MemSpy: Analyzing Memory System Bottlenecks in Programs

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

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

Abstract

To cope with the increasing difference between processor and main memory speeds, modern computer systems use deep memory hierarchies. In the presence of such hierarchies, the performance attained by an application is largely determined by its memory reference behavior— if most references hit in the cache, the performance is significantly higher than if most references have to go to main memory. Frequently, it is possible for the programmer to restructure the data or code to achieve better memory reference behavior. Unfortunately, most existing performance debugging tools do not assist the programmer in this component of the overall performance tuning task.

This paper describes MemSpy, a prototype tool that helps programmers identify and fix memory bottlenecks in both sequential and parallel programs. A key aspect of MemSpy is that it introduces the notion of data oriented, in addition to code oriented, performance tuning. Thus, for both source level code objects and data objects, MemSpy provides information such as cache miss rates, causes of cache misses, and in multiprocessors, information on cache invalidations and local versus remote memory misses. MemSpy also introduces a concise matrix presentation to allow programmers to view both code and data oriented statistics at the same time. This paper presents design and implementation issues for MemSpy, and gives a detailed case study using MemSpy to tune a parallel sparse matrix application. It shows how MemSpy helps pinpoint memory system bottlenecks, such as poor spatial locality and interference among data structures, and suggests paths for improvement.

1 Introduction

While processor speeds have increased by more than two orders of magnitude over the last decade, main memory (DRAM) speeds have barely increased by a factor of two [10]. In response to this ever increasing speed of processors, modern computer systems are designed with sophisticated memory systems that include small on-chip caches, large external caches, and interleaved main memory. The memory hierarchies are even deeper and more complex for multiprocessors. One consequence of these deep memory hierarchies is that cache miss latencies have become extremely large when counted in processor clocks. As a result, if an application is to attain good performance, it must exhibit memory reference behavior that exploits caches well — the reference pattern must exhibit high spatial, temporal, and processor locality [1].

The memory reference behavior of an application, at the most basic level, depends on the intrinsic nature of the application; however, the programmer still has considerable flexibility in manipulating the algorithm, data structures, and program structure to change the memory reference patterns in order to better exploit the memory hierarchy. We find that a tool designed to help in this task must (i) separately report processor and memory time, so that programmers can discern when the memory behavior is the bottleneck, (ii) link bottlenecks back to *data* objects, as well as code, and (iii) give memory statistics at a level of detail that allows the programmer to identify and fix the bottlenecks. The latter, in our experience, requires detailed information on which code and data objects are responsible for the most cache misses, and reasons for why those misses occurred. Understanding the cause of misses is important, since some of those misses may be essential misses (e.g., cold-start misses), while others may be more easily optimized away (e.g., replacement or invalidation misses).

Most existing performance debugging tools [2, 3, 5, 7, 8, 9, 15], do not provide the detailed information mentioned above. In this paper, we describe MemSpy, a prototype tool that provides such information and helps programmers improve the memory reference behavior of applications. The paper outlines two case studies showing the usefulness of MemSpy's detailed information in tuning applications with poor memory referencing behavior.

While the detailed information provided by MemSpy is beneficial, it comes at the price of higher overhead. For applications whose performance is limited by memory reference behavior, the power of the information provided by MemSpy warrants the extra overhead; however, one may first want to perform more general code optimizations using simpler tools. Consequently, we envision MemSpy as part of a hierarchy of performance debugging tools. At higher levels we expect tools that have low overhead and that provide only basic application statistics. These tools will identify coarse bottlenecks in the program. Lower in the hierarchy we expect tools like MemSpy that provide detailed outputs with somewhat higher overheads.

In *Proc. 1992 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*

The remainder of the paper is structured as follows. The next section describes related work. Following that, in Section 3, we present an overview of MemSpy. Next, we present a case study using MemSpy to tune the performance of a parallel sparse matrix application. Section 5 gives the implementation details and presents data on MemSpy's execution time overhead. In Section 6, we discuss our experiences with the system, and Section 7 presents conclusions.

2 Related Work

In recent years, there has been a surge of interest in developing tools to support application performance debugging, rather than simply correctness debugging. Many performance debuggers now exist, occupying points along a spectrum from high-level, low-overhead tools to more detailed, higher overhead tools. At one end of this spectrum, tools such as gprof are intended to produce simple, high-level statistics with minimal overhead. At the other end, tools like SHMAP and MemSpy are intended to give more detailed information to the user with commensurate increase in overhead. This section discusses a selection of relevant tools over the range of this spectrum, and motivates our choice for MemSpy's level of detail.

Gprof [9] is a commonly used execution profiler for sequential programs. By giving a hierarchically arranged profile of the execution time of a program's procedures, it offers a high level view of which procedures have the greatest potential for optimization. Gprof, however, does not distinguish between computation time and memory system time, and it therefore provides no help in locating memory system bottlenecks.

Quartz [2] is an execution profiler for parallel programs; in many ways, it is an extension of gprof into the parallel domain. Quartz reports *normalized processor time* as its primary metric. This is defined as the total time all processors spend in each section of code, divided by the number of other processors concurrently busy when that section of code was being executed. Quartz presents normalized processor time statistics for a program's procedures, and also reports the amount of normalized processor time spent in critical sections. Normalized processor time emphasizes the point that optimizing an application's less parallel code can have more impact on overall performance than improving code executing with high parallelism. Like gprof, Quartz aggregates computation time and memory system time together, making it difficult to determine when the memory behavior is a bottleneck. However, Quartz is quite good at focusing the user's attention on those procedures that are most critical to performance; we have incorporated some of Quartz's functionality into MemSpy.

MTOOL [7, 8] is a system specifically designed to detect memory bottlenecks in both sequential and parallel programs. MTOOL's basic performance metric is the difference between a program's actual execution time with non-ideal memory system behavior, and the execution time of the same code with an ideal memory system. This difference is the amount of execution time for which the processor was stalled due to memory system delays. This information is presented for loops and procedures within the program. While MTOOL is a useful tool for focusing attention on the primary memory bottlenecks in the code, it provides no statistics on the specific behavior (cold-start misses, interference, sharing, etc.) responsible for the problems. Furthermore, since MTOOL's output is procedure and loop oriented, it often provides little or no insight into which data objects are responsible for the poor memory system behavior.

Another tool for studying memory referencing patterns in programs is SHMAP [5]. This system annotates sequential FORTRAN programs, collects memory reference traces, and produces an animated picture of references to the program's main data objects. While SHMAP is useful for detecting patterns in references to array data objects, it offers only limited help for references to more complex data structures, such as lists and trees. SHMAP also offers little summary information about the program's behavior; miss rates are computed only on a per-processor, rather than per-data-object or per-procedure basis, and the user must glean information on cache replacements by carefully examining the animation. For long running simulations of program execution, watching the animations and discerning patterns may become quite tedious.

In summary, current performance monitors exist at many points of the spectrum of possible tradeoffs between efficiency and level of detail in their output. We introduce MemSpy to provide a useful level of detail in memory system performance statistics that has not yet been explored; further, we examine methods for reducing MemSpy's overhead as much as possible.

3 MemSpy Overview

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 that focuses the programmer's attention on the problem areas in the application. Then, it helps the programmer fix the bottlenecks by providing detailed information on the application's memory behavior at these bottlenecks. MemSpy's key features can be summarized as follows:

- Both data oriented and code oriented output.
- An initial attention-focusing mechanism based on the fraction of time spent stalled for memory.
- Detailed information on the causes of poor memory performance.
- Applicability to both serial and parallel applications.

Traditionally, performance monitoring tools have presented primarily code oriented output; that is, the statistics are presented for different procedures, loops, or source lines in the code. However, many performance bottlenecks are more naturally viewed in terms of data oriented statistics, where statistics are presented for different application data objects. In contrast to earlier approaches, we believe

that both data and code oriented statistics are important for performance debugging; they provide orthogonal views of program performance, and the combination of the two can be quite powerful. For example, consider `pthor` [17], a parallel logic simulation application in the SPLASH benchmark suite [16]. In `pthor`, the `ElementArray`, an array of logic elements, is responsible for more of the program's cache misses than any other data object. However, these misses are distributed across several procedures. Code oriented output cannot emphasize `ElementArray`'s performance problems as well as data oriented output, because no single section of code is the bottleneck. In this case, the bottleneck lends itself to data oriented viewing.

The blocked matrix multiplication code ($Z = X \times Y$) shown in Figure 1 gives another example of the power of combining data oriented statistics with code oriented statistics. Blocked algorithms such as this operate on submatrices or blocks of the original matrix, so that data fetched into the cache are reused before replacement. The bulk of the computation is performed in line 13. In this line, the appropriate elements of X and Y are multiplied, and the result is accumulated in an element of Z. Code oriented statistics are useful for focusing the programmer's attention on this section of the code. However, with code oriented statistics alone, it would be difficult to determine which of the matrices in the loop is causing the bottleneck. With data oriented statistics, one learns that the bottleneck on this line is almost entirely due to misses generated by the Y matrix.

```
 1) BlkMultiply(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 1: Blocked matrix multiply code.

For both code and data oriented statistics, it is important to provide the user with a focusing mechanism; that is, a metric or display that initially helps the user locate bottlenecks in the code. In MemSpy, the primary focusing mechanism is the *percentage of total memory stall time* associated with each monitored object. That is, code and data objects are ranked according to the fraction of stall time they are responsible for. We have found this more useful than other metrics such as cache miss rates for identifying problem sections in the program. This is because code or data segments with high miss rates, but low total stall time (because there are few references) do not impact performance as much as segments with lower miss rates but more total stall time.

Figure 2 shows an example of the initial display produced by MemSpy for blocked matrix multiply. In this run, we multiply two 295 x 295 element matrices together; we use a block size of 64 so that a single block just fits into the simulated 32Kbyte cache. In the output table, data objects are ranked across the horizontal axis, and code objects are ranked vertically. This output matrix presents a concise breakdown of how code and data objects contributed to the program's total memory stall time. The legend at the top gives the names of the data and code objects. Here, we can see that 85% of the program's memory stall time occurs in the Y variable in `BlkMultiply()`, the routine shown in Figure 1. It is surprising that Y is responsible for so much stall time (and so many cache misses), since the Y block is sized to fit in the cache. As we proceed, MemSpy's output will provide more information about the precise cause of this problem.

```
TOTAL APPLICATION STATISTICS

Execution Time: 649.0M cycles
Total Memory Stall Time: 509.5M cycles

Overall Miss Rate: 37.8%

Total References: 26.93M
-Reads: 19.99M (74.2%) -Writes: 6.94M (25.8%)
Total Misses: 10.19M
-Reads: 9.92M (97.4%)  -Writes: .26M (2.6%)
------------------------------------------
Percentage of Total Memory Stall Time,
 broken down by data and code

Data Bins:                Code Segments:
    0 : Matrix.Y           0 : BlkMultiply
    1 : Matrix.Z           1 : main
    2 : Matrix.X           2 : ClearProduct

                        Data Bins
Code      |        |     0        1        2
Segments  |  Tot%  |   86.4     8.5      5.1
------------------------------------------
  0       |  97.5  |   85.5     7.7      4.3
  1       |   1.7  |    0.9     --       0.8
  2       |   0.8  |     --     0.8       --
```

Figure 2: MemSpy initial output: Blocked matrix multiply.

For each data-bin–code-segment combination, users can request detailed statistics about the behavior of references in that "bin". The display gives information such as: miss rate, read and write statistics, statistics on local versus remote misses, and memory latency statistics. Finally, a key feature of this output is the breakdown of the types of cache misses. Cache misses occur in the following situations: (i) if the line has never been referenced before by this processor, (ii) if the line has been replaced out of the cache since its last reference, or (iii) if the line has been invalidated since its last reference. MemSpy provides statistics which break down the misses occurring due to each of these situations.

Figure 3 is an example of a detailed display from the blocked matrix multiply example.[1] In this case, we are examining the detailed statistics for the Y matrix in the `BlkMultiply` routine. We can see that this particular data-code combination has a miss rate of 68%. This is surprisingly high, because blocking is used specifically to reduce cache misses to Y. The output shows that the misses to Y in this routine are *all* due to cache replacements.[2] MemSpy gives a breakdown of which data objects caused replacements, and we see that roughly 90% of all replacements were caused by Y itself. To summarize, (i) Y has a surprisingly high miss rate, (ii) the misses are primarily due to replacements, and (iii) the replacements are mainly caused by other references to the Y data object. These three facts alert the programmer to the problem of *self-interference*.[3] The programmer can now 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 [11].

```
DETAILED OUTPUT: Matrix.Y in BlkMultiply

Elapsed Time in BlkMultiply: 627.6M cycles
Memory Stall Time in bin: 435.5M cycles

Percentage of Total Memory Stall Time: 85.5%
Percentage of Total Misses: 85.5%
Miss Rate: 67.6%
Percentage of Total Refs: 47.8%

REFERENCES: 12.88M --
  Reads: 12.88M (100.0%) Writes: 0 (0.0%)
MISSES: 8.71M --
  Read misses: 8.71M (100.0%)
  Write misses: 0 (0.0%)

1st Ref Miss: 0 (0.0%)
Inval Miss: 0 (0.0%)
Repl Miss: 8.71M (100.0%)

Memory Stall Time (Cycles):
Total = 435.5M, Avg per reference= 33.8
Memory Read Stall Time = 435.5M,
Memory Write Stall Time = 0
Avg Memory Read Stall Time = 33.8

Causes of Replacements:
  bin Matrix.Y: 90.0%
  bin Matrix.Z:  6.9%
  bin Matrix.X:  3.1%
```

Figure 3: MemSpy detailed output: Y in `BlkMultiply`.

[1] In this figure, the *Percentage of Total Memory Stall Time* equals the *Percentage of Total Misses* because in this case we use a simple memory simulator, with a constant latency for all cache misses.

[2] There are no first reference misses in this routine, because all the matrix elements are first referenced in a separate initialization procedure.

[3] To discover the effect shown here, Lam et al. [11] manually instrumented the code to gather data similar to MemSpy's. MemSpy automates and generalizes this process, making these statistics more accessible to programmers.

This example has illustrated the usefulness of MemSpy's output. Section 4 also shows a more detailed case study using MemSpy. However, MemSpy's detailed statistics have a cost. Information at MemSpy's level of detail is available in two ways: simulation or hardware tracing. The prototype version of MemSpy uses simulation to gather the necessary information. We are optimizing this simulator-based version and examining the basic performance limitations of this approach. We also intend to create a version of MemSpy that uses the hardware tracing facilities of the DASH multiprocessor [12] to gather this information. To further reduce the cost of gathering MemSpy's detailed information, we view MemSpy as part of a hierarchy of performance debugging tools. High level tools provide coarse-level information to focus the user on these selected performance bottlenecks; then, MemSpy can be used to monitor particular data objects or sections of code. In this way, one pays for MemSpy's detail only when it is useful.

There are several other important issues that arise as a result of MemSpy's detailed, data oriented output. How does MemSpy decide which data object a particular reference belongs to? Should it keep separate statistics for each individually allocated range of memory? Or for each class of data objects? Furthermore, how can we automatically assign the bins names that best correspond to variable names the user recognizes? For example, memory may be allocated and assigned temporarily to pointer name `tmp`, before being assigned to a more intuitively named variable. We also need to optimize the speed of statistics gathering in general. These issues will be examined more closely in Section 5.

4 A Performance Tuning Case Study

In this section, we present a step by step description of how MemSpy may be used to tune a parallel application. As we proceed through the case study, it is important to note how simple statistics become much more powerful when presented for individual data structures as well as for sections of code. The application we have chosen is `tri`. It is a parallel program that implements the *triangular system solve* phase of the incomplete Cholesky conjugate gradient (ICCG) algorithm. (The ICCG algorithm is a widely used iterative method for solving large sparse systems of equations that arise in engineering applications.) The work shown here is an illustration of work previously described in [14]. In the original study, the authors had gathered these statistics using a version of the ICCG code with very low level instrumentation added by hand. MemSpy automates and generalizes this process.

The statistics shown here were collected using MemSpy with a simulated bus-based multiprocessor consisting of 4 processors, each with 64Kbytes of cache. The cache line size is assumed to be 64 bytes and the miss latency is assumed to be 50 clock cycles. This roughly models the architecture of the Silicon Graphics 4D/340 multiprocessor on which the original `tri` study was done.

```
for i = 1 to N {
  x[i] = b[i];
  for j = 1 to i-1 {
    x[i] = x[i] - M[i][j] * x[j];
    }
  }
```

Figure 4: Serial pseudo-code for `tri`.

4.1 Example Application: `Tri`

The basic problem solved by `tri` is: $Mx = b$, where M is a sparse, lower triangular matrix with unit diagonal, and x and b are vectors. M and b are known; x is to be computed. The pseudo-code in Figure 4 gives a straightforward, serial solution to this problem. Since M is lower triangular, j is always less than i in the summation and the sum involves only $x[j]$ that have already been computed.

The actual parallel solution we study differs from Figure 4 in several ways. First, the sparse matrix M is stored in the following compact format. The non-zero elements of M are stored contiguously by row in the one dimensional array $M.nz$. Another array, col, stores the column number of each non-zero in $M.nz$. A third array stores pointers to the beginning of each row in $M.nz$. To parallelize `tri`, the algorithm attempts to compute values for several $x[i]$ concurrently. Of course, not all iterations can be performed at once, because computing $x[i]$ in row i may require $x[j]$ from a previous row j. To exploit parallelism, the dependencies between rows (various $x[i]$) can be determined in advance, and an acyclic dependency graph built. By doing a topological sort on this graph, we can assign each row ($x[i]$) to a discrete level of computation so that it depends only on rows in lower levels (i.e., those $x[i]$ that have been computed earlier). In the version of `tri` we begin with here, processors are assigned the rows from each level in a round-robin fashion. Figure 5 shows the pseudo-code executed by each processor.

```
 1) For each "row" assigned to me {
 2)   /* initialize accumulator variable*/
 3)   accum = b[row];
 4)   For each non-zero entry,j,in this row{
 5)     /* wait until x[j] is ready */
 6)     while (!Ready[col[j]]) ;
 7)     /* update accum using M.nz and x */
 8)     accum = accum - M.nz[j] * x[col[j]];
 9)     }
10)   /* set x[row] to its final value */
11)   x[row] = accum;
12)   /* x[row] is now usable by others */
13)   Ready[row] = 1;
14)   }
```

Figure 5: Parallel `tri` implementation.

4.2 Performance of Original `Tri` Code

When we run the original `tri` code using the benchmark matrix BCSSTK15 [6], we find that the speedup with 4 processors is very low, only a factor of 1.4. To explore the cause, we use MemSpy to first look at the total number of misses. These numbers are shown in a summarized form on line 2 of Table 1. This figure also shows the breakdown of cache misses among program data objects.

The first thing that stands out is that the total number of cache misses rises sharply, by a factor of 3.3, as we go from the sequential to the multiprocessor version of the code. Though the total time spent doing real work has remained roughly the same, the time spent stalled for memory has more than tripled. Furthermore, in the parallel version, when one process is stalled waiting for memory, others may be forced to spin-wait until that process gets the needed memory item and produces the elements the other processes wait for. Thus, memory behavior is likely to be the prime reason for the poor speedups. To see how the misses may be reduced, we now look at the composition of misses in the two cases. We see that the number of misses has increased for all data objects;[4] however since $M.nz$ causes the most misses in the multiprocessor version, we focus first on its behavior.

We first note that the non-zero elements of matrix M are accessed only once in both the sequential and parallel version of the code; thus, ideally the total number of misses for the matrix M should not increase as we go from the sequential to the parallel code. Yet the data show that the number of misses increases by over 50%. When we request more detailed information about the bin $M.nz$ from MemSpy, it shows us the data in Figure 6. It indicates that most of the misses (over 90%) are first reference (cold) misses and not invalidation or replacement misses.

```
Percentage of Total Memory Stall Time: 42.2%
Percentage of Total Misses: 42.2%
Percentage of Total Refs: 55.9%
Miss Rate: 9.4%

1st Ref Miss: 16482 (91.4%)
Inval Miss: 0 (0.00%)
Repl Miss: 1556 (8.6%)

Causes of Replacements:
   bin double*.M.nz:  55.5%
   bin x:             21.7%
   bin int*.Ready:    12.3%
   bin b               9.9%
```

Figure 6: MemSpy detailed output: bin $M.nz$ in `tri`.

Once MemSpy points out that most of the misses are first reference misses, it is not so hard for the application programmer to figure out that the real cause for increased misses is poor spatial locality for $M.nz$. In particular, the number of non-zeroes per row of M is very small in input

[4] Since the $Ready$ data vector is not needed for the uniprocessor version, it obviously causes no misses there.

Table 1: Summary of MemSpy output after various tuning steps.

Version	Cache Misses (x 1000)					Execution Time (x 1000 cycles)	Speedup
	Total	$M.nz$	$Ready$	x	$other$		
Sequential	12.9	11.2 (86.7%)	—	1.2 (9.3%)	0.5 (3.9%)	2,580	1.0
Original Parallel	41.6	17.8 (42.2%)	11.9 (27.9%)	10.5 (24.6%)	2.3 (5.4%)	1860	1.4
Tuning Step 1	39.2	11.3 (28.8%)	13.2 (33.7%)	14.2 (36.2%)	0.5 (1.3%)	1742	1.5
Tuning Step 2	18.1	11.2 (61.9%)	—	6.4 (35.4%)	0.5 (2.8%)	967	2.6
Tuning Step 3	16.0	11.2 (70.0%)	—	4.3 (26.9%)	0.5 (3.1%)	890	2.9

matrices for the tri computation.[5] Since cache lines are 8 double words long (64 bytes), each cache line contains multiple rows. In the parallel code, successive rows are frequently assigned to different processors, and as a result, when a processor fetches the contents of a row it needs, it also fetches useless data (adjacent rows relevant only to other processors). This does not occur in the uniprocessor code where adjacent rows are accessed consecutively by the same processor.

We emphasize that MemSpy has facilitated this observation about spatial locality by allowing us to isolate the miss statistics for $M.nz$, and letting us compare the uniprocessor and multiprocessor values. Without such detailed data oriented statistics, the lack of spatial locality would be difficult to infer.

4.3 Step 1: Restoring Spatial Locality

The goal of this tuning step is to improve the spatial locality of references to $M.nz$ in `tri`. This is accomplished by symmetrically reordering the rows and columns of the matrix $M.nz$, so that the row indices of rows assigned to a particular processor are contiguous and appear in the order in which the rows are processed. The details of the reordering method are discussed in [14].

When the program is rerun, using the new ordering scheme for spatial locality, MemSpy produces the new miss composition data summarized on line 3 of Table 1. This output indicates that now only 29% of the misses are due to the $M.nz$, with 34% of the misses in the $Ready$ vector, and 36% of the misses in the x vector. Misses in $M.nz$ have been reduced from 17.8K to 11.3K, and are now only 1% greater than misses in $M.nz$ in the sequential version. The reordering for spatial locality has been effective in reducing the $M.nz$ misses to almost the intrinsic number required by the application.

While the misses in $M.nz$ have been reduced significantly, this change leads only to a very minimal improvement in overall performance, about 6%. MemSpy again tells us (as seen in Table 1), that this is because the decrease in misses for $M.nz$ is partly offset by an increase in misses for the $Ready$ and x vectors. Figure 7 shows the detailed output for the $Ready$ vector after step 1. Here, 81% of $Ready$'s misses are due to replacements, and 87% of these replacements are caused by references to the x vector. The introduction of the new ordering scheme, which renumbers the rows in the x and $Ready$ vectors, has resulted in a pathological memory mapping; cross-interference between the x and $Ready$ vectors in the cache causes the misses in these data objects to increase dramatically.[6]

```
Percentage of Total Memory Stall Time: 33.7%
Percentage of Total Misses: 33.7%
Percentage of Total Refs: 26.2%
Miss Rate: 13.7%

1st Ref Miss: 988 (7.5%)
Inval Miss: 1502 (11.4%)
Repl Miss: 10749 (81.1%)

Causes of Replacements:
   bin x: 86.7%
   bin double*.M.nz: 13.3%
```

Figure 7: MemSpy detailed output: $Ready$ in step 1.

We again note that without a tool like MemSpy, it would be difficult to understand the effects of this tuning step. In fact, one might have jumped to the wrong conclusion that reordering $M.nz$ was not effective in improving spatial locality; in reality, MemSpy shows that the reordering was effective, but that the potential improvement was offset by interference in the x and $Ready$ vectors. The following two subsections will discuss further steps taken to reduce the $Ready$ and x misses.

4.4 Step 2: Reducing $Ready$ Traffic

Following the reduction in $M.nz$ traffic, two other data objects, x and $Ready$ have become the leading contributors to the cache misses. Although x generates more misses than $Ready$, we first show the effect of reducing the misses due to $Ready$ because it is more readily apparent.

The $Ready$ vector indicates when a particular x element has been computed and is ready for use by later computations. After step 1, the MemSpy output shows (see Figure 7) that the $Ready$ misses constitute roughly one third of all misses. Of these, a majority are due to cross-interference between x and $Ready$ (indicated by replacements), a small fraction (7.5%) are partly intrinsic and partly due to lack

[5] For example, if M comes from a partial differential equation corresponding to a 5-point stencil, each row has two off-diagonal non-zeroes.

[6] This cross-interference is data dependent, and does not occur as severely in other matrices we have studied.

of spatial locality (indicated by first reference misses), and another small fraction (11.5%) are due to sharing or invalidations.

To reduce misses in $Ready$, one might first consider ways of reducing cross-interference and sharing. However, Rothberg and Gupta, in fact, devised a new form of self-scheduling that allows $Ready$ to be eliminated entirely. This method takes advantage of the NaN (Not a Number) value provided for by the IEEE 754 Standard for Binary Floating Point Arithmetic. The NaN value is stored into each element of the x vector before the `tri` phase begins. Then, instead of using the $Ready$ vector to indicate an x element has been computed, processes waiting for x elements can simply spin on the x value itself. When the value changes from NaN to a valid floating point value, it is ready for use.

This change substantially improves program performance due to two effects on the memory system behavior of the program. As shown in Table 1, $Ready$ misses are eliminated entirely; furthermore, misses due to the x vector are also substantially reduced due to a decrease in the cross-interference described above. The next subsection focuses on improving the performance of x.

4.5 Step 3: Reducing Traffic due to x

Cache misses for x primarily occur when an x element produced by one processor is subsequently used by another processor. Thus, the goal of this step is to devise strategies for assigning x elements to processors such that each element primarily depends on other x elements assigned to the same processor. This reduces the need for interprocessor communication of these values, and reduces the x traffic. Rothberg and Gupta investigate several heuristics for accomplishing this, and MemSpy is helpful in comparing the effects of these different heuristics.

For brevity, we present results for only the final heuristic proposed by Rothberg and Gupta. In it, each $x[i]$ is assigned to the processor that currently owns the most previous elements required to compute that $x[i]$. MemSpy shows (see line 5 of Table 1) that misses due to the x vector decrease from 6.4K to 4.3K—around 41% of these misses are first reference misses, 12% are due to invalidations, and 47% are due to replacements. MemSpy further indicates that almost all (99%) of the replacements are due to the $M.nz$ matrix. Since `tri` streams through the data in the very large M matrix, these replacements are essentially unavoidable.

4.6 Summary

This case study has highlighted how MemSpy may be used to tune an application's memory behavior. In the first tuning step, MemSpy was used to calculate miss counts for the $M.nz$ data. These played a key role in pointing out that poor spatial locality was the cause of the increase in misses. Based on this information, we reordered the matrix to improve spatial locality. MemSpy's information on the causes of misses was also instrumental in helping us understand the cross-interference that resulted from reordering. Without MemSpy, it would have been difficult to separate the two effects. In Step 2, we eliminated $Ready$ misses. MemSpy's data oriented output was key in indicating that $Ready$ was responsible for a large amount of traffic. In the final tuning step, a heuristic for improving x access patterns was examined. Here again, MemSpy's miss counts were useful in showing the improvement in x behavior. Furthermore, MemSpy's data indicating which data object caused replacements was also useful. By knowing that most of x's replacements were caused by $M.nz$, we were able to reason that they are largely unavoidable.

5 MemSpy Implementation

As we have shown, MemSpy presents detailed statistics on low-level memory system events. Gathering data at this level requires support from either a software memory system simulator or a hardware tracing system. This section discusses the implementation details of the prototype version of MemSpy, which uses the former, software-based approach. MemSpy is implemented as part of a memory simulator using the Tango [4] system to instrument the code for memory monitoring. In this section, we first give some necessary background information on Tango memory simulations. Following that, we discuss issues in generating data and procedure oriented statistics, labeling the data oriented statistics with intuitive names from the user program, and designing the user interface. Finally, we present data on MemSpy's performance.

5.1 Tango Memory System Simulation

Tango is a software simulation and tracing system, used by MemSpy in monitoring the memory system behavior of programs. Its tracing and memory simulation facilities are useful in both the sequential and parallel domains.[7]

When using Tango, the application to be studied is first instrumented by a special preprocessor. At each memory reference, the instrumentation adds procedure calls to a memory simulator. The memory simulator procedure then calls MemSpy procedures to maintain statistics on simulator events such as cache hits, cache misses, etc. The simulator maintains the state of each processor's cache, while the additional MemSpy code tracks the causes and frequency of misses. The modular interface between MemSpy and the memory simulator allows MemSpy to be implemented easily with a variety of memory simulators. Because this method uses no intermediate trace files, one can run detailed simulations of large benchmarks without the disk space limitations imposed by trace-file based approaches.

5.2 Grouping Statistics into Bins

MemSpy presents data and code oriented statistics. To do this, both the code "axis" and the data "axis" of the application are subdivided into logical units; we call these units *code segments* and *data bins*. Statistics are then maintained

[7] Tango simulates multiprocessors by multiplexing the execution of several application processes on a single real processor.

for each pairing of code segment and data bin; each such pairing is referred to as a *statistical bin*. The following subsections describe the methods of determining appropriate code and data divisions.

5.2.1 Separation of Statistics by Code Objects

Along the code axis, MemSpy separates statistics by procedures. It is straightforward to determine which procedure the process is currently in, because Tango supports event logging on procedure entry and exit. These entry and exit events are passed to the memory system simulator, and using them, MemSpy maintains a procedure stack for each process. In this way, the current procedure is always known, and can be used to select the appropriate procedure bin in which to place statistics.

5.2.2 Separation of Statistics by Data Objects

Along the data axis, MemSpy separates statistics by *data bins*. Some data bins correspond to a single data object in the application source code. In other cases, it is appropriate to group together statistics from several data objects into a single data bin. Thus, a data bin may contain statistics from several non-contiguous ranges of memory. The following paragraphs discuss (i) how the memory space is divided into data bins and (ii) how these data bins are given names which are intuitive and useful to the programmer using MemSpy.

Data Division As a first approach to this data binning problem, the program's entire memory space could be divided into memory ranges, where each memory range corresponded to a single data object in the program, and statistics are kept for each individual memory range. However, considering each individual data object to be a separate statistical unit would likely result in cases where there are many bins with very similar behavior. For example, in LocusRoute, a CAD wire routing program from the SPLASH benchmarks, the program allocates storage for thousands of wires. Since all the wires have similar memory behavior, keeping separate statistics on each wire is not as useful as aggregating statistics for all wires. To automatically aggregate statistics for all wires, we might use an approach which groups into a single data bin *all memory ranges allocated at the same point in the source code*. However, the opposite extreme, combining too many data objects together in a single data bin, must also be avoided. For example, in a benchmark program which performs LU decomposition, the program's main data structures are two matrices which are allocated at exactly the same point in the source code, within a `NewMatrix` routine. Here, the programmer would like to view separate statistics for each matrix, since their memory behavior is quite different. Because of cases like this, MemSpy maintains separate statistics for *all memory ranges allocated at the same point in the source code with identical call paths*. That is, data allocated in different calls to a procedure from different call paths will be monitored in separate bins.[8] We claim that data objects allocated at the same point in the source code via the same call path are usually similar in memory behavior, and their statistics, in general, should be aggregated.

To implement this proposed method of data division, MemSpy needs to be able to map every possible memory address to its corresponding data bin. To maintain mappings between ranges of memory and the data to which they correspond, one needs to know the size and starting positions of all memory allocated by the application. In general, programs use three types of memory allocation: (i) static, (ii) stack, and (iii) dynamic. In this version of MemSpy, we automatically maintain mappings only for dynamically allocated data. This fits in well with the parallel programming model we currently use, in which all shared memory must be heap allocated.[9] When users want to monitor a variable which is not heap allocated, they can manually add a procedure call into the application to define that mapping. For MemSpy to maintain mappings for static and stack allocated data, it would require data type information, in order to know the sizes of the individual data objects. A later version of MemSpy will provide the compile time instrumentation support necessary to produce mappings for statically and stack allocated variables.

For mappings of dynamically allocated data objects, MemSpy maintains a log of all heap allocated memory, and records which memory ranges belong to which program variables. Logging memory allocations from the heap is fairly straightforward; we simply instrument the code to log (i) the pointer returned by the malloc routine, (ii) the size of the allocated block of memory, (iii) the name of the variable to which the malloc return value is assigned. (Naming will be discussed in more detail later.) This instrumentation generates events which become part of the input event trace for the MemSpy memory simulator. MemSpy then builds up a data structure to store these memory ranges.

We have found this method for data division to be quite effective in practice. However, there will still be cases in which the user would like some manual control over the division of data. We are interested in extending the current scheme to allow the user to give suggestions or directives on how the statistical bins should be composed, as well as to provide automatic support for static and stack allocated memory objects.

Data Bin Naming In assigning names to data bins, we want to use symbolic variable names from the source program since these have some intuitive meaning to the programmer. Furthermore, clearly, the names should be unique. To satisfy the first requirement, intuitiveness, consider each static appearance of a malloc in the code: we name the associated bin with a string that concatenates the *data type* and *variable name* of the pointer which receives the malloc return value. However, as stated above, multiple data bins are created for the same malloc if the malloc is encountered through different procedure call paths. Thus, to guarantee uniqueness, the names are disambiguated by prepending a string summarizing the state of the call stack.

[8] The exact method used for tracking the call path is similar to that used by Zorn and Hilfinger in their memory allocation profiler, *mprof* [18].

[9] Our parallel programming model uses C language programs augmented with Argonne National Laboratory parallel programming macros [13]. In this model, all shared memory is dynamically allocated using the G_MALLOC macro.

The final full name is of the form:

```
"ProcName.return_pc.ProcName.return_pc...
.DataType.VarName"
```

This method has both strengths and weaknesses. By prepending the bin name with call stack information, we guarantee a unique name for each bin. However, in our experience with MemSpy, we have found that a short version of the name: `DataType.VarName` is usually unique and sufficiently intuitive for the programmer. It works especially well when important program variables are directly assigned the pointer returned by malloc, so that the variable name in the short form is a familiar program name. However, sometimes the allocated memory is assigned to a temporary variable and then later assigned to a more "significant" variable in the program. In these cases, the data bin will receive the name of the temporary variable, rather than the preferred name. Another weakness of this method appears in cases where the long form is necessary to distinguish between data bins; the name it produces, with program counter values interspersed, is often inconvenient or difficult to read. Both of these weaknesses are hidden from the user by allowing the user to rename variables to a new unique name of their choosing.

5.3 Storing Information on Causes of Misses

Statistics on the causes of application misses are an important part of MemSpy; to provide this data, MemSpy needs to store information to explain the cause of each miss. Cache misses are caused by one of the following: (i) the line has never been referenced before by this processor, (ii) the line has been replaced out of the cache since its last reference, or (iii) the line has been invalidated since its last reference. To distinguish between these three cases, 2 bits of state information are required for each memory line in use by each processor.

To store this state information, MemSpy defines a one dimensional array that is indexed by the lower bits of the referenced address. The array contains the state bits indicating the cause of the miss. It also contains the remaining upper portion of the address, to act as an identifier. The array size can be varied depending on the size of the application's data set. If the array is defined to be smaller than the data set of the application, then several referenced addresses might index into the same location of the array; we define a hash table to handle these overflow cases. The overflow state information is hashed based on the referenced address and stored in linked lists. Clearly, there is a tradeoff here: A smaller primary array will have less space overhead, but with poor performance for applications with large data sets that overflow into the hash table. A larger array will handle a larger data space more efficiently, but with higher space overhead. One could improve the performance of this system by taking advantage of temporal locality in the reference patterns. If an object from the overflow table has just been referenced, it is likely to be referenced again soon; performance may be improved by moving its state information out of the overflow table and into the primary array.

5.4 User Interface

The user interface of a performance monitor must guide the user towards bottlenecks in the code, and then give the information necessary to remedy them. This subsection gives an overview of MemSpy's user interface. The current user interface has been intentionally kept quite simple.

A MemSpy session begins by presenting initial data using the focusing mechanism *Percentage of Total Memory Stall Time* as the primary means of sorting the output. That is, for each code object and data bin pair, MemSpy computes the ratio of the memory stall time incurred in this statistical bin, compared to the total memory stall time in the program. When MemSpy output is first displayed, this information is presented as an ordered matrix in which one axis shows the different data bins, and the other axis shows the different procedures. Each row and each column of the matrix are sorted, so that the upper left corner of the matrix contains the procedure-data pair with the highest percentage of total memory stall time, and the numbers decrease as one moves down and to the right. A sample output was shown in Figure 2. The initial display also summarizes information on the program's execution time, and aggregate cache memory statistics. From this starting point, users have several options available to them. These options include displaying more information about a bin, renaming data bins, or combining bins and displaying the total information.

The most basic operation a user can perform after starting up MemSpy is to request a display for a particular statistical bin using the `DisplayAll` command. This display, shown for example in Figure 3, gives detailed information about the statistical bin. This data allows the user to reason about the types of memory system problems in the application. For example, if a particular data object has a high miss rate, the misses are primarily due to replacements, and the replacements are primarily caused by other references to the same data object, one concludes that self-interference is a problem.

The `DisplayAll` command may also be used on combinations of multiple data and/or code divisions. That is, one may request the statistics of a particular data object in several procedures, or several data objects within a procedure, and so on. By building the basic information given by MemSpy into other useful combinations, the user can adapt the output to the specific high-level structure of the code.

Other commands allow the user to manipulate the names of the data bins to allow for easier debugging. A `fullname` command allows the user to see a data bin's full name, including the stack trace. Note that, to save space, the main display gives only the partial names of the data bins in the form `data_type.variable_name`. With `fullname`, the user can distiguish between data bins whose partial names are identical. The `rename` command allows the user to change the label of a data bin to a more appropriate name. (The most effective method for assigning intuitive names to data bins is still an open question. Until we arrive at a more satisfactory conclusion, we find this intermediate approach, giving the bin a unique name that the user is then free to change, quite useful.)

In the future, we will extend the user interface to give the

user greater control over monitoring. For example, the user can currently request that only a subset of code segments be monitored; we would like to extend this to give the user control over which *data* objects are monitored as well. The user should also be allowed to direct the automatic division of data into bins, in cases where a non-default binning is needed. We will also provide the user with a database of statistics from previous runs. This will allow the user to easily compare results from a current version of an application with previous results. Finally, we are currently implementing a graphical user interface, to make MemSpy more convenient to use.

5.5 Performance

This section presents preliminary performance results for the MemSpy system. While the prototype system is largely unoptimized, the current execution time overheads seem reasonable. We also briefly outline methods for improving MemSpy's performance, a major thrust of future research.

Table 2 compares the execution times on a DECstation 3100 for three benchmark applications. Execution times are presented for three cases: (i) Actual uniprocessor benchmark runs, with neither simulation nor monitoring. (ii) Tango simulations of the benchmarks without MemSpy monitoring, and (iii) Tango simulations of uniprocessor benchmark runs with MemSpy monitoring as well. The table shows that MemSpy's overhead, when compared to a uniprocessor run with no monitoring, ranges from a factor of 22 to a factor of 58 for these benchmarks.

In order to understand what contributes to this overhead, let us examine the sequence of operations needed to log an event with MemSpy. For each memory reference, the original assembly code for the application is instrumented with a procedure call to the Tango system. Within the procedure, temporary registers (i.e., those whose values are not preserved across procedure calls) are first saved, so that registers used by the memory simulator will not overwrite the values expected in them by the application. Next, the Tango memory simulator procedure is called. Within the memory simulator, different MemSpy routines are called to update the data required for MemSpy's statistics, such as whether the reference is a hit or a miss, a read or write, and so on. In Table 2, simulation overhead refers to time spent in the memory simulator procedure; MemSpy overhead refers to time spent in the special MemSpy routines only.

From Table 2, we see that the Tango simulation overhead dominates the additional MemSpy overhead in monitoring an application. For the simple simulator used here, more than half of this overhead is in saving and restoring all temporary registers before calling the memory simulator. One can reduce this overhead by customizing the register save routine so that it only saves the temporary registers actually used by the MemSpy memory simulator. For example, 10 double precision saves and restores of floating point registers could be eliminated from the current version. This would result in a roughly 50% reduction in register save-restore time for each memory reference. Furthermore, note that many of the integer registers are used only when simulating a cache miss, not when simulating a cache hit; by postponing these register saves until after a cache miss is actually detected, we can significantly reduce the overhead of invoking the memory simulator on cache hits, the more common case.

Overhead in MemSpy itself ranges from 30 to 44% of the total overhead in these benchmarks. This MemSpy overhead is comprised of (i) time spent determining the bin to which a reference's statistics belong, and (ii) time spent updating statistics, such as counting hits, misses and information on the causes of misses. The first factor, searching for the appropriate statistical bin, is the prime contributor to MemSpy's overhead; it accounts for roughly 30% of a program's total execution time. The search for a bin requires the traversal of a tree data structure containing the mappings from dynamically allocated address ranges to bins. At the root of the tree is an array of n pointers; the array is indexed by the upper $\log_2(n)$ bits of the search address, and each pointer corresponds to a different portion of the address space. In turn, each of these pointers may point to another array whose elements correspond to sub-portions of that memory region, and so on. Where a portion of memory contains only a single address range, the bin information is stored, and no further arrays are required. In `pthor`, with roughly 50,000 different heap allocated memory ranges, bin searches require an average of 3.7 pointer indirections through the tree.

One could further reduce the MemSpy overhead by allowing the user the option of keeping statistics only for cache misses, not for cache hits. In the current version of MemSpy, all references require an address-to-bin translation. By not monitoring hits, we could do bin lookup only for misses. This would lead to a significant performance improvement since bin lookup comprises roughly one third of the application overhead. Without statistics for cache hits, MemSpy could not produce data on cache miss *rates* or total reference counts. However, one could still view counts of misses, breakdowns of total misses, and data on causes of misses, some of MemSpy's primary features.

We feel that with these optimizations, MemSpy can be made 5 to 10 times faster for uniprocessor simulations. This overhead is likely to be quite acceptable to many users given the detailed information MemSpy is providing the user.

Running MemSpy to simulate multiprocessor, rather than uniprocessor, executions has two additional sources of overhead. These are related to the fact that Tango interleaves the execution of the multiple application processes on a uniprocessor. First, the Tango execution time for a multiprocessor run can be no smaller than the total execution times for each thread being run. This is because the threads are run sequentially (although interleaved). Second, additional overhead is incurred when context switching between threads: all non-temporary registers must be saved on a context switch. These factors lead to higher execution time overheads for multiprocessor runs of MemSpy. For example, running MemSpy on a 4 process matrix multiplication, with the same input data as the uniprocessor run shown in Table 2, has an overhead of 120.4 as compared to the uniprocess overhead of 21.7. One can reduce this overhead somewhat by optimizing context switching in the simulation. If we make the assumption that context switches are only necessary on cache misses, not on all references as currently assumed, we can greatly reduce the number

Table 2: MemSpy execution time overhead.

Application	Time (s) No Simulation No MemSpy	Time (s) Simulation, No MemSpy	Time (s) Simulation, and MemSpy	Simulation Overhead alone	MemSpy and Simulation Overhead
Tri	4.5	72.0	101.0	16.0	22.4
MatMult	54.3	659.0	1179.3	12.1	21.7
Pthor	9.0	313.0	521.4	34.8	57.9

of context switches attempted by the application, with little effect on the simulation results. Finally, future versions of MemSpy may use the hardware trace facilities available on the DASH multiprocessor to gather memory reference statistics without the overheads inherent to Tango's sequential simulation-based approach.

6 Discussion

MemSpy's statistics have proven useful in understanding the memory system behavior of several applications. First, our initial focusing mechanism, *Percentage of Total Memory Stall Time*, is effective in pointing the user towards problem areas in the code. Second, we have found the breakdown of the *causes of misses* to be quite useful. Knowing whether the memory system problem is one of interference, sharing, or poor spatial locality is a large step toward solving the problem, and MemSpy's statistics on causes of misses give the user much of the information needed to diagnose these problems. However, one level of reasoning that is still left to the user is deciding whether misses are intrinsic to the program, or whether they are "excess" misses that one can hope to optimize away. For example, in the `tri` code, misses in $M.nz$ accounted for 70% of total misses after tuning. By examining the code, the user can conclude that these misses are intrinsic, and cannot be significantly reduced. In some cases, a comparison of multiprocessor misses to uniprocessor misses can act as a guide in determining what fraction of the misses are intrinsic.

We anticipate several extensions to MemSpy's user interface. These include integrating MemSpy into a hierarchy of tools, to provide a complete performance tuning system; thus, a high-level tool like Quartz would provide initial information on code bottlenecks, and subsequent runs with MTOOL and MemSpy would give greater detail on specific memory performance bottlenecks. Within MemSpy itself, we intend to implement a database to store information about previous runs. Such a database would allow the user to easily compare statistics from the current run with statistics from previous runs of the same program.

The current MemSpy prototype is simulator based, which gives it several advantages and disadvantages. Simulation allows an application to be tuned with different sets of architectural parameters, and can be useful in evaluating expected performance of an application on machines not yet available. However, MemSpy's reliance on simulation degrades its performance and somewhat limits its usefulness. Clearly, improvements in simulation performance would make MemSpy a more viable tool for a wider range of applications. We intend to optimize the performance of the simulation-based version of MemSpy. Furthermore, for many applications, one can run them in ways that reduce execution times while still giving realistic memory behavior. For example, with many numerical applications, one can run them for a small number of iterations and then extrapolate their performance to more realistic numbers of iterations; the `tri` code is one such example of this. Another way to observe realistic behavior with less simulation time is to study cases where both problem size and processor cache sizes have been proportionately scaled down.

We are also investigating a MemSpy implementation using the hardware performance monitor on the DASH multiprocessor. DASH's hardware monitor collects traces of bus activity which can then be processed to generate MemSpy statistics. This approach promises a significant performance improvement over the current simulator driven prototype. Furthermore, it allows for a more complete view of program execution, including effects like virtual to physical memory mapping, scheduling, and multiprogramming which are often more difficult (though not impossible) to account for in simulation-based approaches.

7 Conclusions

In summary, we have found MemSpy's statistics to be effective in explaining many of the unknowns of memory system behavior for both parallel and sequential programs. MemSpy's data oriented statistics offer an orthogonal view to code oriented statistics, and give the user greater leverage in tuning memory performance. Statistics on the causes of an application's cache misses are also an important aid in performance debugging that has not been adequately provided previously. We envision using MemSpy as part of a hierarchy of performance debugging tools: higher level tools provide initial insight into program behavior, while MemSpy provides detailed information on memory system behavior to address memory performance bottlenecks.

8 Acknowledgments

We would like to thank Helen Davis, Doug Pan, and Ed Rothberg for their comments on previous versions of this paper. Thanks also go to Ed Rothberg for providing the case study applications. This work was supported in part by the Digital Equipment Corporation Systems Research

Center and DARPA contract N00039-91-C-0138. Anoop Gupta is also supported by a National Science Foundation Presidential Young Investigator Award.

References

[1] A. Agarwal and A. Gupta. Memory Reference Characteristics of Multiprocessor Applications under MACH. In *Proc. ACM SIGMETRICS Conference on the Measurement and Modeling of Computer Systems*, pages 215 – 225, May 1988.

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

[3] Z. Aral and I. Gertner. Non-Intrusive and Interactive Profiling in Parasight. In *Proc. ACM SIGPLAN Parallel Programming: Experience with Applications, Languages and Systems (PPEALS)*, pages 21–30, July 1988.

[4] H. Davis, S. R. Goldschmidt, and J. Hennessy. Tango: A Multiprocessor Simulation and Tracing System. In *Proc. International Conference on Parallel Processing*, pages 99–107, Aug. 1991.

[5] 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.

[6] I. Duff, R. Grimes, and J. Lewis. Sparse Matrix Test Problems. *ACM Transactions on Mathematical Software*, 15:1–14, 1989.

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

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

[9] 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.

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

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

[12] D. Lenoski, J. Laudon, T. Joe, D. Nakahira, L. Stevens, A. Gupta, and J. Hennessy. The DASH Prototype: Implementation and Performance. To appear in *Proc. Nineteenth Annual International Conference on Computer Architecture*, May 1992.

[13] E. Lusk, R. Overbeek, et al. *Portable Programs for Parallel Processors*. Holt, Rinehart and Winston, Inc., 1987.

[14] E. Rothberg and A. Gupta. Parallel ICCG on a Hierarchical Memory Multiprocessor— Addressing the Triangular Solve Bottleneck. Technical Report CSL-TR-90-449, Stanford University Computer Systems Laboratory, Sept. 1989. To appear in *Parallel Computing '92*.

[15] Z. Segall and L. Rudolph. PIE: A Programming and Instrumentation Environment for Parallel Processing. *IEEE Software*, pages 22–37, Nov. 1985.

[16] J. P. Singh, W.-D. Weber, and A. Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. Technical Report CSL-TR-91-469, Stanford University, Apr. 1991.

[17] L. Soule and A. Gupta. An Evaluation of the Chandy-Misra-Bryant Algorithm for Digital Logic Simulation. In *Proc. Sixth Workshop on Parallel and Distributed Simulation*, Jan. 1992.

[18] B. Zorn and P. N. Hilfinger. A Memory Allocation Profiler for C and Lisp. Technical Report UCB/CSD 88/404, University of California, Berkeley, Feb. 1988.