041	0.045	0.058	0.026	0.028	0.031	0.042
32K	0.040	0.041	0.045	0.055	0.025	0.026	0.028	0.037	0.017	0.018	0.020	0.027

associative caches. Our algorithm is a generalization of an earlier algorithm [19], [34]. We find all-associativity simulation is tremendously effective, allowing dozens of caches to be evaluated in time that is within a small constant factor of the time needed to simulate one cache with wide associativity.

Our empirical examination of associativity and miss ratio provides data and insight into how miss ratio is affected by changes in associativity. In particular:

- We show how to divide cache misses into *conflict, capacity*, and *compulsory* misses, using only average miss ratios from alternative caches. Increasing associativity but not cache size can only reduce conflict misses. Increasing cache size but not associativity increases the number of sets, and therefore may decrease conflict and capacity misses. Compulsory misses cannot be reduced without increasing block size or prefetching.
- By applying a model from [26] to a wide variety of caches, we show that the difference between set-associative and fully-associative miss ratios (the rate of conflict misses) can be predicted by assuming blocks map to sets uniformly and independently, resulting in too many active blocks mapping to a fraction of the sets.
- We find empirically that *miss ratio spread*, the relative change in miss ratio caused by reducing associativity, is relatively invariant for caches of significantly different size and miss ratio. Our data show that reducing associativity from eight-way to four-way, from four-way to two-way, and from two-way to direct-mapped causes relative miss ratio increases of about 5, 10, and 30 percent, respectively. We also use miss ratio spreads to provide design target miss ratios for caches with limited associativity.

ACKNOWLEDGMENT

We would like to thank R. Katz, D. Patterson, and other members of the SPUR project for their many suggestions that improved the quality of our research, H. Stone for comments on [15], and S. Dentinger, G. Gibson, and V. Madan for reading and improving drafts of this paper.

REFERENCES

[1] A. Agarwal, R. L. Sites, and M. Horowitz, "ATUM: A new technique for capturing address traces using microcode," in *Proc. 13th Int. Symp. Comput. Architecture*, June 1986, pp. 119–129.

[2] A. Agarwal, M. Horowitz, and J. Hennessy, "Cache performance of operating systems and multiprogramming workloads," *ACM Trans. Comput. Syst.*, vol. 6, no. 4, pp. 393–431, Nov. 1988.

[3] ——, "An analytical cache model," *ACM Trans. Comput. Syst.*, vol. 7, no. 2, pp. 184–215, May 1989.

[4] C. Alexander, W. Keshlear, F. Cooper, and F. Briggs, "Cache memory performance in a UNIX environment," *Comput. Architecture News*, vol. 14, no. 3, pp. 14–70, June 1986.

[5] J. Baer and W. Wang, "On the inclusion properties for multi-level cache hierarchies," in *Proc. 15th Annu. Int. Symp. Comput. Architecture*, Honolulu, HI, June 1988, pp. 73–80.

[6] L. A. Belady, "A study of replacement algorithms for a virtual-storage computer," *IBM Syst. J.*, vol. 5, no. 2, pp. 78–101, 1966.

[7] J. Bell, D. Casasent, and C. G. Bell, "An investigation of alternative cache organizations," *IEEE Trans. Comput.*, vol. C-23, no. 4, pp. 346–351, Apr. 1974.

[8] B. T. Bennett and V. J. Kruskal, "LRU stack processing," *IBM J. Res. Develop.*, pp. 353–357, July 1975.

[9] J. M. Chambers, W. S. Cleveland, B. Kleiner, and P. A. Tukey, *Graphical Methods for Data Analysis.* Boston, MA: Duxbury, 1983.

[10] J. Cho, A. J. Smith, and H. Sachs, "The memory architecture and the cache and memory management unit for the Fairchild CLIPPER Processor," Comput. Sci. Div. Tech. Rep. UCB/Comput. Sci. Dep. 86/289, Univ. of California, Berkeley, Apr. 1986.

[11] D. W. Clark, "Cache performance in the VAX-11/780," *ACM Trans. Comput. Syst.*, vol. 1, no. 1, pp. 24–37, Feb. 1983.

[12] M. C. Easton and R. Fagin, "Cold-start versus warm-start miss ratios," *Commun. ACM*, vol. 21, no. 10, pp. 866–872, Oct. 1978.

[13] I. J. Haikala and P. H. Kutvonen, "Split cache organizations," CS Rep. C-1984-40., Univ. of Helsinki, Aug. 1984.

[14] M. D. Hill, DineroIII Documentation, Unpublished Unix-style Man Page, Univ. of California, Berkeley, October 1985.

[15] ——, "Aspects of cache memory and instruction buffer performance," Ph.D. dissertation, Comput. Sci. Div. Tech. Rep. UCB/Comput. Sci. Dep. 87/381, Univ. of California, Berkeley, Nov. 1987.

[16] ——, "A case for direct-mapped caches," *IEEE Comput. Mag.*, vol. 21, pp. 25–40, Dec. 1988.

[17] K. R. Kaplan and R. O. Winder, "Cache-based computer systems," *IEEE Comput. Mag.*, vol. 6, pp. 30–36, Mar. 1973.

[18] J. S. Liptay, "Structural aspects of the System/360 Model 85, Part II: The cache," *IBM Syst. J.*, vol. 7, no. 1, pp. 15–21, 1968.

[19] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger, "Evaluation techniques for storage hierarchies," *IBM Syst. J.*, vol. 9, no. 2, pp. 78–117, 1970.

[20] R. L. Mattson, "Evaluation of multilevel memories," *IEEE Trans. Magn.*, vol. MAG-7, no. 4, pp. 814–819, Dec. 1971.

[21] F. Olken, "Efficient methods for calculating the success function of fixed space replacement policies," Masters Report, Lawrence Berkeley Laboratory LBL-12370, Univ. of California, Berkeley, May 1981.

[22] S. Przybylski, M. Horowitz, and J. Hennessy, "Performance tradeoffs in cache design," in *Proc. 15th Annu. Int. Symp. Comput. Architecture*, Honolulu, HI, June 1988, pp. 290–298.

[23] T. R. Puzak, "Analysis of cache replacement algorithms," unpublished Ph.D. dissertation, Dep. Elec. Comput. Eng., Univ. of Massachusetts, Feb. 1985.

[24] D. R. Slutz and I. L. Traiger, "Evaluation techniques for cache memory hierarchies," IBM Tech. Rep. RJ 1045 (#17547), May 1972.

[25] A. J. Smith, "Two methods for the efficient analysis of memory address trace data," *IEEE Trans. Software Eng.*, vol. SE-3, no. 1, pp. 94–101, Jan. 1977.

[26] ——, "A comparative study of set associative memory mapping algorithms and their use for cache and main memory," *IEEE Trans. Software Eng.*, vol. SE-4, pp. 121–130, Mar. 1978.

[27] ——," Cache memories," *Comput. Surveys*, vol. 14, no. 3, pp. 473–530, Sept. 1982.

[28] A. J. Smith, "Cache evaluation and the impact of workload choice," in *Proc. 12th Int. Symp. Comput. Architecture*, June 1985, pp. 63–73.

[29] ——, "Bibliography and readings on CPU cache memories and related topics," *Comput. Architecture News*, Jan. 1986, pp. 22–42.

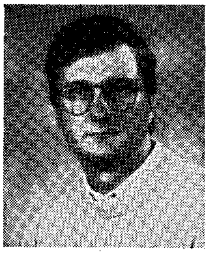
[30] ——, "Line (block) size choice for CPU caches," *IEEE Trans. Comput.*, vol. C-36, no. 9, pp. 1063–1075, Sept. 1987.

[31] J. E. Smith and J. R. Goodman, "Instruction cache replacement policies and organizations," *IEEE Trans. Comput.*, vol. C-34, pp. 234–241, Mar. 1985.

[32] W. D. Strecker, "Cache memories for PDP-11 family computers," in *Proc. 3rd Int. Symp. Comput. Architecture*, Jan. 1976, pp. 155–158.

[33] J. G. Thompson, "Efficient analysis of caching systems," Comput. Sci. Div. Tech. Rep. UCB/Comput. Sci. Dept. 87/374, Univ. of California, Berkeley, Oct. 1987.

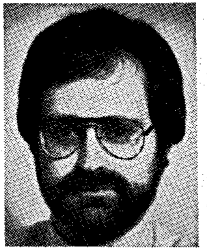
[34] I. L. Traiger and D. R. Slutz, "One-pass techniques for the evaluation of memory hierarchies," IBM Tech. Rep. RJ 892 (#15563), July 1971.



Mark D. Hill (S'81–M'87) received the B.S.E. degree in computer engineering from the University of Michigan, Ann Arbor, in 1981, and the M.S. and Ph.D. degrees in computer science from the University of California, Berkeley, in 1983 and 1987, respectively.

He is currently an Assistant Professor in the Computer Sciences Department at the University of Wisconsin, Madison. While at U.C. Berkeley, he was a principal contributor to SPUR, a project that built a shared-bus multiprocessor. His research interests center on computer architecture, with an emphasis on performance considerations and implementation factors in memory systems.

Dr. Hill is a member of ACM and a 1989 recipient of the National Science Foundation's Presidential Young Investigator award.



Alan Jay Smith (S'73–M'74–SM'83–F'89) was born in New Rochelle, NY. He received the B.S. degree in electrical engineering from the Massachusetts Institute of Technology, Cambridge, and the M.S. and Ph.D. degrees in computer science from Stanford University, Stanford, CA, the latter in 1974.

He is currently a Professor in the Computer Science Division of the Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, where he has been on the faculty since 1974, and was Vice Chairman of the EECS department from July 1982 to June 1984. His research interests include the analysis and modeling of computer systems and devices, computer architecture, and operating systems. He has published a large number of research papers, including one which won the IEEE Best Paper Award for the best paper in the IEEE Transactions on Computers in 1979. He also consults widely with computer and electronics companies.

Dr. Smith is a member of the Association for Computing Machinery, the Society for Industrial and Applied Mathematics, the Computer Measurement Group, Eta Kappa Nu, Tau Beta Pi, and Sigma Xi. He was chairman of the ACM Special Interest Group on Operating Systems (SIGOPS) from 1983 to 1987, was on the board of directors of the ACM Special Interst Group on Measurement and Evaluation (SIGMETRICS) from 1985 to 1989, was an ACM National Lecturer (1985–1986) and an IEEE Distinguished Visitor (1986–1987), is an Associate Editor of the *ACM Transactions on Computer Systems* (TOCS), a subject area editor of the *Journal of Parallel and Distributed Computing* and is on the editorial board of the *Journal of Microprocessors and Microsystems*. He was program chairman for the Sigmetrics '89/Performance '89 Conference.