Efficient Trace-Sampling Simulation Techniques for Cache Performance Analysis*

Tien-Fu Chen

Department of Computer Science
National Chung Cheng University
Chiayi 62107, Taiwan, ROC

Abstract

In this paper, we focus on the simulation techniques in order to reduce the space and time requirements for simulating large caches. First, we propose a space sampling technique to perform trace reduction for time and space. Our approach is to perform stratified sampling *based on an index of locality. Our results show that the technique can provide accurate estimate of performance metric using only a small portion of trace references. Alternatively we also propose a time sampling approach, which performs sampling on loop iterations and requires that references between inter-loop intervals be fully simulated. We show that the time sampling technique may give representative performance results for the entire loop execution. Depending on different workloads, the approach has been shown to be very effective in reducing simulation time at the cost of small estimate errors.*

1 Introduction

As processor speed has been increased dramatically over the last few years, memory latency and bandwidth have also progressed but at a much slower pace. It is therefore essential that we investigate techniques to reduce the effects of the imbalance between processor and memory cycle times. The introduction of caches between the processor and memory modules has been shown to be an effective way to bridge this gap. However, as the gap is increasing and the cache plays a more important role in the architecture, a careful cache design choice is therefore crucial to the design of future computer system. As a result, efficient evaluation methodology of memory system becomes an essential and vital step towards a design of high-performance system.

*This work was supported by National Science Council of R.O.C. under grant NSC 83-0408-194-004.

In general, there are three ways to evaluate the performance of a cache memory: hardware monitoring, analytical modeling, and simulation. The first approach provide the most accurate results, however it requires expensive hardware and has limited applicability. The second approach[2] requires most inexpensive calculations, but only gives roughly estimated results. Simulation, especially driven by traces[4], has been the most widely used method among the three ways. It generally works in the following way: A program of interest (or benchmark) is really executed and at the same time a history of its memory references (called *trace*) is captured. The resulting address trace is then used to drive a simulation model under study. By varying the parameters in the simulation model, we can investigate various choices of a new cache architecture design. The drawbacks of the trace-driven simulation approach are: (1) it requires a large storage to record the traces, (2) it takes a long simulation time, and (3) the impact of varying architectural choices may not be reflected in the traces. Among of them, the effect of the third factor is still unknown, since it is essentially hard to measure counter-effects on the final results. As cache sizes are growing, these problems may be getting worse because it would require very large traces (e.g., billions of references) to obtain steady-state performance[7].

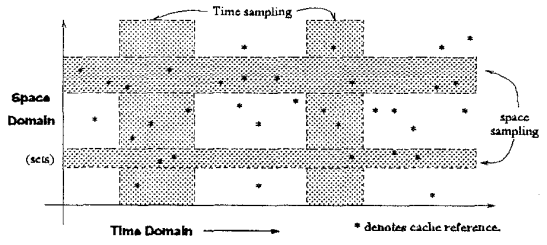

Figure 1. Sampling by space and time division

1080-241X/96 $5.00 © 1996 IEEE
Proceedings of SIMULATION '96

Since trace simulation is generally time-consuming, we may avoid the long simulation process by using *trace-sampling* techniques. The key idea is to observe only a small portion of the cache simulation and make the performance measurement from a collection of these observations (also called *samples*). As shown in Figure 1, the sampling techniques may be applied in either time domain or space domain, or combined both. By time sampling, the cache performance is observed only in several time-contiguous trace intervals and the rest portions of the trace stream are ignored. By space sampling, the performance is observed for references accessing portion of overall cache sets. Since the cache is usually relatively small to the main memory, it is partitioned into many sets and each set contains one or more data lines[11]. A basic operation of simulating the cache is to select a set based on the data address and then to compare the tag with the data reference. Under such an organization, those references to one set are conceptually independent of references to the other set.

In this paper, we focus on the simulation techniques mentioned above, namely simulation by sampling in order to reduce the space and time requirements for simulating large caches. The rest of the paper is organized as follows: the next section gives a review of related work. In Section 3, we describe a simulation methodology which may provide efficient technique to deal with huge trace problems for simulating large caches.

In Section 4, we propose a space sampling technique (i.e., set sampling). The basic idea is to quantify the locality and to divide the entire cache sets into "similar" groups. The sampling is done by a random choice within similar groups based on a given ratio, that is, by *stratified sampling*. The overall reference traces can be significantly reduced without generating too much error.

Section 5 gives a technique of time sampling. Instead of randomly cutting trace by fixed time intervals, we perform sampling based on the loop iterations. The assumption is that the behavior of loop iterations is basically similar so that sampling on loop iterations may give representative performance results for the loop execution. The execution between loops is fully simulated to preserve the behavior characters which may have large variations. Fortunately, these execution parts only share a small portion of the overall execution stream. Finally, we conclude the paper and suggests directions for future work in Section 6.

2 Related Work

The primary resource that the trace-driven simulation relies on is the set of trace. However, traces are generally huge, requiring hundreds of megabytes per trace. There are in general three approaches to minimize the trace storage size. The first approach *abstract execution* is proposed by J. Larus[14]. In his scheme, the trace contains only a subset of the addresses with which the use of static analysis can combine to generate a complete trace. Thus, traces are stored in terms of *abstract* records of program execution.

The second approaches *trace reduction* decreases the size of traces by cutting partial information. Smith[20] pioneered this work by proposing a trace deletion scheme for memory paging studies. To extend Smith's technique, Puzak[18] introduced a trace stripping technique that reduces program traces by preprocessing them using a smaller cache filter. Wang and Baer[21] extended Puzak's cache filter to create a reduced trace that may contain exactly the same number of cache misses and write-backs as the original trace. In contrast to Puzak's cache filter, Agarwal et al.[3] proposed a block filter to exploit cache spatial locality. The block filter considers a window of w consecutive references a time and leaves only a reference to represent spatial locality contained within the reference window.

The third approach *trace compression* aims at reducing the storage size without losing any trace information. The standard sequential data compression algorithm such as the Lempel-Ziv scheme[22] may not work well in the trace compression because the addressing patterns does not always repeat as expected. Samples[19] presented a trace compression technique which checks the difference between a current address and the last address and keeps only those relatively large differences in the trace.

As to the sampling technique, Laha et al.[13] gave a sampling study by observing cache behavior in several trace intervals in a single time-contiguous trace. Since the length of sampling intervals was heuristically experimented, it is questionable what a good interval should be derived for general applications. Kessler et al.[12] compared trace-sampling techniques for very large caches. Their results indicate that sampling can be a good way to achieve good reduction. However, since their observations were based on large caches, where variations between sets tend to be small, the results may not be applicable to other organizations. Liu and Peir[16, 17] examined techniques for sampling of cache sets based on congruence classes. Their approach requires traces to be pre-processed and the technique is generally limited to a single organization. They group up the sets whose miss rates are similar and assume that the sets in the same group as have similar behavior. For each group, they selected a few sets to stand for that group. As indicated by Heidelberger[10], there is more information obtained by running a single set for a longer time that there is in examining a collection of highly correlated references produced over a shorter period of time. Hence, the key observation is to avoid simulation on sets with high correlation.

Another trace sampling had been studied by Fu and Patel[9]. In their study, model simulation is performed

using sampled intervals which are generated by taking a number of samples from a complete trace. Unfortunately, the algorithm to deal with fill prediction problem (cold start) is based on heuristic inference. Concurrent simulation of cache sets[10] had been studied in the area of parallel simulation. To sum up, these sampling techniques still bear a similar disadvantage that they can only measure cache misses. More recently Lauterbach[15] proposed a parallel simulation scheme by distributing the simulation of the samples across many workstations. He avoided the use of a large samples by taking the advantage of filling the cache with the warm-start state.

Laha et al.[13] studied the sampling by taking a number of sampled intervals. References within an inter-sample interval are not collected and simulation is performed only within those selected samples. The primary problems to use partial intervals to represent the overall metric are that the cache memory may contain unknown states that would have been filled by references in the inter-sample interval (also known as *cold-start problem*) and that the intervals are generally selected in a systematic way. Agarwal et al.[1] suggested that the cache state at the beginning of an interval can be *stitched* with the cache state at the end of the previous interval. However, the approach requires that references nearly at the end of inter-sample interval be not skipped. Fu and Patel[9] also performed the study of sampling in obtaining accurate cache simulation results from a representative sampled trace. Their efforts are focused on the determination of initial cache state. Overall, all of these approaches select the interval in a random manner at a length of randomly chosen size.

3 Evaluation Methodology

3.1 Trace generation

To generate the traces in this study, we use the `QPT` system[6] developed at the U. of Wisconsin. It is a profiling and program tracing system with rewriting a executable program file by either inserting code to record the execution frequency of every basic block or control-flow edge, or by inserting code to capture every instruction and data references.

Since the size of trace files are generally very huge (e.g., Nasa requires 300MB for the compressed file and Eqntott requires 200MB space), we employ the *on-the-fly* technique, that is, the output of the trace generator is directly fed to the simulator without writing traces to a disk. The advantages of the on-the-fly simulations include the saving of disk storage and the reduction of data transmission time (especially when the disk storage is not local to the simulating environment). However, the cost is that traces must be regenerated for each simulation of the architectural model. We alleviate the problem by simulating multiple cache configurations for each trace generation pass so that the cost of regenerating traces is amortized among several configuration studies. The other drawback is that the operating system may alternatively execute the two processes where context switching time becomes another extra overhead due to the fact that the trace producer and the trace consumer (simulator) are different processes.

3.2 Benchmarks and metric

In this study, we focus on only the simulation techniques. We vary the size and set-associativity of memory caches over a range of 64K-2M bytes and direct-mapped to four-way. We do not consider other features or cache enhancements like prefetching, write strategies.

We perform simulations on SUN SPARC 10 workstations running the SunOS 4.1.3 system, using standard CC and Fortran compilers. We use ten programs from the SPEC benchmark suite.

The performance metric we used is *misses per instruction* (MPI) instead of miss ratio. The two main reasons are that miss ratio may be misleading in estimation by sampling and that MPI can give the more common measure *cycle per instruction* (CPI) by multiplying a factor of cache miss latency. The original *MPI* value of the full trace (called MPI_{full}) is:

$$MPI_{full} = \frac{\sum_{i=0}^{m-1} miss_i}{\sum_{i=0}^{m-1} instruction_i} \quad (1)$$

where $miss_i$ and $instruction_i$ are the miss count and the instruction of the ith set respectively in a cache with m sets.

4 Space Sampling

When sampling in space domain is concerned, the first intuition is *set-sampling*. Because the simulation space can be divided into multiple disjoint subspaces, we can simulate the activity in each of the spaces independently. The *set-sampling* we consider is a technique to produce a *reduced trace*, which is a fraction of a program trace, typically containing only a fraction of references in the full trace. Instead of the huge full trace, we could just use a reduced trace to drive simulations and to estimate the performance of a cache memory systems so that the long simulating time could be saved.

For set sampling, the simplest method is to select sets randomly from a cache. However, the selection does not imply any relationship with the behavior of other cache sets.

Hence, the result from random selection may not necessarily give representative results for the entire cache memory.

Kessler et al.[12] had proposed a *constant-bit* method to overcome the disadvantages of random sampling. The scheme selects all of the references whose set index addresses contain certain constant bits, for instance, those have the binary value 0000 in address bits 8-11. The key observation in the method is to select references rather than sets. However, the selection rule is very regular, so it may not reflect the actual behavior of the trace. The problem occurs when the workloads use the address space systematically and the trace has a certain degree of regularity according to the constant bits (e.g., frequent accesses to a fixed stride vector), we will get a biased estimation.

4.1 Quantify the locality

A program or a process usually tends to use only a portion of its address space at certain times, which is known as the *principle of locality*. In fact, this principle has two dimensions: *temporal locality* and *spatial locality*.

Our basic assumption is that those sets which have similar degree of locality should have a certain relationship, that is, are highly correlated. We will define two index values to quantify the degree of locality. However, the concepts of temporal and spatial locality are very abstract, so we generally concern about the term "locality" only. Then, we divide the sets into several *strata*, that is, to stratify the cache sets so that sampling can be done based on each stratum respectively.

Now we wish to quantify the concept of locality in terms of concrete values, so that we can estimate the degree of similarity for cache sets. Given that the smallest access unit is a block, we consider the locality indices for data blocks. We use two parameters to quantitatively represent the locality: distance locality and neighborhood locality. First, we specify a *serial number* for each address reference and define the *distance* between two references which are mapped to the same set as the difference of their serial numbers. Let's consider the distance between consecutive references for one data block. Clearly, the longer the distance, the weaker the locality is. Since the dropping pace of locality with respect to distance can be approximated by an exponential function, we use the function e^{-d} to represent that.

When the distance is equal to one for a block, the block has been referenced two times in a row. Hence, it has the highest locality value. When the distance is larger than a threshold value, the locality becomes very insignificant. We estimate the degree of locality for one data block by adding up all locality function values calculated from each pair of consecutive references for that block. We call the added-up value as "*distance locality*", which is more related to temporal locality.

The other measure we used for estimating locality is to count the references frequencies for each block associated with its neighbor blocks. By the principle of spatial locality, a block tends to have a closer relationship with its neighbor blocks than other blocks in the entire address space. We assign a counter for each block. The counter of a block will be incremented whenever either this block itself is accessed or one of its neighbor blocks is referenced. We call the value of the counter "*neighborhood locality*", which is related to spatial locality.

We go through the entire trace to calculate the locality values for each block. Finally, we combine these two values by multiplication and obtain a single general locality value with each data block.

4.2 Sampling by stratification

Our goal is to draw a few sets out of the smallest cache (described in the next section) as a sample for generating reduced traces. The reduced trace is a collection of address references which access the sampled sets. Since many blocks may be mapped to the same cache set while they are placed in the smallest cache. We obtain an index value for each set by computing the sum of general locality of all blocks associated with this set. The basic idea is to stratify sets by locality index based on the index to perform set sampling. The assumption is that the sets with similar index values would have similar properties and can to be assigned in the same stratum.

From sampling techniques[8], we know that:

If n sampling events are observed from N populations of X (for example, X is the MPI of cache sets), we let $f = \frac{n}{N}$.

When a random sampling is applied, the variation of sampling estimates will be

$$\sigma^2_{\bar{x}'} = \frac{N^2}{n}(1-f)\sigma^2_X$$

where σ^2_X is the variation in population.

When proportionally sampling is performed on stratified sets, we have

$$\sigma^2_{\bar{x}'} = \frac{N^2}{n}(1-f)(\sigma^2_X - \sigma^2(\bar{X}_h))$$

where $\sigma^2(\bar{X}_h)$ is the variation among stratum means.

As a result, the overall variation of sampling estimates can be minimized by a proper stratification. The key point is that we use the locality index to stratify those cache sets into a number of *strata*, in each of which the variance of metric is minimal, whereas the difference in the stratum means $\sigma^2(\bar{X}_h)$ is maximal.

In this study, the similarity is defined as the difference of index values below 10%. In that manner, the total sets

Table 1. Reduction ratios for different selection ratio

	Trace	Selection Ratio r_s				
		1/4	1/8	1/16	1/32	1/64
Ratio of set selected	Eqntott	27.15 %	15.14 %	9.18 %	6.15 %	4.69 %
	Espresso	28.52 %	16.89 %	11.04 %	8.50 %	7.52 %
	Fpppp	30.86 %	20.02 %	14.84 %	12.79 %	12.21 %
	Matrix	26.66 %	14.55 %	8.50 %	5.37 %	4.00 %
	Nasa	27.64 %	15.82 %	9.96 %	6.93 %	5.47 %
	Spice	28.32 %	16.70 %	11.23 %	8.59 %	7.23 %
	Tomcatv	26.56 %	14.45 %	8.40 %	5.37 %	3.81 %
	Xlisp	27.83 %	15.92 %	10.25 %	7.42 %	5.96 %
Ratio of reduced trace	Eqntott	29.52 %	17.12 %	11.00 %	7.42 %	6.09 %
	Espresso	39.36 %	31.94 %	27.00 %	24.95 %	24.87 %
	Fpppp	30.75 %	22.55 %	18.98 %	17.57 %	17.00 %
	Matrix	27.42 %	19.15 %	14.30 %	11.48 %	9.87 %
	Nasa	26.97 %	16.50 %	11.05 %	7.07 %	5.29 %
	Spice	30.69 %	19.71 %	14.59 %	12.24 %	10.75 %
	Tomcatv	29.32 %	16.61 %	8.93 %	5.06 %	4.55 %
	Xlisp	28.55 %	17.04 %	11.59 %	8.78 %	7.99 %

of the smallest cache are split into many strata. For each stratum, we just pick out a few sets (given by a ratio) to represent all other sets in the stratum, and the sample trace is obtained by collecting references to the selected sets.

The next task is to pick up sample sets from the entire cache based on stratification sampling concept. Considering the a stratum of sets with similar index values, we establish a selecting ratio r_s ($r_s < 1$)and pick the ratio of number of sets from each stratum. Upon the sampling ratio r_s on cache sets number, a collection of trace from sampled sets does not necessarily give a fraction of r_s of full trace. There are two level of sampling to be performed. For a stratum that contains sufficient n number of sets, that is, $n > \frac{1}{r_s}$. Then we should pick $\lceil r_s n \rceil$ sets out of this stratum. Other small stratum as a group are randomly sampled at the same ratio r_s.

4.3 Conflicting blocks

Since the sampling on the space domain is to provide a cache filter such that other configurations may also use the reduced trace to obtain the estimated results with less simulation cost. As a result, we should take into account the block conflict problem in selecting sets.

Let's consider those blocks that may conflict while they are mapped into the same cache set. The situations may not be identical in different cache configurations. Two blocks of data conflict within a 32K cache, they may or may not collide in an 128K cache. Base on the inclusion property[5], we can show that two blocks conflict in an 128K cache must have a collision in a 32K one if the associativity of these two cache are identical. Therefore we infer the following observation based on inclusion property:

Observation: Given two caches A and B with different size have the same block size and the same associativity, the size of A (A_{size}) is larger than the size of B (B_{size}). If there are two data blocks conflicted in the cache A, then they will always conflict in the cache B.

Suppose these two blocks are placed at set S in cache A, then in cache B, they would all be located at the same set ($S \ mod \ B_{size}$). At the same time, if two blocks are belonged to two different sets in cache B, they will be separated in cache A. Hence, in order to provide a fully independent sets, we need to use the smallest cache in our cache filtering process, that is, to choose a minimal configuration in the study to obtain a filtering cache for set sampling.

4.4 Estimating *MPI* and errors for a sample

As described in Section 3, we use the *MPI* as the performance metric. What we need is to estimate the overall *MPI* using set sampling technique. We use a 32K cache as the smallest cache to simulate full traces. The smallest cache has m sets (m = 1024 in our experiments), and there are n sets selected in a set sample S. The original *MPI* value of the full trace is defined as Equation (1).

Then we use the *all-instructions* method[12] to calculate the estimated MPI_S for sample S as follows:[1]

[1] We do not estimate MPI_S as $\frac{1}{n}\sum_{i \in S} \frac{miss_i}{instruction_i}$, since the arithmetic mean of the miss ratio per set may contribute significant errors[18].

Table 2. Coefficients of variance by space sampling

		Selection Ratio r_s	1/4	1/8	1/16	1/32	1/64
Trace	Cache size	$MPI_{full} \times 1000$	Coefficients of variance				
Eqntott	128 K	4.213	1.32 %	2.35 %	2.77 %	3.16 %	2.89 %
	256 K	2.451	1.78 %	3.16 %	4.28 %	5.14 %	5.20 %
	512 K	0.849	2.98 %	5.43 %	5.84 %	6.06 %	6.76 %
	1024 K	0.263	5.77 %	13.86 %	19.39 %	28.04 %	35.75 %
Espresso	128 K	0.077	11.42 %	13.24 %	21.38 %	34.06 %	26.93 %
	256 K	0.045	10.99 %	12.53 %	14.63 %	25.32 %	24.64 %
	512 K	0.022	1.29 %	2.12 %	3.62 %	4.65 %	5.74 %
	1024 K	0.022	1.15 %	1.98 %	3.14 %	4.28 %	5.20 %
Fpppp	128 K	0.224	6.04 %	7.47 %	6.60 %	7.60 %	8.57 %
	256 K	0.186	3.18 %	3.86 %	4.48 %	4.86 %	4.59 %
	512 K	0.174	1.96 %	3.17 %	5.04 %	6.87 %	6.15 %
	1024 K	0.173	0.72 %	1.26 %	2.21 %	2.79 %	2.89 %
Matrix	128 K	1.688	3.39 %	5.53 %	5.36 %	6.74 %	8.17 %
	256 K	0.141	1.75 %	2.62 %	2.57 %	3.32 %	4.01 %
	512 K	0.025	3.70 %	5.36 %	8.57 %	10.21 %	14.81 %
	1024 K	0.022	0.33 %	0.70 %	1.37 %	2.31 %	2.85 %
Nasa	128 K	9.744	0.18 %	0.30 %	0.36 %	0.39 %	0.50 %
	256 K	0.290	7.12 %	11.13 %	14.11 %	13.68 %	19.86 %
	512 K	0.003	2.09 %	4.92 %	8.06 %	12.28 %	15.47 %
	1024 K	0.002	0.47 %	0.98 %	1.73 %	2.87 %	3.57 %
Spice	128 K	7.144	1.54 %	2.76 %	3.51 %	4.97 %	6.29 %
	256 K	3.203	0.73 %	1.36 %	1.55 %	2.29 %	2.43 %
	512 K	0.679	1.26 %	1.90 %	2.75 %	4.87 %	6.03 %
	1024 K	0.351	0.24 %	0.41 %	0.64 %	0.87 %	1.08 %
Tomcatv	128 K	6.044	0.35 %	0.52 %	0.68 %	0.48 %	0.85 %
	256 K	5.947	0.18 %	0.27 %	0.36 %	0.27 %	0.48 %
	512 K	5.790	0.10 %	0.15 %	0.20 %	0.19 %	0.32 %
	1024 K	4.938	0.09 %	0.13 %	0.20 %	0.27 %	0.37 %
Xlisp	128 K	0.064	3.00 %	4.18 %	7.23 %	8.10 %	10.35 %
	256 K	0.001	2.39 %	3.37 %	5.73 %	6.46 %	7.79 %
	512 K	0.001	2.39 %	3.37 %	5.73 %	6.46 %	7.79 %
	1024 K	0.001	2.39 %	3.37 %	5.73 %	6.46 %	7.79 %

$$MPI_S = \frac{n}{m} \frac{\sum_{i \in S} miss_i}{\sum_{i=0}^{m-1} instruction_i}$$

where $|S| = n$ and there exist m sets in the cache. For each program trace, we have to take several samples (S_1 to S_K) to make multiple estimations. Therefore the coefficients of variance (CV) for these samples can be computed:

$$CV = \frac{\sqrt{\frac{1}{K}\sum_{j=1}^{K}(MPI_{S_j} - MPI_{full})^2}}{MPI_{full}}$$

where K is the sampling times (number of samples).

4.5 Results

Table 1 shows the results of stratified sampling on cache sets from eight traces. In the table, we show respectively the reduction ratios of number of sets selected and fractions of the total length of trace with respect to selection ratio (r_s). Consider the column labeled "1/4" in Table 1, which presents to the stratified sampling using $1/4^{th}$ of the sets, that is, 25% of sets. Using stratification on sets, we obtain trace reduction by roughly 26%-30% because the number of sets in each stratum may not be equal. When we reduce the fraction of sets in a stratum (the number of sets per stratum) from 1/4 to 1/16 and from 1/16 to 1/64, the ratios are significantly reduced, up to only 3%-10% of original trace. However, the effectiveness of trace reduction apparently depends on the characteristics of workloads, where the

stratum sampling does not give effective the reduction for benchmarks Espresso and Fpppp. Also note that the ratio of sets selected does not necessarily coincide with the ratio of traced trace (e.g., Espresso and Spice). The main reason is that trace references do not equally distributed among all of sets. This also explains that why random systematic sampling like *constant-bit* scheme may not work well for some workloads.

Table 2 gives the errors between stratum sampling and MPI_{full} for different workloads with varying cache sizes and selection ratio. The column labeled $MPI_{full} \times 1000$ gives the "real" value of *MPI* when a full trace of each workload is applied. As expected, the MPI_{full} decreases as the cache size increases. The left part of the table shows the coefficient of variation of the stratum-sampling *MPI* estimates. Generally speaking, most workloads contain relative errors of less than 10%, some even less than 1%. Under the same workload, the relative error is slightly increased as the selection ratio becomes smaller, since the ratio of trace selected is smaller. Increasing the cache size from 64K to 1024K may generally reduce the coefficients of variance. This is because a cache with a larger size has less conflict and capacity misses than a cache with a smaller size so that the locality index can be reflected in a more accurate manner. However, in some workloads, the condition may not hold in some particular sizes. We conjecture that this is due to the fact that those references mapped to sets are severely biased to certain particular location. Based on the reduction in Table 1 and the errors in Table 2, we can find that our stratum sampling can obtain accurate *MPI* estimate (with errors less than 10%) in Eqnott, Matrix, Nasa, Spice, Tomcatv, and Xlisp at a cost of only a small fraction of trace (less than or equal to 10% of overall trace). For the rest of workloads, the technique also gives acceptable error with slightly higher cost (up to 30% of original traces), although some cache configurations with severe conflict may give significant errors.

5 Time Sampling

Time sampling is another alternative technique often used in the experiment with a long execution stream. Instead of randomly cutting trace by fixed-length intervals, we perform sampling based on the loop iterations. The assumption is that the behavior of loop iterations is basically similar so that sampling on loop iterations may give representative performance results for the loop execution. The execution between loops is fully simulated to preserve the behavior characters which may have large variations. Fortunately, these execution parts only share a small portion of the overall execution stream. The results indicate that our time sampling technique can skip a large portion of traces (up to 95%) but still obtain accurate performance metric. The bias effect due to unknown cache state is small in the inter-loop intervals.

5.1 Time sampling by loop

Since the primary goal of time sampling is to collect a representative collection of sampled intervals, sampling on a fixed length interval in a random way may not be representative as expected. Figure 2 shows the MPI versus AMPI with respect to sampled intervals in a length of one million instructions per interval. We can see that the values of MPI per interval are widely different and randomly distributed. As a result, not only results collected from partial trace may be misleading, but a wide variance in MPI per intervals may not give a representative result.

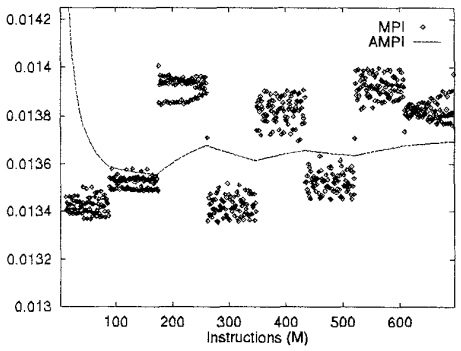

Figure 2. MPI per 1M instructions for Matrix

Instead, we observe the same metric in terms of loop iteration per interval as shown in Figure 3, where intervals between different loops may have different length. The metric MPI of each iteration becomes more stable on a per-loop basis. Hence, the observation of the regularity of loop execution leads to our motivation of time sampling on loop iterations.

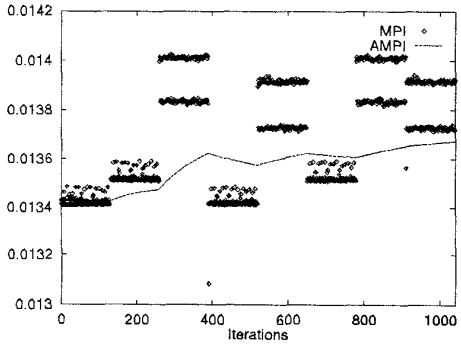

Figure 3. MPI of every iteration for Matrix

The basic idea of our sampling approach is to identify the loop execution starting point and ending point. When

the simulation finds a starting point of a loop, the simulator keeps track of data in each loop iteration. Once the instantaneous performance metric from one iteration is not *too much* different from previous iterations, we may conjecture that the following iterations have similar performance behavior. Hence, we may skip the rest of execution in the loop and then estimate the performance metric of the loop based on the observed data. Even though we perform sampling in the loop, the simulation after one loop will be restarted at the end of loop execution, that is, all of execution streams between loops are fully simulated.

Our assumption is that the simulator is able to identify the starting point and ending point. To achieve that, we need to instrument the benchmark programs so that special hints will be passed to the simulator when the necessary loop points are reached. One way to do that is to rewrite the binary code of the benchmark programs[6]. The loop execution can be identified by a backward branch edge in the control flow graph. Since we currently use SPEC benchmarks, where source codes are available to us, our approach is to insert additional instrumentation assembler language code in the beginning and the end of the loop for the purpose of generating special marker events in the trace.

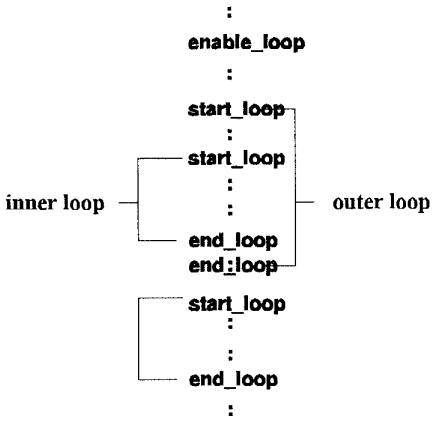

Figure 4. Iterations of nested loops

When the modified code is executed and the loop marker events occur, the simulator would be able to identify a portion of instruction stream as a single iteration and repeat the iterations until the end of loop execution. The new trace stream is observed by the simulator as shown in Figure 4. In this manner, the performance metric of a sample is collected based on the execution stream between the beginning and ending markers.

5.2 Sampling criteria

As the execution of one iteration is identified, now the question becomes the determination of the continuation and skip of iterations in the loop. Our assumption is that when the performance metric of loop iterations is getting stable or periodic, then the rest of execution can be inferred from the previous sections. First we define $AMPI_k$ as the average of MPI in the loop after k iterations:

$$AMPI_k = \frac{\text{total misses \# up to } k \text{ iterations in a loop}}{\text{total instructions \# up to } k \text{ iterations in a loop}}$$

We require that each loop must execute at least 5 iterations before the loop sampling. The reason is to avoid the start-up effect and to obtain reasonable representative data for the sampling. After that, when the difference of AMPI between two adjacent iterations is sufficient small to a predefined rate, we can expect that the AMPI of each iteration afterwards can be estimated to the approximate value. As a result, after at least 5 iterations, the simulator detects the condition:

$$\frac{|AMPI_{n-1} - AMPI_n|}{AMPI_{n-1}} \leq 5\%$$

If the condition holds at the n^{th} iteration, we can estimate the metric of the entire loop using the stable $AMPI$. Hence, the total number of misses for the loop would be:

$$\text{Total misses of loop} = \sum_{i=1}^{n} M_i + (\sum_{i=1}^{N} I_i - \sum_{i=1}^{n} I_i) * AMPI_n$$

where N is the number iterations in the loop. In the above equation, the first term is known after the first n iterations by simulation and the next term is to estimate the miss count. After that, we resume the simulation so that the metric between inter-samples can be collected. Hence,

$$AMPI_{overall} = \sum_{i \in loop_i} AMPI_i * Weight_i + \sum_{k \in inter\_loop_i} AMPI_k * Weight_k$$

where $Weight_i$ is proportional to the instruction count of each sample. Finally, we give the bias of MPI estimate by

$$Bias_t = \frac{AMPI_{overall} - AMPI_{full}}{AMPI_{full}}$$

where $AMPI_{full}$ is the average MPI obtained from a full trace.

Although we skip from n^{th} iterations on, the fraction of reduced trace can be very small since the value $\frac{n}{N}$ is usually small such that most simulation time can be saved. The other advantage of our sampling over traditional time sampling is that the long loop execution may mitigate cold-start bias because programs usually have different working sets for different loop execution.

5.3 Results of Time Sampling

Table 3 gives the results of time-sampling by loop iterations for benchmarks Nasa, Tomcatv, Matrix, and Fpppp. Particularly for Fpppp, we perform three different levels of loop sampling because the entire program consists of several loops at different levels. In Fpppp(1), we insert the loop markers with respect to the most outer loop so that each iteration may contain even up to millions of instructions. Fpppp(2) gives the sampling for the most fine-grained loop iterations, where the similarity among iterations is the most apparent. In the left part of Table 3, the columns below labeled "data reference count" give the fraction ratio of the overall trace which has been really simulated. In most cases, time-sampling by loop iterations skip a large portion of traces (only 2% to 7.4% of data references is actually used).

Table 3. Results of time sampling by loop

	Data reference count		
	Original	Sampling	ratio
nasa	5678M	422 M	7.44 %
tomcatv	2278 M	58 M	2.55 %
matrix	308 M	14 M	4.68 %
fpppp(1)	1743 M	715 M	41.03 %
fpppp(2)	1743 M	1409 M	80.77 %
	MPI		
	Original	Sampling	$Bias_t$
nasa	0.014328	0.014330	0.0115 %
tomcatv	0.013254	0.013076	1.35 %
matrix	0.013685	0.013681	0.029 %
fpppp(1)	0.000852	0.000948	11.25 %
fpppp(2)	0.000800	0.000859	0.77 %

The right part of Table 3 shows the *MPI* estimates and error. We can see the bias of the performance metric is relatively small for programs Nasa, Tomcatv, Matrix, and Fpppp(2). In these programs, the regularity is very significant so that the overall *MPI* can be easily calculated even with a very small fraction of the trace. It indicates that for programs with intensive loop computations to spend much simulation time in obtaining a single result is not necessary. However, if we apply time-sampling at intervals of fixed length, we may not acquire sufficient representative samples. For Fpppp(1), the *MPI* estimate gives slightly large errors to the real *MPI*. Our observation on the program is that the first few iterations in the medium loops have momentous oscillation curve that the simulator tends to over-estimate the *MPI*.

5.4 Effect of unknown cache state

Similar to the time-sampling technique, the loop sampling approach must estimate the initial effect for those intervals without knowledge of initial cache state. Although this is also known as *cold-start problem*, the problem has a different situation in our technique. Note that the cache state usually reflects the state of current *working-set* of the execution. The key observation is that the unknown cache state in our approach is in inter-loop intervals (where the working set has been largely changed), whereas in other time-sampling the cache state is completely unknown to its following references. As a result, we may expect the effect of *cold-start problem* is very minor in our approach.

Table 4. Cold start misses of loop sampling

	# of inter-loop	unknown refer	cold start miss	ratio
nasa	5	122	0	0.0
tomcatv	500	37893	23900	63. %
matrix	8	227	0	0.0
fpppp(1)	1	106	0	0.0
fpppp(2)	617	500	86	17. %

To see the effect of cold-start cache, we gather the number of references in inter-loops which are mapped to a cache set in a unknown state. We also compare the same reference to a full trace simulation to check whether the reference contributes a real miss in the full simulation. Table 4 shows results of cold-start effect. The column labeled "# of inter-loop" indicates the number of inter-loop intervals in the simulation. Only references in the inter-loop interval will introduce cold-start problems. In general, the count of the inter-loop intervals is very few and the corresponding number of references is even small. The column labeled "cold-start miss" shows that the real misses due to unknown cache state are very rare, with only 0% up to 17%, except Tomcatv. In the program, the number of unknown references is relatively large, resulting in significant misses in inter-loop intervals. However, this bias due to the unknown cache state is still relative small, when compared to the overall data references (as shown in Table 3).

6 Conclusion

In this paper, we focus on the development of effective simulation techniques, namely simulation by sampling in order to reduce the space and time requirements for simulating large caches. We have proposed two new sam-

pling techniques to attack the time and space problems in trace-driven simulation.

We propose a space sampling technique (i.e., set sampling). The basic idea is to quantify the locality and to divide the entire cache sets into "similar" groups (called *strata*). The sampling is done by a random choice within similar groups based on a given ratio, that is, by *stratified sampling*. The overall reference traces can be significantly reduced without generating too much error. After that, we give a new time-sampling technique which performs sampling based on the loop iterations. The basic assumption is that the behavior of loop iterations is similar so that sampling on loop iterations may give representative performance metrics for the loop execution.

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, November 1988.

[2] A. Agarwal, M. Horowitz, and J. Hennessy. An analytical cache model. *ACM Transactions on Computer Systems*, 7(2):184--215, May 1989.

[3] A. Agarwal and M. Huffman. Blocking: exploiting spatial locality for trace compaction. In *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 48--57, 1990.

[4] A. Agarwal, R. L. Sites, and M. Horowitz. ATUM: a new technique for capturing address traces using microcode. In *Proc. of the 13th Annual Intl. Symp. on Computer Architecture*, pages 119--127, New York NY (USA), June 1986.

[5] J.-L. Baer and W.-H. Wang. On the inclusion properties for multi-level cache hierarchies. In *Proc. of the 15th Annual Intl. Symp. on Computer Architecture*, pages 73--80, 1988.

[6] T. Ball and J. R. Larus. Optimally profiling and tracing programs. Technical Report #1031, Computer Science Department, University of Wisconsin - Madison, 1991.

[7] A. Borg, R. E. Kessler, and D. W. Wall. Generation and analysis of very long address traces. In *Proc. of the 17th Annual Intl. Symp. on Computer Architecture*, pages 270--281, May 1990.

[8] E. K. Foreman. *Survey Sampling Principles*. Marcel Dekker, Inc., 1991.

[9] J. W. C. Fu and J. H. Patel. Trace driven simulation using sampled traces. In *Proc. of Hawaii International Conf. on System Science*, pages 211--220, 1994.

[10] P. Heidelberger and H. Stoe. Parallel trace-driven cache simulation by time partitioning. Technical Report 68960, IBM Research Report, 1990.

[11] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, San Mateo, CA, 1990.

[12] R. E. Kessler, M. D. Hill, and D. A. Wood. A comparison of trace-sampling techniques for multi-megabyte caches. Technical report, University of Wisconsin, September 1991.

[13] S. Laha, J. A. Patel, and R. K. Iyer. Accurate low-cost methods for performance evaluation of cache memory systems. *IEEE Transactions on Computers*, 37(11):1325--1336, November 1988.

[14] J. R. Larus. Abstract execution: A technique for efficiently tracing programs. Technical Report TR-912, University of Wisconsin - Madison, 1990.

[15] G. Lauterbach. Accelerating architectural simulation by parallel execution of trace samples. In *Proc. of Hawaii International Conf. on System Science*, pages 178--182, 1994.

[16] L. Liu and J.K. Peir. Sampling of cache congruence classes. In *Intl. Conf. on Computer Design: VLSI in Computers and Processors*, pages 552--557, 1992.

[17] L. Liu and J.K. Peir. Cache sampling by sets. *IEEE Transactions on VLSI Systems*, 1(2):98--104, 1993.

[18] T. R. Puzak. *Analysis of Cache Replacement Algorithms*. PhD thesis, University of Massachusetts, 1985.

[19] A. D. Samples. Mache: Non-loss trace compaction. In *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 89--97, 1989.

[20] A. J. Smith. Two methods for the efficient analysis of memory address trace data. *IEEE Transactions on Software Engineering*, 3(1):94--101, January 1977.

[21] W.-H. Wang and J.-L. Baer. Efficient trace-driven simulation methods for cache performance analysis. In *Procs of the Conference on Measurement and Modeling of Computer Systems*, pages 27--36, 1990.

[22] T. Welch. A technique for high performance data compression. *Computer*, 17:8--19, June 1984.