In *Proc. ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems*, May 1993.

[11] M. R. Martonosi. *Analyzing and Tuning Memory Performance in Sequential and Parallel Programs*. PhD thesis, Stanford University, Dec. 1993. Also Stanford CSL Technical Report CSL-TR-94-602.

[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.

isolate and highlight program memory bottlenecks, and how to gather such information efficiently. These ideas are embodied in the MemSpy performance monitoring tool.

Orthogonal to previous code oriented statistics, MemSpy's *data oriented statistics* form an important new dimension in viewing and analyzing program behavior. Together, data and code oriented statistics are a powerful approach for analyzing and tuning memory performance. In addition, by offering detailed characterizations of the predominant causes of misses for each data structure, MemSpy gives users important insights about the cause of memory bottlenecks, and how to fix them.

Finally, key to implementing such detailed statistics is the ability to gather program information efficiently. This work shows that simulation-based performance monitors can offer a feasible, effective, and inexpensive alternative to other data collection methods. By combining the optimizations of *hit bypassing* and *reference trace sampling*, MemSpy overheads dropped to 3 to 10 fold for sequential applications and 8 to 25 fold for parallel applications. Overall, this work demonstrates the utility of MemSpy's detailed, data oriented statistics, and introduces methods for collecting them with execution time overheads that make MemSpy an attractive alternative to other less detailed approaches.

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. L. Hennessy. Mtool: An Integrated System for Performance Debugging Shared Memory Multiprocessor Applications. *IEEE Transactions on Parallel and Distributed Systems*, pages 28–40, Jan. 1993.

[4] S. R. Goldschmidt. *Simulation of Multiprocessors, Speed and Accuracy.* PhD thesis, Stanford University, June 1993.

[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] 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.

[7] P. Lacroute and M. Levoy. Fast Volume Rendering Using a Shear-Warp Factorization of the Viewing Transformation. In *Proc. ACM SIGGRAPH*, July 1994.

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

[9] A. R. Lebeck and D. A. Wood. Cache Profiling and the SPEC Benchmarks: A Case Study. Technical report, Univ. of Wisconsin Computer Sciences Department, Mar. 1992.

[10] M. Martonosi, A. Gupta, and T. Anderson. Effectiveness of Trace Sampling for Performance Debugging Tools.

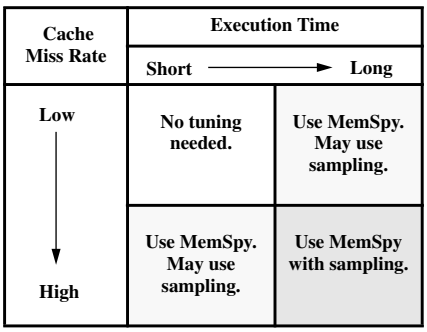

Figure 11: Application characteristics and MemSpy usage.

references have little need for MemSpy tuning, so their potential for higher sampling error is less relevant.

6.4.2 MemSpy Performance Using Sampling

Our primary purpose in using time sampling is to improve MemSpy's performance. To implement sampling, we add per-reference instrumentation, in which a sampling counter is decremented and checked 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, MemSpy saves the application registers and performs the cache hit check described in Section 6.3. Thus, we expect this sampling implementation to offer a modest performance improvement on cache hits (that can be bypassed anyway) and a large performance improvement on cache misses. In other words, as with accuracy, we expect sampling to yield the largest performance benefits on the applications most likely to be used with MemSpy – those with poor memory behavior.

The lightly shaded bars in Figure 7 show simulation overhead using the time sampling approach with the same parameters as in the preceding accuracy discussion. For the sequential benchmarks, sampling results in a 1.7 to 3.1 fold performance improvement over hit bypassing alone. For the parallel benchmarks, improvements are slightly larger, ranging from 2 to 5.4. These improvements lead to attractive execution time overheads ranging from roughly 3 to 10 for sequential applications and 8 to 25 for parallel applications.

7 Conclusions

Both sequential and parallel applications are currently facing a growing gap between processor and memory speeds, and consequently, performance lost due to memory stalls can be substantial. Despite this trend, performance monitoring tools have lagged in providing support for identifying and characterizing memory bottlenecks. This paper has presented our views on what kinds of information are useful for tuning memory performance, how to present such information in ways that

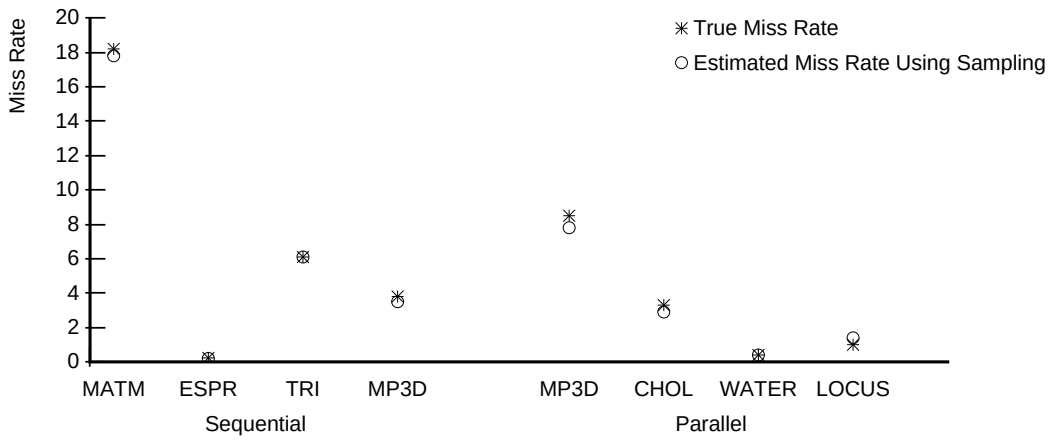

Figure 10: Estimated and true cache miss rates for sequential and parallel applications.

6.4.1 MemSpy Accuracy Using Sampling

Overall, we find that time sampling is quite effective at accurately reproducing cache statistics from a full simulation. For example, Figure 10 compares estimated cache miss rates from sampling (using a 10% sampling ratio) to the program's true miss rate calculated over all references. For the sequential applications, the samples contain 0.5M references. (That is, we simulate 0.5M references, then turn simulation off for 4.5M references before turning it on again.) For the parallel benchmarks, samples are 3M references long, where a sample in this case is a group of references from the *interleaved* reference traces of *all* processors. The largest absolute deviation between the true miss rate (stars) and estimated miss rate (circles) is 0.74%, with small relative deviations as well.

The discussion in [11] presents more comprehensive sampling accuracy results, varying the number of samples taken, the sample length, the cache size, and in the parallel case, the number of processors. Overall, our results show that for sequential benchmarks with small and moderately sized caches, miss rates can be estimated with small absolute deviations ($< 0.5\%$) and relative deviations of 10% or less. These estimates require sample lengths of only 0.5M references or less. For good accuracy when simulating larger caches ($\geq$ 1MB), longer samples are required to prime the cache – 4M references or more. However, this still allows for aggressive sampling ratios on many applications. Finally, we find that simulating parallel applications requires slightly longer samples because parallel machines generally have more total cache. However, required sample length is not proportional to total cache size for parallel applications, because coherence traffic can mitigate the need for longer samples.

As illustrated in Figure 11, trace sampling is successful in MemSpy because the applications most suited to sampling coincide well with the applications most in need of tuning. Applications with high miss rates and many references are most amenable to sampling, because it is easier to sample them with low relative errors. By contrast, applications with low miss rates and few

Figure 9: Time samples in an application reference trace.

bypassing path in Figure 8 shows how the cache hit check is embedded into the register saves. Thus, we initially only save the registers required to check if the reference is a cache hit or cache miss. If the reference is a cache miss, we complete the rest of the register saves and continue simulating. If the reference is a cache hit, we restore the minimal subset of registers and return to the application, bypassing the simulator call entirely. In parallel applications, writes that cause invalidations are always simulated, but no statistics are kept on cache hits.

The middle columns in Figure 7 show the significant performance benefits of using hit bypassing. With this technique, performance improves by factors of 1.5 to 3.4 compared to the baseline code. MemSpy's overhead factors drop to 8 to 17 for sequential code and 30 to 50 for parallel code.

6.4 Optimizing MemSpy Performance Using Sampling

In the second optimization, reference trace sampling, cache behavior is estimated while simulating only portions of a reference trace, rather than simulating the full trace. Intuitively, sampling is a promising technique because we expect to be able to approximate program behavior with reasonable accuracy by taking intermittent snapshots of its activity. For example, sampling of program counter values is already used in several tools such as Gprof [5]. Reference trace sampling promises significant speedup, since one incurs the full simulation overhead only on a fraction of the full reference stream. On the other hand, there is an inherent tradeoff between the fraction of references simulated and the accuracy of the simulation results. Within the context of a performance debugging tool, trace sampling can be used effectively to improve the tool's performance while retaining acceptable accuracy.

We focus here on the use of time sampling which, as shown in Figure 9, is implemented by intermittently turning reference simulation on and off as a reference trace is processed. The two key parameters in the implementation are (i) the *sample length* (or, number of references contained in each sample), and (ii) the *number of samples*. A third dependent parameter is the *sampling ratio*, the ratio of the total number of references within the samples, divided by the total number of references in the run. In this paper, we present accuracy results for one setting of sampling parameters, and briefly summarize other results. References [6], [10], and [11] discuss reference trace sampling in more detail.

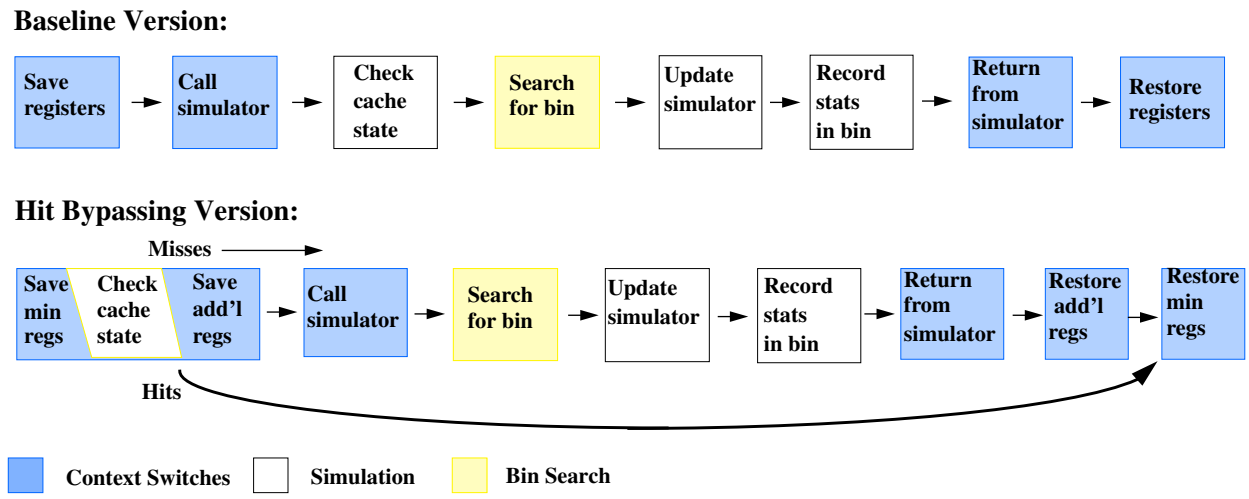

Figure 8: MemSpy control flow: Baseline and Hit Bypassing versions.

of 18 to slightly under a factor of 60. The overheads for the parallel applications, ranging from factors of 64 to 115, are higher than for the sequential benchmarks due to increased complexity in simulating the parallel machine. [1]

For both the sequential and parallel benchmarks, 97% or more of MemSpy's overhead is due to processing memory references. Thus, we focus our discussion specifically on the processing overhead required for memory references in MemSpy. The baseline path in Figure 8 shows the flow of actions MemSpy performs on each simulated memory reference. These actions are categorized (by shading) into three types of overhead: (i) memory simulation itself (roughly 45% of the overhead), (ii) statistics bin searches (30%), and (iii) context switches (register saves and restores) entering and exiting the simulator (25%). Based on this data, the performance optimizations presented in Sections 6.3 and 6.4 will concentrate on reducing time spent in these three phases.

6.3 Optimizing Performance Using Hit Bypassing

Hit bypassing originates from the observation that the "interesting" memory events, from a performance tuning point of view, are events that incur memory stalls. For most architectures, only cache misses incur stalls; by keeping no cache hits statistics, we can avoid the significant overhead on all cache hits, a majority of the references.

In the baseline implementation, we save a full set of registers, and then call the simulator to check if the reference is a cache hit or miss. The cache check itself uses very few registers (roughly 4-7 depending on the simulator and the implementation), and if the reference is a hit, we return almost immediately, restoring the full set of registers although only a handful were used. The hit

[1] For the parallel applications, the execution time overhead is shown relative to a uniprocessor execution time of the program. To compare to an actual multiprocessor execution time, one multiplies these overheads by the expected program speedup.

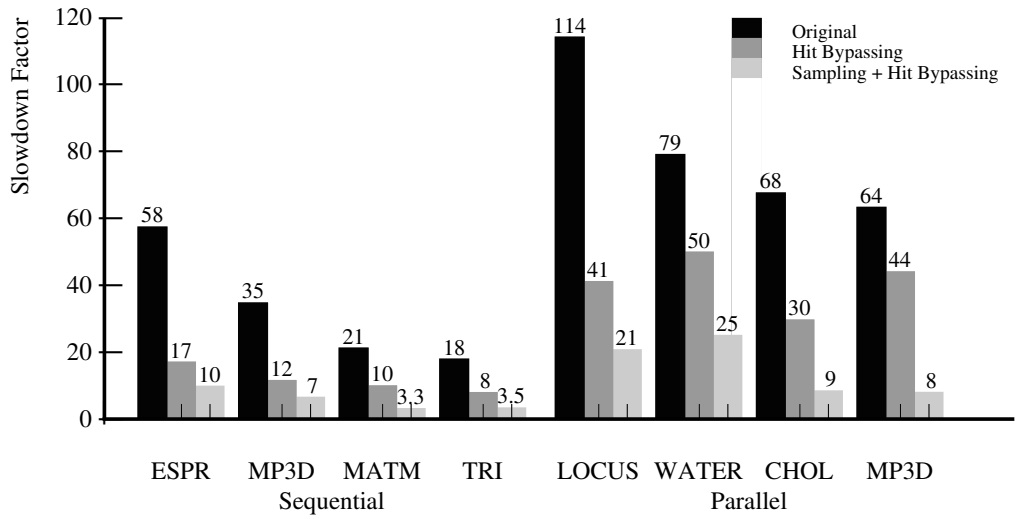

Figure 7: MemSpy performance overhead: baseline, hit bypassing and sampling versions.

tention is not modeled. All heap references and static data references in the code are instrumented for simulation. For multiprocessor simulations, we simulate an invalidation-based protocol. When simulating sequential machines we assume a 128KB direct-mapped data cache with 32 byte lines. For parallel machines, we assume 16 CPUs, each with a 64KB direct-mapped cache with 32 byte lines.

Benchmark Applications By evaluating MemSpy's performance on substantial sequential and parallel applications from the engineering and scientific community, we show that its simulation-based implementation is practical on "real" benchmarks. The four sequential applications are: (i) Blocked matrix multiply as described in Section 4, (ii) Espresso from the SPEC benchmark suite, (iii) Tri, a sparse triangular matrix solver, and (iv) a uniprocessor run of Mp3d, a SPLASH benchmark. The four parallel applications are all taken from the SPLASH benchmark suite: (i) Mp3d, (ii) Cholesky, (iii) Water, and (iv) LocusRoute. These eight applications span a variety of application domains including numerical computations, scientific, and engineering applications. In addition, they span a range of cache miss rates (0.2% to 18.2%) and code sizes (500 to 14000 lines). As such, they demonstrate the variety of applications amenable to MemSpy's performance monitoring.

6.2 Performance of Baseline MemSpy Implementation

The leftmost (black) column for each application in Figure 7 shows the baseline performance overheads for MemSpy as measured on a DECstation 5000/240 workstation. The results are shown as multiplicative factors comparing the time for a MemSpy run to the time for an uninstrumented run of the same program. For the sequential applications, the overheads range from roughly a factor

enough for clear bottlenecks to stand out. Moreover, statistics that are too fine-grained may also be inefficient to implement in terms of (i) storage inefficiency, because more memory is needed to maintain very fine-grained statistics, and (ii) execution time inefficiency, because extra time is needed to manage and update the larger number of statistics bins.

In its code oriented statistics, MemSpy separates information by procedures. Since the MemSpy simulator logs procedure entries and exits, it can easily maintain a shadow of the procedure call stack to track the currently executing procedure. In most programs, procedure-oriented statistics have been fine-grained enough to localize performance bugs, but future versions of MemSpy could allow users to choose statistics on basic blocks, rather than procedures, for finer-grained monitoring when needed.

In its data oriented statistics, MemSpy keeps aggregate data bins that encompass *all memory ranges allocated at the same point in the source code with identical dynamic procedure call paths.* When a heap allocation occurs, the current source code position and stack are noted. If the current program counter, as well as all the program counters on the stack, identically match that for a previously initialized bin, the statistics for this new range of memory are kept in that bin. The rationale for this heuristic is that, in our experience, data objects allocated at the same point in the source code via the same call path are usually similar in memory behavior. When memory is allocated in separate calls to a procedure from different call paths, it is monitored in separate bins.

Integrated together, these methods for monitoring programs and generating data and code oriented statistics at useful granularities have proven quite effective in practice. These techniques have been used successfully on a variety of programs, including the SPLASH parallel benchmarks. In addition, as will be discussed next, the implementation is efficient enough to allow these statistics to be generated with competitive overheads as well.

6 MemSpy Performance

This section begins by describing the measurement setup and presenting MemSpy's execution time overheads for sequential and parallel applications on a "baseline" implementation. While the baseline simulation overhead is already acceptable, we also describe two approaches for reducing MemSpy's runtimes into more interactive regimes: (i) optimizing simulation for the "common case", cache hits, and (ii) generating program statistics based on only samples of references, rather than the full trace.

6.1 Performance Measurement Setup

To evaluate MemSpy's overheads, the results presented in this paper were gathered using a simple memory simulator with a single direct-mapped cache per processor. Cache hits execute in a single processor cycle and cache misses take a fixed, parameterized latency to be serviced. Network con-

detailed information on the causes of misses and the causes of replacements, it would have been difficult to interpret this situation as self-interference. MemSpy has also been used to tune several other programs as well. For example, it has identified performance bugs due to: (i) false sharing and a "vestigial" (incremented but unused) variable in LocusRoute, a SPLASH benchmark, (ii) self-interference in the `ElementArray` in Pthor [12], (iii) poor spatial locality in a sequential volume rendering program, Vrender [7], and (iv) shared accesses to a private variable in a parallel version of Vrender. Together these experiences have confirmed the utility of these detailed, data oriented statistics.

5 MemSpy Implementation

In this section we address two principle design decisions in MemSpy: (i) how to gather this detailed information efficiently and (ii) at what code and data granularity to present this information to users.

5.1 Simulation-Based Monitoring

To present the performance information illustrated in Section 4, MemSpy must monitor at the granularity of individual memory references in the code being studied. To accomplish this, we have implemented MemSpy using a simulation-based approach. Simulation's main advantage is that it can be general and portable, since it relies on no specialized hardware support.

MemSpy is built on top of the Tango Lite reference generator [4]. Tango Lite is a direct execution system that simulates the execution of both multiprocessor and uniprocessor machines on uniprocessor workstations. In a direct execution simulation, "interesting" events are instrumented at compile-time with additional code to call event simulators. For MemSpy, the following events are instrumented: (i) memory references, (ii) procedure calls and returns, (iii) memory allocations, and (iv) synchronizations. For each event, instrumentation code calls the MemSpy simulator that maintains internal information on the state of the simulated memory hierarchy, as well as the profile information required to report MemSpy's statistics.

5.2 Granularity of Data and Code Oriented Statistics

A significant issue in implementing data and code oriented statistics is determining a natural granularity at which to present statistics. MemSpy statistics are organized and managed in terms of *statistics bins* containing program information (such as memory time, number of cache misses, or causes of cache misses) collected either for particular data or code sections or for pairings of data and code sections in the application. Choosing the right granularity in both data and code oriented statistics is important because statistics that are too coarse-grained may not localize bottlenecks well enough. On the other hand, statistics that are too fine-grained may not aggregate activity

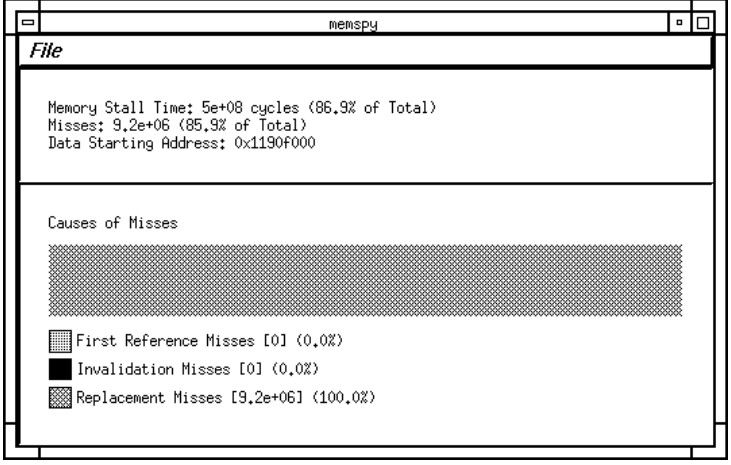

Figure 5: MatMul: MemSpy detailed statistics on `Y` matrix in the `block` procedure.

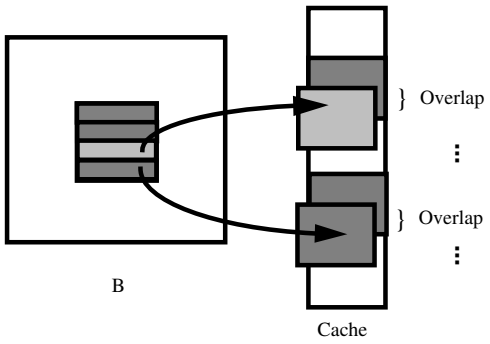

Figure 6: Self interference in blocked matrix.

space constraints) of the causes of these replacements. Surprisingly, over 95% of the replacements are due to the `Y` matrix itself.

Thus MemSpy, in this case, takes users to the point where they know (i) that the bottleneck is in the `Y` matrix, (ii) that it is caused by excessive cache interference, and (iii) that this interference is in fact self-interference, since the replacements are cause by the `Y` matrix itself. From this information, programmers can determine that self-interference occurs here because the sub-rows within the currently used block of `Y` are not stored contiguously, and thus do not map neatly across the whole cache. Rather, as shown in Figure 6, sub-rows are separated from one another by intervening amounts of matrix storage. This leads to cases where some sub-rows map on top of one another in the cache, while other portions of the cache remain unused. The programmer can minimize this effect by choosing a block size with less interference, or by copying the block so that it occupies a contiguous region of memory.

Overall, the example's message is that without MemSpy's data oriented statistics, it would have been difficult to see which matrix was causing the memory bottleneck. Furthermore, without

Figure 4: MatMul: Memory stall time in the `block` procedure attributed to the `X`, `Y`, and `Z` matrices.

To further understand and tune this code, users need information on whether a single matrix is a bottleneck, or whether some interaction between the three matrices is causing the stalls.

4.1.1 Data Oriented Breakdown

To understand the stall time contributed by each data structure, MemSpy lets users click on the memory portion of the `block` routine's bar to request the next display. This display, shown in Figure 4, gives a breakdown of the memory stall time into components incurred by each data structure in this procedure. With these data oriented statistics, one can learn that the bottleneck in this routine is almost entirely due to cache misses on references to the `Y` matrix. These misses are responsible for over 85% of the total stall time in the program. Since the `Y` matrix is actually the one that was blocked for good data reuse, it is surprising that `Y` is responsible for so much stall time.

4.1.2 Detailed Statistics on Causes of Misses

At this point, while MemSpy has provided insight as to *where* the bottleneck is occurring, we still have little understanding of *why* it would be occurring. To get more insight into the problem, we can click on the `Y` bar in Figure 4 to bring up a cause of miss display. The bar chart in Figure 5 breaks down the causes of cache misses for the `Y` matrix in the `block` routine. In this routine, all of `Y`'s misses are caused by previous replacements. That is, the data objects were all previously in the cache, but have been replaced out of the cache before the re-references occurred that resulted in cache misses. (The first reference misses for the `Y` matrix occur in a separate initialization routine.)

The high number of replacement misses incurred by `Y` indicates that the bottleneck is probably related to cache interference effects. To understand the cause of the memory bottleneck, however, users must know which accesses are causing the cache replacements. Clicking on the replacements portion of the "causes of misses" bar in Figure 5, brings up a breakdown (not shown here due to

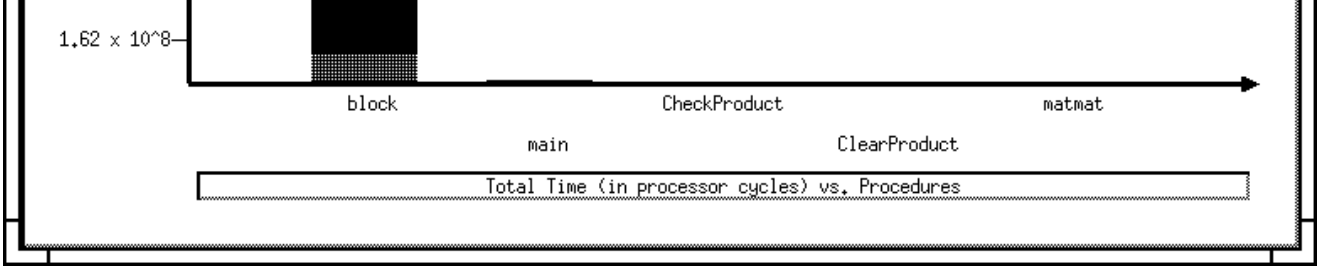

Figure 3: MatMul: MemSpy overview statistics display.

powerful method for focusing the user's attention on problem areas in the code. One also sees how detailed statistics on the causes of cache misses are crucial for understanding why the performance bottleneck is occurring.

4.1 Tuning Using MemSpy

To make the performance bug most evident, we show MemSpy's output on one of the poor performance cases from [8]. We multiply two 293 x 293 element matrices together, using a block size of 56. (A single 56 x 56 block requires roughly 25K bytes, and should easily fit into the 64KB cache.)

MemSpy begins by presenting the output shown in Figure 3. The program procedures are indicated along the x axis of this graph, and the time (in processor cycles) spent on behalf of each procedure is given on the y axis. The bar for each procedure breaks down the elapsed time by how much of it was spent in computation and how much in memory stalls. In addition, for parallel programs, the bar indicates synchronization time as well.

Figure 3 indicates that the bulk of the application's time is spent in the `block` routine. It accounts for over 90% of the execution time. Furthermore, the breakdown of time within the `block` routine shows a clear memory bottleneck. While we expected the bulk of the computation to be spent in `block`, the observation that roughly 80% of the time is spent on memory stalls is surprising, since we expected the 25KB block to easily fit in the 64KB cache for the computation.

Within the `block` routine, most of the execution time is spent in line 13 of the code in Figure 2. In this line, the appropriate elements of `X` (`r`) and `Y` are multiplied, and the result is accumulated in an element of `Z`. Since all three matrices are accessed on source line 13, code oriented statistics alone offer no help in determining the relative contributions of the three matrices towards the bottleneck.

```
 1. block(X, Y, Z, N, B)
 2. Matrix *X, *Y, *Z;
 3. int N,B;
 4. {
 5.   int kk,jj,i,j,k;
 6.   double r;
 7.   for kk = 1 to N by B do
 8.     for jj = 1 to N by B do
 9.       for i = 1 to N do
10.         for k = kk to min(kk+B-1,N) do
11.           r = X[i,k];
12.           for j = jj to min(jj+B-1,N) do
13.             Z[i,j] = Z[i,j] + r*Y[k,j];
14. }
```

Figure 2: Pseudo-code for blocked matrix multiply example.

The lack of detailed support for memory bottleneck identification stems partly from difficulty in efficiently gathering memory system statistics. Gathering detailed memory statistics requires fine-grained monitoring using either specialized hardware or software simulation. The drawback of specialized hardware is that it can limit the generality of the tool, but on the other hand, software simulation can often be too slow. This work offers optimizations that significantly improve the efficiency of simulation-based monitoring.

4 Using MemSpy: A Case Study

This section illustrates MemSpy's detailed, data oriented statistics on a particular program, `MatMul`, which performs a blocked matrix multiply. Although this is a fairly simple application, it is a case where the common intuitions about the code's behavior are incorrect, and tools like MemSpy are helpful in guiding the programmer to the bottleneck.

The blocked matrix multiplication code (computing $Z = X \times Y$) is shown in Figure 2. Unlike standard matrix algorithms, blocked algorithms such as this are coded to operate on sub-matrices or blocks of the original matrix. These sub-blocks are sized to fit in the cache, to maximize reuse of the data. By iterating over all sub-blocks, the full matrix multiplication can be performed, ostensibly with better cache performance due to the blocking.

As Lam et al. reported in [8], the performance of such blocked operations is often erratic, and is quite sensitive to even small changes in the matrix size, the block size, and the cache organization. For a DECstation 3100, they report that a 300 by 300 blocked matrix multiply (with 56 by 56 block size) executes at 4.0 MFLOPS, while by contrast, an only slightly smaller 293 by 293 matrix with the same block size executes at only 2.0 MFLOPS on the same machine. Thus, the goal of this case study is to show how MemSpy can be used to guide programmers through the tuning process for this program. As we proceed through the case study, one can see that data oriented statistics are a

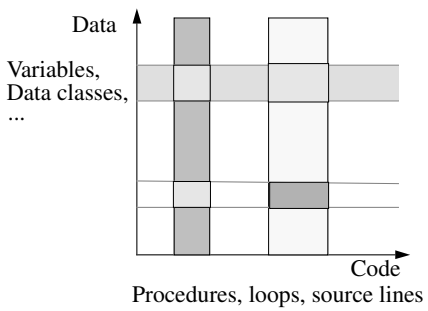

Figure 1: Decomposing programs into data and code statistical bins.

Figure 1 gives an abstract illustration of possible code oriented and data oriented subdivisions in a program. While code oriented statistics only divide program statistics along one dimension, the key contribution of data oriented statistics is that they allow for statistics to be presented along a second dimension of this space, by subdividing the program according to source-level application data structures. Because of the inherent link between memory performance and the access patterns of particular data structures, these statistics can be crucial to reasoning about memory behavior. (Echoing this message, CPROF [9], developed independently, also implemented data oriented statistics.)

Data oriented statistics are especially useful in cases in which a particular data structure may constitute a memory bottleneck, but accesses to it are distributed across several procedures. For example, in Pthor (a SPLASH benchmark [12]) the `ElementArray` is responsible for more of the program's cache misses than any other variable, but these misses are distributed across several procedures. Code oriented output cannot emphasize `ElementArray`'s performance problems as well as data oriented output can, because no single section of code is the bottleneck. In this case, the bottleneck lends itself to data oriented viewing. Furthermore, combinations of data and code oriented statistics can also be instrumental in isolating particular memory bottlenecks.

3.2.2 Statistics on Causes of Cache Misses

After determining where program bottlenecks lie, users need progressively more detail to understand *why* bottlenecks are occurring. As discussed in Section 3.1, cache misses occur for one of three reasons: (i) due to first reference, (ii) due to replacement, or (iii) due to invalidation. By presenting detailed statistics on the frequency and primary causes of cache misses, MemSpy can give the programmer insight about the type of memory performance bottleneck present, and how to tune it.

In spite of the importance of detailed information in understanding memory performance, most existing tools have provided no statistics on memory system behavior at all. Other tools, such as Mtool [3], give only high-level information about which parts of the code cause memory bottlenecks.

performance tuning.

Some programs may suffer from performance bottlenecks due to *cache interference*. Here, multiple memory lines mapping to the same cache line can compete for cache space and result in excessive cache misses. Tools can help programmers identify interference by pointing out data structures with excessive *replacement misses*. Replacement misses occur if a particular memory line has been referenced before, but has been replaced out of the cache by an intervening reference to another line competing for the same space in the cache. In such programs, simple adjustments or coloring strategies to stagger data structures in the cache can significantly improve performance.

As another example, *poor spatial locality* occurs when a program's data access order is not well correlated with the data storage order; because of this, it may not make efficient use of the full line of data fetched on a cache miss. Poor spatial locality can be especially pronounced in parallel code, because data structures that were stored and accessed contiguously in the sequential case may become distributed over several processors in the parallel case. Frequently, poor spatial locality can be deduced from seeing large memory stalls due to *first reference misses*. (Other metrics, such as counts of the number of bytes accessed per cache line between cache misses, can also be useful for noticing poor spatial locality.) Spatial locality can then be improved by restructuring data accesses or storage schemes to pack cache lines more efficiently with useful data.

Finally, in multiprocessors, programs may also have poor memory performance due to *excessive interprocessor sharing*, resulting in large amounts of cache coherence traffic. Since shared data is used for interprocessor communication in shared memory parallel programs, some memory stalls due to sharing are unavoidable. However, an excess of *invalidation misses* can indicate data structures or code sections where restructuring to minimize the required communication would be most fruitful. In addition, programs can also often be restructured to reduce false sharing, in which multiple processors actively read and write *different* variables on the same cache line.

Overall, the message here is that by understanding the underlying *causes* of memory bottlenecks, a programmer gains significant insight towards tuning them.

3.2 MemSpy Statistics for Tuning Memory Bottlenecks

An effective performance tool must offer programmers information on both *where* and *why* bottlenecks are occurring in their code. To answer the "where" question, MemSpy presents statistics in terms of the application's data, as well as code, structures. To answer the "why" question, MemSpy presents statistics on the frequency and causes of cache misses that can be instrumental in guiding programmers and compilers towards effective program transformations. This section describes each of these approaches.

3.2.1 Data and Code Oriented Statistics

2 Previous Performance Monitoring Approaches

For the most part, previous work on performance tools has not focused on support for memory performance tuning. For example, Gprof [5], a widely-used tool, offers per-procedure rankings of execution time spent in the code, but gives programmers little intuition about what is causing a particular bottleneck, and whether memory behavior may be responsible. Other tools such as Quartz [1] have been aimed specifically at parallel programming; they offer support for discerning synchronization, as well as computation, bottlenecks in the code. However, as with Gprof, Quartz offers no specific information for understanding application memory performance.

Mtool [3] is an example of a tool which does provide support for memory performance tuning. Mtool provides information on which code fragments are memory bottlenecks by presenting per-basic-block statistics on the amount of time spent on memory stalls. However, this level of detail offers little insight as to *why* bottlenecks are occurring, or which data structures are most responsible. Without more detailed information, it is often difficult to discern the problem and develop a strategy for remedying it.

At the other extreme of more detailed program information, SHMAP [2] provides a reference-by-reference animation of the program memory behavior. This animation of cache activity can allow programmers to discern memory performance pitfalls, but it offers little summary information or support for automatically analyzing the memory behavior. Especially in irregular, non-scientific code, reference patterns and performance bugs may be difficult to understand through this almost purely visual approach, and the volume of animation data required to analyze real benchmarks can be overwhelming. MemSpy attempts to address some of the shortcomings of previous tools.

3 Tuning Program Memory Behavior

Tools can assist in diagnosing and tuning memory performance bugs by providing specific types of information on program memory behavior. In general, a tool's first step in identifying performance bottlenecks should be to point out where bottlenecks are occurring by pointing out data structures or sections of code where the memory stall time attributed to it is both "large" in an absolute sense, and "larger than expected" in a relative sense. Beyond this, additional tuning support is required to point out why memory bottlenecks are occurring; the nature of this support depends on the specific types of performance bugs encountered, as described below.

3.1 Common Memory Performance Bugs

Three frequently occurring memory performance bugs in sequential and parallel applications are: (i) cache interference, (ii) poor spatial locality, and (iii) interprocessor sharing. Because of their differing characteristics, they are recognized and tuned in different ways. The descriptions in this section will drive our discussion in Section 3.2 of features provided by MemSpy to support memory

cessors, latencies can be over a hundred cycles. These figures are likely to become even larger in the future. Architects have responded to this trend by adding one or more levels of cache into memory hierarchies between the processor and main memory. Even with these hierarchies, however, many programs still have poor performance due to memory stalls.

To improve program memory performance, compilers and programmers can transform the application so that its memory referencing behavior takes better advantage of the memory hierarchy. The challenge in performing these transformations, however, is that an application's referencing behavior and interactions with the memory system can be difficult to statically analyze or reason about. Furthermore, the high-level information collected by many existing performance monitoring tools is not sufficiently detailed to analyze specific memory performance bugs. Thus, the focus of our work has been on devising techniques to efficiently collect detailed statistics on application memory behavior and to effectively organize the large volumes of data collected.

This paper describes MemSpy, a performance monitoring tool designed to help programmers or compilers discern *where* and *why* memory bottlenecks are actually occurring, and to guide them towards the program transformations that improve memory performance. MemSpy provides statistics on the frequency and causes of cache misses; these can be crucial in understanding why particular memory bottlenecks are occurring. In addition, because of the natural link between data reference patterns and memory performance, MemSpy allows users to understand interactions of different data structures and code segments by presenting statistics in terms of *both* data and code structures in the program, rather than solely in terms of code structures.

To gather statistics at this level of detail, MemSpy uses a simulation-based approach. Since efficiency is a key concern in simulation-based performance monitoring, we introduce and evaluate two main optimizations, *hit bypassing* and *reference trace sampling*, that reduce the execution time overhead required to gather such information. Used together, these techniques reduce simulation time by nearly an order of magnitude. MemSpy overheads range from factors of 3 to 10 for sequential applications and 8 to 25 for parallel applications for a simple memory simulator. Overall, our experiences using MemSpy to tune several sequential and parallel applications demonstrate that it generates effective memory performance profiles, at speeds that make it an attractive alternative to previous approaches.

The remainder of the paper is organized as follows. In Section 2, we briefly discuss related work. Section 3 discusses the main types of memory performance bugs, and uses this to motivate a discussion of MemSpy's features and contributions. Section 4 shows an example of MemSpy's use to tune a program's memory behavior. Following this, in Section 5, we discuss the implementation issues of MemSpy's detailed, data oriented statistics. Since the overhead of running the tool is such an important factor in the usefulness of the tool, Section 6 evaluates the performance of a baseline MemSpy implementation and presents two optimizations for improving MemSpy's performance. In Section 7 we present conclusions.

Tuning Memory Performance in Sequential and Parallel Programs

Margaret Martonosi
Dept. of Electrical Eng.
Princeton University
Princeton, NJ 08544

Anoop Gupta
Computer Systems Laboratory
Stanford University, CA 94305

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

Abstract

Recent architecture and technology trends have led to a significant and increasing gap between processor and main memory speeds. Caches hide these latencies to some extent, but when cache misses are frequent, memory stalls can significantly degrade program execution time. This paper describes MemSpy, a performance monitoring system designed to help identify and fix program memory bottlenecks. The natural interrelationship between memory bottlenecks and program data structures motivates MemSpy's introduction of *data oriented statistics* for memory performance information. Furthermore, MemSpy's detailed statistics on the causes of cache misses are crucial for determining sources of memory bottlenecks.

Since tool efficiency is important, a key focus of this work is on performance optimizations for MemSpy's simulation-based monitoring. Two optimizations discussed here, *hit bypassing* and *reference trace sampling*, reduce simulation time by nearly an order of magnitude. Overall, our experiences using MemSpy to tune several sequential and parallel applications demonstrate that it generates effective memory performance profiles, at speeds that make it an attractive alternative to previous approaches.

Keywords:
Memory System Performance, Performance Tuning, Data Oriented Statistics, Simulation, Trace Sampling.

1 Introduction

Processor speeds in modern computers are improving at a much faster rate than main memory speeds, and as a result, relative latencies from processors to main memory have dramatically increased. In uniprocessors, main memory latencies can be tens of processor cycles, while in multipro-

This paper has been accepted for publication and will appear in IEEE Computer. This research was supported by DARPA contract N0039-91-C-0138. In addition, Anoop Gupta was partly supported by an NSF Presidential Young Investigator award, and Thomas Anderson by an NSF Young Investigator award.