EVALUATING ASSOCIATIVITY IN CPU CACHES

by

Mark D. Hill

Alan Jay Smith

Computer Sciences Technical Report #823

February 1989

Evaluating Associativity in CPU Caches[†]

Mark D. Hill

Computer Sciences Department
University of Wisconsin
Madison, Wisconsin 53706
markhill@cs.wisc.edu

Alan Jay Smith

Computer Science Division
EECS Department
University of California
Berkeley, California 94720

Because of the infeasibility or expense of large fully-associative caches, cache memories are often 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 the 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 about 5, 10 and 30 percent respectively, regardless cache size.

[†] The material presented here is based on research supported in part by the Defense Advanced Research Projects Agency monitored by Naval Electronics Systems Command under Contract No. N00039-85-C-0269, the National Science Foundation under grants CCR-8202591 and MIP-8713274, by the State of California under the MICRO program, the graduate school at the University of Wisconsin-Madison, and by IBM Corporation, Digital Equipment Corporation, Hewlett Packard Corporation, and Signetics Corporation.

1. Introduction

Three important CPU cache parameters are cache size, block (line) size and associativity [Smit82]. Cache size (buffer size, capacity) is so important that it is a part of almost all cache studies (for a partial bibliography see [Smit86]). Block size (line size) has recently been examined in detail in [Smit87]. Here we concentrate on associativity (degree of associativity, set size), which is the number of places in a cache where a block can reside. Figure 1 illustrates associativity and defines some terms.

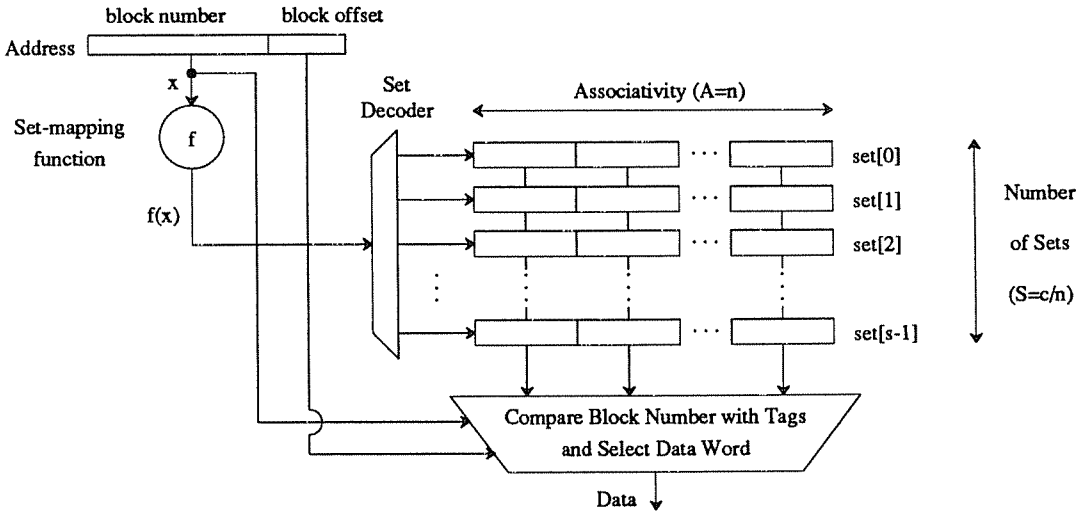


Figure 1. Set-Associative Mapping.

A set-associative cache uses a *set-mapping function* f to partition all *blocks* (data in an aligned, fixed-sized regions 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 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.

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 it equals several low-order bits of the block number. For 256 sets, for example, $f(x) = x \bmod 256$ or $f(x) = x$ AND 0xff, where mod is remainder and AND is bitwise-and.

Determining 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 set 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. On the other hand, increasing associativity often increases cache cost and access time, since more blocks (frames) must be searched in parallel to find a reference [Hill88].

The method we use for examining associativity in CPU caches is *trace-driven simulation.* It uses one or more (address) *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 [Smit85b]. 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. [Matt70] 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 2 reviews previous work on cache simulation algorithms and associativity in caches. In section 3, we explain our methods in more detail and describe our traces. Section 4 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 5 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 [Smit85b] and [Smit87] to caches with reduced associativity.

Readers interested in the effect of associativity on miss ratio but not in cache simulation algorithms, may skip Section 4, as Section 5 is written to stand alone.

2. Related Work

2.1. Simulation Algorithms

The original paper on memory hierarchy simulation is by Mattson et al. [Matt70]. 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[†]. Mattson et al. show inclusion holds between alternative caches that have the same block size, do no prefetching and use the same set-mapping function (and therefore have the same number of sets) for 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. [Bela66]. FIFO does not qualify since cache capacity affects a block's replacement priority. In Section 4.1, 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 associativities, 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 call these three operations FIND, METRIC and UPDATE, and will show that the algorithms discussed in later in Sections 4.2 and 4.3 use the same steps.

The most straight-forward 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

[†] *Inclusion* is different from *multi-level inclusion* defined by Baer and Wang [Baer88]. While inclusion is a property relating alternative caches, multi-level inclusion relates caches in the same cache hierarchy.

more complex implementations of stack simulation, using hash tables, m-ary trees and AVL trees [Benn75, Olke81, Thom87]. While these algorithms are useful for some memory hierarchy simulations, Thompson [Thom87] 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 [Trai71], and with different numbers of sets, block sizes and sub-block sizes (sector and block sizes, address and transfer block sizes) [Slut72]. They require that all alternative caches use LRU replacement, bit-selection for set mapping, and have block and sub-block 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 4.3, 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 [Smit77]. Puzak [Puza85] shows that if all caches simulated use bit-selection and LRU replacement, references that hit in a direct-mapped cache having the fewest number of sets can be deleted without affecting the total number of misses. We will show that this result trivially follows from properties we define in Section 4.1, 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.)

2.2. 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 32K-byte and smaller caches, among the more notable of which are: ([Lipt68], [Kapl73], [Bell74], [Stre76], [Smit82],[†] [Clar83] and [Haik84]); (2) papers that discuss associativity and other aspects of cache design for larger caches ([Alex86], [Agar88] and [Przy88]); and (3) those that discuss only associativity ([Smit78] and [Hill88]). Since caches have been getting larger, papers in category (1) can also be characterized as older, while those in category (2) are more recent.

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

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 larger caches.

Alexander et al. use trace-driven simulation to study small and large caches [Alex86]. Unfortunately, the miss ratios they give are much lower than those that have been measured with hardware monitors and real workloads; see [Smit85b] 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 [Agar88]. 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 (25 percent).

Przybylski et al. [Przy88] 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 [Smit78]. 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 5.2, 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 [Hill88], based on parts of [Hill87]. It shows that many large single-level 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.

3. 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.

Ignoring write-backs, the effective access time of a cache can be modeled as $t_{cache} + miss\_ratio \cdot t_{memory}$. 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 [Hill87] 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 (see Table 1). The first group consists of the second 500,000 references from five traces, three DEC VAX multiprogrammed workloads from ATUM [Agar86] and two samples of IBM 370 code from a trace of IBM's MVS [Smit82]. The second group is made up of 23 traces gathered with ATUM [Agar86]. We refer these workloads as the *five-trace* and *23-trace* groups, respectively. Trace samples that exhibit atypical 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 [Agar88]. 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. [Agar88]. Nevertheless, results for these multiprogrammed traces properly includes cold-start effects whenever a process resumes execution.

We calculate overall miss ratios using the arithmetic average of trace sample miss ratios. The arithmetic mean is reasonable for this task, 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 [Smit87] and [Hill87]) showed that little difference was observed when other averaging methods were used.

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

Table 1. Data on Traces.

These tables present data on the address traces in a five-trace group (top) and a 23-trace group (bottom). 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 [Agar86]. 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 [Smit85b].

The second trace group contains 23 samples of various workloads gathered on a VAX-11 with ATUM [Agar86]. We use this larger trace group selectively to reduce overall simulation time.

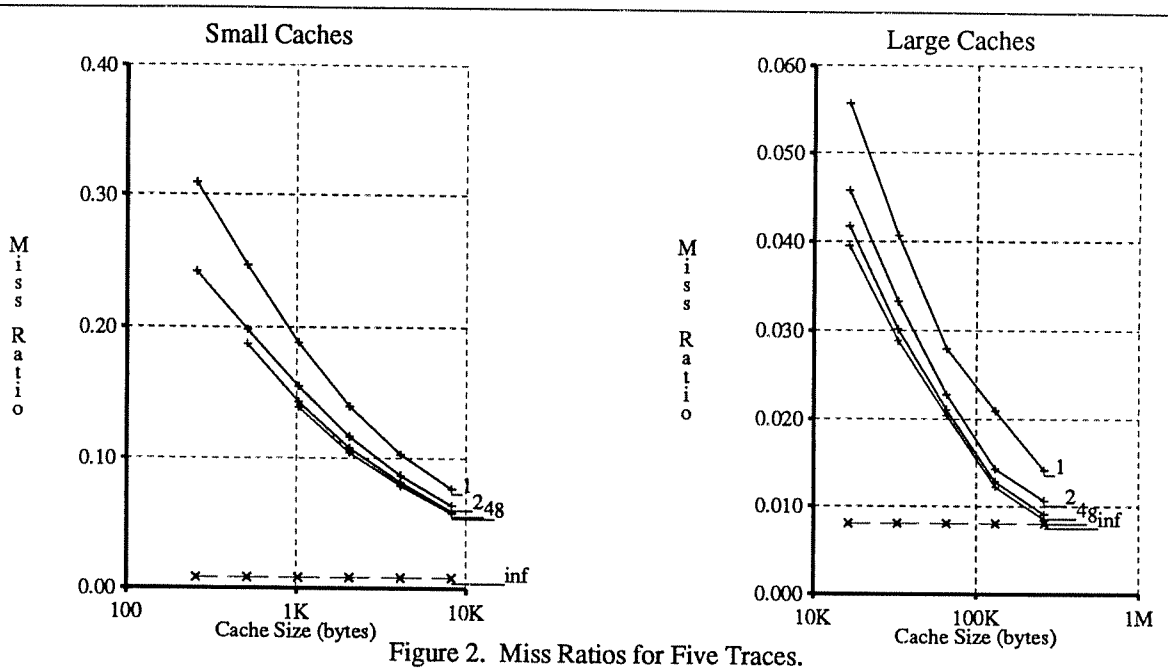


Figure 2. Miss Ratios for Five Traces.

This figure examines 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. Caches are considered warm after 250K references.

The dashed line, (labeled "inf") gives the warm-start miss ratio of an infinite cache, a cache so large that it never replaces any blocks.

Figures 2 and 3 show our miss ratios for various caches and compare some them with other published results. For brevity, only results of the five-trace group are shown here.

4. 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 [Trai71] to one that allows arbitrary set-mapping functions. Finally we compare implementations of the algorithms.

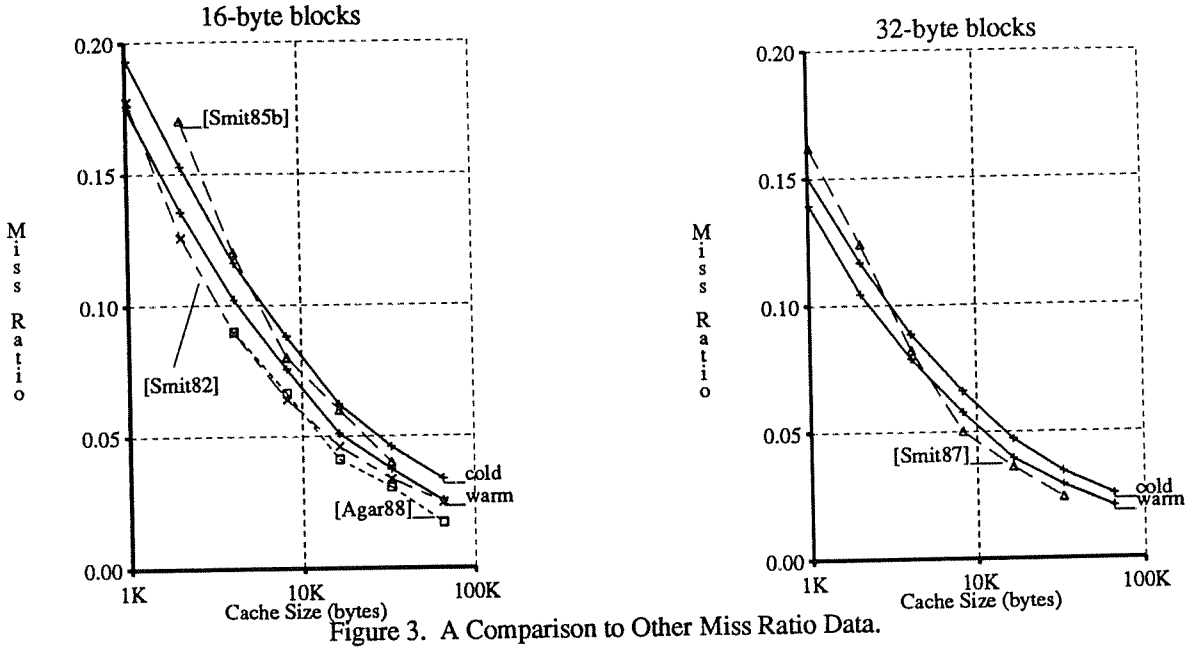


Figure 3. A Comparison to Other Miss Ratio Data.

This figure compares miss ratios for the five-trace group in eight-way set-associative unified caches, having 16-byte (left) and 32-byte (right) blocks, to miss ratios from other sources. Line "cold" measures miss ratios form an empty cache, while line "warm" does not count miss until after 250K references. Since the trace samples include multiprogramming effect, both contain some cold-start missed [East78].

Lines labeled "[Smit85b]" (left) and "[Smit87]" (right) show the design target miss ratios from those papers for fully-associative caches. The line labeled "[Agar88]" (left) shows four-way set-associative miss ratio results from Figure 17 in that paper. Finally, the line labeled "[Smit82]" (also left) shows four- , six- and eight-way set-associative miss ratios taken from hardware monitor measurements on an Amdahl 470 (figure 33 of that paper).

This figure demonstrates that the miss ratios of the five-trace group are consistent with those measured and/or proposed for actual operating environments.

4.1. Properties that Facilitate Rapid Simulation

Two properties useful for simulating alternative direct-mapped and set-associative caches are *set-refinement*† (introduced below) and *inclusion* (introduced in Mattson et al. [Matt70]). 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, $i = 1, 2$.

† *Set-refinement* is called *set-hierarchy* in [Hill87].

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 4.2) and set-associative caches (Section 4.3).

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. [Matt70] show when inclusion holds for alternative caches that use the same set-mapping function (and hence 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 $n_2 \geq n_1$ (non-decreasing 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 non-decreasing 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 $w_1, \ldots, w_{n_2}$[†]. Consider references to $x, w_1, \ldots, w_{n_2}$. 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 $w_1, \ldots, w_{n_2}$ 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 map to its set are more recently referenced.

(2) Let $x_0, \ldots, x_{n_2}$ 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 $x_0, x_1, \ldots, x_{n_2}$. 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 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

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

$(x \bmod 2^j = y \bmod 2^j)$, 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 non-decreasing 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 [Hill87].

(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 [Smit77]. 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 [Puza85].

4.2. 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 useful.

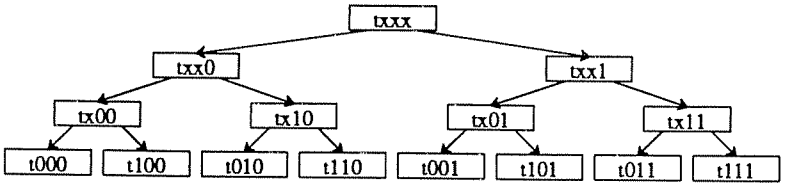
In the last section we showed set-refinement implies inclusion in direct-mapped caches. Forest simulation takes advantage of inclusion, as does stack simulation, by searching for a block from the smallest to largest cache. When a block is found, a hit is implicitly recorded for all larger caches.

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 i-th smallest cache (see Figure 4a). 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 i-th smallest cache.

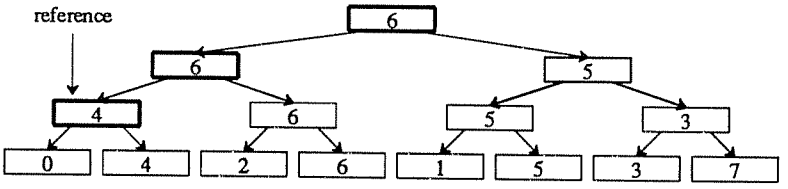
Figure 4bc presents an example of forest simulation, while Figure 5 shows pseudo-code for the algorithm. For each reference, the key idea is to begin at level 1 and proceed downward in the forest until the reference is found or the forest exhausted. At each level, the location of the search is guided 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 i-th smallest direct-mapped cache is $1 - \sum_{j=1}^{i} distance[j]/N$. We will analyze the performance of forest simulation in Section 4.4.

The principal limitation of forest simulation is that it only works for direct-mapped caches. Extending the algorithm to set-associative caches introduces 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 Figure 4b 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.

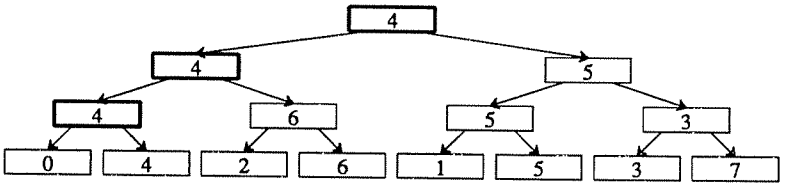
Forest simulation can be extended to simulate N-way set associativity by replacing each node in the forest by an N-element LRU stack. At each reference, rather than just replacing the element at a node with the newest reference, the stack at that node is updated in the normal LRU manner; the descent 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



(a) a forest with bit-selection



(b) before a reference to block 4



(c) after the reference

Figure 4. Forest Simulation Example.

The top tree 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 could 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 that 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.

The middle tree (b) depicts the forest after a series of references. 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 1 and 2 miss and caches 3 and 4 hit.

The bottom tree (c) shows the tree after this reference as been processed. The nodes in bold now contain the referenced block.

consisting of only one element. 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.

```
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
        }
}
```

Figure 5. Forest Simulation.

4.3. 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 4.1). This work generalizes to arbitrary set-mapping functions an algorithm developed for caches using bit-selection only [Matt70, Trai71]. The algorithms discussed in this section can also be extended to handle multiple block sizes and sector sizes [Slut72, Trai71].

In theory, the storage required for all-associativity simulation is $O(N_{unique})$, where N_{unique} is the number of unique blocks referenced in an address trace. 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.

Figures 9 through 11 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 Figures 9 through 11, 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). The left-hand side of Figure 6 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 the right-hand side of Figure 6, 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[†]. 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. Figure 7 illustrates that this method, which we call *all-associativity* simulation.

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

[†] 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 block 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 re-reference 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 block 0 and 2 are now in a different order in different stacks.

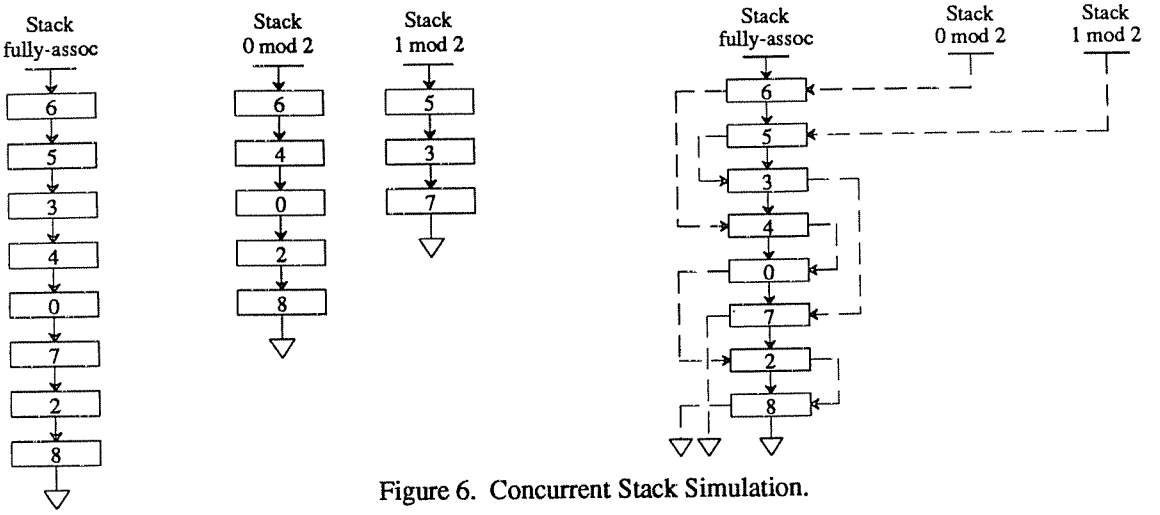


Figure 6. Concurrent Stack Simulation.

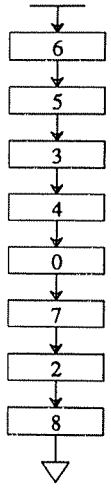
The three singly-linked stacks on the left-hand side of this figure display how the stacks for caches with one set (fully-associative) or two sets using bit selection ($f_1(x)=0$ and $f_2(x)=x \bmod 2$) could look during a simulation. The stack for one set contains a list of all the block numbers recently referenced, listed from most-recently-referenced to least-recently-referenced. We call this stack a *fully-associative* stack, because it models fully-associative caches. The stacks for two sets contain similar lists for the even and odd block numbers. 2^L stacks are required to simulate with bit selection for 2^L sets. A block resides in a cache of c block frames with one set if and only if the block is in the fully-associative stack at a distance of less than or equal to c. A block resides in a cache of c block frames with two sets if and only if it is in the appropriate stack at a distance of less than or equal to $c/2$. A block resides in a cache of c block frames with 2^L sets if and only if it is in the appropriate stack at a distance of less than or equal to $c/2^L$.

The multiply-linked stack on the right-hand side illustrates how a single set of nodes can be used to represent the stacks for caches using bit selection with one and two sets. A second *next* pointer field must be added to each node so that it can be linked into a second stack. The stacks for stack simulations with L different set-mapping functions can share one group of nodes if each node contains storage for L different *next* pointers. This reduces the expected storage requirements with respect to using separate stacks, but does not reduce simulation time.

across many stacks (e.g., 512 stacks for simulating the VAX-11/780's cache [Clar83]).

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 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[‡], this can result in non-trivial savings if L is large (see Figure 8).

[‡] 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 $\sum_{i=0}^{\infty}(i+1)/2^{i+1}=2$.



Stack fully-assoc	Block 2 found?	Fully-Assoc f(x) = 0 Same set?	Fully-Assoc f(x) = 0 stack_count[1]	Two Sets f(x) = x mod 2 Same set?	Two Sets f(x) = x mod 2 stack_count[2]	Four Sets f(x) = x mod 4 Same set?	Four Sets f(x) = x mod 4 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

Figure 7. All-Associativity Simulation Example.

This figure illustrates how all-associativity simulation processes a reference to block 2 for caches with set-mapping functions $f_1(x) = 0$, $f_2(x) = x \bmod 2$, and $f_3(x) = x \bmod 4$. Counter *stack_count[i]* always contains the number of blocks encountered so far in stack $f_i(2)$. Each row of the figure shows that *stack_count[i]* is incremented in response to block y in the stack if and only if $f_i(y) = f_i(2)$.

Processing stops when the reference is found (block 2). The stack distance of block 2 in a cache with set-mapping function f_i is *stack_count[i]*. The stack distances found for block 2 are 7, 4, and 2, respectively.

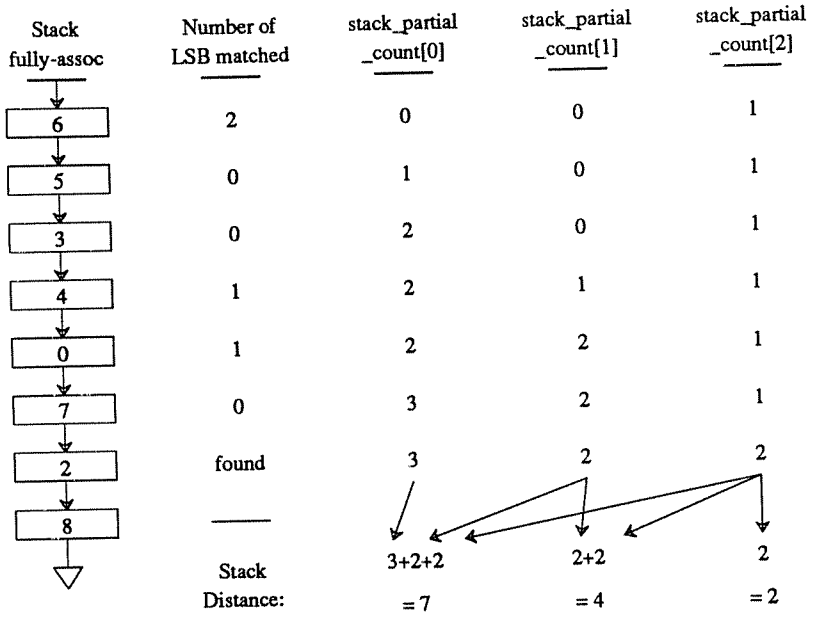


Figure 8. All-Associativity Simulation with Set-Refinement Example.

This figure illustrates how all-associativity simulation with set-refinement processes a reference to block 2 by scanning the stack until block 2 is found (or the stack is exhausted). For each block before the reference is found, the algorithm calculates the largest i for which the reference and the stack node are in the same f_i set, and increments *stack_partial_count*[i]. For bit selection, finding this set-mapping function reduces to determining the number of least-significant bits that match between the block numbers of the reference and the stack node. Once the reference is found, *stack_partial_count*[L] is incremented, the reference's stack distance with set-mapping function f_i is $\sum_{j=i}^{L} stack\_partial\_count[j]$.

```
integer L  /* number of set-mapping functions */
function f1(x), ..., fL(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=fi) = 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 (fi(x)==fi(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)
}
```

Figure 9. All-Associativity Simulation.

```
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. */
```

Figure 10. All-Associativity with Set-Refinement (Storage).

```
For each reference x {
        for i=1 to L { stack_partial_count[i] = 0 }
        read(var x)
        N++
        stack_number = f1(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 (fi(x)==fi(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)
}
```

Figure 11. All-Associativity with Set-Refinement (Key Procedure).

4.4. 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[‡] [Hill85]. Adding stack simulation increased total code size by 150 statements, adding forest simulation, 220 statements. Stack simulation is implemented using linked lists. The forest simulation 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.

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[†]	(4.0)	326.1	(1.054)	402.9	(1.303)
16K to 128K	1-way	1234.4[†]	(4.0)	321.0	(1.038)	332.3	(1.074)
16K to 128K	1-, 2- & 4-way	6308.6[†]	(6.0)	--	--	366.6	(1.185)

Table 2. Simulation Times.

This table shows simulation times for C language implementations of stack, forest and all-associativity simulation. All caches simulated have 32-byte blocks, do no prefetching, use LRU replacement, are unified (data and instructions cached together) and use bit selection. Results in the first row 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 [Smit85b]. We also examined traces from three other architectures [Hill87]. We omit these results here, since they are qualitatively 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 16K-byte direct-mapped (1-way) cache with the MVS trace. Readers interested in simulation performance times for fully-associative caches, driven by traces of memory and disk references should consult [Thom87].

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

We performed experiments, described in Table 2, to answer the following three questions regarding how these implementations perform for CPU cache simulations. Performance simulating other caching systems, e.g., disk caches, will differ [Thom87].

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

(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 2). 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 non-direct-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, ..., another with c sets, another with $2c$ sets, ..., and finally one with $2^s c$ sets. The simulation in the final row of Table 2, for example, required six stack simulations, using 128, 256, ... and 4K stacks, respectively.

5. 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 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.

5.1. Categorizing Set-Associative Misses

The simulation algorithms described earlier facilitate computing the miss ratios for many alternative cache sizes and associativities. This 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[†]).

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 [Agar89]).

Table 3 illustrates this miss ratio categorization for a trace of VAX-11 interactive users under Ultrix. For this trace, we see (1) the absolute size of the conflict miss ratios for set-associative caches (not direct-mapped) are small, making further increases in associativity of limited benefit, (2) the absolute (relative) size of conflict miss ratios for direct-mapped caches gets smaller (larger) with increasing cache size, making increasing associativity absolutely less (relatively more) 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.

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

It has been previously shown by one of the authors [Smit78] 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 i-th most-recently-referenced block in one of s

[†] That is, necessary without violating our assumptions of a fixed block size, LRU replacement, no prefetching and bit-selection. The compulsory miss ratio is equivalent the "cold start" miss ratio, as defined in [East78], for an arbitrary large cache.

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%
64K	1-way	0.0386	0.0105	27%	0.0192	50%	0.0090	23%
64K	2-way	0.0296	0.0015	5%	0.0192	65%	0.0090	30%
64K	4-way	0.0279	-0.0002	-1%	0.0192	69%	0.0090	32%
64K	8-way	0.0275	-0.0006	-2%	0.0192	70%	0.0090	33%
128K	1-way	0.0261	0.0130	50%	0.0041	16%	0.0090	34%
128K	2-way	0.0195	0.0064	33%	0.0041	21%	0.0090	46%
128K	4-way	0.0164	0.0033	20%	0.0041	25%	0.0090	55%
128K	8-way	0.0151	0.0021	14%	0.0041	27%	0.0090	59%

Table 3. Three Miss Ratio Components.

This table illustrates the effect of dividing the miss ratios for three samples of "ue" (see Table 1) into *(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). For an n-way set-associative cache with s sets, having miss ratio $m(A=n, S=s)$, the conflict miss ratio is $m(A=n, S=s) - m(A=n \cdot s, S=1)$, the capacity miss ratio is $m(A=n \cdot s, S=1) - m(A=\infty, S=1)$ and the compulsory miss ratio is $m(A=\infty, S=1)$. Direct-mapped caches are denoted by "1-way".

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. Results for large caches are unstable, since small miss ratio variations can cause a large perturbation in a component's relative size.

That several conflict miss ratios for eight-way set-associative caches are negative is unimportant, since (1) the magnitudes are small (-0.0006), indicating eight-way set-associative caches and fully-associative cache have approximately the same miss ratio, and (2) the behavior is possible. A c-block fully-associative cache misses on all references to blocks not among the c most-recently referenced blocks, while a set-associative cache can hit on some of these blocks. If such references are common, for example, due to a loop spanning $c+1$ blocks, a set-associative or direct-mapped cache can perform better [Smit85a].

sets, while q_i is the probability a reference is made to the i-th 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[†] allows us to express a set-associative LRU distance probability in terms of fully-associative LRU distance probabilities:

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

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

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

Substitution yields:

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

Figure 12 shows actual direct-mapped and set-associative miss ratios and miss ratios predicted with the above equation. Results here and for several other traces [Hill87] yield three conclusions:

(1) The predictions are quite accurate. In most cases the relative error is less than five percent; only rarely is it greater than ten 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

[†] For some event A and a set of mutually exclusive and exhaustive events B_i, Bayes Rule states that: $\text{Prob}(A) = \sum \text{Prob}(A \mid B_i) \cdot \text{Prob}(B_i)$.

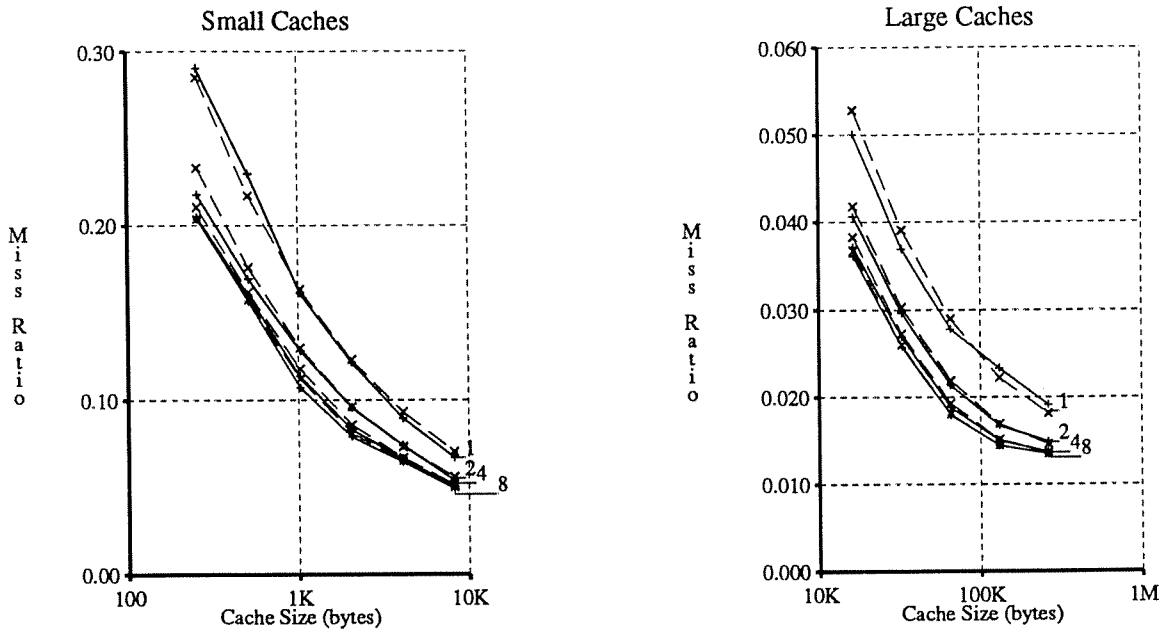


Figure 12. Predicted and Actual Miss Ratios for "mul2".

This figure compares set-associative miss ratio predictions using a model from [Smit78] (dashed lines) with actual miss ratios (solid lines) for various associativities. The data are for unified caches with 32-byte blocks, using trace *mul2*

the above approximation assumes), due to spatial locatity [Smit78].

(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 because it illuminates the cause of (set-)conflict misses, showing that the actual rate of conflict misses is nearly identical to the rate of conflict misses resulting from assuming that active blocks map to sets with independent and equal probability.

5.3. 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, having 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:

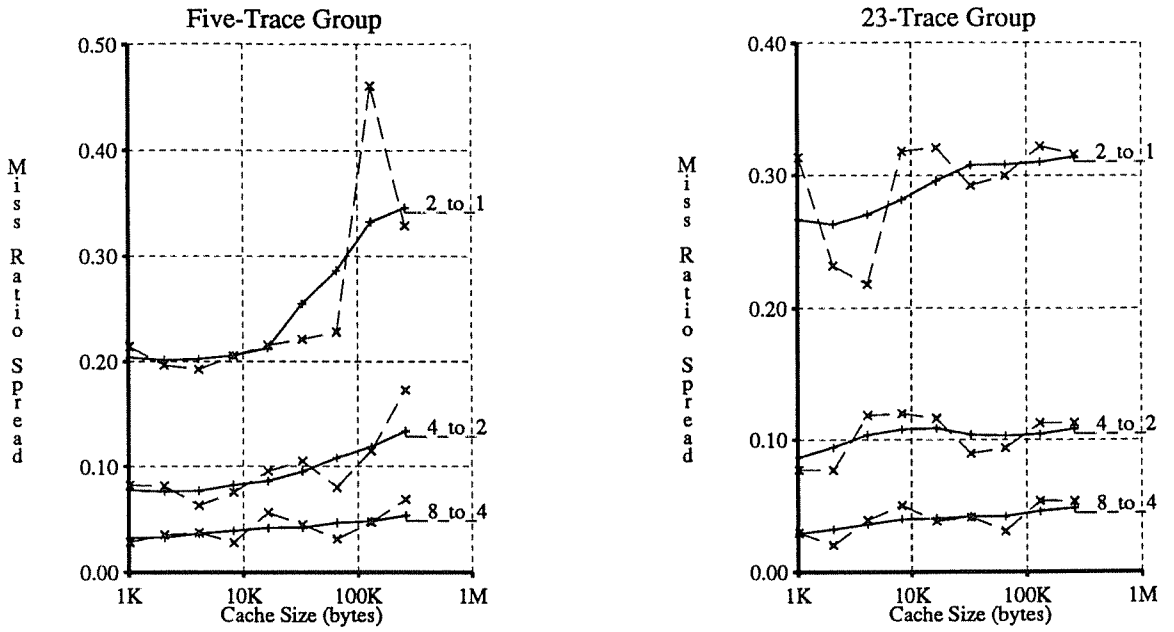


Figure 13. Unified Cache Miss Ratio Spreads.

This figure shows average miss ratio spreads for unified caches using 32-byte blocks and LRU replacement with the five- (left) and 23-trace (right) groups. The miss ratio spread is the relative increase incurred by halving the associativity of a cache. The average miss ratio spread is computed using the ratio of the average miss ratios. Table 4 shows similar results from an alternate computation, taking the average of the miss ratio spreads of individual traces. Dashed lines present raw data, while solid lines are smoothed using a weighted average of adjacent spreads (recommended in [Cham83]). 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.

For the most part, miss ratio spreads vary little with changing cache size. The only major exception to this rule is the miss ratio spread between direct-mapped and two-way set-associative 128K-byte 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 128K-byte caches.

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

Figures 13 and 14 and Table 4 present data from trace-driven simulation. Figure 13 shows some miss ratio spreads of unified caches with 32-byte blocks for the five- and 23-trace groups. Figure 14 examines miss ratio spreads for instruction and data cache with the five-trace group. Table 4 shows miss ratios spreads, calculated in a different way, for many caches with the 23-trace groups. These results together with more data in [Hill87] 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

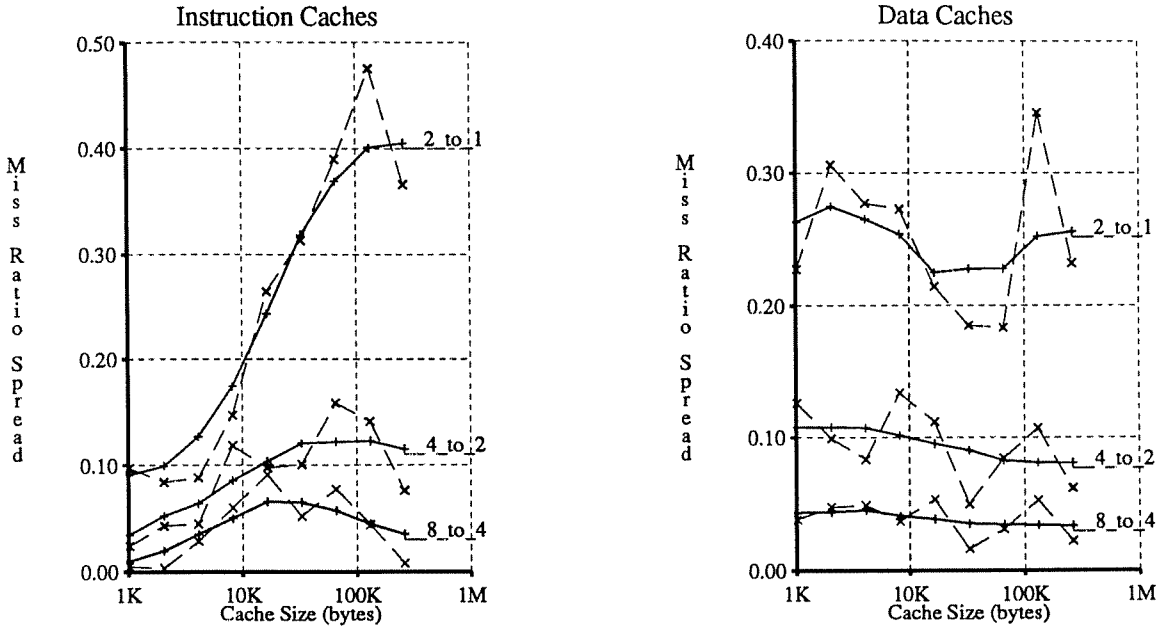


Figure 14. Instruction and Data Cache Miss Ratio Spreads.

This figure shows miss ratio spreads for instruction (left) and data (right) caches using 32-byte blocks and LRU replacement with the five-trace group. The miss ratio spread is the relative increase incurred by halving the associativity of a cache. Dashed lines present raw data, while solid lines are smoothed using a weighted average of adjacent spreads.

Most spreads show little systematic variation with changes in cache size. The spread between two-way set-associative and direct-mapped instruction caches, however, is strongly correlated with cache size. We expect that this is due to looping behavior, which diminishes the miss ratio spreads for small instruction caches [Smit85a].

ratio spreads between small direct-mapped and two-way set-associative instruction caches are smaller than many other spreads due to the looping behavior of instruction reference streams, which minimizes the usefulness of increasing associativity in small instruction caches [Smit85a].

(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.

(4) Miss ratio spreads between unified and data caches are similar. Instruction cache spreads are similar or smaller (see also [Cho86]). Miss ratio spreads between direct-mapped and two-way set-associative instruction caches are significantly smaller than other spreads, as has been observed elsewhere [Smit85a].

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 4. To one significant figure, halving associativity with these traces from eight-way to four-way to two-

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 4. Smoothed Miss Ratio Spreads.

This table displays smoothed miss ratio spreads for caches with the 23-trace group. The miss ratio spread is the relative increase incurred by halving the associativity of a cache. Spreads are smoothed using the method described by Figure 13. These spreads are calculated by using the geometric average of the ratio of miss ratios for individual traces. This method yields slightly larger spreads than those calculated using the ratio of average miss ratios (as in Figure 13). 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.

Relative Miss Ratio Change for the Five-Trace Group							
Cache Type	Block Size	From Direct-Mapped To			From Eight-Way To		
		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 Type	Block Size	From Direct-Mapped To			From Eight-Way To		
		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%

Table 5. Relative Miss Ratio Change.

This table displays the average relative miss ratio changes with the five-trace (top) and the 23-trace group (bottom). The relative miss ratio change from direct-mapped to n-way set-associative is $[m(A=n)-m(A=1)]/m(A=1)$ where $m(A=i)$ is the miss ratio of an i-way set-associative cache. Since most $m(A=n)$'s are less than $m(A=1)$'s, these changes are negative. Similarly, the change from eight-way set-associative is $[m(A=n)-m(A=8)]/m(A=8)$. Since eight-way set-associative miss ratios are near fully-associative miss ratios, these changes give the relative size of the (set-)conflict miss ratio component. All average changes are calculated by using the geometric average of the ratio of miss ratios for each trace at each cache size from 1K to 256K bytes. Except for round-off error, the numbers for the 23-trace group are equivalent to the average miss ratio spreads of Table 4.

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 5. Relative to direct-mapped, the miss ratios for eight-, four- and two-way set-associative are respectively about 34%, 30% and 22% lower. Assuming that eight-way set-associative is effectively fully-associative, the miss ratio increases by 5% for four-way, 17% for two-way and 52% for direct-mapped.

Our examination of miss ratios for caches with different associativities has shown that the miss ratio spread does not change dramatically as caches get larger. Consequently the absolute miss ratio difference decreases as cache gets 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 [Hill88, Przy88].

5.4. Extending Design Target Miss Ratios

In [Smit85b], 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 [Smit85b] presented miss ratios for fully associative caches with 16-byte blocks, broken down into figures for unified, instruction and data caches. In another paper [Smit87], 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.

We use the same method here to extend the design target miss ratios to caches of limited associativity (see Table 6). 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 4.

6. 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 [Matt70]) 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-associative caches. Our algorithms is a generalization of an earlier algorithm [Matt70, Trai71]. 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:

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

Table 6. Design Target Miss Ratios.

In this table we extend the *design target miss ratios* from [Smit85b] and to caches of varying associativity by multiplying those numbers by the smoothed miss ratio spreads of Table 4. These miss ratios may serve as "rules of thumb" for cache designers working with "a 32-bit architecture running fairly large programs and mature (i.e., large) operating system.

We do not extend the design target miss ratios to caches larger than 32K bytes, because the original design target miss ratios in [Smit85b] and [Smit87] are limited to caches of 32K-bytes or less, and the methodology for extending them to larger cache sizes is beyond the scope of this paper.

- 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 [Smit78] to a wide variety of caches, we show that the rate of conflict misses (i.e, why set-associative miss ratios are larger than fully-associative ones) can be predicted by assuming blocks map to sets uniformly and independently, resulting in too many active blocks map to a fraction of the sets.

- Finally, 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.

7. Acknowledgments

We would like to thank Randy Katz, David Patterson and other members of the SPUR project for their many suggestions that improved the quality of our research, Harold Stone for comments on [Hill87], and Sue Dentinger, Garth Gibson and Viranjit Madan for reading and improving drafts of this paper.

8. References

[Agar86] A. Agarwal, R. L. Sites and M. Horowitz, ATUM: A New Technique for Capturing Address Traces Using Microcode, *Proc. Thirteenth International Symposium on Computer Architecture* (June 1986).

[Agar88] A. Agarwal, M. Horowitz and J. Hennessy, Cache Performance of Operating Systems and Multiprogramming Workloads, *ACM Trans. on Computer Systems*, 6, 4 (November 1988).

[Agar89] A. Agarwal, M. Horowitz and J. Hennessy, An Analytical Cache Model, *ACM Trans. on Computer Systems*, 7, 2 (May 1989).

[Alex86] C. Alexander, W. Keshlear, F. Cooper and F. Briggs, Cache Memory Performance in a UNIX Environment, *Computer Architecture News*, 14, 3 (June 1986), 14-70.

[Baer88] J. Baer and W. Wang, On the Inclusion Properties for Multi-Level Cache Hierarchies, *15th Annual International Symposium on Computer Architecture*, Honolulu, Hawaii (June 1988).

[Bela66] L. A. Belady and J. Gecsei, A Study of Replacement Algorithms for a Virtual-Storage Computer, *IBM Systems Journal*, 5, 2 (1966).

[Bell74] J. Bell, D. Casasent and C. G. Bell, An Investigation of Alternative Cache Organizations, *IEEE Trans. on Computers*, C-23, 4 (April 1974), 346-351.

[Benn75] B. T. Bennett and V. J. Kruskal, LRU Stack Processing, *IBM Journal of R & D* (July 1975).

[Cham83] J. M. Chambers, W. S. Cleveland, B. Kleiner and P. A. Tukey, *Graphical Methods for Data Analysis*, Duxbury Press, Boston, (1983).

[Cho86] J. Cho, A. J. Smith and H. Sachs, The Memory Architecture and the Cache and Memory Management Unit for the Fairchild CLIPPER Processor, Computer Science Division Technical Report UCB/Computer Science Dept. 86/289, University of California, Berkeley (April, 1986).

[Clar83] D. W. Clark, Cache Performance in the VAX-11/780, *ACM Trans. on Computer Systems*, 1, 1 (February, 1983), 24 - 37.

[East78] M. C. Easton and R. Fagin, Cold-Start vs. Warm-Start Miss Ratios, *Communications of the ACM*, 21, 10 (October 1978), 866-872.

[Haik84] I. J. Haikala and P. H. Kutvonen, Split Cache Organizations, CS Report C-1984-40., Univ. of Helsinki (August 1984).

[Hill85] M. D. Hill, DineroIII Documentation, Unpublished Unix-style Man Page, University of California, Berkeley (October 1985).

[Hill87] M. D. Hill, Aspects of Cache Memory and Instruction Buffer Performance, Ph.D. Thesis, Computer Science Division Technical Report UCB/Computer Science Dept. 87/381, University of California, Berkeley (November 1987).

[Hill88] M. D. Hill, A Case for Direct-Mapped Caches, *IEEE Computer*, 21, 12 (December 1988), 25-40.

[Kapl73] K. R. Kaplan and R. O. Winder, Cache-based Computer Systems, *Computer*, 6, 3 (March, 1973).

[Lipt68] J. S. Liptay, Structural Aspects of the System/360 Model 85, Part II: The Cache, *IBM Systems Journal*, 7, 1 (1968), 15-21.

[Matt70] R. L. Mattson, J. Gecsei, D. R. Slutz and I. L. Traiger, Evaluation techniques for storage hierarchies, *IBM Systems Journal*, 9, 2 (1970), 78 - 117.

[Matt71] R. L. Mattson, Evaluation of Multilevel Memories, *IEEE Trans. on Magnetics*, MAG-7, 4 (December 1971).

[Olke81] F. Olken, Efficient Methods for Calculating the Success Function of Fixed Space Replacement Policies, Masters Report, Lawrence Berkeley Laboratory LBL-12370, University of California, Berkeley (May 1981).

[Przy88] S. Przybylski, M. Horowitz and J. Hennessy, Performance Tradeoffs in Cache Design, *15th Annual International Symposium on Computer Architecture*, Honolulu, Hawaii (June 1988).

[Puza85] T. R. Puzak, Analysis of Cache Replacement Algorithms, Unpublished Ph.D. Dissertation, Dept. of Electrical and Computer Engineering, University of Massachusetts (February 1985).

[Slut72] D. R. Slutz and I. L. Traiger, Evaluation Techniques for Cache Memory Hierarchies, IBM Technical Report RJ 1045 (#17547) (May 1972).

[Smit77] A. J. Smith, Two Methods for the Efficient Analysis of Memory Address Trace Data, *IEEE Transactions on Software Engineering*, SE-3, 1 (January 1977).

[Smit78] A. J. Smith, A Comparative Study of Set Associative Memory Mapping Algorithms and Their Use for Cache and Main Memory, *IEEE Trans. on Software Engineering*, SE-4, 2 (March 1978), 121-130.

[Smit82] A. J. Smith, Cache Memories, *Computing Surveys*, 14, 3 (September, 1982), 473 - 530.

[Smit85a] J. E. Smith and J. R. Goodman, Instruction Cache Replacement Policies and Organizations, *IEEE Trans. on Computers*, C-34, 3 (March 1985), 234-241.

[Smit85b] A. J. Smith, Cache Evaluation and the Impact of Workload Choice, *Proc. Twelfth International Symposium on Computer Architecture* (June 1985).

[Smit86] A. J. Smith, Bibliography and Readings on CPU Cache Memories and Related Topics, *Computer Architecture News* (January 1986), 22-42.

[Smit87] A. J. Smith, Line (Block) Size Choice for CPU Caches, *IEEE Trans. on Computers*, C-36, 9 (September 1987).

[Stre76] W. D. Strecker, Cache Memories for PDP-11 Family Computers, *Proc. Third International Symposium on Computer Architecture* (January 1976), 155-158.

[Thom87] J. G. Thompson, Efficient Analysis of Caching Systems, Computer Science Division Technical Report UCB/Computer Science Dept. 87/374, University of California, Berkeley (October 1987).

[Trai71] I. L. Traiger and D. R. Slutz, One-Pass Techniques for the Evaluation of Memory Hierarchies, IBM Technical Report RJ 892 (#15563) (July, 1971).