Effectiveness of Trace Sampling for Performance Debugging Tools

Margaret Martonosi and **Anoop Gupta**
Computer Systems Laboratory
Stanford University, CA 94305

Thomas Anderson
Computer Science Division,
Univ. of California, Berkeley, CA 94720

Abstract

Recently there has been a surge of interest in developing performance debugging tools to help programmers tune their applications for better memory performance [2, 4, 10]. These tools vary both in the detail of feedback provided to the user, and in the runtime overhead of using them. MemSpy [10] is a simulation-based tool which gives programmers detailed statistics on the memory system behavior of applications. It provides information on the frequency and causes of cache misses, and presents it in terms of source-level data and code objects with which the programmer is familiar. However, using MemSpy increases a program's execution time by roughly 10 to 40 fold. This overhead is generally acceptable for applications with execution times of several minutes or less, but it can be inconvenient when tuning applications with very long execution times.

This paper examines the use of trace sampling techniques to reduce the execution time overhead of tools like MemSpy. When simulating one tenth of the references, we find that MemSpy's execution time overhead is improved by a factor of 4 to 6. That is, the execution time when using MemSpy is generally within a factor of 3 to 8 times the normal execution time. With this improved performance, we observe only small errors in the performance statistics reported by MemSpy. On moderate sized caches of 16KB to 128KB, simulating as few as one tenth of the references (in samples of 0.5M references each) allows us to estimate the program's actual cache miss rate with an absolute error no greater than 0.3% on our five benchmarks. These errors are quite tolerable within the context of performance debugging. With larger caches we can also obtain good accuracy by using longer sample lengths. We conclude that, used with care, trace sampling is a powerful technique that makes possible performance debugging tools which provide *both* detailed memory statistics *and* low execution time overheads.

1 Introduction

Modern computers exhibit an increasingly wide gap between processor and memory speeds. With increases in processor clock rates outpacing improvements in memory speeds [6], memory system performance has become a significant bottleneck to achieving good overall application performance. Cache misses on typical current generation uniprocessors can incur delays of tens of processor cycles, and in multiprocessor machines, remote memory latencies can be hundreds of cycles. For an application to achieve good performance, it must use locality to exploit the memory hierarchy effectively.

The application programmer has considerable flexibility in tuning the program for better memory system performance. However, tuning the memory behavior of large programs is a complex task requiring detailed information on the program's access patterns. Some performance monitoring systems (such as MTOOL [3, 4]) give only *code oriented* information indicating the amount of memory overhead in particular loops or procedures. This information is useful for initial queries about application behavior; however, it is often not detailed enough to help the user fix the application's performance bottlenecks. Detailed statistics summarizing the frequency and causes of cache misses, as well as the behavior of different data structures in the code, are extremely useful in understanding and fixing memory bottlenecks. Generally these detailed statistics can be gathered in one of two ways, using (i) hardware performance monitors or (ii) software simulation. Software simulation is the more widely applicable of these approaches, because most machines do not provide support for hardware performance monitoring. Furthermore, software simulation allows programmers to evaluate application performance with different cache sizes and speeds.

Unfortunately, detailed software simulation can often be quite slow. However, careful attention to the simulation environment can reduce simulation overhead, making detailed software simulation fast enough to be part of an interactive performance debugging tool. An example of this is MemSpy [10]. MemSpy gives programmers detailed, data-oriented information to help locate and fix memory bottlenecks in their code. Its output presents users with statistics in terms of source level data objects, as well as code objects. MemSpy's simulation overhead is approximately a factor of 10 to 40 times the actual execution time of the program. For applications with relatively short run times (up to a few minutes) this overhead is often acceptable, given the utility of the data it produces. For applications with longer run times, several approaches are available to reduce the simulation time. These include scaling data sets and cache size, or for iterative algorithms, reducing the number of iterations studied. While these approaches are often useful for certain classes of problems, we believe that the complementary approach of trace sampling is a promising and more general method for reducing simulation overhead.

Reference trace sampling is the technique of simulating only randomly chosen portions of a reference trace, rather than simulating the full trace. Intuitively, this promises significant speedup,

In: *Proc. 1993 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*

since one incurs the full simulation overhead only on a fraction of the full reference stream. If one samples such that only one tenth of the references are simulated, one can hope for a speedup of up to a factor of ten. Our results show that significant performance improvements can be obtained using sampling within MemSpy. When simulating one tenth of the total references, we get 4 to 6 fold speedups compared to non-trace-sampled MemSpy. For the benchmarks studied here, this reduces MemSpy's overhead to a factor of 3 to 8. With execution time overheads in this range, MemSpy's performance becomes competitive with other tools [4] that present less detailed statistics.

This paper makes several main contributions. As previously stated, we show that within the context of a performance debugging tool, reference trace sampling can be used effectively to improve the tool's performance. We go on to present results showing how the accuracy of the sampled results varies with key parameters such as number of samples, sample length, and cache size. We find that the parameter settings required for accurate sampled output do not excessively limit performance. For example in 16KB and 128KB caches, when simulating about one tenth of the total references in about 20 samples of 0.5M references each, the absolute errors in cache miss rate never exceed 0.3%. Larger caches (e.g., 1MB) can obtain good accuracy by using longer samples (4 million to 8 million references each), while continuing to only sample one tenth of the references total. Programs running with larger caches will generally make more total data references, in order to make use of the cache, so we do not consider individual samples of this length to be a serious restriction. Finally, we present performance results which show the success of this approach, and present suggestions for further optimizations.

The paper is structured as follows. Sections 2 and 3 give background information on trace sampling and on the MemSpy tool. In Section 4, we describe our sampling implementation within MemSpy. Section 5 describes our architectural assumptions and benchmark applications, before presenting results in Section 6. Section 7 discusses possible extensions to our current approach to incorporate *set sampling*, as well as time sampling, and to support sampling in multiprocessor simulations. We present our conclusions in Section 8.

2 Trace Sampling Background

MemSpy implements reference trace sampling by gathering evenly spaced samples to be simulated from the full reference trace. This is illustrated in Figure 1. The *sampling ratio* is the ratio of the total number of references within the samples, divided by the total number of references. Thus, the performance improvement possible in a sampling system is limited by the reciprocal of the sampling ratio. For example, if one uses a sample length of 0.5M references and a sampling interval of 5M references, the sampling ratio would then be 1/10, and one could not expect more than a factor of 10 speedup over full simulation.

Reference trace sampling is subject to two types of inaccuracies. First, the samples gathered may not be representative of the full trace. This error is common to all forms of sampling, including the program counter sampling already used in a number of performance monitoring systems [1, 3, 4, 5]. This sort of error can generally be controlled by taking a number of samples over the course of the program's execution. Section 6.1 discusses the relationship between accuracy and the number of samples taken.

The second inaccuracy occurs because the state of the cache is

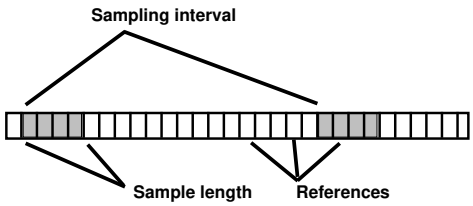

Figure 1: Samples in reference trace.

unknown at the beginning of each sample (because the cache's true state depends on events which have occurred during the unmonitored section of the program.) Therefore, within each sample, the first reference to each cache line (for a direct-mapped cache) is an *unknown reference* that could either be a hit or a miss. Estimating the miss ratio of these unknown references can be a source of inaccuracy when using reference trace sampling. Because anywhere from none to all of the unknown references could actually be misses, miss rate calculations for the sample depend both on the number of *known misses* and the number and miss rate of the unknown references. By using longer sample lengths, we can make the number of unknown references less significant when compared to known misses. Section 6.3 will discuss the relationship between sample length and the accuracy of miss rate statistics.

Several studies have examined the applicability of sampling in the context of architectural studies of cache performance. Laha et al. [8] studied the accuracy of memory reference trace sampling using caches that were 128KB in size and smaller. Their study concludes that sampling techniques allow accurate estimates of the miss rate for caches of this size. However, their results were presented with sample lengths of 60,000 references every 100,000 references, or a sampling ratio of 0.6. Sampling ratios in this range offer little promise for improved performance. Furthermore, they presented simulation data only for caches with 1024 lines or fewer, so sample lengths of 60,000 were in general long enough to prime the cache.

Wood et al. [15] developed a model for estimating the miss rates of unknown references in sampled traces. They found that their model predicted the behavior of unknown references better than several previous methods [8, 14]. Most importantly, they point out that unknown references typically have miss rates significantly higher than the application's steady state miss rate. We will evaluate the effectiveness of the Wood et al. estimators on our benchmarks.

In later work, Kessler et al. [7] studied trace sampling for large secondary caches of 1MB to 16MB, and with sampling ratios down to 1/10. For their benchmarks, they noted that unknown references at the beginning of a sample can dominate known misses, potentially making miss rate estimates quite inaccurate. We believe this large inaccuracy arose in part because the benchmarks they used had few cache misses. (Seven of the eight traces had misses per instruction (MPI) values less than 0.003, while one had an MPI of roughly 0.02[1]. With a cache miss latency of 20 cycles, an MPI of 0.003 corresponds to spending only 6% of program runtime in memory stalls.) Thus, for these benchmarks, the total number of known misses in each sample was often very small.

From a performance debugging point of view, it is not important to pinpoint with high accuracy the miss rate of applications with very low miss rates. The primary purpose of MemSpy is to tune applications with poor memory system behavior. These programs will typically have higher miss rates; their larger proportion of

[1] [7] does not report cache miss rates directly.

known misses to unknown references will allow miss rates to be estimated more accurately. Also, within the context of a performance debugging tool (as opposed to more forward looking architectural studies), it is still quite relevant to focus on smaller caches than those used by Kessler et al. With smaller (e.g., primary, on-chip) caches, sampling's accuracy improves. Finally, within the context of performance debugging, we can tolerate slightly larger errors in the statistics generated. To tune an application, one needs to have a good idea of the location and magnitude of the bottlenecks, but not a perfectly precise cache miss rate value. We will show that trace sampling reproduces MemSpy's statistics with quite acceptable accuracy in most cases.

For these reasons, we feel that trace sampling is a promising approach to reducing the execution time of performance debugging tools like MemSpy. In this paper, our goal is not simply to reiterate the results given by Laha et al. or Kessler et al., but to reevaluate the efficacy of trace sampling in a new context. In this new context, performance debugging, our constraints on the accuracy of results may be somewhat looser, while the performance savings offered by sampling are of central importance.

3 MemSpy Background

This section gives a brief summary of MemSpy's features in order to understand how trace sampling will affect their accuracy.

MemSpy is a performance debugging tool designed to help programmers locate and fix memory bottlenecks in applications. MemSpy first helps in *locating* bottlenecks by providing high level information to guide the user towards potential memory bottlenecks. This high level information (illustrated by the MemSpy output in Figure 2) breaks down the total execution time of the program by procedures. Within each procedure, MemSpy presents a breakdown of how much time was spent in memory stalls due to cache misses, versus how much time was spent in computation.

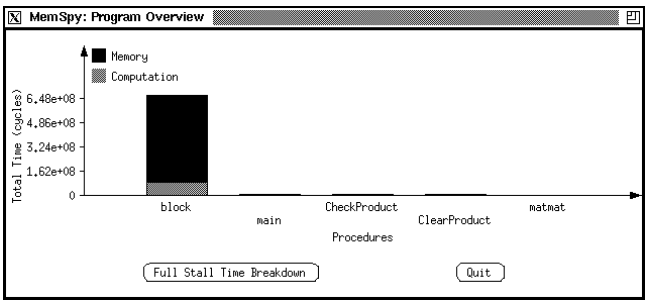

Figure 2: Initial MemSpy output display.

MemSpy can then help in *fixing* bottlenecks by providing more detailed, low-level information on source-level code and data objects. For these MemSpy statistics, data objects correspond roughly to particular classes of data in a program, and code objects correspond to procedures. For each data object in each procedure, MemSpy provides information on the number of cache misses, the percentage of the total memory stall time incurred, and the causes of the cache misses. An example of this sort of information is shown in Figure 3.

MemSpy has been used to tune a number of applications, as described in [10]. The detailed, data-oriented information given by

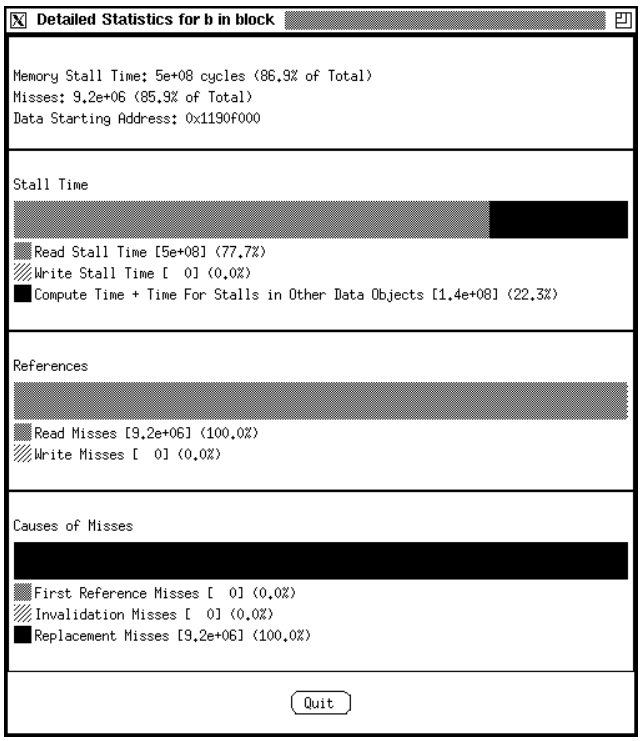

Figure 3: Detailed MemSpy output display.

MemSpy has been instrumental in understanding and fixing a variety of performance bugs, such as interference between different data structures in the cache and excess communication in parallel programs. However, collecting this information comes at the cost of higher execution time overhead than tools which provide only high level statistics [3, 4]. Depending on the reference rate and miss rate of the application, the MemSpy overhead is typically a factor of 10 to 40 times the actual execution time of the application. Table 1 gives the overheads for running the original MemSpy for the applications used in this study, when simulating a 128KB cache[2]. The runs were performed on a DECstation 5000/240 with 80 megabytes of memory. Section 5.2 gives more detailed information on the characteristics of each benchmark.

Table 1: Original MemSpy overheads. (128KB cache)

Application	Actual Exec. Time (sec.)	MemSpy Exec. Time (sec.)	Overhead Factor
MP3D	41.7	954.6	22.9
MATMAT	66.1	998.7	15.1
TRI	73.5	782.1	10.6
ESPRESSO	23.5	866.0	36.9
PTHOR	16.0	672.8	42.1

To understand this performance overhead, we examine the sequence of operations MemSpy uses to simulate a memory reference. At compile time, each memory reference in the original application assembly code is instrumented with a procedure call to

[2]Note that these baseline overheads already represent an improvement over those presented in [10].

the memory system simulator. At run time, within this procedure, MemSpy performs the following actions:

1. Save application registers so that the memory system simulator will not destroy them.
2. Use a cache simulator to determine whether the reference is a cache hit or a cache miss.
3. Update MemSpy statistics for the relevant code and data objects.
4. Restore application registers to their original values.
5. Return control to the application.

Cache simulation and statistics updates can take from roughly 20 cycles (if the reference is a cache hit), to roughly 200 cycles (if the reference is a cache miss). Saving and restoring application registers comprise roughly 60 cycles. Even in applications with poor memory behavior, generally a majority of the references are still cache hits, so we find the bulk of MemSpy's overhead is spent in saving and restoring registers when switching from application to MemSpy and vice versa. While we are currently working on techniques to reduce this overhead, this paper focuses on using trace sampling as an orthogonal technique for improving MemSpy's performance.

4 Implementation of Trace Sampling

Our discussion of the trace sampling implementation within MemSpy is divided into two subsections. The first deals with performance issues in the implementation, while the second deals with issues related to the accuracy of the results.

4.1 Performance Issues

Obviously, our primary purpose in using trace sampling is to improve MemSpy's performance. Therefore, it is important to produce a sampling implementation which promises substantial performance improvements over the current full-trace simulation. We have seen that the bulk of MemSpy's time is spent in the register saves and restores required when switching from application to the memory simulator and vice versa. So, an implementation which simply turns off simulation *within* the MemSpy simulator will not improve performance significantly. Rather, to be worthwhile the implementation must circumvent the overheads of register saves and restores whenever the simulation is turned off.

To accomplish this, we modify the normal MemSpy assembly time instrumentation of memory references. In addition to the usual call to the MemSpy memory simulator, additional instrumentation is added. In this extra instrumentation, a sampling counter is decremented and checked against zero to see if simulation is currently ON or OFF. If simulation is OFF, control branches around the memory simulator procedure call. If simulation is ON, the simulator is called as in a full simulation. Figure 4 illustrates the original and new instrumentation.

Section 6.6 will show that this implementation gives us a sizable performance improvement. A sampling ratio of 1/10 leads to 4 to 6 fold performance improvements. Section 6.6 also discusses ways of further reducing this overhead.

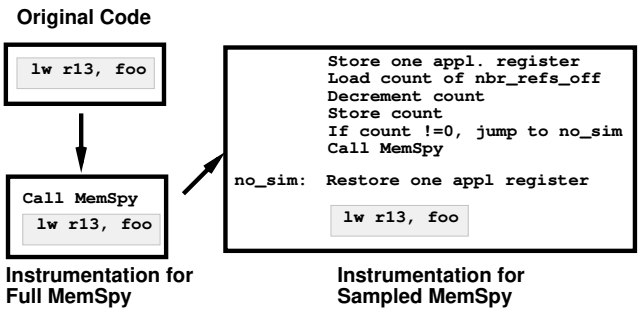

Figure 4: Inlined assembly code for sampling.

4.2 Accuracy Issues

As we noted earlier, trace sampling is subject to two orthogonal forms of inaccuracy:

Error due to non-representative samples: The deviation between the application's true miss rate when fully simulated, and the application's true miss rate measured during sampled regions only.

Error due to unknown references: Within each sample, the deviation between the application's estimated miss rate (including an estimate of the miss rate of unknown references), and the application's true miss rate *during the sampled region*.

For the results presented in this paper, the true miss rate during sampled regions is generally a good estimate of the application's true overall miss rate. The error in this estimate can be controlled by increasing the number of samples taken. The bulk of the error we measure is of the second type: error due to unknown references. For a particular sample, the cache miss rate, m, can be expressed as

$$m = \frac{M_k + \mu U}{H_k + M_k + U}$$

where H_k is the number of *known hits*, M_k is the number of *known misses* in the sample, U is the number of unknown references, and μ is the fraction of unknown references which are actually cache misses. Knowing H_k, M_k, and U, we can estimate m by estimating the unknown reference miss rate, μ. Wood et al. [15] show that miss rates for unknown references are typically higher than the overall application miss rate, so assuming that μ is equal to the steady state miss rate will result in optimistic performance estimates. They introduce a method for estimating μ by estimating the fraction of time that lines in the cache are dead (i.e. will not be referenced again before a new line replaces them in the cache). In Section 6.4, we will evaluate this method's effectiveness within our framework. Note that one can always compute a range of possible m' values by allowing μ to vary from 0 to 1. Thus, m' can be expressed with symmetric error bounds as follows:

$$m' = \frac{M_k + 0.5U}{H_k + M_k + U} \pm \frac{.5U}{H_k + M_k + U} \quad (1)$$

In general, one can reduce the error due to unknown references by lengthening samples so that known misses dominate unknown references. However for a fixed desired sampling ratio, increases in sample length must be traded off against the number of samples taken. Sections 6.1 and 6.3 discuss this tradeoff in more detail.

Because we are studying trace sampling within the MemSpy context, it is important to also study how errors in cache miss rate

translate into errors in the statistics reported by MemSpy. Will errors in cache statistics for individual data structures, as well as overall cache statistics, be acceptable? Will the sampling approach be accurate enough to allow the program's true bottlenecks to be identified? Section 6.5 discusses the errors in MemSpy metrics which we have observed for the benchmarks studied.

5 Experimental Setup

This section briefly presents necessary background information on the architectures and the benchmark applications we examined.

5.1 Architectures Simulated

The statistics shown here were gathered using a very simple memory simulator that does not model network contention. While the model is simple, we have found that it captures most of the important memory system behavior in applications used with MemSpy. In these studies, cache are direct mapped, and have one of the following organizations: (i) 16KB, with 16 byte lines, (ii) 128KB, with 32 byte lines, or (iii) 1MB, with 64 byte lines. The 16KB cache is a typical size for an on-chip first level cache. The 128KB cache represents a reasonable size for an off-chip cache. The 1MB cache, large by today's standards, might be used as a secondary cache.

5.2 Benchmark Applications

The results of trace sampling studies are clearly dependent on the applications used for gathering the results. For our benchmarks, we have found that sampling provides significant execution time benefits with little sacrifice in accuracy. For tuning the memory behavior of these applications, the sampled results are in most ways interchangeable with the full-trace results.

The applications presented here span a wide range of memory behaviors. Two of the benchmarks, MP3D and PTHOR, are applications from the SPLASH benchmark suite [12]. Their memory system performance has been studied extensively in the past. Likewise, the memory referencing characteristics of MATMAT [9] and TRI [11] have also been studied. (We use untuned versions of these applications, to observe behavior typical of applications one might use with MemSpy.) The fifth application, ESPRESSO, from the SPEC benchmark suite [13], has much better memory performance than the others (0.2% cache miss rate on a 128KB cache). This application is useful for showing how sampling behaves on applications with very low miss rates. Of the five, it is the most sensitive, in terms of relative error, to changes in the length of individual samples and the particular μ estimate chosen. Table 2 gives some basic information on each of the applications studied, including their cache miss rates for a 128KB cache.

6 Results

This section presents our results on the accuracy and performance of trace sampling within MemSpy. In general, we find that trace sampling is quite effective at reproducing the statistics from a full trace run of MemSpy. As a preview, Table 3 compares estimated cache miss rates from sampling to (i) the program's true miss rate calculated over all references and (ii) the program's true miss rate

Table 2: Application Characteristics

Application	Data Set Size (MB)	Refs (M)	True Miss Rate (%)
MP3D: Hypersonic Particle-based Simulator	33.4	150.4	3.8
MATMAT: Blocked Matrix Multiply	6.4	139.7	18.2
TRI: Triangular Sparse Matrix Solver	68.2	109.8	6.1
ESPRESSO: Boolean Function Minimizer	1.3	143.8	0.2
PTHOR: Discrete Logic Simulator	4.6	67.2	4.0

calculated over only those references occurring during a sample.[3] In each of the applications, the absolute error in miss rate never exceeds 0.4%. Errors in this range are generally acceptable for use in a performance debugging context. MemSpy's execution time overhead for these applications generally ranged from factors of 3 to 8. This overhead makes MemSpy an attractive alternative to other less detailed performance monitoring tools.

Table 3: Estimated and True Miss Rates. (128KB cache, 10% sampling ratio, 0.5M Refs/sample)

Appl.	Overall True Miss Rate	True Miss Rate During Samples (%)	Estimated Miss Rate During Samples (%)
MP3D	3.8	3.8	3.5
MATMAT	18.2	17.9	17.8
TRI	6.1	6.4	6.1
ESPRESSO	0.23	0.14	0.20
PTHOR	4.0	3.8	3.7

While this data shows reasonable accuracy for trace sampling in one configuration, a more general evaluation of trace sampling for performance debugging must examine several issues. The most important questions to be answered are:

- How does the number of samples taken affect the error due to non-representative samples?
- How does the accuracy vary with cache size?
- How does the length of each sample affect the error due to unknown references?

[3] Except when stated otherwise, all estimated miss rates in this section are reported using a μ of 0.5 as in Equation 1. This allows us to separate the issue of μ estimation from the other issues being studied.

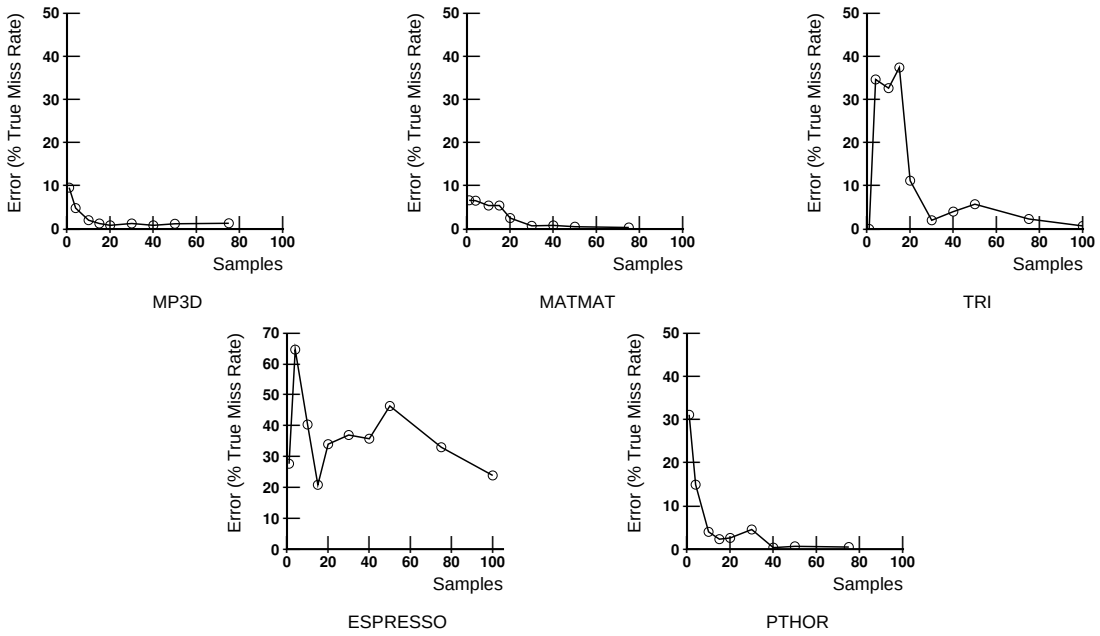

Figure 5: Relative Error vs. Number of Samples. (128KB cache, 0.5M Refs/sample)

- How well can we estimate the miss rate for unknown references?
- How does sampling affect specific MemSpy output, such as the ordering of memory bottlenecks and statistics on the causes of cache misses?
- Subject to constraints on accuracy, what performance improvement can we expect when incorporating sampling into MemSpy?

To some extent, these questions are all interrelated. The performance gains realized from trace sampling are bounded by the sampling ratio, which is the product of the number of samples taken during a run, multiplied by the sample length, and divided by the trace length. However, choosing the sample length and the number of samples has implications for the accuracy of the resulting run as well. The cache size being simulated is also a strong determinant of sampling accuracy, with implications on the required sample length. We must also examine how these errors in the overall miss rate translate into errors in MemSpy metrics. If we understand each of these trends and tradeoffs, we can decide when, and to what extent, additional sampling error may be accepted in exchange for better performance.

6.1 Accuracy vs. Number of Samples

The number of samples taken partly determines how representative the trace will be of the overall program performance. Intuitively, a single large sample will not reproduce the overall program's behavior as well as several smaller ones. Program behavior can vary over the run time of the program, with some phases (such as initialization) characterized by very poor memory system behavior while other phases have much better cache performance. Laha et al. [8] mention the importance of capturing representative samples and present data indicating that using 35 samples was generally sufficient to characterize the miss rates for their Lisp benchmarks. Here, we present more comprehensive data showing how accuracy varies across a wide range of values for number of samples. To do this, we fix the total number of references simulated and vary the number of samples taken. This allows us to study the effect of changing the number of samples, while holding the sampling ratio (and therefore performance) constant. To study the representativeness of different collections of samples, we examine the deviation between the true miss rate during sampled regions, and the application's overall true miss rate. (This is the error due to non-representative samples mentioned in Section 4.2.) Note that we are *not* studying the effects of unknown references in this section.

Figure 5 presents the relative deviation between the true miss rate during sampled regions and the overall true miss rate, plotted against the number of samples taken. (In this case, the sampling ratio was 1/10.) Three of the applications: MATMAT, MP3D, and PTHOR, show excellent behavior. Even after as few as 10 samples, the relative error in these three cases is roughly 5% or less. Except for initialization periods, MP3D, MATMAT, and PTHOR are characterized by relatively constant memory behavior throughout their program. For these types of programs, fewer samples are required to capture the "typical" memory behavior. A fourth program, TRI, shows large relative fluctuations in error for less than 20 samples, but settles down with greater numbers of samples. Comparing the true miss rate over time for TRI with that of MP3D (Figure 6), we see that TRI's miss rate fluctuates much more with time than that of MP3D. Thus, more samples are needed to capture its behavior. Finally, the fifth application, ESPRESSO, shows relative errors greater than 20% even at 100 samples. Espresso has a very low miss rate, so even tiny fluctuations in its absolute miss rate over time show up as large relative swings. When an application has so few known misses over its entire execution, the *relative* error due to sampling is higher. However, the absolute error from sampling may still be quite small. Low-miss-rate applications typically do not require memory system tuning at all, so MemSpy's performance on them is less relevant.

As expected, the presence of phases in the memory behavior of a program (such as TRI) mandates the use of more samples to accurately represent its memory behavior. Here MemSpy can leverage off users' knowledge of how their programs behave. When

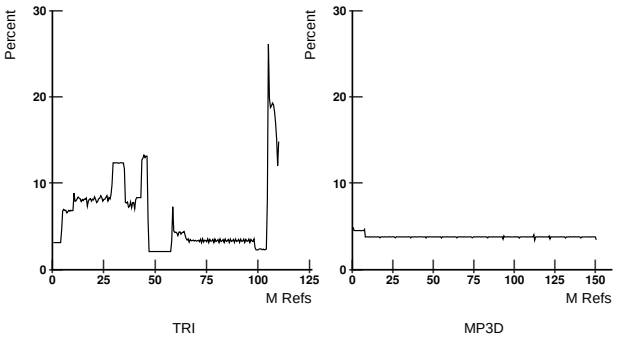

Figure 6: Overall True Miss Rate vs. Time. (128KB Cache)

programmers know that their application has basically constant memory behavior, they can request that a reduced number of samples be taken, for higher performance. While users do not want *full* control of the sampling setup, directives like these allow them to speed up the tuning process in specific cases. Furthermore, since tuning is iterative, users can choose to have only a few samples collected in early runs, and then simulate a higher fraction of references as they move to more detailed tuning.

6.2 Accuracy vs. Cache Size

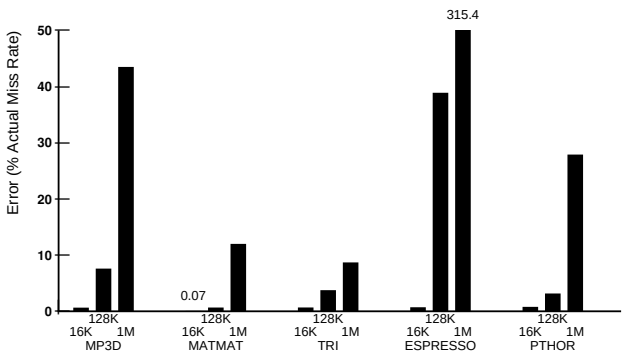

Figure 7: Relative Error vs. Cache Size. (10% sampling ratio, 0.5M Refs/sample)

As previous studies have shown [7, 15], the accuracy of miss rate estimates is a strong function of the cache size used. Figure 7 shows the relative error in cache miss rate estimates for 16KB, 128KB and 1MB caches at a sampling ratio of 1/10, with a sample length of 0.5M references. Here, these errors are shown relative to the true miss rate *during the sample*. Within each sample, the best we can hope to achieve is to re-create the true miss rate of that sample. The error with respect to the overall miss rate may be slightly more or less than the error shown here, depending on how accurately the sampled regions capture the behavior of the full trace.

For the 16KB and 128KB caches, the relative errors are all less than 10%, with one exception. Espresso's miss rate has a *relative* error of nearly 40% for the 128KB cache. However, its absolute miss rate (.144% within the sampled regions) is so small that these large relative errors are neither surprising nor problematic, from a performance debugging point of view. Since ESPRESSO's memory stall time is a small fraction of its total execution time, even the sampled version of MemSpy will accurately point out that ESPRESSO does not have any substantial memory performance bottlenecks.

With 1MB caches, the relative error is greater than 10% for four of the five applications. The higher relative errors noticed here are due to two factors. First, as the cache size increases, more references are needed to prime the cache state. This causes the number of unknown references to increase. Second, as the cache size increases, the application's cache miss rate generally decreases. This causes a decrease in the number of known misses. From Equation 1, both of these effects tend to increase the size of the error bounds on estimated miss rate. To improve the miss rate estimates with large cache sizes, we have two options. First, we can reduce U's significance in the equation, by lengthening each sample taken. Alternatively, we can improve our estimate of μ, the miss rate for unknown references. The following subsections address these two issues. Note first however, that despite the large relative error, the absolute errors remain quite small. Figure 8 shows a plot for the 1MB cache of both the true cache miss rate (stars) and the sampled cache miss rate (circles) with error bounds as expressed in Equation 1. Absolute error of this magnitude is generally considered acceptable when tuning programs.

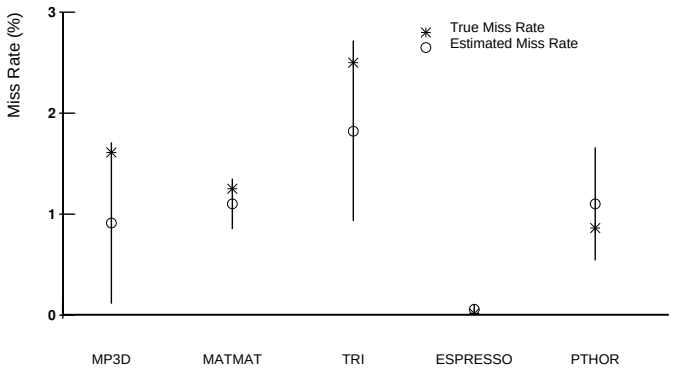

Figure 8: Actual and estimated cache miss rates. (1MB Cache 10% sampling ratio, 0.5M Refs/sample)

6.3 Accuracy vs. Sample Length

Section 6.2 illustrates the fact that a single choice of sample length may not work effectively across a range of cache sizes and application behaviors. For larger caches, unknown references become significant, and one must use longer samples to mitigate their effect.

To gather the data shown here, we divide the trace into contiguous samples, and collect data for all of them, averaging the results. This allows us to study more samples per application than if we restricted ourselves to a sampling ratio of, say, 1/10. The errors are shown relative to the true miss rate during the sampling region.

Figure 9 shows relative errors in miss rate estimates for the five benchmark applications with a 1MB cache. As expected, longer samples dramatically improve accuracy. At 8M references per sample, all applications have relative errors less than 10%. Absolute errors on these applications are all under 0.1%. For even moderately long running applications, 8M references per sample is not a prohibitively long sample length. To collect 50 samples of this size with a sampling ratio of 1/10, the total application would need to have 4 billion data references. Assuming one data reference every 3 instructions, this would be roughly 12 billion

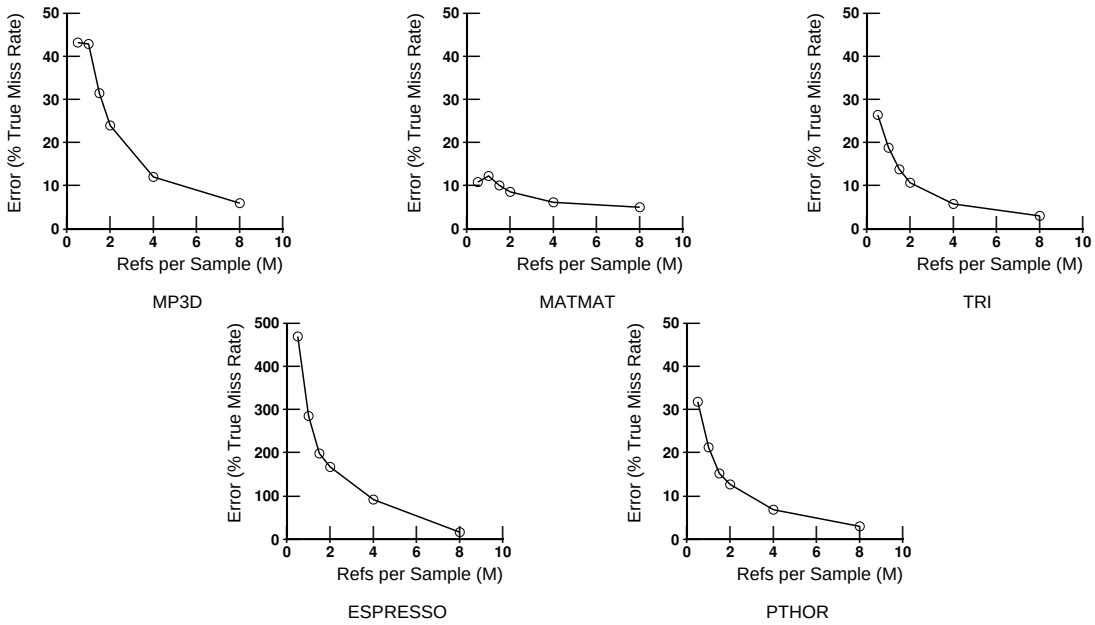

Figure 9: Relative Error vs. Sample Length. (1MB cache)

instructions, or about 2 minutes of execution time on a 100 MIPS machine. We feel that these are not prohibitive requirements on either the run time or the reference rate of a program.

6.4 Miss Rate of Unknown References

Section 6.3 shows that reducing error by using longer samples works quite well. However, many interesting applications do not have run times long enough to use long sample sizes *and* aggressive sampling ratios. For this reason, we now evaluate a proposed model for estimating the miss rate of unknown references, μ, in Equation 1.

Wood et al. [15] developed a renewal theoretic model for estimating the miss rates of unknown references. The model predicts μ based on two characteristics. (We refer to their estimate as μ'.) The first characteristic P_{dead}[4] is the ratio of the amount of time a cache line holds a memory line which will not be referenced again (*dead time*) divided by the total time that memory line spent in the cache (*generation time*). This ratio approximates the probability that the unknown references will be directed at "dead" lines in the cache, and will therefore be cache misses. The second characteristic used in this model is the number of cache lines referenced during a sample. This accounts for the fact that many short and even moderate length samples will not reference all lines of the cache. Thus not all the dead lines estimated by P_{dead} will be referenced.

Column three of Table 4 shows the estimates of μ that are obtained by calculating "true" dead times and generation times based on *all* references in the trace (not just references within the sampled regions). However with sampling, it is not realistic to monitor dead and generation times on *all* references, so the rightmost column shows the same μ estimator calculated using only dead times and generation times for generations which were contained within simulation ON periods. A generation is defined to start and end with a cache miss. Thus, within a sampling system, collecting generation data requires that *two known misses* occur to a particular cache line within a particular sample. With low miss rates and short samples, many cache lines do not meet this requirement, so this model does not measure their behavior. Thus the model is inaccurate in two ways. First, the model is skewed towards counting statistics on shorter generations which can fit within the sample. Second, statistics for cache lines which have such generations are applied to other cache lines as well. For MP3D and PTHOR, the error in the sampled μ' is especially pronounced. The error with ESPRESSO is also large, because it has very few generations total. Here, μ' severely overestimates μ, because cache lines with poor behavior are more likely to be counted than those with good behavior. Intuitively, we see that the accuracy of the Wood et al. model improves when there are more known misses in a sample. However, it is exactly under these conditions that unknown references, U, are small with respect to known misses. When M_k is less dominant, μ simultaneously becomes more important to relative error and less accurate.

We can use a heuristic metric to evaluate the degree of error in μ' by computing the average number of known generations per sample. This metric, G_k in Table 4, is computed by dividing the total number of generations in the sampled regions by the product of the number of samples and the number of cache lines. When there is one generation per cache line per sample, G_k will equal 1. When there are a large number of generations per cache line per sample, G_k will be greater than 1. Intuitively, we expect μ' to

Table 4: Accuracy of μ estimators. (1MB Cache, 0.5Mrefs/sample)

Application	Actual μ	μ' (All Refs)	μ' (Sampled)	G_k
MP3D	.966	.952	.637	2.8
MATMAT	.880	.882	.854	21.3
TRI	.877	.903	.942	11.5
ESPRESSO	.116	.823	.990	0.04
PTHOR	.267	.205	.468	3.3

[4] [15] referred to this as μ_{long}.

be more accurate when a large number of generations were used in calculating it. For our benchmarks, G_k increases monotonically with the absolute deviation of μ' from μ, indicating it is effective in gauging the deviation in the estimate.

Table 5: Using μ' in miss rate calculations. (1MB cache, 0.5M Refs/sample)

Application	Actual Miss Rate	Using μ'		Using $\mu = 0.5$	
		Miss Rate	Error (%)	Miss Rate	Error (%)
MP3D	1.61	1.21	-24.8	.92	-42.9
MATMAT	1.30	1.29	-0.8	1.14	-12.3
TRI	2.38	2.42	1.7	1.93	-18.9
ESPRESSO	.011	0.08	627	0.04	263.6
PTHOR	.92	1.09	18.5	1.11	20.7

Table 5 shows a summary of miss rate estimations based on μ' and the deviation from the true miss rate value. Not surprisingly, in spite of its inaccuracies, μ' performs better than more simplistic models, such as assuming $\mu = 0.5$. For ESPRESSO, however, it overestimates the miss rate for unknown references, which leads to a large absolute error in the estimated miss rate.

We feel that μ' estimation within the MemSpy system could be improved further. A major feature of the MemSpy tool is the presentation of statistics in terms of data objects, as well as code objects. Thus, within that framework, it would be quite natural to keep dead time and generation statistics on individual data structures. These could be used to better estimate the behavior of unknown references to these data structures.

Finally, although an accurate estimation of μ would be helpful in some situations, the ability to provide error bounds around the miss rate estimate makes the accuracy of the estimation less crucial.

6.5 Error in MemSpy Metrics

Until this point, we have presented our results in terms of their effect on the overall cache miss rate. However, MemSpy presents more detailed statistics than just the cache miss rate; we now examine the sensitivity of these statistics to inaccuracies introduced by sampling.

The key statistics presented by MemSpy fall into three basic categories. First, MemSpy presents displays which show memory stall time as a percentage of total execution time (memory stall time plus compute time) for different code and data objects in the application. This allows the user to determine if memory behavior is a significant bottleneck or not. Because of sampling, both the memory stall time (estimated cache miss rate multiplied by the latency of a cache miss) and the total execution time potentially can be inaccurate. However, we find that the sampling of compute time is in general quite accurate. Thus, the main error in the percentage memory stall time follows directly from the error in the cache miss rate. Relative errors in estimating this metric parallel the errors in cache miss rate estimates. The magnitude of these errors is largely determined by the cache miss latency and the relative magnitude of the compute time. When compute time is small, the error in this fraction will roughly match the error in the cache miss rate. However, the presence of a large compute time in the denominator of the expression: $Mem/(Compute + Mem)$ can decrease the sensitivity of this statistic to errors in miss rate.

Table 6: Memory bottleneck identification in PTHOR.

Data Object	Procedure	Memory Stall Time (%)	
		True	Sampled
Element Array	EvalElement	16	15
Element Array	StimFanOuts	11	12
Free List for GateChanges	EvalElement	9	9
Time_Valid	EvalElement	6	6

MemSpy's second type of statistic breaks down memory stall time by procedure–data pairings. In this way, MemSpy allows the user to see which data structures are most responsible for the program's poor memory performance. We can collect this bottleneck information from a full MemSpy run, and compare it to a sampled MemSpy run. For a 128KB cache, and a sampling ratio of 1/10, we find excellent agreement between the sampled and true statistics. For the four applications studied (we omitted ESPRESSO since the overall miss rate indicates it needs little tuning), the orderings of bottlenecks reported by the sampling version exactly matched the orderings for the true version through the top 90% of the memory stall time. As an example, Table 6 shows the bottleneck orderings, and percentage breakdowns for PTHOR.

MemSpy also presents a breakdown of the causes of a bin's cache misses. In sequential programs, a cache miss can either be a cold miss or a replacement miss. Cold misses occur when memory is referenced for the first time in a program. Replacement misses occur when data which was previously in the cache has been replaced by other data before it is re-referenced. Unfortunately, accurate reproduction of these statistics is more challenging. There are two possible types of errors which can cause these statistics to be inaccurate when sampling. The first error occurs if a cache line is referenced for the first time during a period when simulation is turned off. In this case, this cold miss will not be noted, so the first subsequent reference that occurs when simulation is on, will be counted as a cold miss. In reality, it was a replacement miss.

The second type of error pertains to correctly attributing the cause of a replacement miss. For each replacement miss, MemSpy records the data item which caused the replacement, so that it can give statistics on which data items caused replacement misses to other data items. If a cache line is pushed out of the cache during a period when sampling is off, then no statistics will be recorded indicating which data item pushed it out. If a subsequent replacement miss occurs, one of two things may happen. First, intervening references to that line may occur between when sampling is turned on, and when the replacement miss occurs. In this case, a `Cause` data structure is updated to indicate the first of these intervening references is the replacement cause. This information is correct in one sense, since these references would have caused a replacement miss as well; they are simply not the direct cause of replacement that would have been seen in the full trace. In the second case, if no intervening references occur, then the cause of replacement is considered to be unknown. In general, we have found that with severe interference, replacements occur often enough that the sampled version is able to detect and indicate the problem. As the users begin to to fine tune their code, they may choose to simulate a higher fraction of references, to detect the more subtle performance bugs.

6.6 Performance

Having presented statistics on the accuracy of a sampled version of MemSpy, we now evaluate its performance. The goal of implementing trace sampling within MemSpy is to reduce the execution time overhead needed to collect MemSpy statistics. Section 4 gave a description of our current sampling implementation. To reiterate, for every assembly level call to the MemSpy simulator, we add extra instrumentation which decrements a reference counter and branches around the simulator call if simulation is OFF. This introduces an overhead of 6 instructions per instrumented memory reference. When simulation is ON, there is additional overhead to (i) save application registers, (ii) switch to the simulator, (iii) simulate, (iv) restore application registers and (v) return to the application.

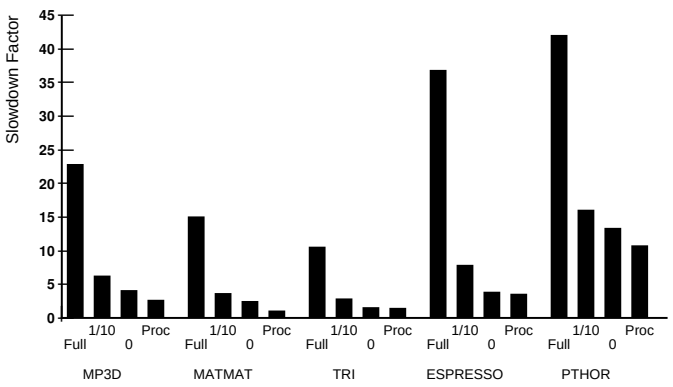

Figure 10: Sampled MemSpy performance overhead. (128K cache)

Figure 10 shows the simulation overhead of all five benchmarks under different configurations. All overheads shown are for simulations of 128KB caches. (Overheads for other cache sizes are similar, although slightly dependent on the application's miss rate.) The bars labeled "Full" show the (multiplicative) overhead of a full MemSpy simulation as compared to the uninstrumented application run. The bars labeled "1/10" show the overhead for sampled runs with a sampling ratio of 1/10. At this ratio, the overheads are reduced by roughly 4 to 6 fold from the original MemSpy implementation. They now range from 2.9 for TRI to 16 for PTHOR. Four of the five benchmarks have overheads less than 8.

These are acceptable overheads in many cases, given the detailed statistics MemSpy provides. Nonetheless, we should examine why the speedup obtained was not closer to 10, the reciprocal of the sampling ratio. The third bar for each application ("0") shows the overhead when the reference instrumentation is present, but the simulator is never called, a sampling ratio of 0. The fourth bar for each application ("Proc") shows the overhead when *no* reference instrumentation is added; all overhead here is due to MemSpy's logging of procedure entries and exits, as well as MemSpy initialization time. The difference in height between the "1/10" bars and the "0" bars represents the cost of reference simulation. The difference in height between the "Proc" and "1/10" bars is the cost of both reference simulation and additional instrumentation for sampling. For most applications, reference simulation is responsible for roughly half of the overhead in a run. MemSpy's procedure logging is responsible for much of the rest of the overhead, with the additional sampling instrumentation responsible for up to about 25% of the overhead. To reduce MemSpy's overhead, one could consider approaches which try to reduce (i) procedure logging overhead, (ii) sampling instrumentation overhead, and (iii) simulation overhead. The following paragraphs discuss each of these three axes for optimization.

Procedure logging overhead could be reduced to some degree through simple optimizations of the logging code; however, procedure events cannot be sampled as with memory events, since procedure calls and returns must occur in matched pairs to to maintain the state of the stack.

The second source of overhead, sampling instrumentation, is defined as the additional instructions needed to switch simulation ON and OFF, as opposed to simulating all references. In the implementation presented here, this overhead is primarily the six additional instructions per memory reference which allow control to branch around the memory simulator when simulation is OFF. To try to avoid this overhead, we have implemented a preliminary version of a more aggressive approach. In this new approach, control switches back and forth between two different versions of the application: one version fully instrumented to simulate all memory references, and another version only instrumented to log procedure entries and exits, not memory references. The program executes in the fully instrumented version when simulation is turned ON, and then switches to the minimally instrumented version when simulation is turned OFF. These mode switches are determined by virtual timer interrupts using the UNIX `setitimer` call. This version is still subject to the overhead of procedure event logging, and in addition, has double the application code size (because there are two versions of all application code), which has a detrimental effect on instruction cache behavior. For these reasons, its performance benefits thus far have been moderate at best. It offers no better than a 20% speedup for the benchmarks presented here. However, further work on efficiently handling the mode switch in this approach may make this an attractive alternative to our initial implementation.

The third overhead, simulation overhead, is comprised of both the time spent to simulate the memory system, as well as the time spent saving and restoring registers in order to "context switch" to the memory simulator. For the simple memory simulator used here, register saves and restores are responsible for more than half of the "simulation" overhead. One way to reduce the time spent performing register saves and restores would be to circumvent these register operations for all cache hits. We have implemented this optimization in the following way. On each memory reference, we save a small subset of the application registers, and then do a preliminary check to determine if the memory reference is a cache hit. If it is a hit, we branch around the memory simulation and remaining register saves and restores. We need only restore the small subset of registers we saved before the hit check. An untuned implementation of this optimization offers a further 12 to 23% speedup to the benchmarks shown here.

7 Discussion

The previous section has shown that, for the benchmarks considered, reference trace sampling is effective at improving MemSpy's performance, with only a small decrease in the accuracy of the reported statistics. This section will addresses several side issues not yet touched upon.

7.1 Applications Suitable for Trace Sampling

While we consider the benchmarks used here to be representative of applications used with MemSpy, not all programs being tuned will be amenable to sampling. In general, we can divide applications roughly into the categories shown in Table 7. This table divides programs according to two characteristics, miss rate and total references. It shows that the applications most suited to sampling generally coincide with the applications that need memory tuning. Applications with high miss rates and many references are most amenable to sampling. A large number of known misses decreases the effect of unknown references; a long reference trace allows enough samples to be taken, with each sample long enough to prime the cache. By contrast, applications with low miss rates and few references have little need for MemSpy tuning, so the high error due to unknown references is not relevant. Applications with high miss rates, but few references will likely have an execution time short enough to be run without sampling.

Table 7: Applications amenable to sampling and MemSpy.

Miss Rate	Number of References		
	Few	Moderate	Many
low	No MemSpy needed	Use MemSpy, without sampling	Use MemSpy, with long samples
med.	Use MemSpy, without sampling	Use MemSpy, possible sampling	Use MemSpy, with medium samples
high	Use MemSpy, without sampling	Use MemSpy, with sampling	Use MemSpy, with short samples

7.2 Avoiding Periodic Behavior

One of the pitfalls of trace sampling is the possibility that the samples will repeatedly coincide with periodic application phases, resulting in a cache miss estimate that is not necessarily representative of the program as a whole, even when gathering a large number of samples. While we have not implemented it here, a straightforward solution is to use samples whose length varies randomly around a chosen mean, with a specified variance. This would introduce some randomness into the sampling interval, to make it less likely to repeatedly coincide with a particular phase of the application.

7.3 Set Sampling

Up to this point, this paper has only treated issues related to time sampling. In set sampling, one simulates the behavior of selected cache lines or sets, rather than simulating the entire cache. From a performance standpoint, an implementation of set sampling offers speedups similar to those in our current implementation of time sampling. As with time sampling, one could augment the application assembly code to branch around the MemSpy procedure call for references which are not to be simulated. As described in [7], one could use a bit mask to select some fraction of the addresses to simulate. This implementation would require the same number of instructions as our current time sampling implementation.

Set sampling is promising from an accuracy standpoint as well. Whereas time sampling suffers from errors due to unknown references, in set sampling the cache state is always known; there are no unknown references to contend with. However, set sampling is still subject to inaccuracies when the sets chosen for sampling are not representative of the overall cache behavior. Since the chosen sets are fixed over the duration of the trace, set sampling is more sensitive than time sampling to error from non-representative samples. A combination of both approaches may be the most promising alternative; thus, for a given performance goal, neither sampling method need be pushed into the extreme regions where it is less accurate.

7.4 Multiprocessor Behavior

MemSpy is designed to be used with both sequential and shared memory parallel programs. However, we have thus far only examined issues related to trace sampling in sequential reference traces. The parallel domain has its own unique characteristics that affect the accuracy and performance of a sampling MemSpy implementation.

On shared memory multiprocessors, "typical" cache miss rates can be considerably higher than on a sequential machine. Data sharing between processors can result in frequent invalidation misses, which are likely to increase the known misses and decrease the effect of unknown references. However, the disadvantage is that a parallel machine can have a much larger amount of state to be primed at the start of each sample. In addition to determining whether the reference is a cache hit or miss, the simulator may also need to determine which other processors have copies of a cache line, in order to determine whether invalidations are needed.

Furthermore, our current parallel simulator is designed to interleave execution of multiple threads on a uniprocessor. In order to simulate a realistic interleaving of program threads, the simulator does frequent context switches between threads, always running the one that is "farthest behind". Maintaining this proper interleaving will interfere with running the program at full-speed when not instrumented, because the simulator will still have to check for context switches. We intend to examine ways of doing periodic low overhead checks to determine whether context switches need to occur, rather than the current method of checking for potential context switches on each memory reference. We will also examine the potential of parallel simulation on a true multiprocessor, to eliminate the need for these context switches, and improve performance.

8 Conclusions

We have presented an analysis of the effectiveness of trace sampling within the context of a performance debugging tool. In general, we found that sampling parameter settings (such as sample length and number of samples) required for good accuracy also allowed significant performance improvements. With sample lengths of roughly 4M references, all benchmarks could be sampled with less than 0.5% absolute error in cache miss rate, even in large 1MB caches. With this setup, MemSpy performance improvements of 4 to 6 fold were obtained. In general, the sample length required to achieve good accuracy will increase as the application's miss rate decreases. This means that performance debugging is an excellent application of memory reference trace sampling. Since we expect the target applications to have fairly

high miss rates, we should be able to use shorter sample lengths to achieve a particular level of accuracy. This in turn allows us to use more aggressive sampling ratios (with higher performance) when generating MemSpy statistics.

We are also able to reproduce the more detailed statistics produced by MemSpy with good accuracy. In the four applications studied, the sampled version of MemSpy produced an ordering of program memory bottlenecks which exactly matched the true program bottlenecks for all bottlenecks totalling up to 90% of the memory stall time. The percentages of memory stall time incurred by different program data structure and procedures also retained useful accuracy, within 20% of their true values.

Performance debugging is especially suited to sampling implementations, because it is an iterative process with different degrees of accuracy warranted at different stages. Sampling allows the MemSpy user to get a fast initial view of program behavior using sampling ratios of 1/10 or less. Then, as the users begin to fine tune performance, they can switch to higher sampling ratios which may provide better accuracy for capturing more subtle details of program behavior. Used with care, sampling can allow accurate estimates of detailed memory system statistics to be produced with execution time overheads that are competitive with other much less detailed performance monitors.

9 Acknowledgments

Thanks to Rohit Chandra for his comments on a draft of this work. This work was supported in part by DARPA contract number N00039-91-C-0138, the National Science Foundation (CDA-8722788) and the Digital Equipment Corporation (the Systems Research Center and the External Research Program). In addition, Anoop Gupta is supported by a National Science Foundation Presidential Young Investigator Award, and Thomas Anderson is supported by a National Science Foundation Young Investigator Award.

References

[1] T. E. Anderson and E. D. Lazowska. Quartz: A Tool for Tuning Parallel Program Performance. In *Proc. ACM SIGMETRICS Conf. on the Measurement and Modeling of Computer Systems*, pages 115–125, May 1990.

[2] J. Dongarra, O. Brewer, J. A. Kohl, and S. Fineberg. A Tool to Aid in the Design, Implementation, and Understanding of Matrix Algorithms for Parallel Processors. *Journal of Parallel and Distributed Computing*, 9:185–202, June 1990.

[3] A. J. Goldberg and J. Hennessy. MTOOL: A Method for Isolating Memory Bottlenecks in Shared Memory Multiprocessor Programs. In *Proc. Intl. Conf. on Parallel Processing*, pages 251–257, Aug. 1991.

[4] A. J. Goldberg and J. Hennessy. Performance Debugging Shared Memory Multiprocessor Programs with MTOOL. In *Proc. Supercomputing*, pages 481–490, Nov. 1991.

[5] S. L. Graham, P. B. Kessler, and M. K. McKusick. An Execution Profiler for Modular Programs. *Software Practice and Experience*, 13:671–685, Aug. 1983.

[6] J. Hennessy and N. Jouppi. Computer Technology and Architecture: An Evolving Interaction. *IEEE Computer*, pages 18 – 29, Sept. 1991.

[7] R. E. Kessler, M. D. Hill, and D. A. Wood. A Comparison of Trace-Sampling Techniques for Multi-Megabyte Caches. Technical Report 1048, Univ. of Wisconsin Computer Sciences Department, Sept. 1991.

[8] S. Laha, J. H. Patel, and R. K. Iyer. Accurate Low-Cost Methods for Performance Evaluation of Cache Memory Systems. *IEEE Trans. on Computers*, pages 1325–1336, Nov. 1988.

[9] M. Lam, E. Rothberg, and M. Wolf. The Cache Performance and Optimizations of Blocked Algorithms. In *Proc. Fourth Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 63–74, Apr. 1991.

[10] M. Martonosi, A. Gupta, and T. Anderson. MemSpy: Analyzing Memory System Bottlenecks in Programs. In *Proc. ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems*, pages 1–12, June 1992.

[11] E. Rothberg and A. Gupta. Parallel ICCG on a Hierarchical Memory Multiprocessor— Addressing the Triangular Solve Bottleneck. *Parallel Computing*, 18(7):719–41, July 1992.

[12] J. P. Singh, W.-D. Weber, and A. Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. *Computer Architecture News*, 20(1):5–44, March 1992.

[13] Spec benchmark suite release 1.0, Oct. 1989.

[14] H. S. Stone. *High-Performance Computer Architecture*. Addison-Wesley, Reading, MA, second edition, 1990.

[15] D. A. Wood, M. D. Hill, and R. E. Kessler. A Model for Estimating Trace-Sample Miss Ratios. In *Proc. ACM SIGMETRICS Conf. on the Measurement and Modeling of Computer Systems*, pages 79–89, June 1991.