A COMPARISON OF TRACE-SAMPLING TECHNIQUES FOR MULTI-MEGABYTE CACHES

by

R. E. Kessler, Mark D. Hill and David A. Wood

Computer Sciences Technical Report #1048

September 1991

A Comparison of Trace-Sampling Techniques for Multi-Megabyte Caches[1]

R. E. Kessler, Mark D. Hill, and David A. Wood

University of Wisconsin
Computer Sciences Department
Madison, Wisconsin 53706
{kessler, markhill, david}@cs.wisc.edu

ABSTRACT

This paper compares the trace-sampling techniques of set sampling and time sampling. Using the multi-billion-reference traces of Borg et al., we apply both techniques to multi-megabyte caches, where sampling is most valuable. We evaluate whether either technique meets a 10% sampling goal: using ≤ 10% of the references in a trace can it estimate the trace's true misses per instruction with ≤ 10% relative error and at least 90% confidence. Our results show that set sampling meets the 10% sampling goal, while time sampling does not. We also find that cold-start bias in time samples is most effectively reduced by the technique of Wood et al. Nevertheless, overcoming cold-start bias requires the use of tens of millions consecutive references.

Index Terms - Cache memory, cache performance, cold start, computer architecture, memory systems, performance evaluation, sampling techniques, trace-driven simulation.

1. Introduction

Computer designers commonly use trace-driven simulation to evaluate alternative CPU caches [SMIT82]. But as cache sizes reach one megabyte and more, traditional trace-driven simulation requires very long traces (e.g., *billions* of references) to determine steady-state performance [BOKW90, STON90]. But long traces are expensive to obtain, store, and use.

We can avoid simulating long traces by using *trace-sampling techniques.* Let the cache performance of a small portion of the trace be an *observation* and a collection of observations be a *sample*. Sampling theory tells

[1] R. E. Kessler was supported in part by a summer internship at Digital Equipment Corporation and graduate fellowships from the National Science Foundation and the University of Wisconsin Alumni Research Foundation. He is now employed by Cray Research, Inc. Mark D. Hill is supported in part by the National Science Foundation (MIPS-8957278 and CCR-8902536), A.T.& T. Bell Laboratories, Cray Research Foundation and Digital Equipment Corporation. David A. Wood is supported in part by the National Science Foundation (CCR-9157366) and the University of Wisconsin Graduate School.

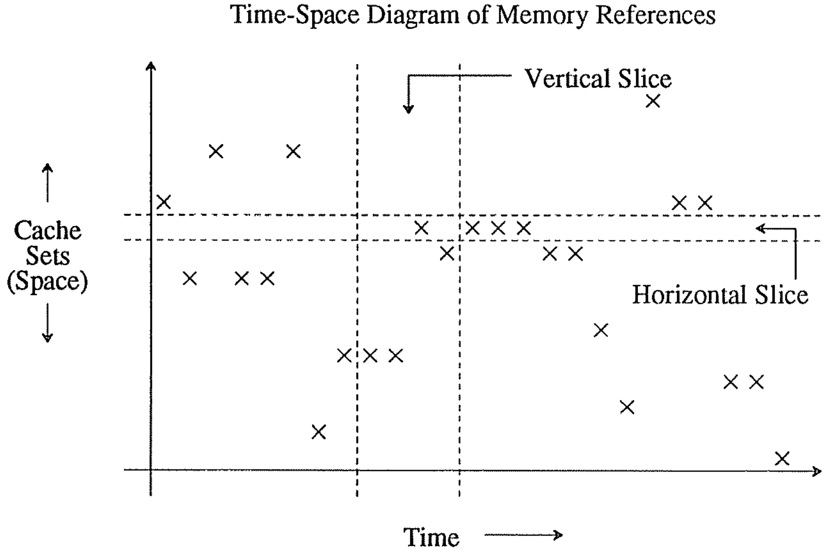


Figure 1. Sampling as Vertical and Horizontal Time-Space Slices.

This figure shows a time-space diagram of a simulation with a very short trace. The time (position within the trace) and cache set of each reference is marked with an ×. An observation in set sampling is the cache performance of one set. References that determine a single set's performance appear in an horizontal slice of this figure. An observation in time sampling is the cache performance of an interval of consecutive references. These references appear in a vertical slice of this figure.

how to predict cache performance of the full trace, given a random sample of unbiased observations [MIFJ90]. With additional assumptions, we can also estimate how far the true value is likely to be from the estimate.

Two important trace-sampling techniques are *set sampling* [HEIS90,PUZA85] and *time sampling* [LAPI88,LAHA88]. An observation in set sampling is the cache performance for the references to a single set (depicted as a horizontal slice in Figure 1), while an observation in time sampling is the cache performance of the references in a single time-contiguous trace interval (a vertical slice in Figure 1)[2].

This study is the first to compare set sampling and time sampling. Using billion-reference traces of large workloads that include multiprogramming but not operating system references [BOKW90], we examine how well these methods predict mean misses per instruction (MPI) for multi-megabyte caches. We say a sampling method is effective if it meets the following goal:

[2] Laha et al. [LAPI88] and Wood et al. [WOHK91] referred to an observation of references in a time-contiguous interval as a "sample". We use *sample* to refer to a collection of observations to be consistent with statistics terminology [MIFJ90].

Definition 1: 10% Sampling Goal

A sampling method meets the *10% sampling goal* if using ≤ 10% of the references in a trace it estimates the trace's true MPI with ≤ 10% relative error and at least 90% confidence.

For set-sampling we find several results. First, calculating the MPI for a sample using instruction fetches to all sets is much more accurate than using only instruction fetches to the sampled sets. Second, instead of selecting the sets in a sample at random, selecting sets that share several index bit values reduces simulation time, facilitates the simulation of cache hierarchies, and still accurately predicts the trace's MPI. Third, and most important, set sampling is effective. For our traces and caches, it typically meets the 10% sampling goal.

For time-sampling, we first compare techniques for overcoming cold-start bias [EASF78], i.e., determining the MPI for a particular trace interval without knowing the initial cache state. We consider leaving the cold-start bias unchanged, recording metrics only during the second half of each interval, recording metrics only for initialized sets [LAPI88, STON90], stitching intervals together [AGHH88], and Wood et al.'s model for predicting the initialization reference miss ratio [WOHK91]. We obtain two results. First, on average, the technique of Wood et al. minimizes the cold-start bias better than the other techniques. Second, for the multi-megabyte caches we studied, interval lengths of tens of millions of instructions and larger are needed to reduce the effects of cold-start.

Then using Wood et al.'s technique to mitigate cold-start bias, we show that time sampling fails to meet the 10% sampling goal, because: (1) many intervals are needed to capture workload variation, and (2) long intervals are necessary to overcome cold-start bias. Thus, for these traces and caches, set sampling is more effective than time sampling for estimating MPI. Time sampling will still be preferred, however, for caches with time-dependent behavior (e.g., prefetching) or interactions between sets (e.g., a single write buffer).

We do not consider other (non-sampling) techniques that reduce trace data storage, such as, Mache [SAMP89], stack deletion and snapshot method [SMIT77], trace (tape) stripping [PUZA85, WANB90], or exploiting spatial locality [AGAH90]. These techniques can be used in addition to the sampling considered in this study. We also do not consider Przybylski's prefix technique [PRZY88], which prepends all previously-referenced unique addresses to each time-observation. This method seems unattractive for multi-megabyte caches where each time-observation requires its own prefix and each prefix must be very large for programs that can exercise multi-megabyte caches.

Section 2 describes our methods. Section 3 and 4 examine set sampling and time sampling, respectively. Finally, Section 5 summarizes our results.

2. Methodology

This section describes the traces, cache configurations, and performance metric we use in later sections.

2.1. The Traces

The traces used in the study were collected at DEC Western Research Laboratory (WRL) [BoKL89, BoKW90] on a DEC WRL Titan [Niel86], a load/store ("RISC") architecture. Each trace consists of the execution of three to six *billion* instructions of large workloads, including multiprogramming but not operating system references. The traces reference from eight to over one hundred megabytes of unique memory locations. These traces are sufficiently long to overcome the cold-start intervals of even the large caches considered in this study. We chose programs with large memory requirements since we predict large application sizes will be more common as main memories of hundreds of megabytes become available.

The traces of the multiprogrammed workloads represent the actual execution interleaving of the processes on the traced system. The **Mult2** trace includes a series of compiles, a printed circuit board router, a VLSI design rule checker, and a series of simple programs commonly found on UNIX[3] systems, all executing in parallel (about 40 megabytes active at any time) with an average of 134,000 instructions executed between each process switch. The **Mult2.2** trace is the Mult2 workload with a switch interval of 214,000 instructions. The **Mult1** trace includes the processes in the Mult2 trace plus an execution of the system loader (the last phase of compilation) and a Scheme (Lisp variant) program (75 megabytes active) and has a switch interval of 138,000 instructions. The **Mult1.2** trace is the Mult1 workload with a switch interval of 195,000 instructions. The **Tv** trace is of a VLSI timing verifier (96 megabytes). **Sor** is a uniprocessor successive-over-relaxation algorithm that uses large, sparse matrices (62 megabytes). **Tree** is a Scheme program that searches a large tree data structure (64 megabytes). **Lin** is a power supply analyzer that uses sparse matrices (57 megabytes).

2.2. Cache Configuration Assumptions

This study focuses on multi-megabyte unified (mixed) caches, where we expect trace sampling to be most useful. We vary the size and set-associativity of these caches over a range of 1-megabyte to 16-megabytes and direct-mapped to four-way. The caches do no prefetching, use write-back and write-allocate policies, and have 128-byte blocks. The non-direct-mapped caches use a random replacement policy. We do not expect the

[3] Trademark AT&T Bell Laboratories.

replacement policy to affect sampling accuracy since, for example, least-recently-used replacement eliminates at most 15% of the cache misses for these caches [KESS91]. The caches use virtual-indexing with PID-hashing, an approximation to real-indexing[4]. We also examined a several real-indexed caches and found that they produced results similar to those in this paper, which is not surprising since real-indexed cache performance is often close to virtual-indexed cache performance.

Since multi-megabyte caches are likely to be used in a cache hierarchy, we simulate them as alternative secondary caches placed behind a fixed primary cache configuration. The primary caches are split (separate) instruction and data caches that are 32-kilobytes each, direct-mapped, 32-byte blocks, do no prefetching, use virtual indexing, and write-back and write-allocate policies. We do not evaluate primary cache tradeoffs in this study since secondary cache performance is unaffected by the primary caches when their sizes differ by at least a factor of eight [PRHH89].

2.3. The Performance Metric: Misses Per Instruction

We measure cache performance with *misses per instruction* (MPI) rather than *miss ratio*[5]. Since we only use MPI to compare the performance of alternative unified secondary caches, MPI is equivalent to Przybylski's *global miss ratio* [PRHH89]. Specifically, a cache's MPI is equal to its global miss ratio times the average number of processor references (instruction fetches and data references) per instruction.

3. Set Sampling

We first examine set sampling, where an observation is the MPI of a single set and a sample is a collection of single-set observations. Section 3.1 discusses how to compute a set sample's MPI and why it should not contain random sets, while Section 3.2 examines how well set sampling predicts MPI_{long}, the MPI of a full trace.

[4] Caches that use virtual-indexing select the set of reference using the reference's virtual address, while those that use real-indexing select with the real address. PID-hashing means that we exclusive-or the upper eight index bits from the virtual address with the process identifier (PID) of the currently executing process.

[5] MPI is better than miss ratio for comparing the performance contributions of several caches in a system (e.g., instruction, data, secondary), because MPI implicitly factors in how often a cache is accessed. Furthermore, MPI times a cache's average miss penalty directly gives the *cycles per instruction* (CPI) lost because of that cache's misses [HENP90].

3.1. Constructing Set Samples

3.1.1. Calculating the MPI of a Sample

Consider a cache with s sets, numbered 0 to $s-1$. For each set i, let $miss_i$ and $instrn_i$ be the number of the misses and instruction fetches to set i. Let S be a sample containing n sets. We consider two ways to calculate the MPI of sample S, $\hat{MPI}_S$. The *sampled-instructions* method divides the mean misses to sets in sample S by the mean instruction fetches to sets in sample S:[6]

$$\hat{MPI}_S = \frac{\frac{1}{n}\sum_{i \in S} miss_i}{\frac{1}{n}\sum_{i \in S} instrn_i} = \frac{\sum_{i \in S} miss_i}{\sum_{i \in S} instrn_i},$$

while the *all-instructions* method divides by the mean instruction fetches to all sets:

$$\hat{MPI}_S = \frac{\frac{1}{n}\sum_{i \in S} miss_i}{\frac{1}{s}\sum_{i=0}^{s-1} instrn_i} = \frac{\sum_{i \in S} miss_i}{\frac{n}{s}\sum_{i=0}^{s-1} instrn_i}.$$

We compare the two methods by computing their coefficients of variation across all set samples $S(j)$ obtained with the *constant-bits* method, described in Section 3.1.2.:

$$CV = \frac{\sqrt{\frac{1}{M}\sum_{j=1}^{M}(\hat{MPI}_{S(j)} - MPI_{long})^2}}{MPI_{long}}, \qquad (1)$$

where M is the number of samples.

Experimental results, illustrated in Table 1, show that the all-instructions method performs much better, never having a coefficient of variation more than one-tenth the sampled-instructions method. The difference is infinite for the Sor and Lin traces because loops confine many instruction fetches to a few sets. We also investigated normalizing $miss_i$ with total references per set and data references per set [KESS91]. These methods

[6] We do not consider calculating $\hat{MPI}_S$ with $\frac{1}{n}\sum_{i \in S}\frac{miss_i}{instrn_i}$, because Puzak [PUZA85] showed estimating miss ratio with the arithmetic mean of the per-set miss ratios is inferior to dividing the misses to sampled sets by the references to sampled sets (the miss-ratio equivalent of the sampled-instructions method). For a sample containing all sets, Puzak's work also implies $\frac{1}{s}\sum_{i=0}^{s-1}\frac{miss_i}{instrn_i} \neq MPI_{long}$.

Trace	$MPI_{long} \times 1000$	Coefficient of Variation (percent)	
		all-instructions	sampled-instructions
Mult1	0.70	2.3%	35.2%
Mult1.2	0.69	1.9%	28.9%
Mult2	0.61	1.9%	24.2%
Mult2.2	0.59	1.3%	24.3%
Tv	1.88	0.6%	139.0%
Sor	7.54	0.3%	∞
Tree	0.59	6.8%	191.9%
Lin	0.09	7.6%	∞

Table 1. Accuracy of MPI Computations.

This table illustrates the accuracy of computing the full trace MPI (column two) for several traces with the all-instructions and sampled-instructions methods. The accuracy is evaluated with the coefficient of variation (Equation 1) for the MPI estimates from a 4-megabyte direct-mapped secondary cache with 16 set samples of 1/16 the full trace each. The set samples are constructed with the constant bits method described in the next section. Results show that the all-instructions method is far superior to the sampled-instructions method.

perform similarly to the sampled-instructions method and not as well as the all-instructions method.

A minor disadvantage of the all-instructions method is that when gathering the references in a sample we must also count instruction fetches to all sets. Since we believe this drawback is out-weighed by the experimental results, we will use the all-instructions method throughout this paper.

3.1.2. The Constant-Bits Method

We now examine two methods for selecting sets to form a sample. We use an example to show a disadvantage of selecting sets at random and introduce the *constant-bits* method to overcome the disadvantage.

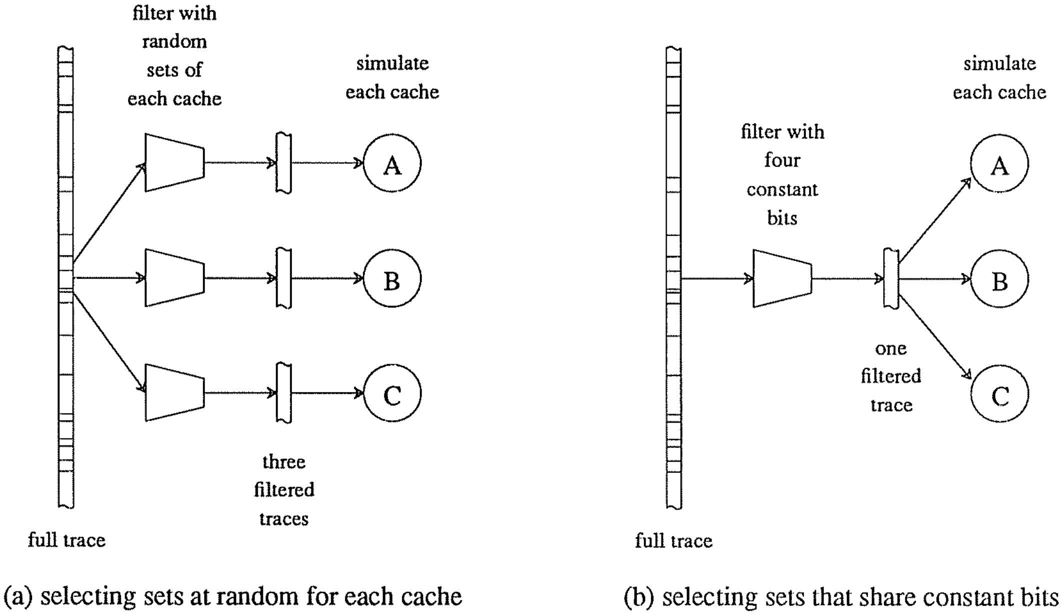
Assume that we want to evaluate three caches with samples that contain about 1/16-th the references in a full trace. Let the caches choose a reference's set with *bit selection* (i.e., the *index bits* are the least-significant address bits above the block offset) and have the following parameters:

Cache A: 32-kilobyte direct-mapped cache with 32-byte blocks (therefore its index bits are bits 14-5, assuming references are byte addresses with bit 0 being least-significant),

Cache B: 1-megabyte two-way set-associative cache with 128-byte blocks (index bits 18-7), and

Cache C: 16-megabyte direct-mapped cache with 128-byte blocks (index bits 23-7).

One method for selecting the sets in a sample is to choose them at random [PUZA85]. To evaluate cache A with references to random sets, we randomly select 64 of its 1024 sets (1/16-th), filter the full trace to extract



(a) selecting sets at random for each cache

(b) selecting sets that share constant bits

Figure 2. Two Methods for Selecting the Sets in a Sample.

This figure illustrates selecting sets for samples of three alternative caches (A, B, and C) using (a) random sets and (b) constant bits. When sets are selected at random, each simulation must be begin by filtering the full trace. With constant-bits, on the other hand, a filtered trace can drive the simulation of any cache whose index bits contain the constant bits.

references to those sets, and then simulate cache A. For cache B, we select 128 of its 2048 sets, filter and simulate. Similarly for cache C, we use 8192 of its 131072 sets. As illustrated in Figure 2a, selecting sets at random requires that each simulation begin by extracting references from the full trace. Furthermore, since primary and secondary caches usually have different sets, it is not clear how to simulate a hierarchy of cache when sets are selected at random.

We introduce a new method, called *constant-bits*, that selects references rather than sets. The constant-bits method forms a filtered trace that includes all references that have the same value in some address bits. This filtered trace can then be used to simulate any cache whose index bits include the constant bits[7] [KESS91]. For example, we can filter a trace by retaining all references that have the binary value 0000 (or one of the other 15

[7] This description assumes *bit selection*, i.e., the set-indexing bits come directly from the address of the memory access [SMIT82]. The scenario is more complicated with other than simple bit-selection cache indexing. In particular, since we use PID-hashing in this study, we ensured that the hashed index bits did not overlap with the constant bits. Note that though we use virtual-indexing, one can apply the constant-bits technique to real-indexed caches, and to hierarchical configurations with both real and virtual indexed caches if the constant bits are below the page boundary.

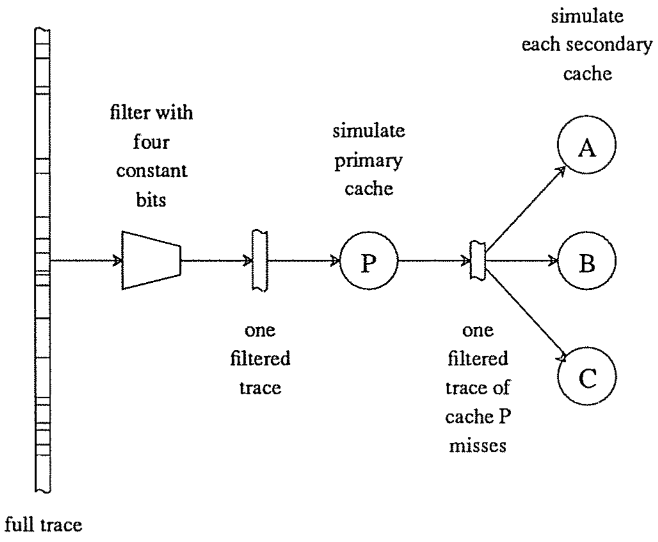


Figure 3. Using Constant-Bits Samples with a Hierarchy.
This figure illustrates how to use constant-bits samples to simulate a primary cache (P) and three alternative secondary caches (A, B and C).

values) in address bits 11-8. If the filtered trace is used with cache A, it will select all sets with binary index xxx0000xxx, where "x" is either 0 or 1. Since this index pattern has six x's, it identifies 64 (2^6) of the 1024 sets in cache A. For caches B and C, the filtered trace selects sets with indices xxxxxxx0000x and xxxxxxxxxxxx0000x, respectively. More generally, we can then use this filtered trace to select 1/16-th of the sets in any cache whose block size is 256 bytes or less and whose size divided by associativity exceeds 2 kilobytes. These include both primary caches (32-byte blocks, 32 kilobytes, direct-mapped) and all secondary caches (128-byte blocks, 1-16 megabytes, 1-4-way set-associative) considered in this paper.

Constant-bits samples have two advantages over random samples. First, as illustrated in Figure 2b, using constant-bits samples reduces simulation time by allowing a filtered trace to drive the simulations of more than one alternative cache. Second, constant-bits samples make it straightforward to simulate hierarchies of caches (when all caches index with the constant bits). As illustrated in Figure 3, we may simulate the primary cache once and then use a trace of its misses to simulate alternative secondary caches.

A potential disadvantage of constant-bits samples is they may work poorly for workloads that use their address space systematically (e.g., frequent accesses to a large, fixed stride vector). Experimental evidence, however, suggests that constant-bits sampling is effective. Figure 4 illustrates the accuracy of constant bits sampling

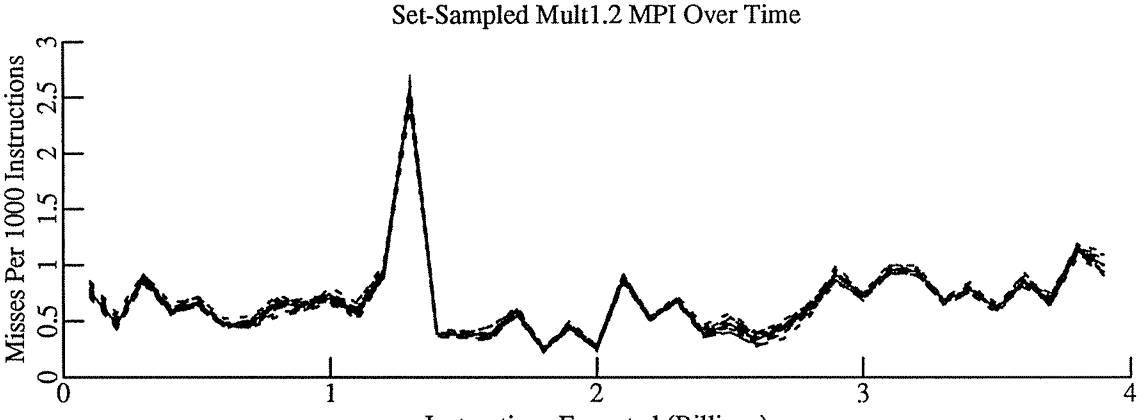


Figure 4. Set Sampling on the Mult1.2 Trace.

For every 100 million instructions, this figure shows the actual MPI's (solid line) with the predicted MPI's from each of 16 different set samples (dotted lines) for the Mult1.2 trace with a 4-megabyte direct-mapped cache. Each sample includes only references that have the same value for address bits 11-8 (i.e., bits 11-8 are the constant bits), assuming that references are byte addresses with bit 0 being least-significant Since four bits are used to select references, each of the 16 samples contains an average of 1/16-th of the trace.

for the Mult1.2 trace. For every 100 million instructions, it plots the true MPI for the interval and the MPI obtained from 16 set samples (each about 1/16 of the references of the full trace). In this example, the set samples are almost indistinguishable from the true MPI. More generally, we found constant-bits samples to be equally or more accurate than random samples with multi-megabyte caches [KESS91]. Thus, we use the constant-bits method to construct set samples throughout the rest of this paper.

3.2. What Fraction of the Full Trace is Needed?

This section examines how well set samples estimate the MPI of a full trace. For reasons discussed above, we construct samples with the constant-bits method and calculate MPI estimate for a sample with the all-instructions method. We first look at the accuracy of set sampling when MPI_{long} is known; then show how to construct confidence intervals for MPI_{long} when it is not known.

In Figure 4 we saw qualitatively that for one trace, cache, and sample size, the MPI variations between set samples and MPI_{long} were modest compared to temporal variations. Table 2 quantifies the long run error between samples and MPI_{long} for several traces, direct-mapped cache sizes, and sample sizes. We measure errors with coefficient of variation calculated using Equation 1. Table 3 gives the corresponding results for two-way set-associative caches.

Set-Sampling Coefficients of Variation (percent)

Trace	Size	$MPI_{long} \times 1000$	Fraction of Sets in Sample		
			1/4	1/16	1/64
Mult1	1M	1.55	1.7%	4.3%	N/A
	4M	0.70	1.4%	2.3%	4.8%
	16M	0.33	1.0%	1.6%	2.7%
Mult1.2	1M	1.45	0.8%	2.9%	N/A
	4M	0.69	0.9%	1.9%	4.1%
	16M	0.32	0.4%	1.5%	3.2%
Mult2	1M	1.24	0.8%	3.4%	N/A
	4M	0.61	1.0%	1.9%	2.9%
	16M	0.26	1.1%	2.3%	3.3%
Mult2.2	1M	1.18	0.4%	2.7%	N/A
	4M	0.59	0.6%	1.3%	2.5%
	16M	0.27	0.7%	1.8%	3.4%
Tv	1M	2.63	0.7%	1.9%	N/A
	4M	1.88	0.2%	0.6%	2.1%
	16M	1.03	0.5%	0.6%	2.0%
Sor	1M	14.77	0.1%	0.4%	N/A
	4M	7.54	0.1%	0.3%	0.7%
	16M	1.97	0.0%	0.0%	0.1%
Tree	1M	2.16	4.1%	5.6%	N/A
	4M	0.59	5.3%	6.8% †	13.6% †
	16M	0.30	1.8%	4.1%	6.5%
Lin	1M	1.16	0.5%	3.3%	N/A
	4M	0.09	2.0%	7.6% †	15.0% †
	16M	0.02	0.0%	0.3%	0.5%

Table 2. Set Sampling Coefficients of Variation for Direct Mapped.

This table shows the actual MPI of the full trace, MPI_{long}, for direct-mapped caches, and the coefficient of variation of the set-sampling MPI estimates, calculated using Equation 1. We construct samples with the constant-bits method. Samples containing 1/4 the sets in the cache have bits 9-8 constant. Samples for 1/16 and 1/64 use bits 11-8 and 12-7, respectively. Some entries marked "N/A" are not available, because the PID hashing overlapped with the constant bits. Except where marked with a dagger (†), at least 90% of the samples have relative errors of less than or equal to ±10%.

The key result is that, for this data and for four-way set-associative caches not shown here, set sampling generally meets the 10% sampling goal. Consider the columns labeled "1/16" in Tables 2 and 3, which correspond to samples using 1/16-th of the sets and therefore will contain less than 10% of the trace on average. Only Lin and Tree with 4-megabyte direct-mapped caches, marked with daggers, fail to have at least 90% of the samples with relative errors of less than or equal to ±10%. (And they both have only 2 of 16 samples with more than ±10% relative error.)

We also observe two other interesting trends in the data. First, reducing the fraction of sets in a sample (and hence the number of sets per sample) from 1/4 to 1/16 and from 1/16 to 1/64 increases the coefficient of variation. If the per-set MPI's were independent and identically distributed, then reducing the number of sets in a sample by

Set-Sampling Coefficients of Variation (percent)

Trace	Size	$MPI_{long} \times 1000$	Fraction of Sets in Sample		
			1/4	1/16	1/64
Mult1	1M	1.19	1.2%	2.2%	N/A
	4M	0.55	1.0%	1.7%	3.0%
	16M	0.26	0.8%	1.6%	2.3%
Mult1.2	1M	1.18	0.7%	1.6%	N/A
	4M	0.56	0.5%	1.2%	2.2%
	16M	0.28	0.5%	1.3%	2.1%
Mult2	1M	1.01	0.3%	1.9%	N/A
	4M	0.52	0.6%	1.2%	2.0%
	16M	0.24	0.9%	1.9%	3.3%
Mult2.2	1M	0.98	0.3%	1.8%	N/A
	4M	0.51	0.5%	1.5%	1.9%
	16M	0.22	0.9%	2.1%	3.5%
Tv	1M	2.31	0.2%	0.6%	N/A
	4M	1.76	0.3%	0.3%	1.6%
	16M	0.98	0.3%	0.7%	1.9%
Sor	1M	14.66	0.0%	0.3%	N/A
	4M	7.76	0.0%	0.2%	0.5%
	16M	1.92	0.0%	0.0%	0.1%
Tree	1M	1.81	2.3%	3.7%	N/A
	4M	0.49	0.8%	1.5%	3.8%
	16M	0.26	0.3%	0.4%	1.1%
Lin	1M	1.10	0.3%	2.6%	N/A
	4M	0.06	1.2%	6.0%	9.8% †
	16M	0.02	0.0%	0.3%	0.5%

Table 3. Set Sampling Coefficients of Variation for 2-Way.

This table shows the MPI of the full trace for two-way set-associative caches, and the coefficient of variation of the MPI estimates, similar to Table 2. Except where marked with a dagger (†), at least 90% of the samples have relative errors of less than or equal to ±10%.

four should double the coefficient of variation [MIFJ90, STON90]. Indeed, there is good evidence that this is the case (see, for example, the row for Mult1.2 with a 4-megabyte cache). Second, increasing associativity from direct-mapped to two-way reduces corresponding coefficients of variation by more than 50%. We conjecture that set sampling works better for two-way set-associative caches because they have fewer conflict misses than direct-mapped caches [HILS89]. A high rate of conflict misses to a few sets can make those sets poor predictors of overall behavior.

Finally, in practical applications of set sampling, we want to estimate the error of an MPI estimate, using only the information contained within the sample (i.e., not using knowledge of MPI_{long} as did Tables 2 and 3). We do this using 90% confidence intervals, calculated from the sample mean and sample standard deviation by the standard technique [MIFJ90]. Our estimate of the sample standard deviation includes a finite population correction, which is important when the sample size is a substantial fraction of the population (e.g., when each

90% Confidence Intervals that Contain MPI_{long}

	Fraction of Sets in Sample					
Trace	1/4		1/16		1/64	
	fraction	percent	fraction	percent	fraction	percent
Mult1	3/4	75%	16/16	100%	61/64	95%
Mult1.2	4/4	100%	16/16	100%	60/64	94%
Mult2	3/4	75%	15/16	94%	61/64	95%
Mult2.2	4/4	100%	16/16	100%	63/64	98%
Tv	4/4	100%	16/16	100%	51/64	78%
Sor	4/4	100%	16/16	100%	64/64	100%
Tree	2/4	50%	12/16	75%	47/64	73%
Lin	4/4	100%	16/16	100%	62/64	97%
All		89%		93%		91%

Table 4. Set-Sampling Error Prediction.

For a 4-megabyte direct-mapped secondary cache and various traces and fraction of sets, this table gives the fraction and percent of 90% confidence intervals that contained MPI_{long}. Since the percentages are near 90%, confidence intervals usefully estimate how far MPI_S is likely to be from MPI_{long}.

sample includes 1/4-th of all sets) [KESS91, MIFJ90].

For large (≥ 30 observations) random samples, sampling theory predicts 90% of the 90% confidence intervals will contain the true mean. For various constant-bits set samples and a 4-megabyte direct-mapped cache, Table 4 displays the fraction of 90% confidence intervals that actually contain MPI_{long}. Since the results in Table 4 are usually similar to 90%, the confidence interval calculation is a useful method for estimating the error of a set-sample, given information from within that sample alone.

3.3. Advantages and Disadvantages of Set Sampling

The most important advantage of set sampling is that, for our simulations, it meets the 10% sampling goal (Definition 1). A set sample automatically includes references from many execution phases, so an individual sample can accurately characterize the MPI of a full trace, including its temporal variability. The reduced trace data requirements of set sampling allow for simulation of longer traces, and therefore more algorithmic phases, in a smaller amount of time. Besides the data reduction, set sampling also reduces the memory required to simulate a cache. A set sample containing 1/16 of the full trace needs to simulate only 1/16 of the sets.

Set sampling does have its limitations. Even with the constant bits method, the full trace must be retained if one wishes to study caches that do not index with the constant bits. Furthermore, set sampling may not accurately model caches whose performance is affected by interactions between references to different sets. The effectiveness of a prefetch into one set, for example, may depend on how many references are made to other sets before the prefetched data is first used. Similarly, the performance of a cache with a write buffer may be affected by how often write buffer fills up due to a burst of writes to many sets.

4. Time Sampling

The alternative to set sampling is time sampling. Here an observation is the MPI of a sequence of time-contiguous references and is called an interval. Section 4.1 discusses determining the MPI for a sample, while Section 4.2 examines using a sample to estimate MPI for the full trace.

4.1. Reducing Cold-Start Bias in Time Samples

To significantly reduce trace storage and simulation time, we must estimate the true MPI for an interval without knowledge of *initial cache state*, i.e., the cache state at the beginning of the interval. This problem is simply the well-known *cold-start problem* applied to each interval [EASF78]. Below we examine how well the following five techniques mitigate the effect of the cold-start problem in multi-megabyte caches.

COLD COLD assumes that the initial cache state is empty. While this assumption does not affect misses to full sets or hits to any set, it causes COLD to overestimate MPI, because references that appear to miss to non-full sets may or may not be misses when simulated with the (true) initial cache state. These potential misses are often called *cold-start* misses [EASF78].

HALF HALF uses the first half of the instructions in an interval to (partially) initialize the cache, and estimates MPI with the remaining instructions.

PRIME PRIME estimates MPI with references to ''initialized'' sets. A set in a direct-mapped cache is initialized once it is filled [STON90], while a set in a set-associative cache is initialized after it is filled and a non-most-recently-used block has been referenced [LAPI88].

STITCH STITCH approximates the cache state at the beginning of an interval with the cache state at the end of the previous interval [AGHH88]. Thus one creates a trace for a sample by *stitching* it's intervals together.

INITMR Like COLD, INITMR simulates an interval beginning with an empty initial cache state. Instead of assuming that all cold-start misses miss, however, INITMR uses Wood et al.'s $\hat{\mu}_{split}$ to estimate the fraction of cold-start misses that would have missed if the initial cache state was known [WOHK91]. The estimate is based on (1) the fraction of time that a cache block frame holds a block that will not be referenced before it is replaced, and (2) the fraction of the cache loaded during the cold-start simulation of an interval. When we could not estimate (1) with the references in an interval, we assume it to be 0.7.

For a particular trace and cache, we evaluate a cold-start technique as follows. We select the number of instructions in an interval, called the *interval length*, and collect a sample S of n=30 intervals spaced equally in the trace. We use the cold-start technique to estimate the MPI for each interval, $\hat{mpi}_i$, and calculate an MPI estimate for sample S with[8]:

$$\hat{MPI}_S = \frac{1}{n} \sum_{i=1}^{n} \hat{mpi}_i .$$

Since we have the full trace, we can simulate each interval with its initial cache state to determine the interval's true MPI, mpi_i, and calculate the true MPI for the sample, MPI_S, with $\frac{1}{n} \sum_{i=1}^{n} mpi_i$. We evaluate how well a technique reduces cold-start bias in a sample S with[9]:

$$BIAS_S = \frac{\hat{MPI}_S - MPI_S}{MPI_S}.$$

It is important to note that MPI_S is not the same as MPI_{long}. In Section 4.2, we will examine how well a time sample predicts the full trace MPI; here we seek to mitigate the cold-start bias of $\hat{MPI}_S$.

We evaluate $BIAS_S$ for five cold-start techniques, eight traces, four interval lengths (100 thousand, 1 million, 10 million, and 100 million instructions), three cache sizes (1, 4, and 16 megabytes) and two associativities (direct-mapped and four-way). Since space precludes us from displaying 192 cases for each cold-start technique, we present several subsets of the data.

For a 10-million-instruction interval length, Tables 5 and 6 display $BIAS_S$ for direct-mapped and four-way set-associative caches, respectively. The data show several trends. First, COLD, HALF and STITCH tend to overestimate MPI_S. COLD does so because it assumes that all cold-start misses miss. Similarly, HALF tends to

[8] Since with time sampling each interval has the same number of instructions, it is meaningful to compute $\hat{MPI}_S$ with the arithmetic mean of the $\hat{mpi}_i$'s.

[9] We calculate $BIAS_S$ for PRIME with the secondary cache's local miss ratio rather than MPI, because counting the number of instructions is not straightforward when some sets are initialized but others are not. Since $BIAS_S$ is a relative error, we expect that calculating it with local miss ratio will be comparable to calculating it with MPI.

Trace	Cache Size	$MPI_S \times 1000$	COLD	HALF	PRIME	STITCH	INITMR
Mult1	1M	1.45	+18%	+5%	-18%	+23%	+0%
	4M	0.62	+77%	+27%	-50%	+52%	-11%
	16M	0.28	+233%	+114%	-80%	+131%	-12%
Mult1.2	1M	1.57	+16%	+2%	-18%	+2%	+2%
	4M	0.77	+66%	+25%	-51%	+27%	-5%
	16M	0.37	+200%	+103%	-80%	+90%	-3%
Mult2	1M	1.21	+18%	+2%	-26%	+23%	-3%
	4M	0.60	+70%	+31%	-62%	+53%	-24%
	16M	0.25	+264%	+168%	-85%	+147%	-9%
Mult2.2	1M	1.18	+19%	+15%	-27%	+29%	-1%
	4M	0.62	+71%	+50%	-61%	+56%	-13%
	16M	0.29	+233%	+180%	-84%	+141%	-3%
Tv	1M	2.55	+4%	-0%	-33%	+32%	-2%
	4M	1.76	+15%	+9%	-56%	+37%	-4%
	16M	0.95	+79%	+61%	-76%	+71%	+37%
Sor	1M	15.68	+0%	-0%	-5%	-11%	-0%
	4M	8.08	+18%	+2%	-18%	-8%	+6%
	16M	2.00	+190%	+60%	-76%	-8%	+114%
Tree	1M	2.00	+13%	-0%	-10%	+29%	-1%
	4M	0.51	+107%	+8%	-50%	+43%	+24%
	16M	0.30	+217%	+35%	-77%	+69%	+18%
Lin	1M	0.75	+20%	+7%	-29%	-0%	+16%
	4M	0.06	+1113%	+535%	-62%	+217%	+903%
	16M	0.01	+4648%	+2248%	---%	+873%	+1037%

Table 5. Bias of Cold-Start Techniques With Direct-Mapped Caches.

This table displays $BIAS_S$ for five cold-start techniques, eight traces, interval length of 10 million instructions, three direct-mapped cache sizes (1, 4, and 16 megabytes).

overestimate MPI_S when the first half of the trace does not sufficiently fill the cache. HALF can underestimate the sample's MPI, however, when the second half of most of a sample's intervals have a lower MPI than the whole of each interval. We believe STITCH overestimates MPI_S, because (due to temporal locality) references are less likely to miss when simulated with an interval's true initial state than with the final state from the previous interval [Wood90]. Second, PRIME underestimates MPI_S for direct-mapped caches. PRIME calculates MPI_S by effectively assuming that cold-start misses are as likely to miss as any other reference. Wood et al. [WoHK91] have shown, however, that this assumption is false, and that cold-start misses are much more likely to miss than randomly-chosen references. PRIME is more accurate for four-way set-associative caches, where the heuristic of ignoring initial references to a most-recently-referenced block mitigates the underestimation. Third, INITMR did not consistently underestimate or overestimate MPI_S. Finally, the large biases for the Lin trace with 4- and 16-megabyte caches are probably not important, because the true MPI's are so small.

Trace	Cache Size	$MPI_S \times 1000$	COLD	HALF	PRIME	STITCH	INITMR
Mult1	1M	0.94	+21%	-5%	-6%	+36%	-11%
	4M	0.44	+106%	+29%	-51%	+80%	-4%
	16M	0.22	+313%	+157%	-99%	+167%	-8%
Mult1.2	1M	1.20	+15%	-5%	-9%	+6%	-7%
	4M	0.60	+81%	+21%	-40%	+43%	+1%
	16M	0.32	+232%	+118%	-57%	+104%	-3%
Mult2	1M	0.92	+14%	-5%	-18%	+33%	-16%
	4M	0.49	+84%	+34%	-64%	+68%	+2%
	16M	0.22	+316%	+202%	-78%	+170%	-9%
Mult2.2	1M	0.96	+16%	+10%	-14%	+38%	-10%
	4M	0.52	+84%	+54%	-52%	+73%	-1%
	16M	0.25	+285%	+221%	+15%	-161%	-14%
Tv	1M	2.14	+4%	-2%	-22%	+32%	-2%
	4M	1.53	+14%	+6%	+12%	+39%	-8%
	16M	0.82	+99%	+75%	+195%	+87%	+32%
Sor	1M	15.46	+0%	-0%	+0%	-11%	-0%
	4M	8.57	+9%	-1%	-12%	-8%	-2%
	16M	2.17	+158%	+34%	-81%	-4%	+60%
Tree	1M	1.60	+11%	-3%	-9%	+35%	-6%
	4M	0.41	+124%	-5%	-32%	+70%	+18%
	16M	0.25	+263%	+38%	+83%	+77%	-17%
Lin	1M	0.69	+26%	+6%	+9%	+6%	+21%
	4M	0.02	+2763%	+1322%	+81%	+778%	+1797%
	16M	0.01	+4648%	+2248%	---%	+873%	+1037%

Table 6. Bias of Cold-Start Techniques With Four-Way Set-Associativity.

This table displays $BIAS_S$ for five cold-start techniques, eight traces, interval length of 10 million instructions, three four-way set-associative cache sizes (1, 4, and 16 megabytes).

Table 7 addresses which cold-start technique is best. For each the five cold-start techniques, we compute $Bias_S$ for all 192 cases. We award a point in the "10%" category for biases less than ±10% and award one in the "Win" category for the cold-start technique closest to being unbiased. Multiple points are awarded in the case of ties. The final row of Table 7 gives totals. HALF and INITMR have twice the "10%" score of the other approaches, while INITMR has more "Wins" than all the other approaches combined. While HALF performs well in many cases, INITMR performs best overall.

Table 8 illustrates how well INITMR performs with three direct-mapped caches (1, 4, and 16 megabytes) and all four interval lengths (100,000, 1,000,000, 10,000,000, and 100,000,000 instructions). As expected, it reduces bias more effectively as the interval lengths get longer or cache size gets smaller, because cold-start becomes less dominant. The most striking aspect of this data is that INITMR, the best method, still performs terribly for intervals containing 100,000 and 1,000,000 instructions. This should not be not surprising, since the number of block frames in the caches (e.g., 8192 for 1-megabyte caches) far exceeds the number of true misses in

Cache	Interval	COLD		HALF		PRIME		STITCH		INITMR	
Size	Length (Mill)	10%	Win	10%	Win	10%	Win	10%	Win	10%	Win
1M	0.1	2	0	2	5	0	2	0	0	0	9
	1	2	1	4	4	3	5	1	2	3	5
	10	4	2	15	13	7	1	4	3	12	8
	100	16	5	16	7	16	6	6	2	16	12
4M	0.1	0	0	0	0	0	1	1	2	1	13
	1	0	0	0	1	1	2	2	1	4	13
	10	1	0	6	6	0	1	2	1	10	8
	100	7	1	14	4	3	2	5	3	12	7
16M	0.1	0	0	0	0	0	0	0	0	0	16
	1	0	0	0	0	0	0	1	2	0	14
	10	0	0	0	0	0	1	2	4	6	11
	100	0	0	3	2	1	0	5	9	5	5
All	0.1	2	0	2	5	0	3	1	2	1	38
	1	2	1	4	5	4	7	5	5	7	32
	10	5	2	21	19	7	3	8	8	28	27
	100	23	6	33	13	20	8	16	14	33	24
All	All	32	9	60	42	31	21	29	29	69	121

Table 7. Scoring of Different Cold-Start Techniques.

This table displays scores of the cold-start techniques for 192 cases: the eight traces, four interval lengths (100 thousand, 1 million, 10 million, and 100 million instructions), three cache sizes (1, 4, and 16 megabytes) and two associativities (direct-mapped and four-way). We award a point in the "10%" category if $-10\% \le Bias_S \le 10\%$ and award one in the "Win" category for the cold-start technique closest to being unbiased ($\log |Bias_S|$ closest to zero). Multiple points are awarded in the case of ties.

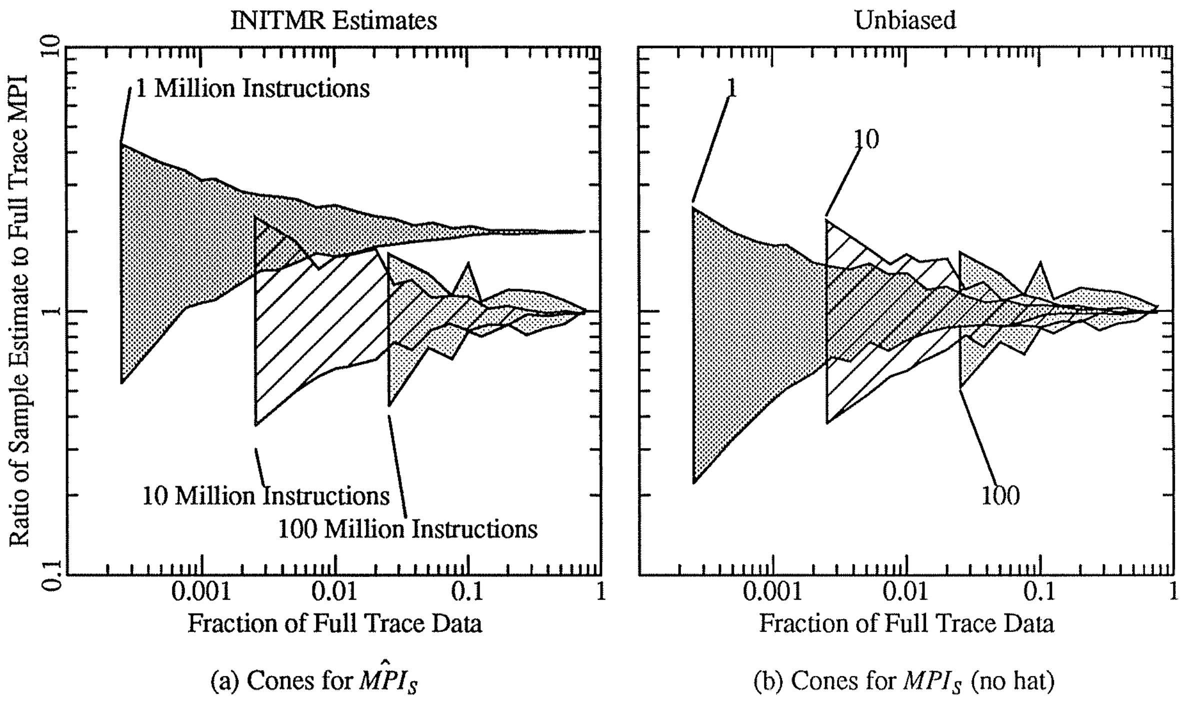
these intervals (e.g., 1550 equals 1,000,000 instructions times a 0.00155 MPI for Mult1). Furthermore, it appears that INITMR does not adequately mitigate cold-start bias unless interval lengths are, at least, 10 million instructions for 1-megabyte caches, 100 million instructions for 4-megabyte caches, and more than 100 million instructions for 16-megabyte caches. These results are consistent with the rule-of-thumb that trace length should be increased by a factor of eight each time the cache size quadruples [STON90].

As Table 8 also illustrates, however, we can determine when INITMR is likely to perform well. We marked each entry in the table with an asterisk ("*") if, on average, the interval length was sufficient to (a) fill at least half the cache and (b) there were at least as many misses to full sets as cold-start misses. All values $Bias_S$ marked with an asterisk are less than ±10%. Nevertheless, they imply that for multi-megabyte caches each interval should contain more instructions than have previously been present in many "full" traces.

Trace	Cache Size	$MPI_{long} \times 1000$	Interval Length (Millions of Instructions)			
			0.1	1	10	100
Mult1	1M	1.55	86%	47%	0%*	0%*
	4M	0.70	156%	120%	-11%	-3%*
	16M	0.33	281%	335%	-12%	-17%
Mult1.2	1M	1.45	103%	21%	2%*	0%*
	4M	0.69	123%	63%	-5%	-2%*
	16M	0.32	400%	100%	-3%	-17%
Mult2	1M	1.24	49%	20%	-3%*	0%*
	4M	0.61	48%	39%	-24%	0%*
	16M	0.26	212%	146%	-9%	-3%
Mult2.2	1M	1.18	127%	24%	-1%*	0%*
	4M	0.59	127%	60%	-13%	0%*
	16M	0.27	170%	106%	-3%	8%
Tv	1M	2.63	36%	-10%	-2%*	0%*
	4M	1.88	34%	-9%	-4%	0%*
	16M	1.03	145%	39%	37%	12%
Sor	1M	14.77	-41%	-3%*	0%*	0%*
	4M	7.54	-27%	44%	6%*	0%*
	16M	1.97	83%	386%	114%	-2%*
Tree	1M	2.16	249%	36%	-1%*	0%*
	4M	0.59	1407%	121%	24%	-7%*
	16M	0.30	796%	198%	18%	-37%
Lin	1M	1.16	-30%	-14%	16%	1%*
	4M	0.09	1437%	946%	903%	113%
	16M	0.02	2567%	1318%	1037%	176%

Table 8. Accuracy of INITMR Time-Sample MPI Estimates.

This table displays $BIAS_S$ for INITMR with eight traces, four interval lengths, three direct-mapped cache sizes (1, 4, and 16 megabytes). We mark entries with an asterisk ("*") if, on average, interval lengths are sufficient to (a) fill at least half the cache and (b) there are at least as many misses to full sets as cold-start misses.



(a) Cones for $\hat{MPI}_S$

(b) Cones for MPI_S (no hat)

Figure 5. Cones for Time Sampling with Mult1.2.
This figure displays cones for $\hat{MPI}_S$ (left) and MPI_S (right) for the Mult1.2 trace and a 4-megabyte direct-mapped cache. For an interval length and sample size (whose product gives the fraction of the trace used) the height of a cone displays the range of the middle 90% of estimates from many samples.

4.2. What Fraction of the Full Trace is Needed?

This section examines how accurately time samples estimate MPI_{long}, the MPI of the full trace. We estimate the MPI of a sample S, $\hat{MPI}_S$, with the arithmetic mean of MPI estimates for each interval in the sample, where we use INITMR to reduce cold-start bias of each interval.

Figure 5a illustrates how we summarize the data[10]. For the Mult1.2 traces and a 4-megabyte direct-mapped cache, it plots $\hat{MPI}_S/MPI_{long}$ on the logarithmic y-axis and the fraction of the full trace contained in the sample on the logarithmic x-axis. Consider the cone at the far left. We use 3000 1-million-instruction intervals to calculate its shape. The left edge, near 0.00025, gives the fraction of the trace used in a sample of one interval. We determine the end-points of the left edge with the empirical distribution of $\hat{MPI}_S$ for single-interval samples. The upper end-point gives the 95-th percentile, while the lower gives the 5-th percentile. Thus, the length of the left edge is

[10] We use a visual display here instead of coefficient of variation, because we believe it provides more insight. We did not use a visual display with set sampling, because we did not have enough samples to smooth the data.

the range of the middle 90% of the $\hat{MPI}_S$'s. We compute other vertical slices similarly. A vertical line (not shown) in the same cone at 0.01 (40 × 0.00025), for example, gives the range of the middle 90% of the MPI_S's for samples of 40 intervals each. The other two cones are for interval lengths of 10 million instructions (300 intervals) and 100 million instructions (30 intervals). The right graph gives similar data for MPI_S, where we calculate the MPI of each interval with its true initial cache state.

A time sample would meet the 10% sampling goal (Definition 1) if the sample's size times the length of each interval were less than 10% of the trace (e.g., to the left of x-axis value 0.1 in Figure 5a), the lower point on the appropriate cone falls between above 0.9 and 1.1 (on the y-axis) Unfortunately, none of the three cones for Mult1.2 qualify. The cone for 1-million-instruction intervals is narrow enough but biased too far above 1.0, while the cones of 10 million and 100 million instructions are too wide.

We found similar results for the rest of the traces, displayed in Figures 6a and 6b. The cones for the multiprogrammed traces are similar to those of Mult1.2, although Mult2 and Mult2.2 have more cold-start bias. The cones for the single applications, Tree, Tv, Sor and Lin, are more idiosyncratic, reflecting application-specific behavior. The cones of Sor, for example, are skewed by Sor's behavior of alternating between low and high MPI (with a period of around 300 million instructions [BoKW90]). For these traces and caches (and for direct-mapped and four-way, 1- and 16-megabyte caches [Kess91]), time sampling fails to meet the 10% sampling goal.

Nevertheless, this data provides several insights into time sampling. First, the cones for MPI_S (Figure 5b) are vertically centered on 1.0 and have a shape similar to those of $\hat{MPI}_S$ (left). This data and data for other traces (not shown) suggest that MPI_S and $\hat{MPI}_S$ have different means but similar distributions. Therefore, it appears that looking for better ways of mitigating cold-start bias in an interval (or sample) can be decoupled from examining how well samples tend to predict MPI_{long}.

Second, the height of the cones tends to vary as one over the square root of the sample size (number of intervals per sample). This suggests that $\hat{mpi}_i$'s are behaving as independent and identically distributed random variables [MiFJ90].

Third, even if we eliminate cold-start bias, accurate estimates of MPI_{long} must use hundred of millions of instructions to capture temporal workload variations. With Mult1.2 and a 4-megabyte direct-mapped cache, Figure 5b shows that MPI_S is within 10% of MPI_{long} (for 90% of the samples examined) only with samples of 200 intervals of length 1 million instructions, 65 10-million-instruction intervals, or 20 100-million-instruction intervals[11]. This is roughly a factor of three decrease in sample size as interval length is multiplied by ten.

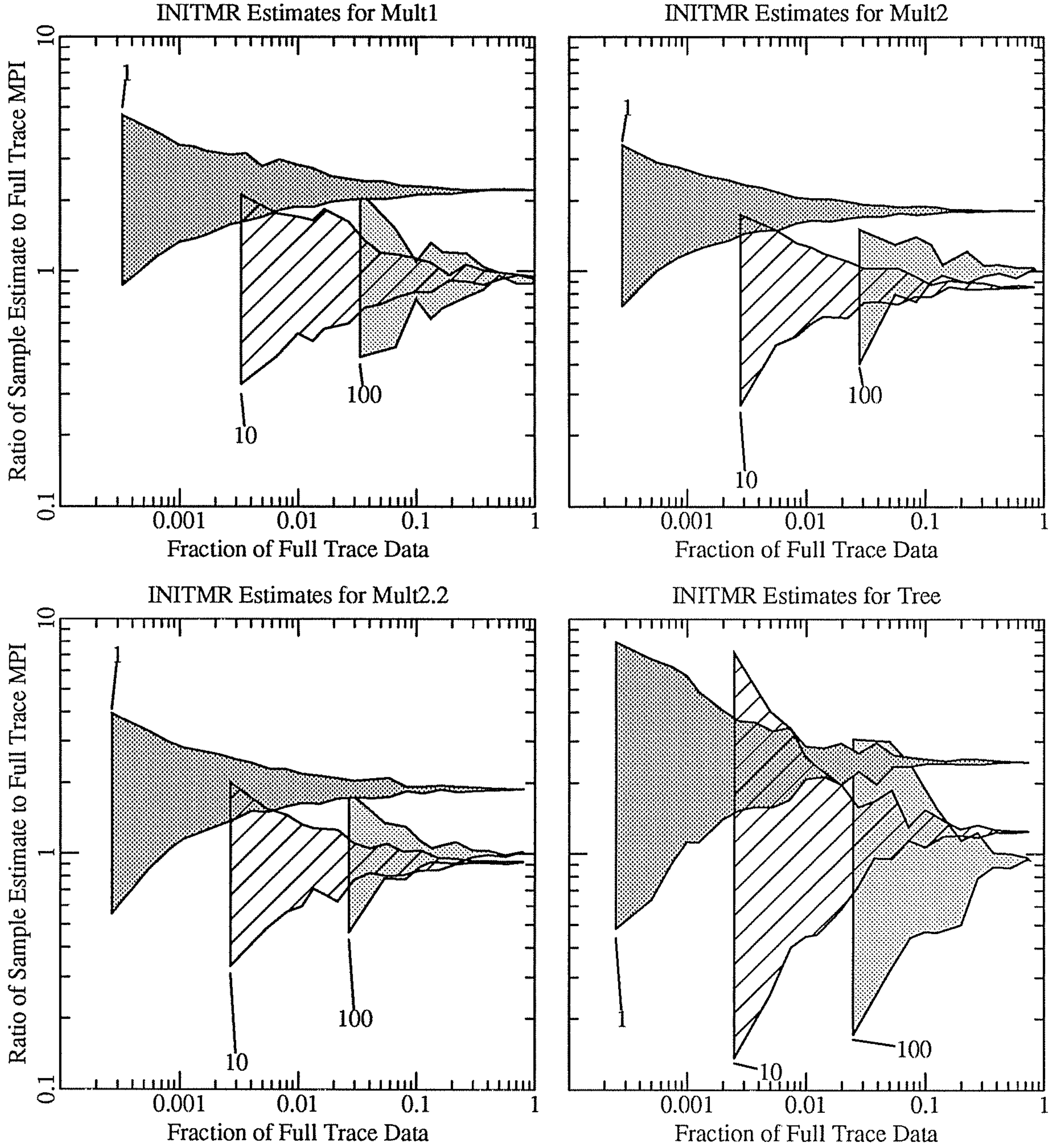
[11] For much smaller caches, Laha et al. found a sample size of 35 intervals to be sufficient [LaPI88].

Finally, we investigate whether the error in $\hat{MPI}_S$ can be estimated from information within the sample itself. We calculate 90% confidence intervals [MIFJ90] and then investigate whether they contain the true mean approximately 90% of the time. In most cases, however, the 90% confidence intervals do not contain MPI_{long} 90% of the time, because cold-start bias (that was not removed by INITMR) prevents the distribution of $\hat{MPI}_S$ from being centered on MPI_{long}. Furthermore, the confidence intervals provide no information on the magnitude of cold-start bias. Confidence intervals did work in a few cases where samples contained 30 or more intervals and interval lengths were long enough to make cold-start bias negligible [KESS91]. These cases, however, failed to meet the 10% sampling goal because the samples contained much more than 10% of the trace. Confidence intervals also worked for MPI_S (whose expected value is MPI_{long} because it has no cold-start bias), when samples contain at least 30 intervals.

4.3. Advantages and Disadvantages of Time Sampling

The major advantage of time sampling is that it is the only sampling technique available for caches with timing-dependent behavior (e.g., that prefetch or are lockup-free [KROF81]) or shared structures across sets (e.g., write buffers or victim caching [JOUP90]). Furthermore, the cold-start techniques for time sampling can be applied to any full-trace simulation, since a ''full'' trace is just a long observation from a system's workload.

However, in these simulations, time sampling fails to meet the 10% sampling goal for multi-megabyte caches, because it needed long intervals to mitigate cold-start bias and many intervals to capture temporal workload variation. These results suggest that unless researchers develop better cold-start techniques, set sampling is more effective than time sampling at estimating the MPI of multi-megabyte caches.



INITMR Estimates for Mult1

INITMR Estimates for Mult2

INITMR Estimates for Mult2.2

INITMR Estimates for Tree

Figure 6a. Cones for Time Sampling with Mult1, Mult2, Mult2.2, and Tree.
Similar to Figure 5a, these figures display cones for $\widehat{MPI}_S$ with the Mult1, Mult2, Mult2.2, and Tree traces.

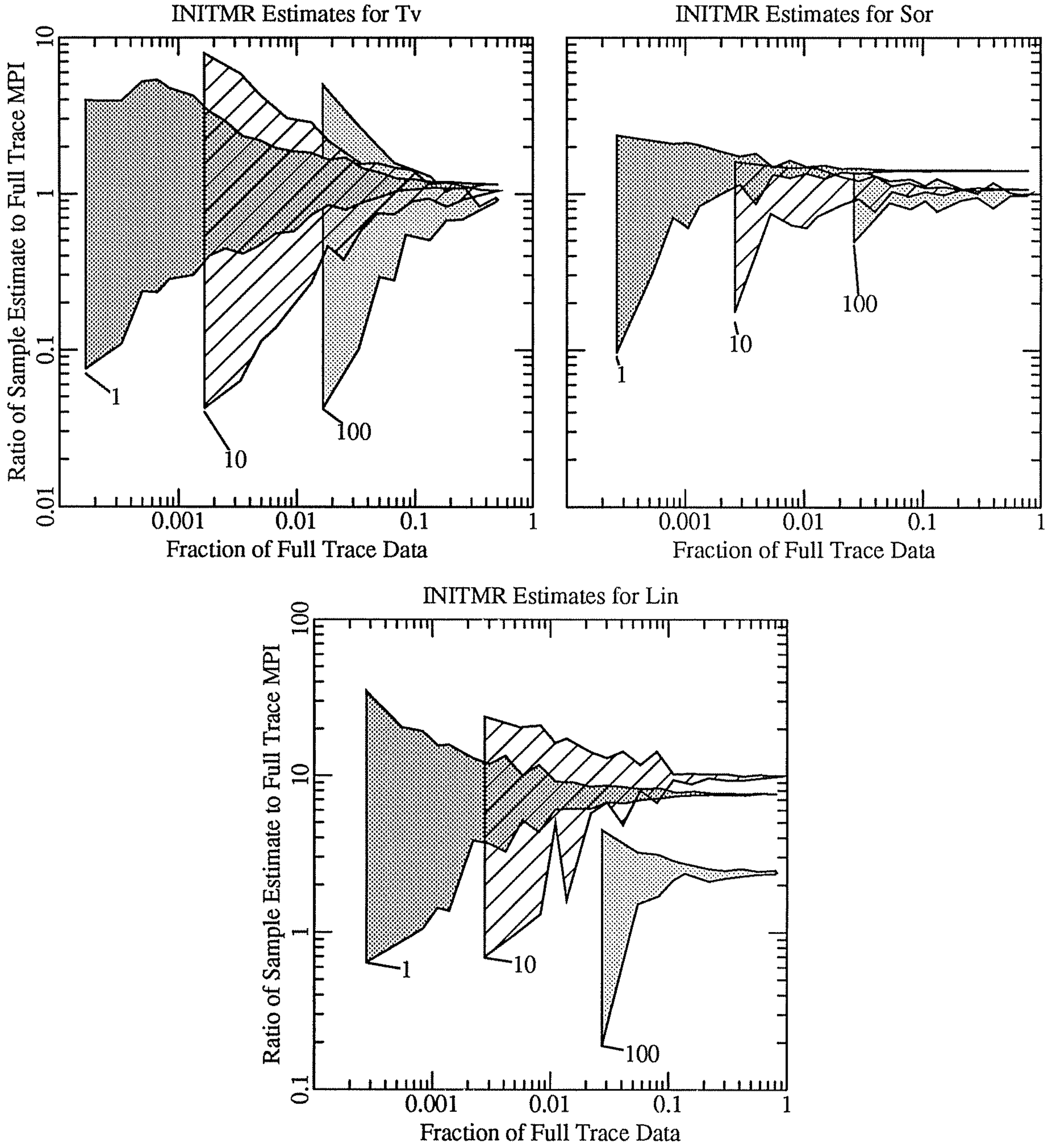


Figure 6b. Cones for Time Sampling with Tv, Sor, and Lin.

Similar to Figure 5a, these figures display cones for $\hat{MPI}_S$ with the Tv, Sor, and Lin traces. Note that Lin uses a different y-axis scale.

5. Conclusions

A straightforward application of trace-driven simulation to multi-megabyte caches requires very long traces that strain computing resources. Resource demands can be greatly reduced using set sampling or time sampling. Set sampling estimates cache performance using information from a collection of sets, while time sampling uses information from a collection of trace intervals. This study is the first to apply both techniques to large caches, where they are most useful. We use billion-reference traces of large workloads that include multiprogramming but not operating system references [BoKW90]

For set sampling we obtained several results. First, calculating the MPI (misses per instruction) for a sample using the number of instruction fetches to all sets is much more accurate than using only the number of instruction fetches in the sample. Second, constructing samples from sets that share some common index bit values works well, since such samples can be used to accurately predict the MPI of multiple alternative caches and caches in hierarchies. Third, sets behave sufficiently close to normal that confidence intervals are meaningful and accurate. Last and most important, set sampling meets the 10% sampling goal: using ≤ 10% of the references in a trace it estimates the trace's true MPI with ≤ 10% relative error and at least 90% confidence.

Results for time sampling include the following. First, Wood et al.'s $\hat{\mu}_{split}$ was the most effective technique for reducing cold-start bias, although using half the references in a trace interval to (partially) initialize a cache often performed well. Second, interval lengths must be long to mitigate cold-start bias (10 million instructions for 1-megabyte caches, 100 million instructions for 4-megabyte caches, and more than 100 million instructions for 16-megabyte caches). Third and most important, for these traces and caches, time sampling does not meet the 10% sampling goal: we needed more than 10% of a trace to get (trace) interval lengths that adequately mitigated cold-start bias and have enough intervals in a sample to make accurate predictions.

Thus, we found that for our traces, set sampling is more effective than time sampling for estimating MPI of the multi-megabyte caches. Time sampling will be preferred, however, when set sampling is not applicable, such as for caches that have time-dependent behavior (e.g., prefetching) or structures used by many sets (e.g., write buffers).

As with any experimental work, our results are sure to hold only for the specific cases examined. Nevertheless, we expect our results to extend to other similar cache configurations and to other user-mode traces from similar workloads. It is an open questions whether our results apply to traces dominated by operating system activity or radically different user-mode workloads.

6. Acknowledgments

We would like to thank The Western Research Laboratory of Digital Equipment Corporation, especially Anita Borg and David Wall, for the traces used in this study. Joel Bartlett, Renato De Leone, Jeremy Dion, Norm Jouppi, Bob Mayo, and Don Stark all were a tremendous help in providing traceable applications. Paul Vixie and Colleen Hawk helped to store the traces. Paul Beebe and the Systems Lab were able to satisfy our *enormous* computing needs. Mike Litzkow and Miron Livny adapted Condor to the requirements of these simulations. Harold Stone gave comments on an earlier version of this work, while Sarita Adve, Vikram Adve and Garth Gibson scrutinized this paper.

7. Bibliography

[AGHH88] A. AGARWAL, J. HENNESSY and M. HOROWITZ, "Cache Performance of Operating System and Multiprogramming Workloads," *ACM Transactions on Computer Systems*, vol. 6, no. 4, November 1988, pp. 393-431.

[AGAH90] A. AGARWAL and M. HUFFMAN, "Blocking: Exploiting Spatial Locality for Trace Compaction," *Proceedings of the Conference on Measurement and Modeling of Computer Systems*, 1990, pp. 48-57.

[BOKL89] A. BORG, R. E. KESSLER, G. LAZANA and D. W. WALL, "Long Address Traces from RISC Machines: Generation and Analysis," Research Report 89/14, Western Research Laboratory, Digital Equipment Corporation, Palo Alto, CA, September 1989.

[BOKW90] A. BORG, R. E. KESSLER and D. W. WALL, "Generation and Analysis of Very Long Address Traces," *Proceedings of the 17th Annual International Symposium on Computer Architecture*, 1990, pp. 270-279.

[EASF78] M. C. EASTON and R. FAGIN, "Cold-Start vs. Warm-Start Miss Ratios," *Communications of the ACM*, vol. 21, no. 10, October 1978, pp. 866-872.

[HEIS90] P. HEIDELBERGER and H. S. STONE, "Parallel Trace-Driven Cache Simulation by Time Partitioning," IBM Research Report RC 15500 (#68960), February 1990.

[HENP90] J. L. HENNESSY and D. A. PATTERSON, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann Publishers Inc., San Mateo, CA, 1990.

[HILS89] M. D. HILL and A. J. SMITH, "Evaluating Associativity in CPU Caches," *IEEE Transactions on Computers*, vol. 38, no. 12, December 1989, pp. 1612-1630.

[JOUP90] N. P. JOUPPI, "Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers," *Proceedings of the 17th Annual International Symposium on Computer Architecture*, 1990, pp. 364-373.

[KESS91] R. E. KESSLER, "Analysis of Multi-Megabyte Secondary CPU Cache Memories," Ph.D. Thesis, Computer Sciences Technical Report #1032, University of Wisconsin, Madison, WI, July 1991.

[KROF81] D. KROFT, "Lockup-Free Instruction Fetch/Prefetch Cache Organization," *Proceedings of the 8th Annual International Symposium on Computer Architecture*, 1981, pp. 81-87.

[LAPI88] S. LAHA, J. H. PATEL and R. K. IYER, "Accurate Low-Cost Methods for Performance Evaluation of Cache Memory Systems," *IEEE Transactions on Computers*, vol. 37, no. 11, November 1988, pp. 1325-1336.

[LAHA88] S. LAHA, "Accurate Low-Cost Methods for Performance Evaluation of Cache Memory Systems," Ph. D. Thesis, University of Illinois, Urbana-Champaign, Illinois, 1988.

[MIFJ90] I. MILLER, J. E. FREUND and R. JOHNSON, *Probability and Statistics for Engineers*, Prentice Hall, Inc., Englewood Cliffs, NJ 07632, Fourth Edition 1990.

[NIEL86] M. J. K. NIELSEN, "Titan System Manual," Research Report 86/1, Western Research Laboratory, Digital Equipment Corporation, Palo Alto, CA, September 1986.

[PRZY88] S. A. PRZYBYLSKI, "Performance-Directed Memory Hierarchy Design," Ph.D. Thesis, Technical Report CSL-TR-88-366, Stanford University, Stanford, CA, September 1988.

[PRHH89] S. PRZYBYLSKI, M. HOROWITZ and J. HENNESSY, "Characteristics of Performance-Optimal Multi-Level Cache Hierarchies," *Proceedings of the 16th Annual International Symposium on Computer Architecture*, 1989, pp. 114-121.

[PUZA85] T. R. PUZAK, "Analysis of Cache Replacement Algorithms," Ph.D. Thesis, University of Massachusetts, Amherst, MA, February 1985.

[SAMP89] A. D. SAMPLES, "Mache: No-Loss Trace Compaction," *Proceedings of the International Conference on Measurement and Modeling of Computer Systems*, 1989, pp. 89-97.

[SMIT77] A. J. SMITH, "Two Methods for the Efficient Analysis of Memory Address Trace Data," *IEEE Transactions on Software Engineering*, vol. 3, no. 1, January 1977, pp. 94-101.

[SMIT82] A. J. SMITH, "Cache Memories," *Computing Surveys*, vol. 14, no. 3, September 1982, pp. 473-530.

[STON90] H. S. STONE, *High-Performance Computer Architecture*, Addison-Wesley, Reading, MA, Second Edition 1990.

[WANB90] W. WANG and J. BAER, "Efficient Trace-Driven Simulation Methods for Cache Performance Analysis," *Proceedings of the Conference on Measurement and Modeling of Computer Systems*, 1990, pp. 27-36.

[WOOD90] D. A. WOOD, "The Design and Evaluation of In-Cache Address Translation," Ph.D. Thesis, Computer Science Division Technical Report UCB/CSD 90/565, University of California, Berkeley, CA, March 1990.

[WOHK91] D. A. WOOD, M. D. HILL and R. E. KESSLER, "A Model for Estimating Trace-Sample Miss Ratios," *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, 1991, pp. 79-89.