Tuning Memory Performance of Sequential and Parallel Programs

Margaret Martonosi
Princeton University

Anoop Gupta
Stanford University

Thomas E. Anderson
University of California at Berkeley

When programmers know where and why memory bottlenecks occur, they can make the appropriate program transformations to enhance performance, with the help of this tool's detailed statistics.

Because modern computer processors are speeding up at a much faster rate than main memory, relative latencies from processors to main memory have dramatically increased. Main memory latencies can reach tens of processor cycles in uniprocessors and over a hundred cycles in multiprocessors. These figures are likely to grow even larger over the next decade, and systems architects have responded 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 memory performance if cache misses are common.

To improve program memory performance, programmers and compiler writers can transform the application so that its memory-referencing behavior better exploits the memory hierarchy. The challenge in achieving these program transformations is overcoming the difficulty of statically analyzing or reasoning about an application's referencing behavior and interactions. In addition, many performance-monitoring tools collect high-level information that is inadequately detailed to analyze specific memory performance bugs.

Here we describe MemSpy, a performance-monitoring tool we designed to help programmers discern where and why memory bottlenecks occur. MemSpy guides programmers toward program transformations that improve memory performance through detailed statistics on cache-miss causes and frequency. Because of the natural link between data-reference patterns and memory performance, MemSpy helps programmers comprehend data structure and code segment interactions by displaying statistics in terms of both the program's data *and* code structures, rather than for code structures alone.

MemSpy uses cache simulations to gather detailed memory statistics. Since efficiency is a key concern in simulation-based performance monitoring, we have evaluated two performance optimizations—*hit bypassing* and *reference-trace sampling*—that reduce the execution time overhead required to gather such information. Together, these techniques reduce simulation time by nearly an order of magnitude. For a simple memory simulator and sequential applications, the time to run a program with MemSpy is 3 to 10 times as long as the time to run the program normally. For parallel applications, the overhead increases to factors of 8 to 25. Our experience in using MemSpy to tune several sequential and parallel applications demonstrates that it effectively profiles memory performance at speeds that make it an attractive alternative to other approaches. (See "Previous performance-monitoring approaches" sidebar.)

0018-9162/95/$4.00 © 1995 IEEE

TUNING PROGRAM MEMORY BEHAVIOR

Memory performance bottlenecks can be diagnosed and tuned by using tools that provide specific information on program memory behavior. The first step for such tools is to locate bottlenecks by identifying data structures or code where the memory stall time attributed to it is both *large* in an absolute sense, and *larger than expected* in a relative sense. The next step—determining *why* memory bottlenecks are occurring—depends on the types of performance bugs encountered, as described next. Based on where and why bottlenecks are occurring, programmers can devise fixes to avoid them.

Common memory performance bugs

Three frequently occurring memory performance bugs in sequential and parallel applications are *cache interference*, *poor spatial locality*, and *interprocessor sharing*. Because of their differing characteristics, they are recognized and tuned in different ways. Tools like MemSpy provide information that will help distinguish these cases.

CACHE INTERFERENCE. Bottlenecks arise from interference when multiple memory lines mapping to the same cache line compete for cache space and cause excessive misses. Tools can help programmers identify interference by pointing out data structures with excessive *replacement misses*. These misses occur if a particular memory line has been referenced but replaced out of the cache by an intervening reference to another line competing for the same cache space. In such programs, simple adjustments or coloring strategies to stagger data structures in the cache can significantly improve performance.

POOR SPATIAL LOCALITY. Performance can also suffer from poor spatial locality, which occurs when a program's data access order is not well correlated with the data storage order. As a result, the program might not efficiently use the full line of data fetched on a cache miss. Poor spatial locality can be especially pronounced in parallel code, because data structures that are stored and accessed contiguously in a sequential program might be distributed over several processors in a parallel program. Poor spatial locality can frequently be deduced from seeing many memory stalls due to *first-reference misses*. (Other metrics can also be useful for identifying poor spatial locality—for example, counting the number of bytes accessed per cache line between cache misses.) Spatial locality can then be improved by restructuring data accesses or storage schemes to pack cache lines more efficiently with useful data.

INTERPROCESSOR SHARING. Finally, in multiprocessors, the performance of some programs might suffer due to excessive interprocessor sharing, resulting in heavy cache-coherence traffic. Because 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 that would benefit from restructuring to minimize the required communication. These misses occur when a processor re-references

Previous performance-monitoring approaches

Earlier performance tools did not generally support memory performance tuning per se. For example, Gprof[1] is a widely used tool that ranks procedures by expended execution time but gives programmers little insight into what causes bottlenecks and whether memory behavior may be responsible. Other tools (for example, Quartz[2]) have been designed specifically to identify sychronization and computation bottlenecks in parallel programs. However, like Gprof, Quartz doesn't specifically examine application-memory performance.

Mtool,[3] on the other hand, supports memory performance tuning. It identifies code fragments that constitute memory bottlenecks by presenting statistics on the amount of memory stall time per basic block. (Stall time is the wait time associated with cache misses.) However, this level of detail doesn't explain why bottlenecks occur or which data structures are most responsible.

At the other extreme of detail, SHMAP[4] (Shared-Memory Access Pattern) provides a reference-by-reference animation of program memory behavior. Cache animation can sometimes help programmers distinguish memory performance pitfalls, but the lack of summary information or automatic analysis can make tuning difficult. Reference patterns and performance bugs may be particularly hard to understand in irregular, nonscientific code. Here, the volume of animation data can be overwhelming.

MemSpy, through statistics on data and code, and also on cache-miss causes and frequency, addresses some of these tools' shortcomings.

References

1. S.L. Graham, P.B. Kessler, and M.K. McKusick, "An Execution Profiler for Modular Programs," *Software: Practice & Experience*, Vol. 13, No. 8, Aug. 1983, pp. 671-685.
2. T.E. Anderson and E.D. Lazowska, "A Tool for Tuning Parallel Program Performance," *Proc. ACM SIGmetrics Conf. Measurement and Modeling of Computer Systems*, ACM, New York, May 1990, pp. 115-125.
3. A.J. Goldberg and J.L. Hennessy, "Mtool: An Integrated System for Performance Debugging Shared Memory Multiprocessor Applications," *IEEE Trans. Parallel and Distributed Systems*, Vol. 4, No. 1, Jan. 1993, pp. 28-40.
4. J. Dongarra et al., "A Tool to Aid in the Design, Implementation, and Understanding of Matrix Algorithms for Parallel Processors," *J. Parallel and Distributed Computing*, June 1990, Vol. 9, No. 2, pp. 185-202.

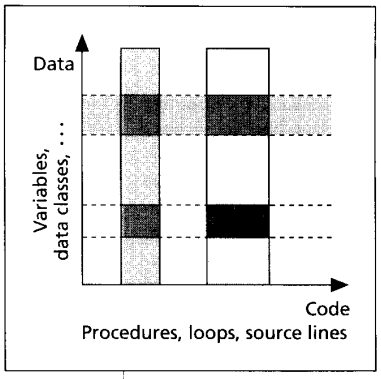

Figure 1. Data-oriented statistics offer a second dimension to the information programmers can use about memory performance. This diagram abstractly illustrates a program divided into data and code statistics bins.

data after a cache invalidation from another processor. Programs can also be restructured to reduce false sharing, in which multiple processors actively read and write different variables on the same cache line.

From the characteristics of these performance bugs, we can see that understanding the causes of cache misses can greatly help a programmer tune memory for enhanced performance.

MemSpy's approach to memory tuning

Effective performance tools must give programmers information regarding the location and cause of code bottlenecks. In determining where they exist, MemSpy presents statistics about the application's data and code structures. In determining the cause of bottlenecks, MemSpy gives statistics on the frequency and causes of cache misses, thereby guiding programmers (and compilers) toward effective program transformations.

```
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. Pseudocode for MatMul, the blocked-matrix multiply program in the case study.

DATA- AND CODE-ORIENTED STATISTICS. Figure 1 abstractly illustrates possible code- and data-oriented subdivisions in a program. Because of the inherent link between memory performance and the program's access patterns to particular data structures, these statistics can be crucial to reasoning about memory behavior. (Cprof,[1] developed independently by Lebeck and Wood, is the only other tool that features data-oriented statistics.)

Data-oriented statistics are especially helpful in cases where a particular data structure constitutes a memory bottleneck and accesses to that structure are distributed across several procedures. For example, in Pthor (a SPLASH benchmark2[2]), the ElementArray structure causes more cache misses than any other program variable, but these misses are distributed across several procedures. Code-oriented statistics cannot emphasize ElementArray's performance problems as well as data-oriented statistics can, because no one section of code causes the bottleneck. In this case, therefore, the bottleneck lends itself to data-oriented viewing. Furthermore, combinations of data- and code-oriented statistics can be instrumental in isolating memory bottlenecks resulting from a single data structure in a large procedure.

STATISTICS ON CAUSES OF CACHE MISSES. After isolating program bottlenecks, programmers need progressively more detail to understand the causal relationships. Recall that cache misses occur for one of three reasons: first reference, replacement, or invalidation.

Regardless of how important detailed information is to understanding memory performance, most existing tools have provided no statistics on memory behavior, or, like Mtool,[3] give only high-level information to identify code that causes bottlenecks. The lack of details stems partly from the difficulty of gathering fine-grained memory statistics, which require fine-grained monitoring through specialized hardware or software simulation. Specialized hardware, however, can limit the tool's general applicability; on the other hand, software simulation can be too slow. As we will show, techniques developed for MemSpy improve the efficiency of simulation-based monitoring.

USING MEMSPY: A CASE STUDY

For case study purposes, we ran MemSpy with a program called MatMul, which performs a blocked-matrix multiply. Although MatMul is a simple application, it is also a case where the common intuitions about the code's behavior are incorrect; namely, choosing block sizes to fit into the cache is not necessarily sufficient for good performance. Tools—specifically MemSpy—guide the programmer in locating and eliminating bottlenecks such as the ones we demonstrate in this blocked code.

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

As Lam et al.[4] reported, the performance of such blocked operations is often erratic, and is sensitive to even small changes in matrix size, block size, and cache organization. For a DECstation 3100, they report that a 300×300-element blocked-matrix multiply (with 56×56 block size) executes at 4 million floating-point operations per second (Mflops), while by contrast, an only slightly smaller 293×293 matrix with the same block size executes at only 2 Mflops on the same machine.

Our case study illustrates how data-oriented statistics powerfully focus the programmer's attention on problematic code and how cache-miss statistics on the causes

of cache misses are intrinsic to understanding why the performance bottleneck is occurring.

Tuning using MemSpy

To emphasize MatMul's performance bug (in this case, cache interference), we show MemSpy's statistics on one of Lam's[4] poor-performance cases. We multiply two 293 × 293-element matrices together, using a block size of 56. (A single 56 × 56 block requires roughly 25 Kbytes and should easily fit into the 64-Kbyte cache.)

MemSpy first displays the output shown in Figure 3. The program procedures—Block, CheckProduct, Matmat—are indicated along the *X* axis of the graph, and the processor-cycle time spent on behalf of each procedure appears on the *Y* axis. The bar for each procedure categorizes the elapsed time according to how much was spent in computation and how much in memory stalls. (For parallel programs, the bar would indicate synchronization time as well.)

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

In the block routine, the execution time is concentrated in line 13 of Figure 2's pseudocode. In this line, the appropriate elements of *X* (r, in line 13) 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 do not help determine the relative contributions of the three matrices toward the bottleneck. To further understand and tune this code, programmers need information on whether the bottleneck is caused by a single matrix or by an interaction between matrices.

DATA-ORIENTED BREAKDOWN. To display more information about the stall time contributed by each data structure, MemSpy lets programmers click on the memory portion of the block routine's bar. This display, shown in Figure 4, breaks down the memory stall time into components incurred by each data structure in the procedure. The data-oriented statistics show that this routine's bottleneck is almost entirely due to cache misses on references to the *Y* matrix. These misses produce over 85 percent of the total stall time in the program. Since the *Y* matrix is the one that was blocked for good data reuse, it is surprising that *Y* is responsible for so much stall time.

DETAILED STATISTICS ON CAUSES OF MISSES. While MemSpy has thus far enlightened us as to where the bottleneck is, we don't yet know its cause. To learn more, we can click on the *Y* bar in Figure 4 to display information on the causes of misses. The bar chart in Figure 5 breaks down the cache-miss causes 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

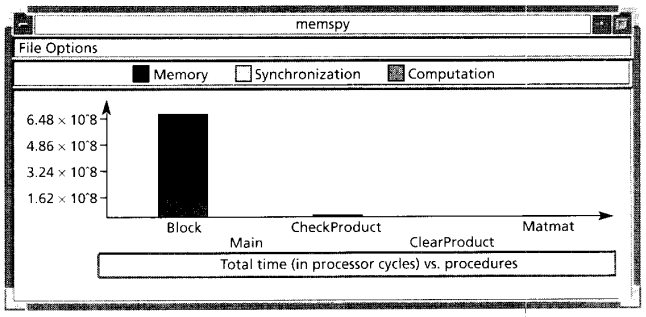

Figure 3. In code-oriented display, MemSpy presents overview statistics for the MatMul program.

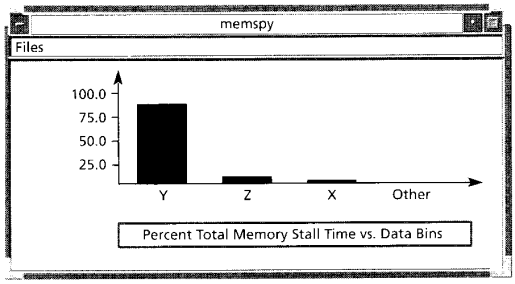

Figure 4. In data-oriented display, MemSpy presents memory stall time in the block procedure attributed to the *X*, *Y*, and *Z* matrices in the MatMul program.

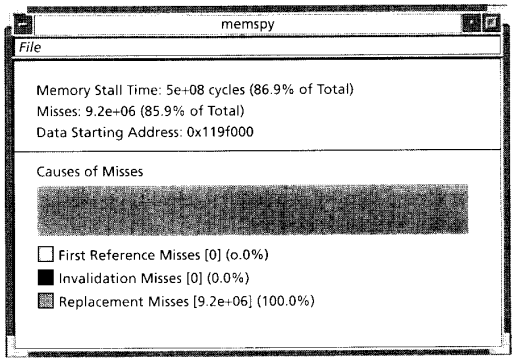

Figure 5. MemSpy displays detailed statistics on the *Y* matrix in the block procedure of the MatMul program.

cache misses. (The first-reference misses for the *Y* matrix occur in a separate initialization routine.)

The many replacement misses incurred by *Y* indicate that the bottleneck is probably related to cache-interference effects. To understand the cause of the memory bottleneck, however, programmers must know which accesses are causing the cache replacements. Clicking on the replacements portion of the "cause of misses" bar in Figure 5 displays a breakdown (not shown here) of what causes these replacements. Surprisingly, over 95 percent of the replacements are due to the *Y* matrix. MemSpy, in

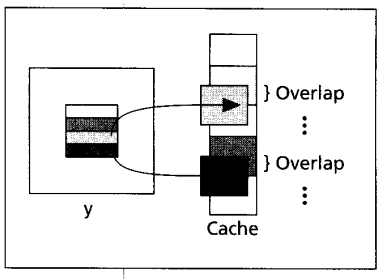

Figure 6. This diagram illustrates self-interference in a blocked matrix *Y*, caused by subrows that map to common lines in the cache.

this case, informs us that (1) the bottleneck is in the *Y* matrix, (2) it's caused by excessive cache interference, and (3) this interference is in fact self-interference, since the replacements are caused by the *Y* matrix itself.

Programmers can determine that self-interference occurs here because the subrows within the currently used block of *Y* aren't stored contiguously and thus do not map neatly across the whole cache. Rather, as shown in Figure 6, subrows are separated by intervening amounts of matrix storage. This leads to cases where some subrows map on top of one another in the cache, while other portions of the cache remain unused. This effect can be minimized if the programmer chooses a block size with less interference or copies the block so that it occupies a contiguous region of memory.

Without MemSpy's data-oriented statistics, programmers would find it difficult to see which matrix was causing the memory bottleneck. Furthermore, without detailed information on the causes of misses and replacements, programmers would have difficulty interpreting this situation as self-interference.

Other applications of MemSpy

In addition to the MatMul case study, MemSpy has also been used to tune other programs as well. For example, it has identified performance bugs due to

- false sharing and a "vestigial" (incremented but unused) variable in LocusRoute, a SPLASH benchmark;
- self-interference in the ElementArray in Pthor[2];
- poor spatial locality in a sequential volume-rendering program, Vrender[5]; and
- shared accesses to a private variable in a parallel version of Vrender.

MEMSPY IMPLEMENTATION

Having confirmed the utility of MemSpy's detailed, data-oriented statistics through our case study discussion, we now examine the issues involved in building MemSpy, which will explain our rationale in creating this tool as we did. In implementing tools like MemSpy, there are two principal design decisions: how to gather detailed information efficiently, and at what code and data granularity to display this information.

Simulation-based monitoring

To display the performance information illustrated in Figures 3 through 5, MemSpy must monitor code at the granularity of individual memory references. To accomplish this, we implemented MemSpy based on simulation. (Simulation's main advantage is that it can be general and portable, since it does not require specialized hardware support.)

MemSpy was built on top of the Tango Lite reference generator.[6] 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, events of interest are instrumented at compile-time with additional code to call event simulators. For MemSpy, we instrumented the following events:

- memory references,
- procedure calls and returns,
- memory allocations, and
- synchronizations.

For each event, instrumentation code executes 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.

Granularity of data- and code-oriented statistics

For data- and code-oriented statistics to be effective, it's essential for the tool to present them at useful granularities. MemSpy statistics are organized and managed as *statistics bins* containing program information such as memory time, number of cache misses, and causes of cache misses. The information is collected for either specific data or code sections or for data-code pairings in the application.

Choosing the right granularity for both data- and code-oriented statistics is important because statistics that are too coarse-grained may not localize bottlenecks. On the other hand, excessively fine-grained statistics may not adequately aggregate activity to distinguish bottlenecks. Moreover, statistics that are too fine grained may also be inefficient to implement because of *storage inefficiency* (more memory is needed to maintain very fine grained statistics) and *execution time inefficiency* (extra time is needed to manage and update the larger number of statistics bins).

In its code-oriented statistics, MemSpy maintains information separately 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 sufficiently fine grained 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, and the program counters on the stack, identically match the program-counter stack for a previously initialized bin, the statistics for this new memory range 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.

These methods for monitoring programs and generating statistics at useful granularities have proven effective. These techniques have been used successfully on a variety of programs, including the SPLASH parallel benchmarks. In addition, as we next discuss, the implementation is efficient enough to allow these statistics to be generated with competitive runtime overheads as well.

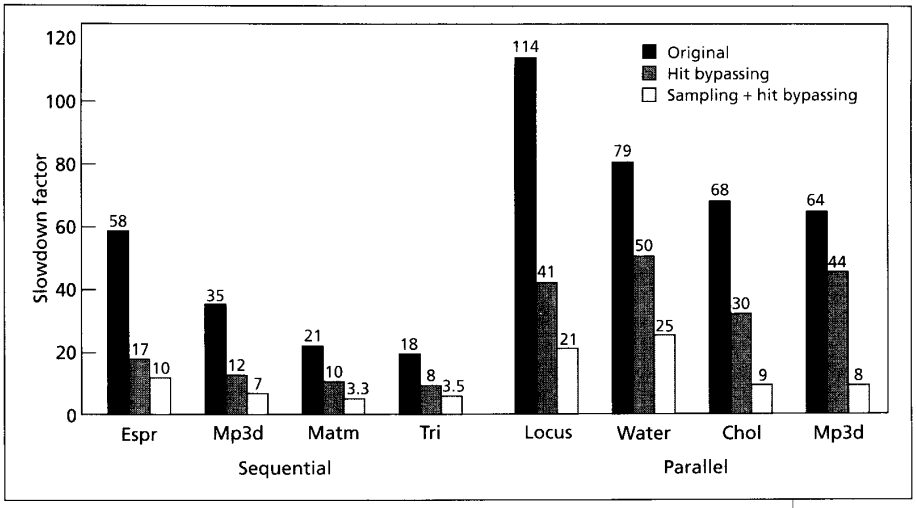

Figure 7. The eight groupings of three columns each show MemSpy's execution time overhead figures for baseline, hit bypassing, and sampling (with hit bypassing) versions run on the eight sequential and parallel benchmark applications.

MEMSPY PERFORMANCE

To evaluate MemSpy's expected runtime performance, we begin by presenting its execution time overheads for sequential and parallel applications on a baseline implementation. All performance measurements were made on a DECstation 5000/240.

While the baseline simulation overhead is already acceptable, we also present the results of two performance optimizations that further reduce MemSpy's overhead, bringing it into more interactive regimes:

1. *hit bypassing*, which optimizes simulation for the more-common cache hits, and
2. *sampling*, which generates program statistics based on samples of references, rather than the full trace.

Measurement setup

To evaluate MemSpy's overheads, we used a simple memory simulator with a single direct-mapped cache per processor. In this simulator, cache hits execute in a single processor cycle; cache misses require a fixed, parameterized latency to be serviced. Network contention is not modeled. All heap references and static data references in the code are instrumented for simulation. For multiprocessors, we simulate an invalidation-based protocol. When simulating sequential machines, we assume a 128-Kbyte direct-mapped data cache with 32-byte lines. For parallel machines, we assume 16 CPUs, each with a 64-Kbyte direct-mapped cache with 32-byte lines.

We ran eight programs spanning a variety of domains including numerical computation, scientific, and engineering applications. They featured a range of cache-miss rates (0.2 to 18.2 percent) and code sizes (500 to 14,000 lines). The four sequential applications were blocked-matrix multiply; Espresso from the SPEC benchmark suite; Tri, a sparse triangular matrix solver; and a uniprocessor run of Mp3d, a SPLASH benchmark. The four parallel applications were from the SPLASH benchmark suite: Mp3d, Cholesky, Water, and LocusRoute. By evaluating MemSpy's monitoring performance with such diverse, substantial applications, we demonstrate the real-world utility of MemSpy's statistics.

MemSpy's baseline performance

The leftmost column of each benchmark grouping in Figure 7 shows MemSpy's baseline performance overheads. (The other columns reflect hit-bypassing and sampling overheads.) We compute the multiplicative (slowdown) factors by dividing the time for a MemSpy run by the time for an uninstrumented run of the same program. For the sequential applications, baseline execution-time overheads range from a factor of 18 to a factor of 58. Baseline overheads for the parallel applications, ranging from factors of 64 to 114, exceed those for the sequential benchmarks because of increased complexity in simulating the parallel machine. (For the parallel applications in Figure 7, the overhead is shown relative to the program's execution on a uniprocessor. To estimate overheads relative to execution time on a multiprocessor, the overheads shown would be multiplied by the expected program speedup.)

For both sequential and parallel benchmarks, 97 percent or more of MemSpy's overhead results from processing memory references. Figure 8 on the next page shows the action sequence MemSpy performs on each simulated memory reference. These actions are categorized into three overhead types:

1. *context switches*—the register save and restore operations when entering and exiting the simulator (25 percent),
2. *memory simulation* (roughly 45 percent of the overhead), and
3. *statistics bin searches* (30 percent).

As will be seen, the hit-bypassing and sampling optimizations reduced the time MemSpy spends in each of these three categories.

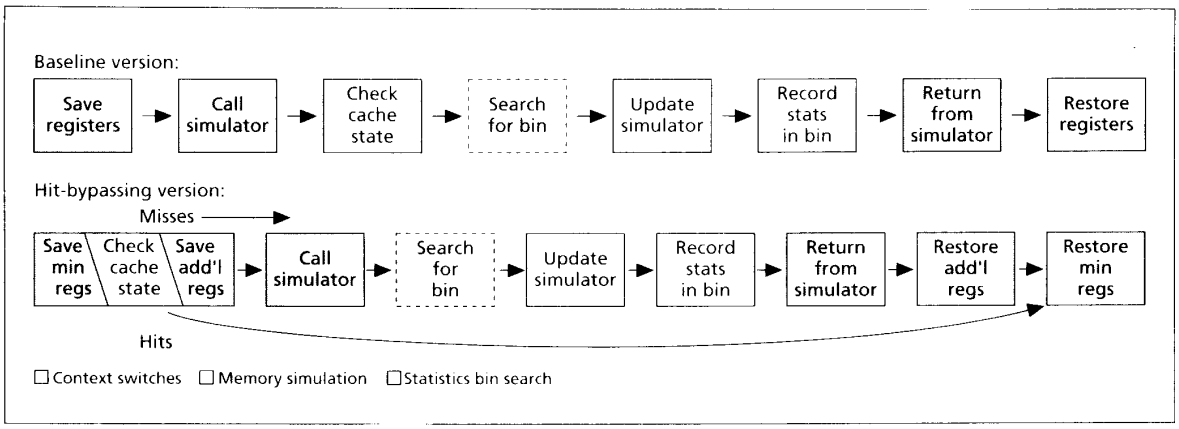

Figure 8. These flowcharts show the sequence of actions MemSpy takes in the baseline implementation and also with the hit-bypassing optimization.

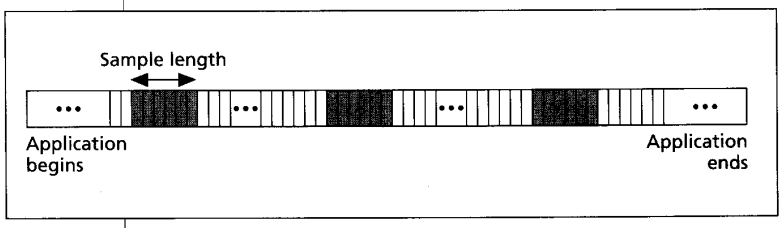

Figure 9. The diagram illustrates time samples in an application reference trace. Shaded regions correspond to the groups of references MemSpy will simulate.

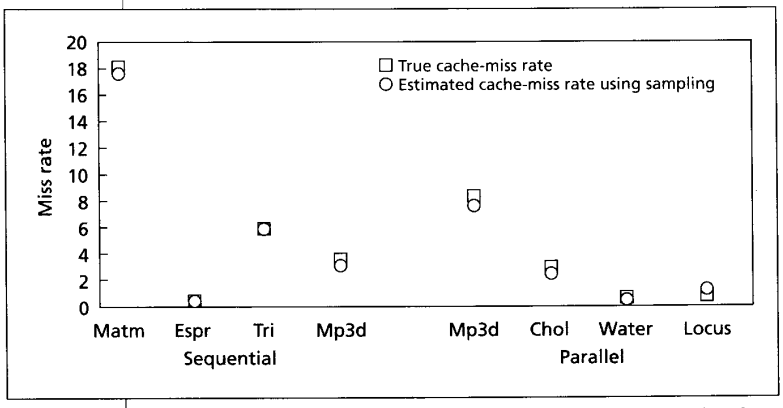

Figure 10. Estimated and true cache-miss rates are shown for the benchmark sequential and parallel applications.

Optimizing MemSpy performance with hit bypassing

Hit bypassing originates from the observation that memory events of interest, from a performance-tuning perspective, are events that incur memory stalls. For many architectures, only cache misses incur stalls. By keeping statistics only on cache misses, we can minimize the overhead incurred on cache hits, which are almost always the majority of the references.

In the baseline implementation, we save a full set of registers, then call the simulator to check whether the reference is a cache hit or miss. The cache check itself uses very few registers (roughly four to seven, depending on the simulator and the implementation). If the reference is a hit, we return to the application, restoring all registers, although only a handful were used. The hit-bypassing path in Figure 8 shows how the cache-hit check is embedded into the register saves. Thus, we initially save only the registers required to check whether the reference is a cache hit or miss. If the reference is a cache miss, we complete the rest of the register saves and continue simulating. If the reference is a 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's groupings show the significant performance benefits of hit bypassing. With this technique, performance improves by factors of 1.5 to 3.4 compared to the baseline code. MemSpy's overhead factors now range from 8 to 17 for sequential code and from 30 to 50 for parallel code.

Optimizing MemSpy performance with sampling

In general, reference-trace sampling refers to the process of estimating cache behavior based on simulating only portions of a reference trace, rather than the full trace. Sampling is intuitively promising because it should let us approximate program behavior reasonably accurately without incurring the performance cost of a full simulation. (In fact, other forms of sampling, such as program-counter sampling, are already used in performance tools like Gprof.[7]) On the other hand, there is an inherent trade-off between the speed and accuracy of the tool:

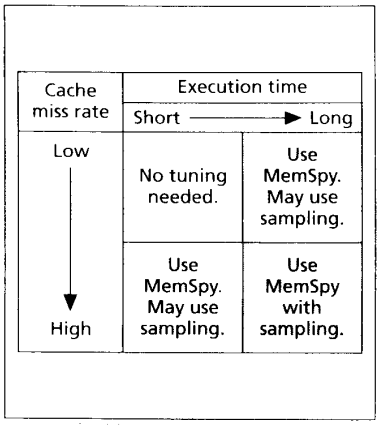

Figure 11. This illustration of application characteristics and MemSpy usage demonstrates how applications most suited to sampling coincide with those most in need of tuning.

Simulating fewer references will generally make the tool faster, while simulating more references will tend to make the tool more accurate. Our work shows that within the context of a performance debugging tool, trace sampling can be used effectively to improve the tool's performance while retaining acceptable accuracy.

Time sampling, 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 the *sample length* (number of references contained in each sample), and the *number of samples*. A third dependent parameter is the *sampling ratio*, which is the total references in the samples divided by the total number of references in the run. Reference-trace sampling is discussed in more detail elsewhere.[8-10]

SAMPLING'S EFFECT ON ACCURACY. Our results show that time sampling accurately reproduces cache statistics derived from a full simulation. For example, based on a 10 percent sampling ratio, Figure 10 compares estimated cache-miss rates from sampling to the program's true miss rate calculated over all references. For the sequential applications, the samples contain 0.5 million 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 and estimated miss rates is 0.74 percent, with small relative deviations as well. A detailed report[10] discusses more comprehensive sampling accuracy results, based on varying numbers and lengths of samples, cache sizes, and in the parallel case, numbers of processors.

Our results show that for sequential benchmarks with caches smaller than 1 Mbyte, miss rates can be estimated with absolute deviations under 0.5 percent and relative deviations of 10 percent or less. These estimates require sample lengths of only 0.5M references or less. To achieve accuracy when simulating 1-Mbyte caches or larger, at least 4M references per sample are required to prime the cache. However, this still allows for aggressive sampling ratios on many applications. Simulating parallel applications requires slightly longer samples because parallel machines generally have more total cache memory. However, required sample length is not linearly 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 references have little need for MemSpy tuning, so their potential for higher sampling error is less relevant.

SAMPLING'S EFFECT ON PERFORMANCE. Our primary purpose in using time sampling is to improve MemSpy's monitoring speed. 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 earlier. We expect at least a modest performance improvement on cache hits (which can be bypassed anyway) and a large performance improvement on cache misses. 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.

Referring back to Figure 7, the lightly shaded bars show simulation overhead for time sampling with the sampling parameters discussed earlier. 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.

BOTH SEQUENTIAL AND PARALLEL APPLICATIONS are currently facing a growing gap between processor and memory speeds; consequently, performance slowdowns due to memory stalls can be substantial. Despite this trend, most performance-monitoring tools have lagged in providing support for identifying and characterizing memory

bottlenecks. MemSpy, however, can significantly help tune memory performance by isolating these bottlenecks through detailed statistics.

The MemSpy monitoring tool lets programmers view and analyze program behavior through the additional dimension of data-oriented statistics. Together with code-oriented statistics, these are a powerful aid to memory performance analysis and tuning. In addition, by characterizing the predominant causes of cache misses for each data structure, MemSpy helps programmers better understand the causes of memory bottlenecks and how to fix them.

Finally, key to providing such detailed statistics is the ability to gather program information efficiently. By combining hit-bypassing and reference-trace sampling optimizations, MemSpy overheads dropped from baseline factors of 18 to 58 for sequential applications to 3 to 10. Similarly, the overheads dropped from baseline factors of 64 to 114 for parallel applications to 8 to 25. We have shown that simulation-based performance monitors like MemSpy can offer a feasible, effective, and low-overhead alternative to other data collection methods. ■

Acknowledgment

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.

References

1. A.R. Lebeck and D.A. Wood, "Cache Profiling and the SPEC Benchmarks: A Case Study," *Computer*, Vol. 27, No. 10, Oct. 1994, pp. 15-26.
2. J.P. Singh, W.D. Weber, and A. Gupta, "Stanford Parallel Applications for Shared-Memory," *Computer Architecture News*, Mar. 1992, Vol. 20, No. 1, pp. 5-44.
3. A.J. Goldberg and J.L. Hennessy, "Mtool: An Integrated System for Performance Debugging Shared Memory Multiprocessor Applications," *IEEE Trans. Parallel and Distributed Systems*, Vol. 4, No. 1, Jan. 1993, pp. 28-40.
4. M. Lam, E. Rothberg, and M. Wolf, "The Cache Performance and Optimizations of Blocked Algorithms," *Proc. ASPLOS 4*, ACM, New York, Apr. 1991, pp. 63-74.
5. P. Lacroute and M. Levoy, "Fast Volume Rendering Using a Shear-Warp Factorization of the Viewing Transformation," *Proc. ACM Siggraph*, ACM, New York, July 1994.
6. S.R. Goldschmidt, *Simulation of Multiprocessors, Speed and Accuracy*, doctoral dissertation, Stanford Univ., Stanford, Calif., June 1993.
7. S.L. Graham, P.B. Kessler, and M.K. McKusick, "An Execution Profiler for Modular Programs," *Software: Practice & Experience*, Vol. 13, No. 8, Aug. 1983, pp. 671-685.
8. R.E. Kessler, M.D. Hill, and D.A. Wood, "A Comparison of Trace-Sampling Techniques for Multimegabyte Caches," Tech. Report 1048, Computer Sciences Dept., Univ. of Wisconsin, Madison, Wis., Sept. 1991.
9. M. Martonosi, A. Gupta, and T. Anderson, "Effectiveness of Trace Sampling for Performance Debugging Tools," *Proc. ACM SIGmetrics Conf. Measurement and Modeling of Computer Systems*, ACM, New York, May 1993.
10. M.R. Martonosi, *Analyzing and Tuning Memory Performance in Sequential and Parallel Programs*, doctoral dissertation, Stanford Univ., Stanford, Calif., Dec. 1993. Also Stanford Computer Systems Laboratory Tech. Report CSL-TR-94-602.

Margaret Martonosi *is an assistant professor of electrical engineering at Princeton University. Her research interests include both architecture and applications issues in high-performance computers, with a focus on efficient computer-performance analysis and performance-monitoring tools.*

Martonosi earned a BS from Cornell University, and an MS and a PhD from Stanford University, all in electrical engineering. She is a 1995 recipient of the NSF Faculty Early Career Development Award. She is a member of IEEE and the IEEE Computer Society.

Anoop Gupta *is an associate professor of computer science and electrical engineering at Stanford University. He co-led the design and construction of the Stanford DASH multiprocessor and is currently working on the next-generation machine. Prior to joining Stanford, he was on the research faculty of Carnegie Mellon University. His research interests are in hardware and software design for scalable parallel computer systems.*

Gupta received a Digital Equipment faculty development award from 1987 to 1989 and the NSF Presidential Young Investigator Award in 1990, and he held the Robert Noyce faculty scholar chair in the School of Engineering at Stanford for 1993-1994. He received a PhD from Carnegie Mellon University.

Thomas E. Anderson *is an assistant professor in the Computer Science Division at the University of California at Berkeley. His research interests include operating systems, computer architecture, multiprocessors, high-speed networks, massive storage systems, and computer science education.*

Anderson received an AB in philosophy from Harvard University, and an MS and PhD in computer science from the University of Washington. He won the NSF Young Investigator Award in 1992 and the Alfred P. Sloan Fellowship in 1993. Anderson has coauthored numerous award papers presented at recent conferences, including SIGmetrics, symposia on operating systems principles, ASPLOS, and USENIX conferences.

Readers can contact Martonosi at martonosi@princeton.edu, Gupta at ag@cs.stanford.edu, and Anderson at tea@cs.berkeley.edu.