On the Use of Trace Sampling for Architectural Studies of Desktop Applications

Patrick Crowley and Jean-Loup Baer
Department of Computer Science & Engineering
Box 352350
University of Washington
Seattle, WA 98195-2350
{*pcrowley, baer*}*@cs.washington.edu*

Abstract

This paper examines the feasibility of performing architectural studies with trace sampling for a suite of desktop application traces on Windows NT. This paper makes three contributions: we compare the accuracy of several sampling techniques to determine cache miss rates for these workloads, we present victim cache and branch prediction architecture studies that demonstrate that sampling can be used to drive such studies, and we show how sampling may be used to accurately and efficiently derive the parameters for Agarwal's analytical cache model [1]. Of the sampling techniques used for the cache miss ratio determinations, stitch, *which assumes that the state of the cache at the beginning of a sample is the same as the state at the end of the previous sample, narrowly outperforms the more complex* INITMR *technique of Wood et al. [12] for these workloads. These two techniques are more accurate than the others and are reliable for caches up to 64KB in size.*

1. Introduction

Trace-driven simulation is a common approach for evaluating memory systems. Unfortunately, it also demands large amounts of space and time, particularly for large caches and long running applications. These demands can be greatly reduced by employing sampling techniques at the expense of providing only a statistical estimate of the properties of a full trace. Previous studies [10, 7, 6] contain results for other workloads and caches and discuss the conditions under which sampling may, or may not, be used.

This work was supported in part by NSF Grant MIP-9700970 and by a gift from Intel Corporation.

Our interest in using sampling is three-fold. First, we are interested in the behavior of commonly used desktop applications. When compared to benchmarks such as SPEC95, these applications have larger working sets, are feature rich, and, of course, can run for billions and billions of instructions. Hence, traces based on exhaustive or elaborate executions of such applications will be extremely large. In this work, we consider the usefulness and limitations of trace sampling for a suite of five publicly available desktop application traces for Windows NT on the Intel X86 platform. Second, we want to demonstrate the utility of these sampling techniques for architectural studies. Although it has been shown that trace sampling is not very accurate for metrics such as hit rate when simulating large, multi-megabyte caches [6], we want to demonstrate that sampling is useful to assess trends not only for caches but also for other architectural structures whose state depends on the processing of past references. Such techniques permit the testing of a wide range of architectural parameters in a relatively short amount of time. Third, we want to show that we can use sampling to estimate parameters for analytical cache models in a manner at least as precise as estimates based on whole reference traces and at a much lower computational cost.

In order to determine the feasibility of sampling for the desktop application traces, we present cache miss rate results for a large set of cache sizes and sampling techniques. Most of our results are derived from traces of limited size (from 0.1 billion to 1.5 billion references) so that we can compare sampling results to “true” results. We have also performed sampling experiments on much longer traces and, in general, they show the same trends as those based on a smaller number of samples. Two architectural studies are presented here that apply sampling techniques. The first study demonstrates how sampling, while only an approximation of actual miss rate, may be used to assess trends in victim cache performance [5], and the second study uses sampling to assess trends in branch prediction techniques. Finally, we demonstrate that sampling may be used to accurately estimate parameters for Agarwal's analytical cache model [1].

The remainder of the paper is organized as follows. Section 2 briefly reviews the details of trace sampling and previously published results. Section 3 introduces the benchmarks and sampling techniques used in this study. Section 4 presents and discusses a selection of sampled cache miss rate results. The victim cache and branch prediction architectural studies are given in Section 5. The application of sampling techniques to the estimation of parameters for the analytical model is presented in Section 6. The paper concludes in Section 7.

2. Trace sampling

In trace sampling an observation, or *sample*, is obtained by recording a fixed number, the *sample size*, of consecutive references from a reference stream[1]. Another fixed number of references are ignored before the next observation is made. The *sampling ratio* is the percentage of total references used in all the observations.

Sampling theory states that sets of random, unbiased observations from a population may be used to make inferences about that population. As described above, observations in trace sampling are not random; they are systematic since they are evenly spaced throughout the trace. This non-random pattern is not a problem, though, since systematic observations can be used to make even more precise inferences than random observations under certain circumstances [2] (that is, when the variance of systematic observations is greater than the variance of the population). Unfortunately, however, trace sampling neither involves unbiased observations nor a sufficient alternative. The problem is that the state of the cache is unknown at the start of each observation. In other words, since portions of the trace are unexamined between observations, it is unknown whether the first reference to each cache block will be a hit or a miss. Such references are referred to as *unknown* [12] or *cold-start* [7] references.

A number of techniques have been employed to mitigate the bias due to unknown references. One approach is to make assumptions about, or construct, the state of the cache at the start of each sample. These assumptions may include: assuming an empty cache(i.e., assume that a complete context switch occurred between samples; hereafter denoted *cold*), assuming the state at the end of the previous sample [1] (*stitch*), and using some number of references to prime the cache [4] (e.g., 20% of the sample, denoted *prime-20*, and 50%, denoted *half*). The efficacy of these assumptions depends on workload, cache organization, and choice of sampling parameters(i.e., sample size and sampling ratio.) If complete context switches occur in a cache between samples from a given trace, then assuming an empty cache at the start of each sample, as is the case with *cold*, will be an accurate assumption. If most misses are due to conflicts in a small number of cache lines, then *stitch* may work well since only a small portion of the working set is likely to change between samples. Priming the cache will be effective if unknown references are few relative to the sample size and are mostly included in the priming set.

[1]In statistics, the term *sample* is used to denote an entire collection of observations. Like most other studies, we eschew this usage.

Another approach is to directly determine or approximate the miss ratio of unknown references, which we denote here as μ. For example, *cold* can also be thought of as an estimator that assumes all unknown references miss. In [7], unknown references are not included in the estimate of overall miss rate. That is, unknown references are used to prime the cache but are not counted as hits or misses. As noted in [12], this implicitly assumes that the miss ratio for unknown references is equivalent to the miss ratio for all other references. By employing a renewal-theoretic model that depends on the percentage of time a given cache block frame is alive or dead, Wood et al. show that μ is higher than the overall miss rate [12].

This model is used to estimate μ by observing the probability that a reference to a cache line occurs within a dead time (where time is measured in total references, and dead time implies that the next reference to that cache line will miss). This suggests that if a random time t has probability P of occuring within a dead time for a given cache line, then P is also the probability that an unknown reference will miss in that cache line. This probability can be measured in a full trace by observing the average live and dead time lengths for each cache line in a cache. In a sampled trace, this probability must be estimated with observations within each sample. This sampled probability is the basis for *INITMR*, the miss rate estimator described in [6] and [12].

Accurately coping with unknown references is particularly important when sampling for large caches, where the number of unknown references can easily dominate the number of known misses. Very large caches typically correspond to a very small number of misses, and, hence, are inherently at odds with sampling [6]. As we will see, however, when known misses dominate unknown references, several approaches will be effective.

Table 1. Benchmarks used for this study. The traces of these applications were produced on a dual Pentium Pro 200 system running Windows NT Workstation 4.0 service pack 3.

Application	Description	Instructions Executed (millions)
acrord32	Adobe Acrobat Reader 3.0: Reader for portable document format (PDF) files. The benchmark loads acrobat.pdf (a 277 KB file) from the standard acrobat reader distribution, and navigates through the document three different ways: through the hyperlinks in the document itself, through the forward and back button provided by acrobat reader, and through a view of the document outline provided by acrobat reader. Finally, the benchmark searches for the word "buy" in the document before closing the program.	408
netscape	Netscape Navigator 3.1 web browser. The benchmark opens four web pages: `www.cs.washington.edu`, `www.cnn.com`, `www.mtv.com`, and `www.washington.edu`. These pages were viewed on March 18, 1998. The java module for netscape was turned off because Etch (our instrumentation tool) does not handle the dynamically generated code generated by the java just-in-time compiler.	92
photoshp	Adobe Photoshop 4.0 image editing package. The benchmark loads fruit.jpg (a 591 KB still-life photograph of fruit) from the standard distribution and applies the *color pencil*, *accented edges*, *diffuse glow*, and *add noise* photo filters to the image.	1,511
powerpnt	Microsoft PowerPoint 7.0b slide preparation package. The benchmark loads in a 311 KB 18-page presentation (the presentation included five pages of graphs and six pages of figures in addition to text) in slide mode, scrolls through 3 pages, edits a figure, and continues scrolling through until the end of the document. The benchmark then goes into the outline mode and creates a new page and goes back into the slide mode to move text around. Finally, the benchmark goes into slide sorter mode and moves some slides around.	209
winword	Microsoft Word 7.0 word processor. The benchmark simulates a user typing in seven paragraphs in an eight page document (document size is 29K). The benchmark then performs four search and replace commands on the document before saving a text version of the file. The interactive spell checker was turned on.	351

Table 2. Application object file characteristics.

Application	Executable Size (MB)	Size with DLLs (MB)	# DLLs used (shared)
acrord32	2.26	9.73	34 (24)
netscape	3.17	9.95	28 (24)
photoshp	3.65	13.5	44 (25)
powerpnt	4.36	12.5	26 (21)
winword	3.78	11.2	26 (21)

3. Methodology

3.1. Benchmarks

Table 1 describes the 5 personal desktop applications that we used as benchmarks and their corresponding workloads. Table 2 presents the size of the original binaries as well as DLL usage. A comparison between the execution characteristics of these applications with those of the integer SPEC95 suite can be found in [9].

3.2. Sampling Techniques

After experimenting with various sample sizes and sampling ratios, we settled on a sample size of 500,000 references and a sampling ratio of 0.1. The process of tuning these parameters for a given workload is important [10, 6]. The rationale for our choice for these Windows NT desktop application traces is discussed in a technical report [3].

Table 3 describes the sampling techniques considered in this study. As noted in the previous section, they differ by the state of the cache at the beginning of a sample, or, alternatively, by the method of estimating μ. The techniques not mentioned earlier are *true-sample*, *non-uniform*, *hot* and *tepid*. *true-sample* simulates the caches over the full trace and reports the miss ratio observed over the regions that are sampled with the other techniques. It is therefore an unbiased estimator of the miss ratio for the entire trace. Its accuracy depends on how "fine-tuned" our sampling parameters, *sample size* and *sampling ratio*, are to a given cache

Table 3. Sampling techniques for coping with unknown references.

Technique	Description
true-sample	starts each observation with correct cache state
cold	assumes that the cache is empty at the beginning of each observation (i.e., each unknown reference misses)
hot	assumes that each unknown reference hits
tepid	arithmetic mean of cold and hot
INITMR	calculates the miss rate based on the μ_{split} estimator from [12]
prime-20	uses the first 20% of each observation to prime the cache
half	uses the first 50% of each observation to prime the cache
stitch	uses the end state of the previous observation as the initial cache state for the current observation
non-uniform	same as cold, except the observations are not evenly spaced (jittered by 20% of the sample size

and workload. While *true-sample* is not a practical method, it is however the basis for comparisons with the other techniques which, in addition to the same sampling errors, will have unknown reference biases. *non-uniform* is similar to *cold* but uses non-uniform sampling intervals. *hot* assumes that all unknown misses hit; that is, an unknown miss rate of 0%. Note that *hot* and *cold* form definite bounds for the other unknown reference miss rate estimators. *tepid* is simply the arithmetic mean of *cold* and *hot*, which is equivalent to assuming a 50% miss rate for unknown references.

4. Determination of Miss Ratios

We simulated direct mapped and 4-way set-associative instruction and data caches with sizes ranging from 8KB to 128KB and direct-mapped and 4-way set-associative combined caches with sizes ranging from 256KB to 4MB. Due to space considerations, select examples from these configurations will be presented here, but complete results are available [3]. In the figures to follow, each data point is the arithmetic mean of the miss ratios observed with one sampling method over each of the samples taken from the trace. The error bars for each data point correspond to the 90% interval of confidence based on the distribution of miss ratios of the systematic samples [2]. The actual miss ratio for the entire trace is indicated by the solid bar.

Figure 1 and Figure 2, respectively, display the miss rates corresponding to the simulations of direct-mapped instruction caches for the *acrord32* application and of 4-way set-associative data caches for *powerpnt*. These figures are representative of the complete set of simulations. The following observations can be made:

1. *true-sample* sometimes underestimates the true miss rate (cf. Figure 1) and sometimes overestimates it (cf. Figure 2). Recall that *true-sample*'s accuracy is linked to the choices of sample size and sampling ratio. By choosing a single (sample size, sampling ratio) pair for all applications, we cannot tune these parameters for each application. Note however that when *true-sample* underestimates (resp. overestimates), it does so consistently for all cache sizes for a given application. Also, in all cases, the real miss rate is within the 90% interval of confidence of *true-sample*.

2. All techniques work well, i.e., give results within the 90% interval of confidence for caches up to 32 KB. Among these techniques, *stitch* works best and, *stitch* and *INITMR* both give good, reliable results on all traces for caches up to 64 KB. All techniques, except those priming the cache and *hot*, tend to overestimate the *true-sample* miss ratio as a result of underestimating the miss ratio of the unknown references. With larger caches, the bias gets larger since the number of unknown references is also larger. The priming techniques have a slightly different behavior since the statistics are gathered on a smaller number of references. Nonetheless, their accuracy for caches of 32 KB and more is always inferior to that of *stitch*.

3. A general trend is that confidence intervals decrease with cache size. It is not the case though that we are more confident with the results for larger caches, rather, the miss rates are simply smaller and, hence, so are the confidence intervals. To make comparisons between confidence intervals of different cache sizes, it is necessary to consider the confidence interval *as a percentage of the miss rate.* We see here that this percentage remains roughly constant.

4. *cold* and *non-uniform* yield the same results, suggesting that completely systematic samples are sufficient

and there is no need to inject randomness in intervals between samples.

Figure 3 depicts the results based on the *winword* trace for large, direct-mapped combined data and instruction caches. In this case, we see that the errors due to the choice of the sampling parameters are very small: *true-sample* is highly accurate. However, the bias due to unknown references is extremely high for all techniques except *stitch*, *hot*, and *INITMR*. What we had seen for the larger caches in Figure 1 becomes even more pronounced. For these larger caches, no technique has a confidence interval that consistently overlaps that of *true-sample*, although *stitch* comes closest overall. If one were satisfied with a 95% rather than a 90% confidence interval, *stitch* might be sufficient.

Long Trace Sampling Results: For the previous results we have used traces of limited size (from 0.1 billion to 1.5 billion references) in order to compare sampling results to "true" results. We have constructed a set of longer traces (based on billions of references) of the same suite of applications to test a greater range of features more exhaustively and to determine whether it is necessary or instructive to use larger workloads to drive these applications. Figure 4 gives results for three of the sampling techniques based on a "long" trace of *netscape*. This trace contains samples amounting to 10% of the original reference stream which contained approximately 2.5 billion instruction and data references. Figure 4 compares the true miss rates for the original short trace to some of the sampled miss rates. Since we know *stitch* to be an accurate sampling technique for these workloads and since our emphasis is on minimizing simulation time, we report only those techniques that involve the minimum amount of computation. For *netscape* and these data caches, the differences are not dramatic. This suggests that while the longer trace contains more instructions corresponding to more features, they are not substantially different, with respect to the cache, from the features represented in the shorter trace. The full set of results for long traces and cache configurations contains cases where differences are either insignificant or pronounced (e.g., the sampled miss ratios for data caches based on the longer *acrord32* trace were found to be nearly 70% higher.)

5. Trace sampling for architectural studies

Trace-driven simulation is used not only for the evaluation of cache parameters but also for studying hardware assists to the memory hierarchy or to the processor core. Often these hardware assists contain structures which, like caches, have states that depend on the recent history of data references or instruction execution. In this section, we show that trace sampling is sufficient to give accurate trends on the efficiency of these assists and thus can save considerable simulation time. The two hardware assists that we choose to demonstrate this effect on are victim caches and branch prediction mechanisms.

5.1. Victim caches

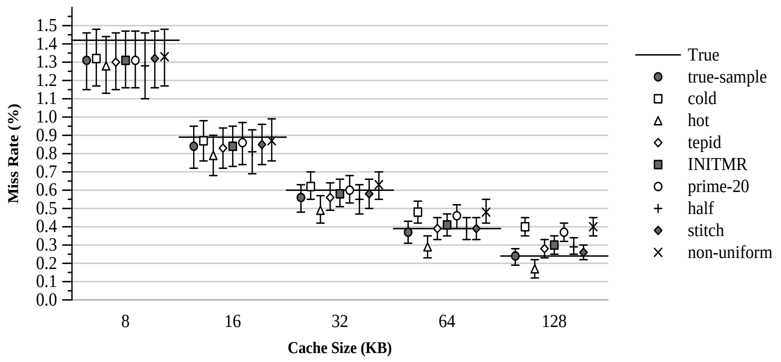
The scenario we consider here is that of an architect who wishes to gather an efficient estimate of the expected decrease in miss ratio for data caches when those caches are augmented with victim caches with between one and five entries. Our purpose here is not to study victim cache trends, as that has been done elsewhere [5], but to demonstrate how sampling techniques may be efficiently used in this regard.

The results are presented in Figure 5 for the *netscape* trace. Figure 5 compares the true results for the victim cache simulation to the *stitch* sampled results. The sampled results were obtained in one tenth of the time required to generate the true miss rates since only one tenth of the original trace was used. In addition to the actual miss rates being very similar, the true miss rate is always either within the 90% confidence interval or slightly outside, the trends are precisely the same. Even if the actual miss rates didn' t agree as well, the predicted trends would be correct. In particular, the trends would be accurately represented even for large caches.

5.2. Branch prediction

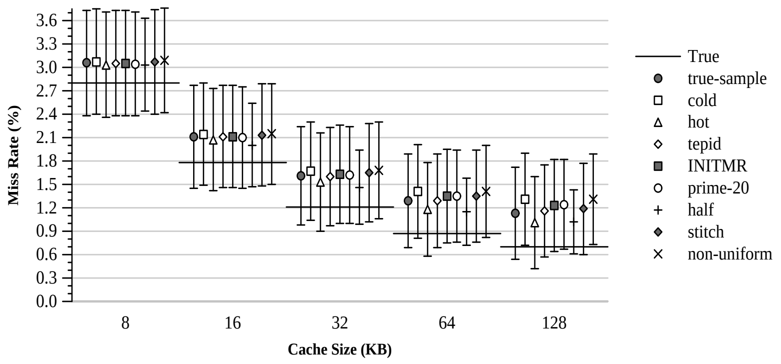
Trace sampling has traditionally been used in cache memory simulations and the previous experiment on victim caches falls into that category. The benefits of trace sampling can be extended to other architectural studies that may involve large workloads and expensive simulation. Branch prediction is one such technique that is typically simulated over the same workloads used to drive cache memory simulations. As these techniques increase in complexity, so do the costs of simulation.

Branch prediction mechanisms rely on one or more tables that encode the result of previous branch outcomes and that are used to predict the outcome of the current branch instruction. The number of branch outcomes n necessary to "warm" up the predictor depends on the particular implementation. Note that the situation is different from that of a "cold" miss in the cache. In a cache, the first reference to a cache line within each sample is an unknown reference because the result of the reference (hit or miss) depends on the previous reference; all the remaining references to that cache line within that sample are correctly identified. In the branch predictors, the first n references to a given entry in a branch predictor will be unknown, given that a prediction depends on the previous n references. The first n references to an entry will still yield a prediction, and that prediction



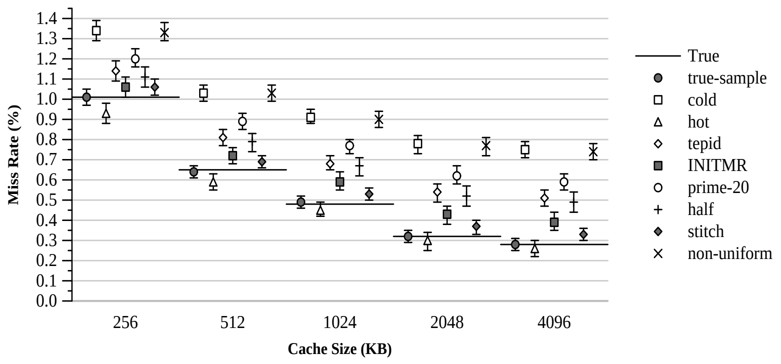
acrord32 (direct, icache) sampling results

Figure 1. Simulation results for *acrord32*.



powerpnt (4-way, dcache) sampling results

Figure 2. Simulation results for *powerpnt*.



winword (direct, ccache) sampling results

Figure 3. Simulation results for *winword*.

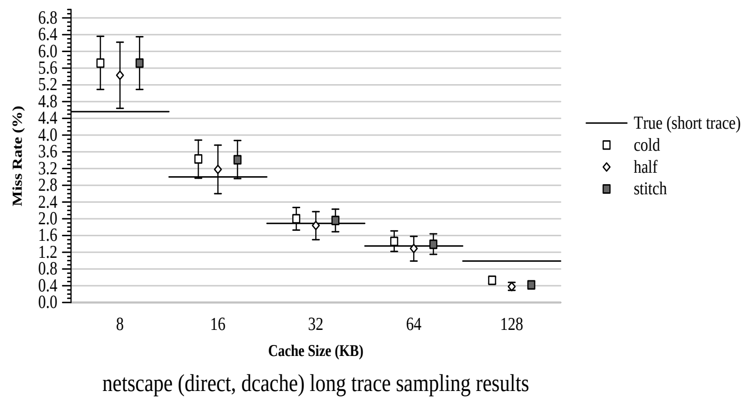


Figure 4. Simulation results for the long *netscape* trace. *true-sample* was not available since we did not record all references, and *prime-20* and *non-uniform* are omitted due to their their similarities to *half* and *none*, respectively.

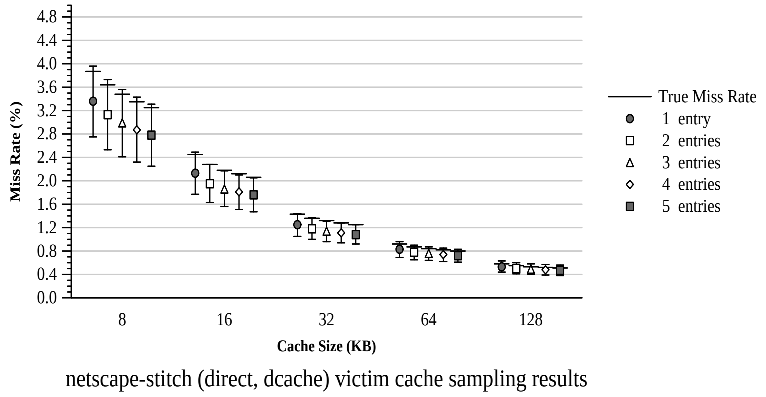


Figure 5. Victim Cache Simulation results for *netscape*.

may be correct, but we won't know if that prediction is the one the branch predictor would have made in the absence of unknown references.

In this section, we investigate the accuracy of trace sampling for two conditional branch predictors with different values of n and with various predictor table sizes. The first predictor is a simple bimodal predictor that uses a two-bit saturating counter per branch. Here, $n = 1$ since it takes two wrong predictions before changing the prediction from "taken" to "non-taken" and vice-versa. The second is a *gshare* predictor that uses a global shared history of branch patterns XOR'd with the program counter to obtain the index into a table of two-bit saturating counters [13, 11]. It is more difficult to assess the value of n because of the dual effects of the history of branches and the value of the PC but it is certainly significantly greater than one. We consider predictor tables ranging in size from 512 to 32,768 entries.

The results from the *acrord32* trace for some of the sampling techniques for the bimodal predictor are found in Figure 6. Again, since we know *stitch* to be an accurate sampling technique for these workloads and since our emphasis is on minimizing simulation time, we report only those techniques that involve the minimum amount of computation. As can be seen, the results using *stitch* are extremely good even for large tables. Because individual branches are most often either always taken or always not taken, keeping the results of past predictions, even distant ones, is beneficial. Starting from scratch, as in *cold*, might yield an unacceptable number of false initial predictions although the trend of almost no gain in prediction accuracy after the tables reach 8K entries is correct.

Figure 7 displays the results for the *gshare* predictor over the same trace. First we can observe that *stitch* is still quite accurate up to 8K entries. Moreover, *stitch* can lead to the correct conclusion that the *gshare* predictor is more accurate than the bimodal predictor only when the predictor tables are sufficiently large (greater than 8K entries for this particular example). On the other hand, none of the other techniques are good in that respect. This is due to the fact that the *gshare* predictor makes predictions according to path history. That is, each branch has multiple entries in the table indicating the taken/not taken decisions made to arrive at it. This effectively increases the number of unknown references for branches seen early in a sample. If the number of entries is sufficiently large and the path history sufficiently long, there is not enough time to build history, a history that was retained in *stitch* and which, from our previous observation, should not change much.

6. Sampling for Analytical Cache Models

In this section, we investigate the use of sampling techniques to estimate parameters that will subsequently be used to solve equations in analytical models of cache behavior. Our vehicle for experimentation is Agarwal's analytical cache model [1].

Like many models based on Mean Value Analysis [8], Agarwal's analytical cache model consists of equations whose inputs are characterized by the mean values of parameters measured from a reference trace. Instead of simulating many cache configurations for a given reference stream, the reference stream is used to collect these input parameter values. These parameters, which are as independent of cache design parameters as possible, are plugged into the analytical model to yield mean value predictions of performance for a variety of cache configurations. Thus, the need for expensive functional simulations is avoided.

Analytical cache models are useful in the sense that they provide a quick way to obtain high-level estimates of cache behavior. They might also pin-point areas in the design space where more accurate, and hence more expensive, evaluation techniques need to be applied. Because quick results are one of the main advantages to analytical modeling, it is necessary that collecting the input parameters to the model be done in a rapid fashion. Even if collecting input parameters does not take as long as a full-fledged simulation, the process can become unwieldy for traces of several billion instructions. Our goal here is to show that these parameters can be gathered via sampling and when the sampled parameters are fed back in the model, they provide answers as accurate as if the parameters had been gathered on the full trace. Our goal, however, is not to increase the accuracy of the model.

The analytical model on which we experiment predicts the miss ratio of associative caches of fixed block size but with various cache capacities and set-associativities. The miss ratio is computed as a composite of effects due to cold misses, non-stationary misses (i.e., misses which occur because of changes in locality), and conflict misses. To gather the parameters necessary to solve for the overall miss rate, the reference trace in [1] is partitioned into some number of time granules of fixed size (e.g., 10,000 or 500,000 references.) The parameters derived from the *entire* trace are:

- u, the average number of unique memory blocks referenced per time granule, i.e., a reasonable estimate of the working set size
- U, the total number of unique memory blocks referenced over the entire trace, i.e., the footprint of the program in main memory
- c, the collision rate, i.e., the average number of times within a time granule that a block that will be referenced in the future is purged from the cache. Agarwal reports that this rate is more or less constant for caches with a given block size provided that the cache size is greater than half the working set size u.

The cold miss effects are proportional to u, the non-stationary effects are proportional to $(U-u)/number\ of\ granules$, and the conflict effects proportional to c and the cache architectural parameters. When we use samples instead of the whole trace, we observe that:

- The average u_s of unique memory blocks referenced per sample is very close to u
- $U_s - u_s \approx (U-u) * sampling\ ratio$, i.e., the term corresponding to non-stationary effects will be similar (U_s is the total number of unique memory blocks referenced over all samples)
- c_s measured on the sample trace is almost the same as c measured on the whole trace.

We obtain the miss rate estimates yielded by the analytical cache model in the manner originally suggested in [1] for input parameters derived from both the entire trace and samples. We used time granule sizes of 10,000 and 500,000 references, but found that, as indicated in [1], the results were insensitive to this parameter. Figure 8 gives the results of applying the model to the *netscape* trace for various cache sizes and two separate estimations of c. The first bar gives the true miss rate, the next two bars give the miss rates for the model using the entire trace and the sampled trace respectively when c is computed with a 16KB direct-mapped cache, while the last two bars are for c computed with a 32 KB direct-mapped cache. The most important observation for the thesis of this paper is that the results from the model using parameters derived from the entire trace and those derived from the sample trace are indistinguishable. Thus, sampling is efficient in that regard. A secondary observation is that the model is most accurate when c is computed for the target size of the cache (this is of course no surprise). Finally, it appears that the larger the cache for which c is computed, the better the accuracy. We have observed that for these workloads, and these working set sizes in particular, c is stable for cache sizes 32 KB and greater. This agrees with the expectation described in [1], that c should be stable for cache sizes greater than half the working set size.

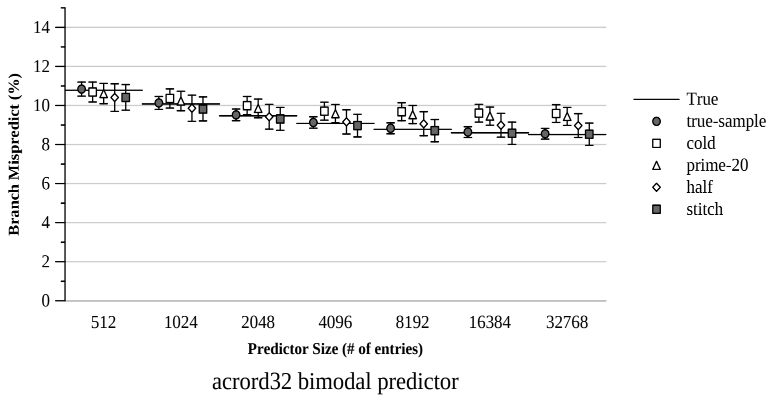


Figure 6. Bimodal Branch prediction sampling results for the *acrord32* trace.

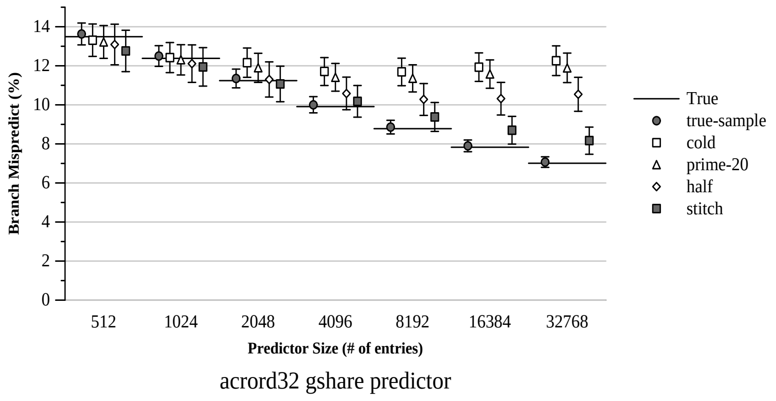


Figure 7. *gshare* Branch prediction sampling results for the *acrord32* trace.

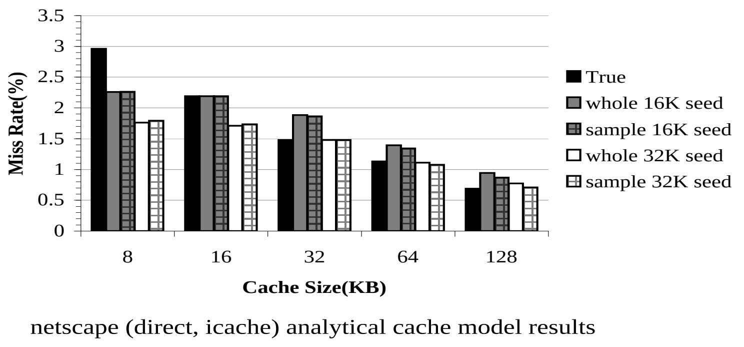


Figure 8. Analytical Cache Model results for *netscape*. In this figure, the label 16K Seed implies that the collision rate, *c*, was measured on this trace for a 16K Cache.

7. Conclusion

Commonly used Windows NT desktop applications can run for billions of instructions. Performing architectural studies on traces of billions of references is not feasible from both time and space perspectives. An alternative methodology is to use trace sampling.

In this paper we have studied the use and accuracy of trace sampling for architectural studies of five Windows NT desktop applications. We have shown that among the choices of sampling techniques for the determination of cache miss ratios, *stitch* (which assumes that the state of the cache at the beginning of a sample is the same as the state at the end of the previous sample) and the more complex *INITMR* are the best at overcoming the difficulties inherent with the problem of the unknown references at the beginning of each sample. Using these sampling techniques resulted in the accurate, i.e., within 90% confidence intervals, determination of cache miss ratios for caches of sizes up to 64 KB.

We also used trace sampling to successfully determine the trends in the use of hardware assists such as victim caches and branch predictors. For these types of time consuming studies, trace sampling can reduce the computational effort by an order of magnitude without loss of insight in the usefulness of the architectural enhancements.

Finally, we have shown that trace sampling is extremely accurate in determining the parameters necessary to drive analytical cache models. In this case also, trace sampling provides large savings in computation time without loss of precision.

References

[1] A. Agarwal, J. Hennessy, and M. Horowitz. Cache performance of operating system and multiprogramming workloads. *ACM Transactions on Computer Systems*, 6(4):393–431, Nov. 1988.

[2] W. G. Cochran. *Sampling Techniques*. John & Wiley Sons, 1977.

[3] P. Crowley and J.-L. Bear. On the use of trace sampling for architectural studies od desktop applications. Technical Report UW-CSE-98-12-05, University of Washington, Dec. 1998.

[4] J. W. C. Fu and J. H. Patel. Trace driven simulation using sampled traces. In *Proceedings of the Twenty-Seventh Hawaii International Conference on System Sciences Vol. I: Architecture*, pages 211–220, Jan. 1994.

[5] N. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proc. of 17th Int. Symp. on Computer Architecture*, pages 364–373, 1990.

[6] R. Kessler, M. D. Hill, and D. A. Wood. A comparison of trace-sampling techniques for multi-megabyte caches. *IEEE Transactions on Computers*, 43(6):664–675, June 1994.

[7] S. Laha, J. H. Patel, and R. K. Iyer. Accurate low-cost methods for performance evaluation of cache memory systems. *IEEE Transactions on Computers*, 37(11):1325–1335, Nov. 1988.

[8] E. Lazowska, J. Zahorjan, G. Graham, and K. Sevcik. *Quantitative System Performance*. Prentice-Hall, Inc., 1984.

[9] D. C. Lee, P. J. Crowley, J.-L. Baer, T. E. Anderson, and B. N. Bershad. Execution characteristics of desktop applications on windows nt. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, June 1998.

[10] M. Martonosi, A. Gupta, and T. Anderson. Effectiveness of trace sampling for performance debugging tools. In *Proceedings of ACM Sigmetrics Conf. on Measurement and Modeling of Computer Systems*, pages 248–259, 1993.

[11] S. McFarling. Combining branch predictors. Technical Report TN 36, DEC-WRL, 1993.

[12] D. A. Wood, M. D. Hill, and R. E. Kessler. A model for estimating trace-sample miss ratios. In *Proceedings of the ACM SIGMETRICS Conference for the Measurement and Modeling of Computer Systems*, pages 79–89, June 1991.

[13] T.-H. Yeh and Y. Patt. Alternative implementations of two-level adaptive branch prediction. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 124–134, 1992.