Chapter 3

Case Studies

The previous chapter outlined classes of memory performance bugs and discussed information useful in identifying and fixing them. In this chapter, we present case studies that demonstrate the usefulness of such detailed, data oriented information, as embodied in MemSpy.

The first application, a blocked matrix multiplication program, shows MemSpy's use in isolating a case of self-interference. We show that even in such simple, loop-oriented applications, memory performance bugs can be difficult to discern without data oriented statistics.

The second example, a parallel triangular sparse system solver, illustrates several memory performance problems that are characteristic of shared memory parallel programs. For example, this case study shows instances of true and false sharing, and it also illustrates the fairly common problem of decreased spatial locality in parallel applications.

The applications shown here were previously described in [LRW91] and [RG92], where the authors developed and evaluated optimizations using extensive, manually-added statistics instrumentation. The aim of this chapter is to show how MemSpy automates and generalizes this process, making such information more accessible to all programmers. Additional case studies are presented in Appendix A.

3.1 Sequential Case Study: Matrix Multiply

The first example is a sequential program, `MatMul`, which performs a blocked matrix multiply. MatMul is an interesting example for a number of reasons. First, it represents a case where the code was specifically written using *data blocking* to improve cache memory behavior, but in spite of this (to the programmer's surprise) poor memory performance was discovered. Second, although this application is conceptually simple, it still required detailed, data oriented statistics to tune it. The fact that such statistics are needed on a fairly simple application like this one lends strength to our argument that more complex applications will likely call for them as well.

3.1.1 Problem Description

The blocked matrix multiplication code being studied is shown in Figure 3.1. This code multiplies matrix `X` by matrix `Y` to form matrix `Z`. Unlike standard matrix algorithms, blocked algorithms such as this are coded to operate on sub-matrices or blocks of the original matrix as shown in Figure 3.2. These sub-blocks are sized to fit in the cache, to maximize reuse of the data. By iterating over all sub-blocks, the full matrix multiplication can be performed, ostensibly with better cache performance due to the blocking.

As was reported by Lam et al. [LRW91], the performance of such blocked operations is often erratic. It is extremely sensitive to even small changes in the matrix size, the block size, and the cache organization. For example, on a DECstation 3100, the authors report that a 300 by 300 blocked matrix multiply (with 56 by 56 block size) executes at 4.0 MFLOPS. By contrast, an only slightly smaller 293 by 293 matrix with the same block size executes at only 2.0 MFLOPS on the same machine. Thus, the goal of this case study is to show how MemSpy can be used to point out to programmers what the performance bug is. As we proceed through the case study, it is important to note how data oriented statistics are a powerful method for focusing the user's attention on problem areas in the code. Note also how detailed statistics on the causes of cache misses are crucial for understanding *why* the performance bottleneck is occurring.

```
 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 3.1: Pseudo-code for blocked matrix multiply example.

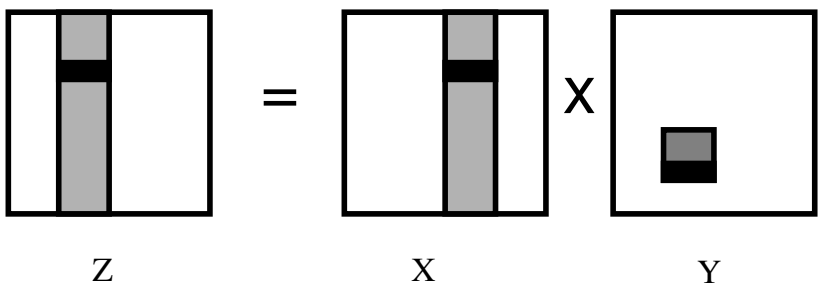

Figure 3.2: Reference patterns in blocked matrix multiply.

3.1.2 Tuning Using MemSpy

To make the performance bug most evident, we will show MemSpy's output on one of the poor performance cases from [LRW91]. We multiply two 293 x 293 element matrices together, using a block size of 56. Thus, a single 56 x 56 block requires 25,088 bytes, and should easily fit into the simulated 64KB cache.

MemSpy begins here by presenting the output shown in Figure 3.3. (The displays shown in case studies throughout this dissertation are screen dumps of actual MemSpy

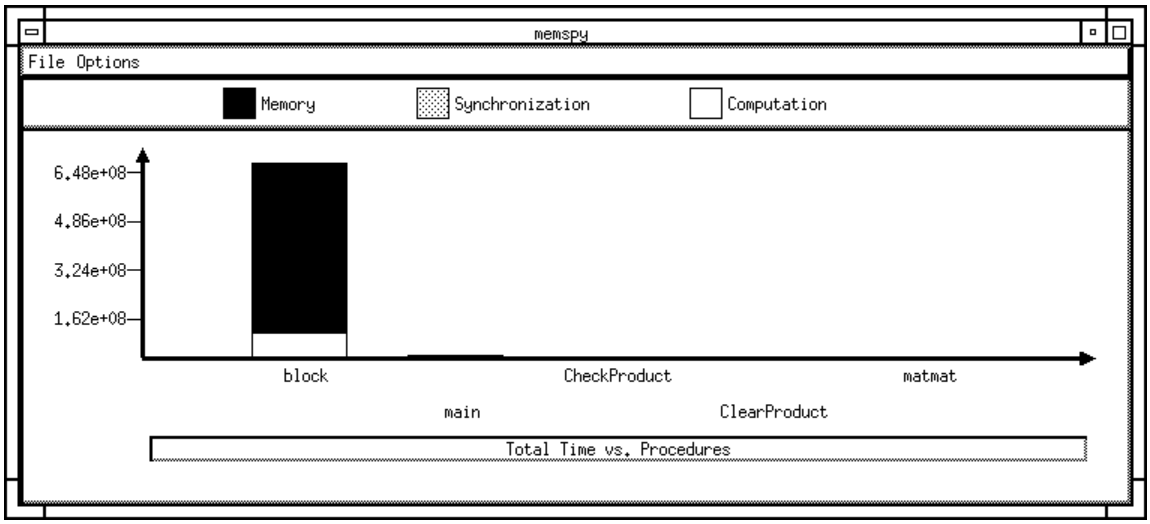

Figure 3.3: MatMul: MemSpy overview statistics display.

output. We use black and white stipple patterns here, but in the actual tool users may choose color displays as well.) The program procedures are indicated along the x axis of this graph, and the time (in processor cycles) spent on behalf of each procedure is given on the y axis. The bar for each procedure breaks down the elapsed time by how much of it was spent in computation and how much in memory stalls. In addition, for parallel programs, the bar indicates synchronization time as well.

Figure 3.3 indicates that the bulk of the application's time is spent in the `block` routine. It accounts for over 90% of the execution time. Furthermore, the breakdown of time within the `block` routine shows a clear memory bottleneck. While we expected the bulk of the computation to be spent in `block`, the observation that roughly 80% of the time is spent on memory stalls is surprising, since we expected the processing on the 25KB block to easily fit in the 64KB cache.

The bulk of the computation in this application is performed in line 13 of the code in Figure 3.1. In this line, the appropriate elements of `X` (`r`) and `Y` are multiplied, and the result is accumulated in an element of `Z`. From this portion of the example, we see that code oriented statistics could be useful for focusing the programmer's attention on this section of the code. However, since all three matrices are accessed on source line 13,

code oriented statistics alone offer no help in determining the relative contributions of the three matrices towards the bottleneck. To further understand and tune this code, users need information on whether a single matrix is a bottleneck, or whether some interaction between the three matrices is causing the stalls.

Data Oriented Breakdown

To understand the stall time contributed by each data structure, users can click on the "memory" portion of the `block` routine's bar to request the next display. This display, shown in Figure 3.4, gives a breakdown of the memory stall time into components incurred by each data structure. With these data oriented statistics, one can learn that the bottleneck in this routine is almost entirely due to cache misses on references to the `Y` matrix. These misses are responsible for over 85% of the total stall time in the program. Note that the `Y` matrix is actually the one that was blocked for better data reuse. Thus, it is surprising that `Y` is responsible for so much stall time (and so many cache misses).

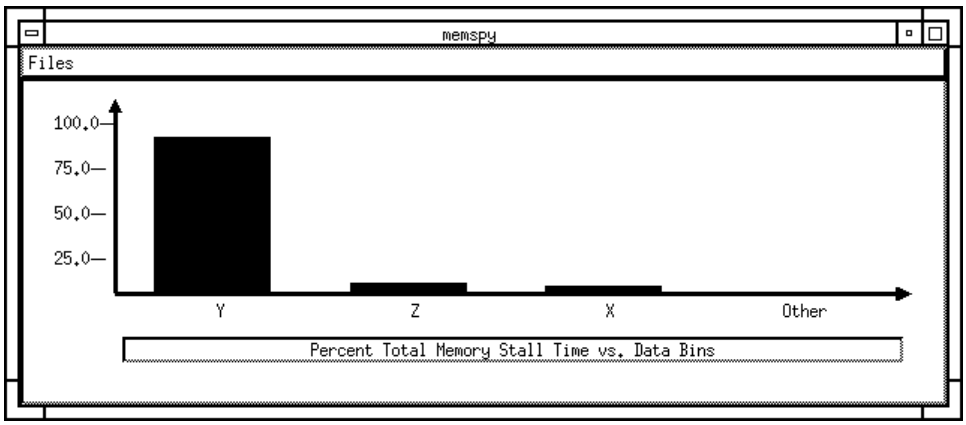

Figure 3.4: MatMul: Memory stall time in the `block` procedure attributed to the `X`, `Y`, and `Z` matrices.

The intuitiveness of this data oriented stall time breakdown belies the significant implementation issues that arise when gathering and presenting these statistics. Critical

issues lie in dividing the program data space into data oriented components and in extracting program names to label the statistics. That is, interesting issues arise in determining (i) when to aggregate statistics on data structures versus viewing statistics on individual data structures and (ii) how to derive program names for statistics in the face of this aggregation. These will be treated in Chapter 4.

At this point, we note that while MemSpy has already provided insight as to *where* the bottleneck is occurring, we still have little understanding of *why* it would be occurring. To get more insights on the problem, we proceed to more detailed statistics.

Detailed Statistics on Causes of Misses

MemSpy allows users to request more detailed statistics about the behavior of references for a particular procedure-data pair. By clicking on the `Y` bar from Figure 3.4, the user brings up the display shown in Figure 3.5.

Figure 3.5 gives more detailed statistics on the behavior of the `Y` matrix in the `block` routine. Most relevant here is the bottom-most bar chart in the display. This graph breaks down the causes of cache misses for the `Y` matrix in the `block` routine. The display shows that in this routine, all of `Y`'s misses are caused by previous replacements. That is, the data objects were all previously in the cache, but have been replaced out of the cache before the re-references occurred that resulted in cache misses. (Note that the entire `Y` matrix is initialized in a separate routine. This is where the first reference misses occur.)

The high number of replacement misses incurred by `Y` is an important piece of information. As discussed in Chapter 2, it indicates that the bottleneck is probably related to cache interference effects. To understand the mechanism causing the memory bottleneck, however, still more information is required. Namely, the user needs to know which accesses are causing the cache replacements. One might expect some small number of replacements due to matrix `X` or matrix `Z`. However, the large number of misses is quite unexpected. By clicking on the replacements portion of the "causes of misses" bar in Figure 3.5, we get a breakdown of the causes of these replacements. This breakdown is shown in Figure 3.6. Surprisingly, over 95% of the replacements are caused by the `Y` matrix itself. This indicates that the performance bug is, in fact, self-interference.

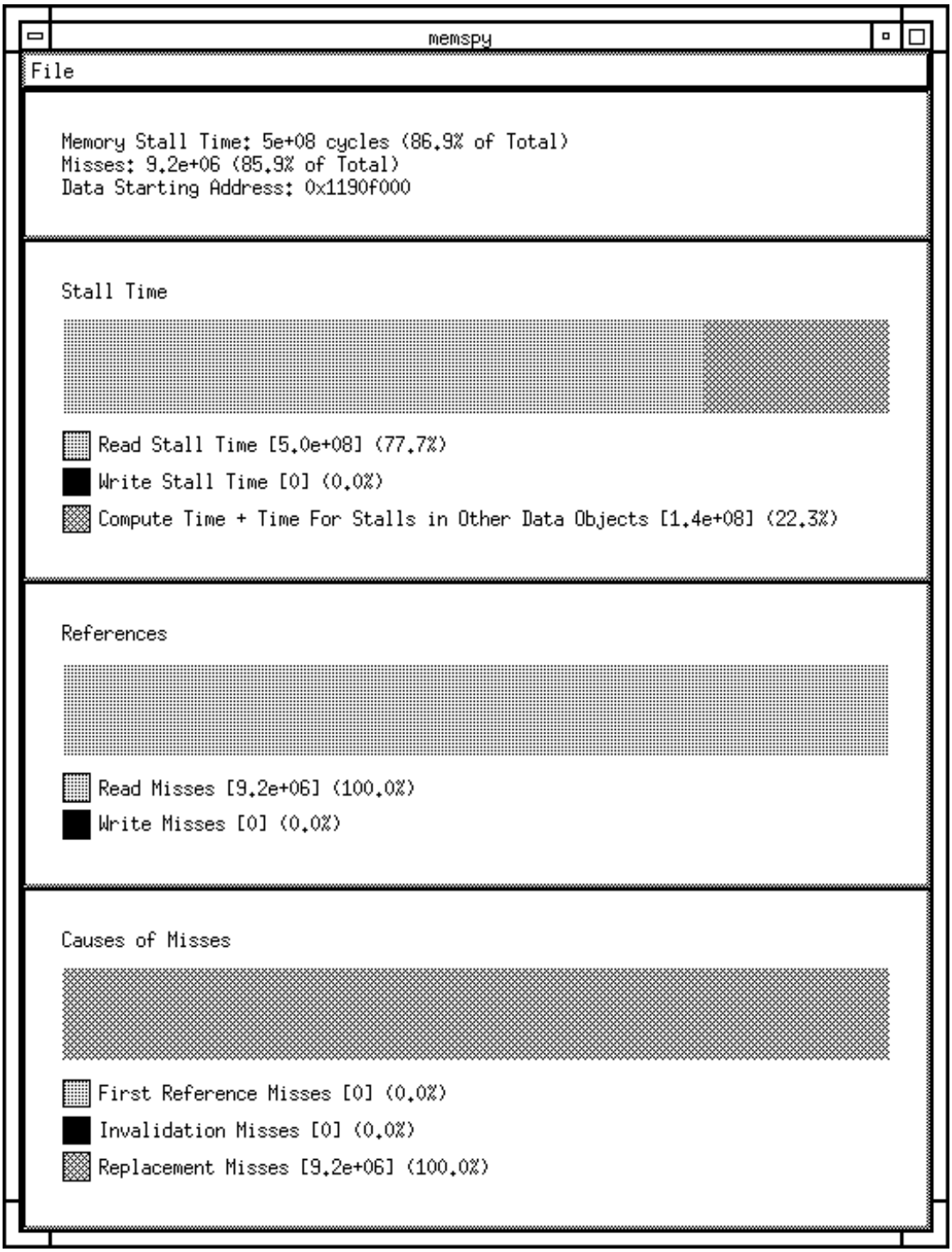

Figure 3.5: MatMul: MemSpy detailed statistics on `Y` matrix in the `block` procedure.

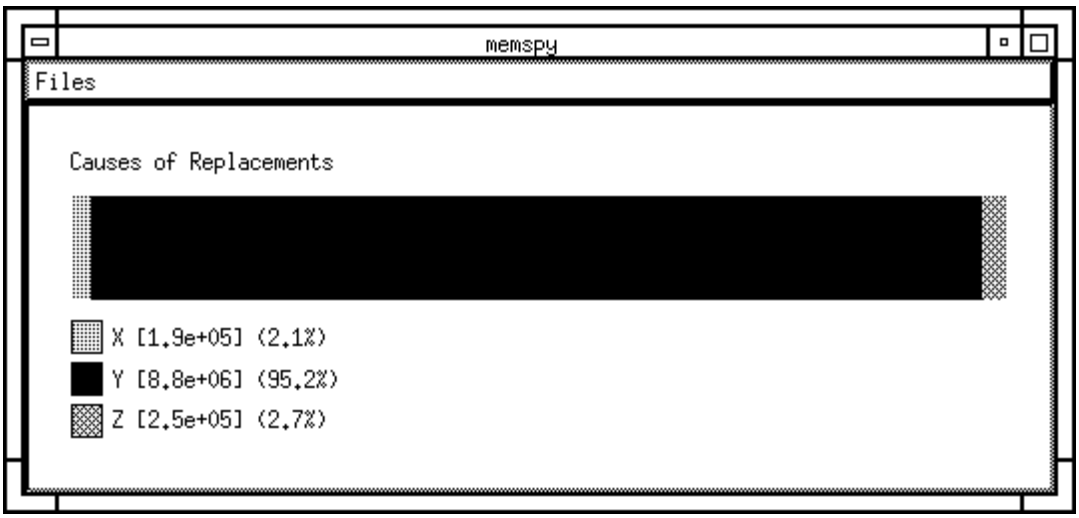

Figure 3.6: MatMul: Causes of replacements for `Y` matrix in the `block` procedure.

Thus in this case, MemSpy takes users to the point where they know that the bottleneck is in the `Y` matrix, that it is caused by excessive cache interference, and that this interference is in fact self-interference, since the replacements are cause by the `Y` matrix itself. The tool gives no further information, but from here, the user can reason about the application to determine the precise fix to the problem in this case.

In particular, self-interference occurs here because the sub-rows within the currently used block of `Y` are not stored contiguously, and thus do not map nicely across the whole cache. Rather, as is shown in Figure 3.7, sub-rows are separated from one another by arbitrary intervening amounts of storage. This leads to cases where some sub-rows map on top of one another in the cache, while other portions of the cache remain unused. Knowing that this is a problem, the programmer can minimize this effect by choosing a block size with less interference, or by copying the block so that it occupies a contiguous region of memory [LRW91].

3.1.3 Example Summary: Matrix Multiply

To summarize, MemSpy has provided step-by-step output that showed (i) `Y` has a surprisingly high miss rate, (ii) the misses are primarily due to replacements, and (iii) the

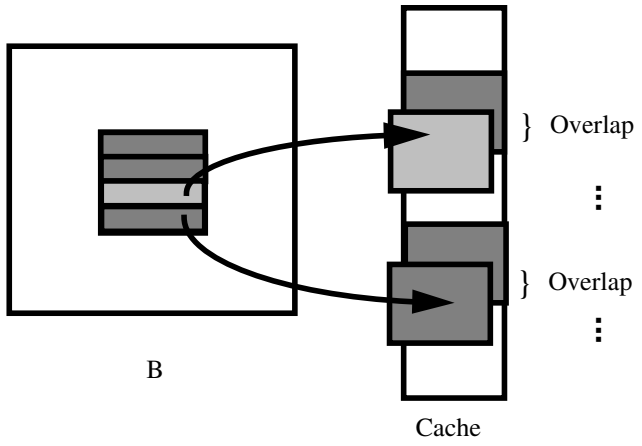

Figure 3.7: Self interference in blocked matrix.

replacements are mainly caused by other references to the `Y` data object. These three facts lead the programmer to identify the problem of self-interference in the `Y` matrix.

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

3.2 Parallel Case Study: Tri

The second case study is a parallel program (`Tri`) which performs 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. As we move to the parallel domain, new issues and new types of performance bugs arise. The example is interesting because it illustrates several such bugs and the role MemSpy plays in identifying them.

First, when parallelizing applications it is common to see a reduction in spatial locality, because data structures are often distributed across processors. In the first step of the case study, MemSpy highlights this performance bug by pointing out a larger than expected number of first reference misses on the main sparse matrix data structure of the

```
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 3.8: Serial pseudo-code for Tri.

problem. An optimization to improve the spatial locality leads to disappointingly small improvements in program performance, however. MemSpy's ability to separate statistics by data structure allows users to distinguish that while the heuristic has improved memory behavior in one data structure, it has degraded it in two others. For one of the data structures we illustrate a high-level change in the program's task synchronization to reduce references to it. For the other data structure, we use MemSpy to evaluate a task scheduling heuristic that can reduce the sharing and improve the spatial locality in the problem.

3.2.1 Problem Description

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 inputs; x is the solution vector to be computed. The pseudo-code in Figure 3.8 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 3.8 in several ways. First, since the input matrix `M` is sparse, the data structures are modified to store the inputs more compactly. The non-zero elements of all rows of M are stored contiguously 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`. In fact for a dense matrix `M`, the dependencies would fully serialize the problem. For sparse `M`, the non-zeroes of `M` represent dependencies between parallel tasks. 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 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 3.9 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 3.9: Pseudo-code for parallel Tri implementation.

3.2.2 Tuning Using MemSpy

Starting with the original parallel code, we describe tuning steps which improve (i) the spatial locality in the matrix `M`, (ii) the synchronization method used, and (iii) the sharing behavior and spatial locality in the vector `x`.

Performance of Original Tri Code

When we run the original Tri code using the benchmark matrix BCSSTK15 [DGL89], we find that the speedup with 4 processors is very low, only a factor of 1.6. To explore the cause, we use MemSpy to identify whether memory performance is a bottleneck.

Figure 3.10 gives the initial MemSpy overview of how execution time is spent in the original parallel version of Tri.[1] The total time spent doing useful computation (shown by the unshaded area in the bar) is roughly the same as that for the sequential code. (The MemSpy displays for the sequential code are not shown here.) Thus, the parallel version spends little time doing computations that were not also present in the sequential version. Furthermore, this application has no significant explicit synchronization time, since the synchronization is implemented implicitly, through spin-loops on shared variables. However, this breakdown shows that there is a significant memory bottleneck in the solver procedure, with over half of the time spent in memory stalls. The time spent stalled for memory in the parallel version is more than triple that for the sequential version. Thus, memory behavior is likely to be the prime reason for the poor speedups. To better understand why these memory stalls are occurring, we can click on the memory portion of the bottleneck bar, to bring up the data oriented breakdown shown in Figure 3.11.

The data breakdown in Figure 3.11 shows that of the solver's total memory stall time, roughly 45% can be attributed to accesses to the non-zeroes of the matrix M, referred to in the code as `M.nz`. The remainder of the stall time is spent primarily in the solution vector `x` and a synchronization vector `ready`. Since `M.nz` causes the most misses in the multiprocessor version, we focus first on its behavior.

To get more detailed information on `M.nz`'s memory behavior, we click on its bar to bring up the data shown in Figure 3.12. This display indicates that the `M.nz` data structure incurs 19K misses during the program execution. By comparison, the corresponding MemSpy display for the sequential version of this code (not shown here) indicates only 11K misses for the `M.nz` data structure. So, in converting to a parallel approach, the number of misses in `M.nz` has increased by over 70%.

[1] Since we are studying only one phase of the problem here, we gather statistics only in the `ForwardSolve` routine. Here the specific name is `ForwardSolvePar_Self`.

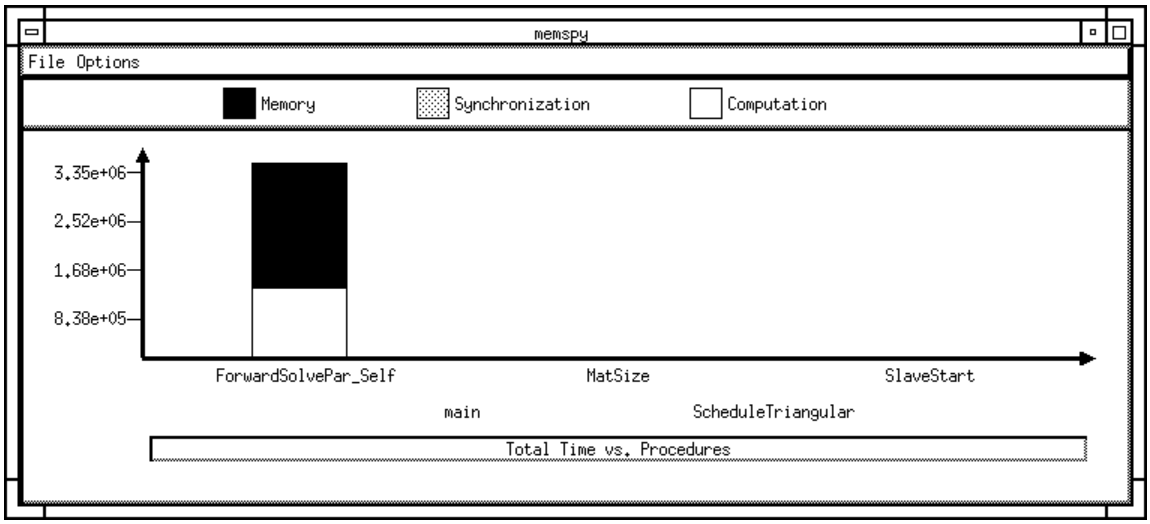

Figure 3.10: Tri: MemSpy overview statistics for original parallel code.

We 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 70%. Furthermore the detailed statistics in Figure 3.12 indicate that most of the misses (roughly 65%) are first reference (cold) misses and not invalidation or replacement misses.

Once MemSpy points out that most of the misses are first reference misses, it is fairly intuitive for the application programmer to figure out that the real cause for increased misses is poor spatial locality in references to `M.nz` in the parallel version of the code. In particular, the explanation for this is related to the fact that the number of non-zeroes per row of `M` is very small in typical input matrices. (For example, if `M` comes from a partial differential equation corresponding to a 5-point stencil, each row has only two off-diagonal non-zeroes.) Since cache lines are 8 double words long (64 bytes), each cache line contains non-zeroes from 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

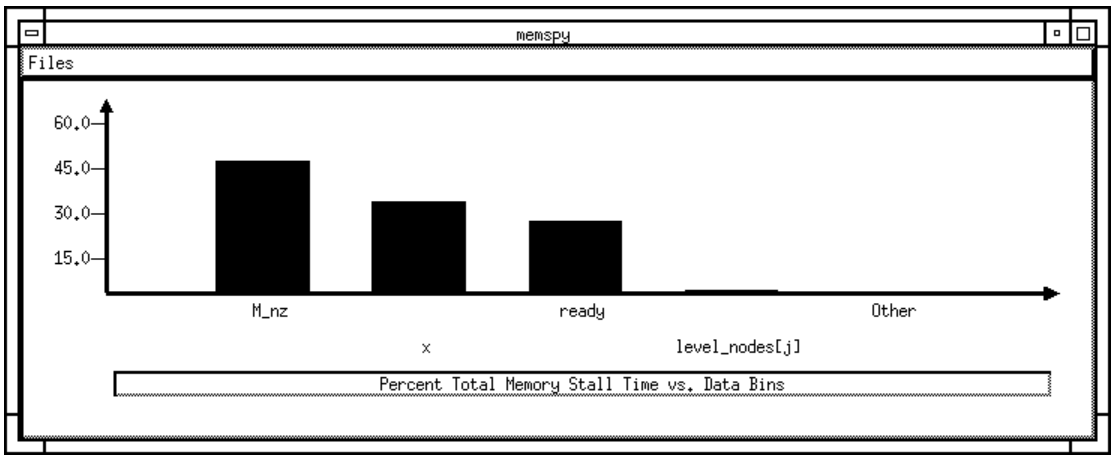

Figure 3.11: Tri: MemSpy data breakdown for original parallel code.

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 statistics on a per-data-structure basis. Without such detailed data oriented statistics, the loss of spatial locality would be difficult to infer.

Step 1: Restoring Spatial Locality

The goal of this tuning step is to improve the spatial locality of references to `M.nz`. This is accomplished by symmetrically reordering the rows and columns of the matrix `M.nz`, so that the 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 [RG92].

When the program is rerun, using the new ordering scheme for spatial locality, MemSpy output indicates that this change leads only to a very minimal improvement in overall performance, about 8%. The overview output shown in Figure 3.13 shows that memory performance continues to be a problem, only slightly improved from before.

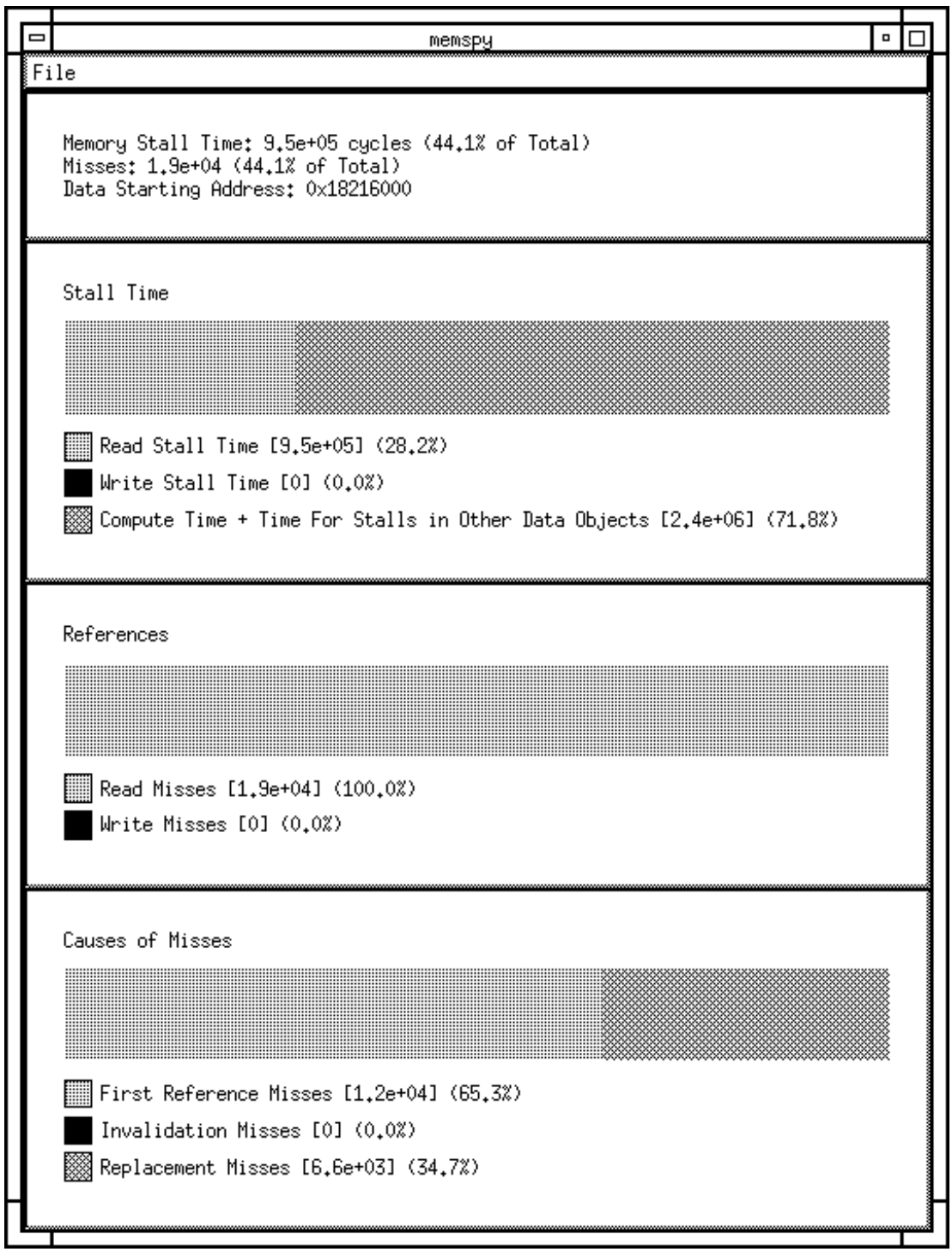

Figure 3.12: Tri: MemSpy detailed statistics for the `M.nz` data structure.

Without more detailed information, the user might surmise that the reordering heuristic was not effective at improving `M.nz`'s memory behavior. However, with more detailed MemSpy statistics, we can see that misses in `M.nz` have been reduced from 19K to 13K, and are now only 18% greater than misses in `M.nz` in the sequential version. The reordering for spatial locality has been effective in reducing the `M.nz` misses closer to the intrinsic number required by the application. However, this reordering has also affected the behavior of `x` and `ready`.

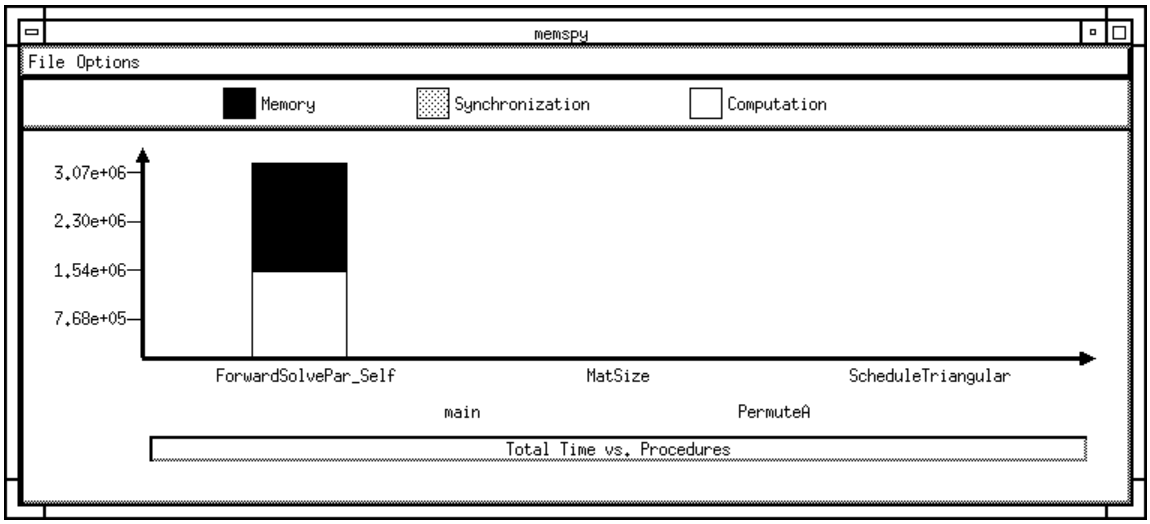

Figure 3.13: Tri: MemSpy overview statistics for step one.

Figures 3.14 and 3.15 show the detailed output for the `x` and `ready` vector after step 1. We see that 72% of `x`'s misses are due to replacements, as are 68% of `ready`'s misses. The programmer can get further information by clicking on the replacement misses portions of the "causes of misses" bars. This causes MemSpy to bring up the new information shown in Figures 3.16 and 3.17. These show that 71% of `x`'s replacements are caused by `ready` and 83% of `ready`'s replacements are caused by `x`. These large numbers of replacements caused by another data structure are known as *cross-interference*, and the interference results in unnecessary memory stall time. (This cross-interference is data dependent, and does not occur as severely in other matrices we have studied.) Thus while the reordering heuristic led to a reduction in misses in `M.nz`, it was offset by

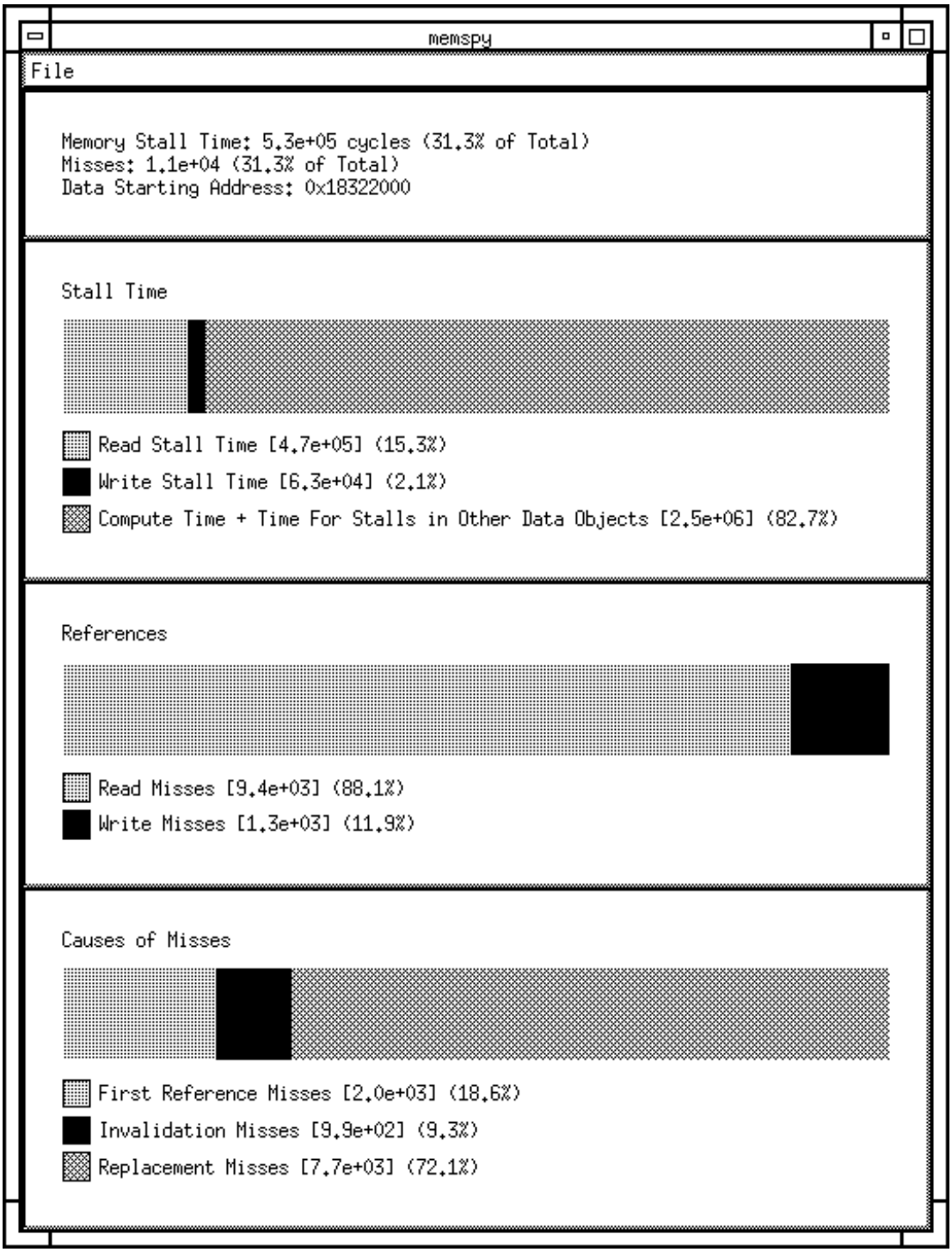

Figure 3.14: Tri: MemSpy detailed statistics for the x vector in step one.

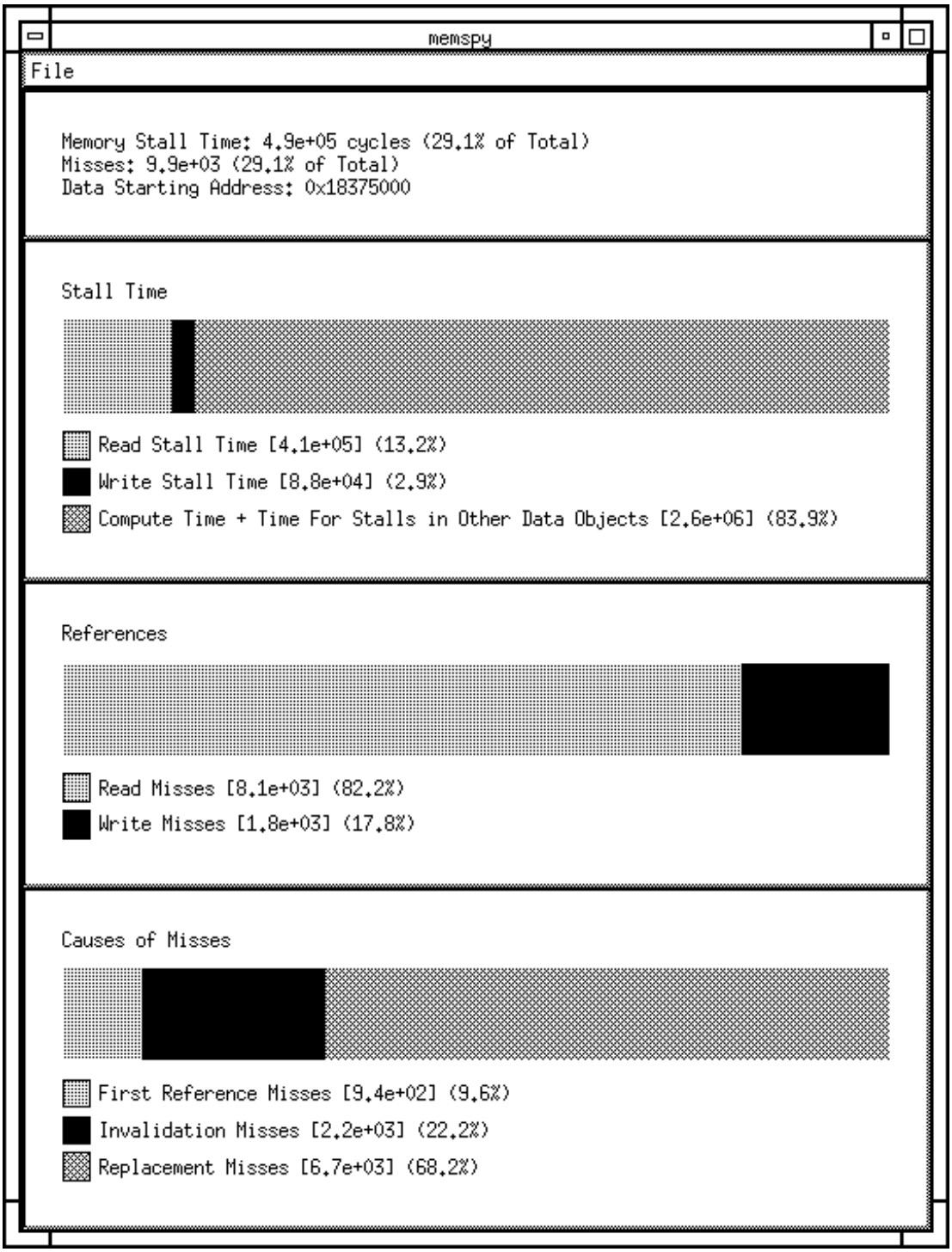

Figure 3.15: Tri: MemSpy detailed statistics for the `ready` vector in step one.

an increase in misses due to cross-interference. Without MemSpy, such countervailing effects are difficult to isolate and understand. The following two subsections will discuss further steps taken to reduce the this cross-interference and improve the behavior of `ready` and `x`.

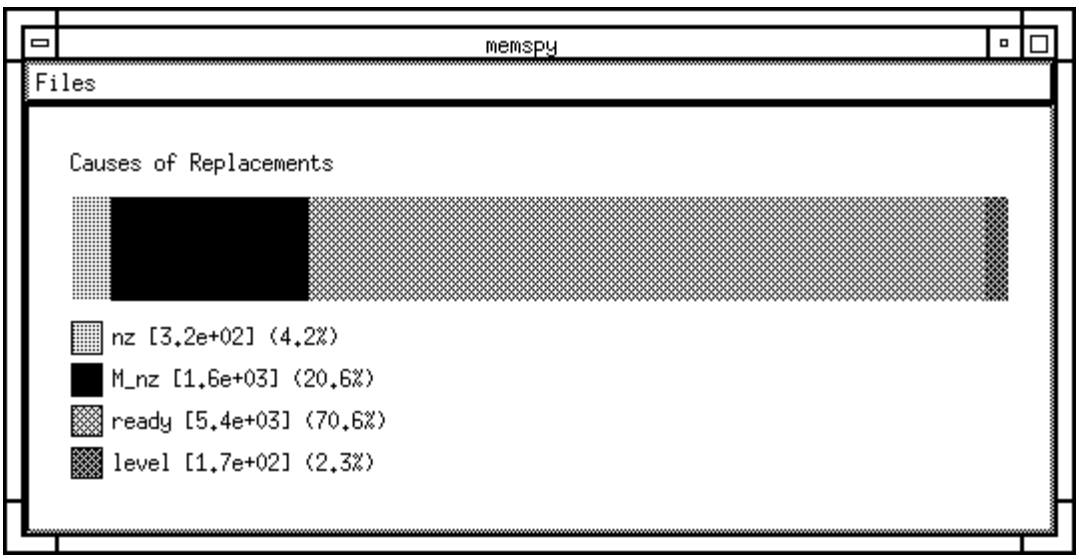

Figure 3.16: Tri: MemSpy causes of replacements for `x` in step one.

Step 2: Reducing `Ready` Traffic

The `ready` vector is used to indicate when a particular `x` element has been computed and is ready for use by later computations. After step 1, the `ready` misses constitute roughly one third of all misses. Of these, the majority are due to cross-interference between `x` and `ready` (indicated by replacements). Another 22.2% are due to invalidations or sharing, and a small amount (9.6%) are first reference misses.

To reduce misses in `ready`, one might first consider ways to reduce the cross-interference or sharing. However, Rothberg and Gupta in fact devised a new form of self-scheduling that allows `ready` to be eliminated entirely [RG92]. 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

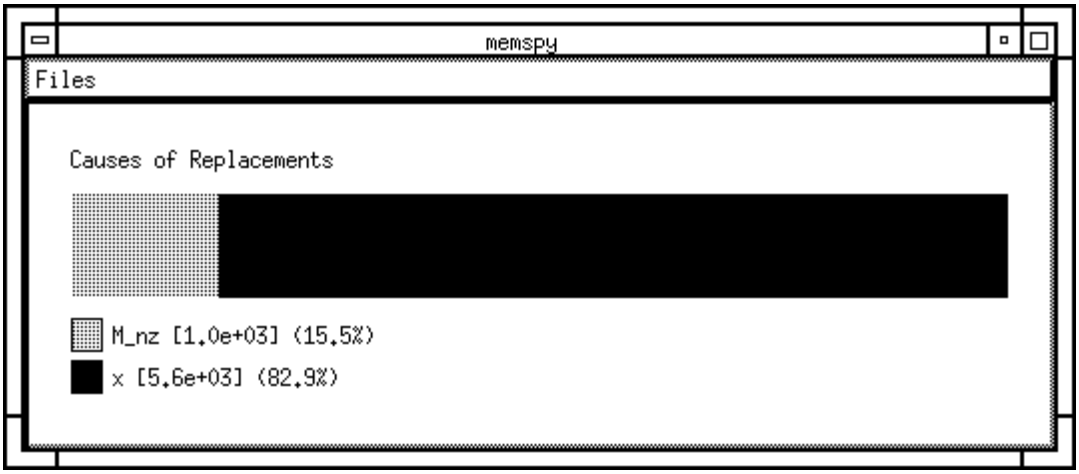

Figure 3.17: Tri: MemSpy causes of replacements for `ready` in step one.

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 as shown in Figure 3.18. The improvement comes due to two effects on the memory system behavior. First, as shown in Figure 3.19, `ready` misses are eliminated entirely. Furthermore, misses due to the `x` vector are also substantially reduced due to a decrease in the cross-interference previously described. As with the previous examples, MemSpy's ability to isolate and quantify the performance of different data structures drives the tuning process here. The next subsection focuses on improving the performance of `x`.

Step 3: Reducing Traffic due to `x`

Having reduced the cross-interference misses for `x`, cache misses for `x` now primarily occur when an `x` element produced by one processor is subsequently used by another processor. It would be preferable for `x[i]` to be produced and then used by the same 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

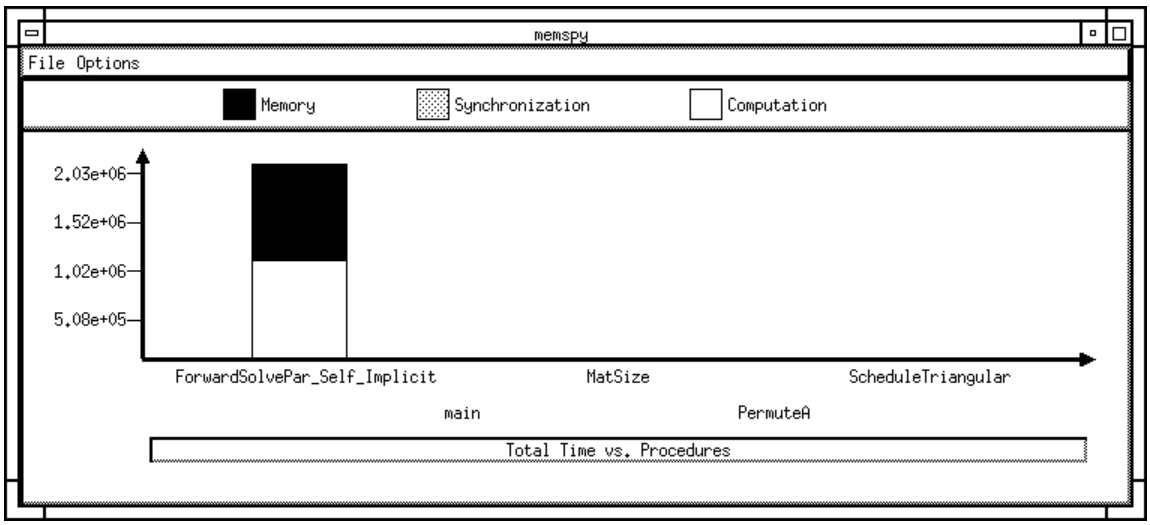

Figure 3.18: Tri: MemSpy overview statistics after step two.

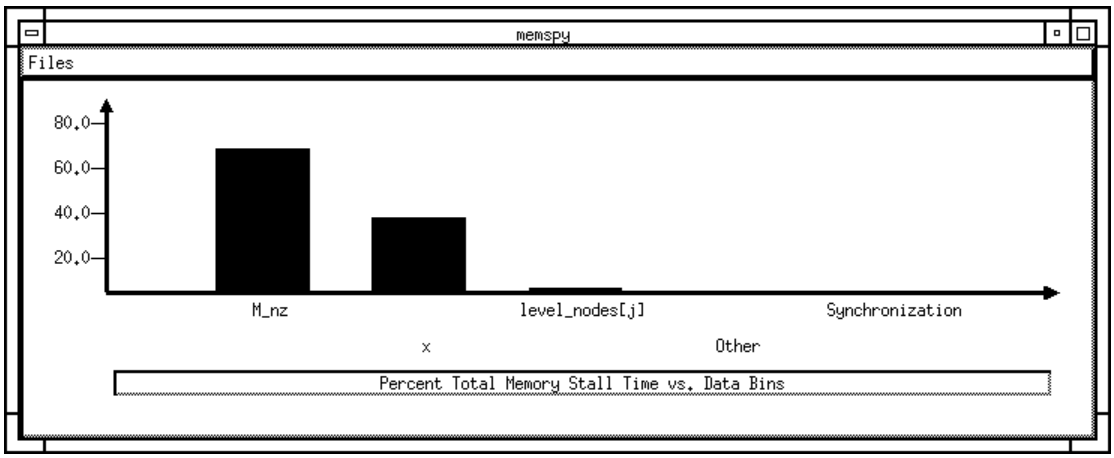

Figure 3.19: Tri: MemSpy data breakdown after step two.

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. 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 Figure 3.20) that misses due to the `x` vector decrease from 6.4K to 4.3K with this heuristic. At this point around 30% of these misses are first reference misses, 8% are due to invalidations, and 62% are due to replacements. MemSpy further indicates that almost all (99%) of the replacements at this point are due to the `M.nz` matrix. Since Tri streams through the data in the very large M matrix, these replacements are essentially unavoidable (unlike the interference due to `ready` that was previously noted).

3.2.3 Example Summary: Tri

This case study has highlighted how MemSpy may be used to tune a parallel application's memory behavior. To summarize, Tables 3.1 and 3.2 give a step-by-step synopsis of the memory behavior for the main program variables, as well as overall program performance at each tuning step. In the first tuning step, MemSpy was used to calculate miss counts for the `M.nz` data. These data oriented statistics played a key role in pointing out that poor spatial locality was the cause of the increase in misses when parallelizing the application.

Table 3.1: Tri: Summary of MemSpy output throughout tuning sequence.

Version	Cache Misses (x 1000)			
	Total	`M.nz`	`ready`	`x`
Seq.	13	11 (85%)	—	1.7 (13%)
Orig Par.	43	19 (44%)	10 (23%)	13 (30%)
Step 1	34	13 (38%)	10 (29%)	11 (32%)
Step 2	20	13 (65%)	—	6.6 (33%)
Step 3	16	11 (69%)	—	4.3 (27%)

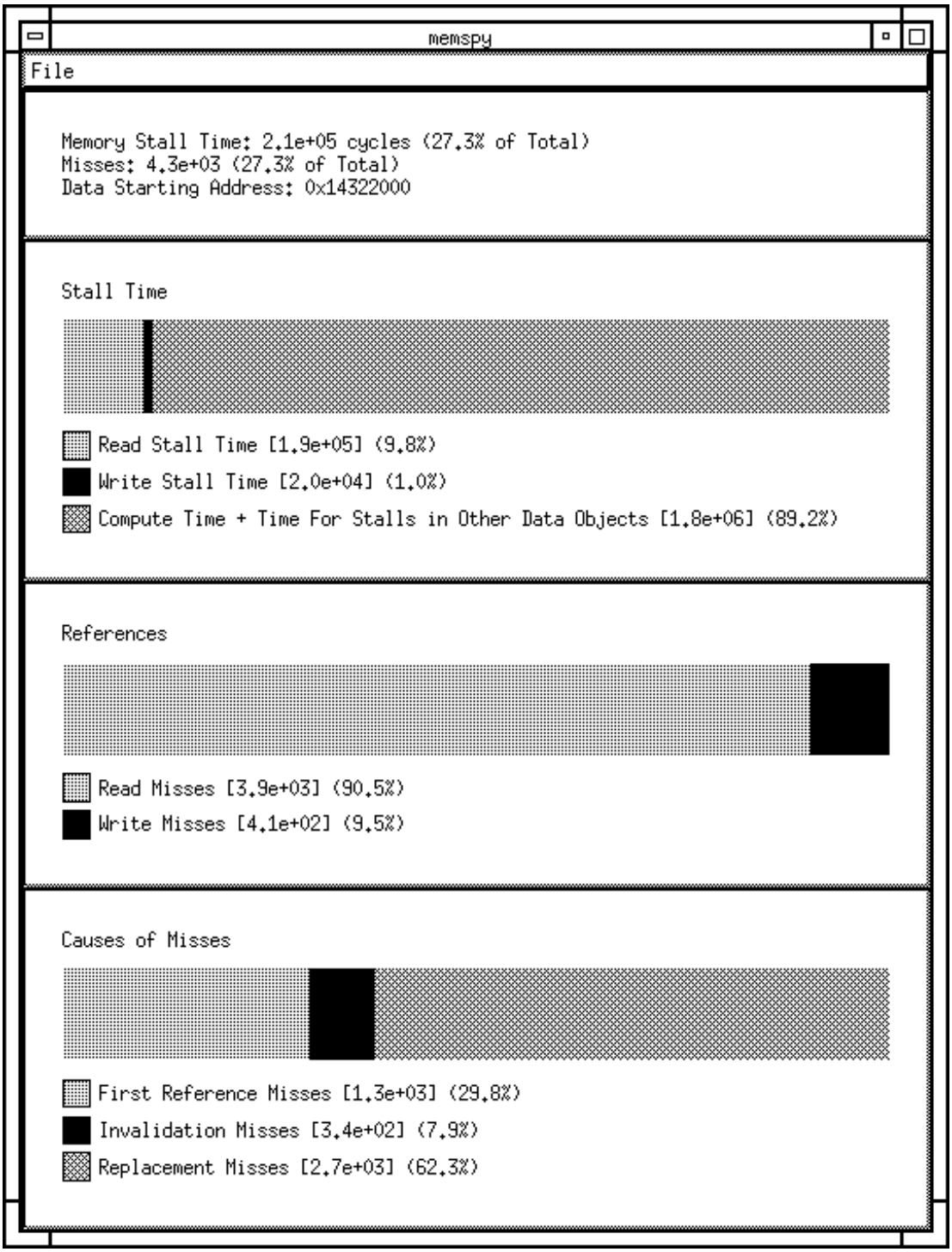

Figure 3.20: Tri: MemSpy data breakdown for x after step three.

Table 3.2: Tri: Summary of simulated performance throughout tuning sequence.

Version	Execution Time (x1000 processor cycles)	Speedup
Seq.	2694	1.00
Orig Par.	1677	1.61
Step 1	1537	1.75
Step 2	1016	2.65
Step 3	989	2.72

Based on this information, we reordered the matrix which improved spatial locality. MemSpy's information on the causes of misses was instrumental in helping us understand the cross-interference that resulted from this 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.

3.3 Chapter Summary

This chapter has illustrated MemSpy's use on both sequential and parallel applications. In the sequential application, blocked matrix multiply, an instance of *self-interference* was identified. Here, MemSpy's data oriented statistics were useful in initially identifying which of three matrices was primarily responsible for the memory stalls. Also, MemSpy's detailed statistics on the causes of cache misses were useful in identifying the problem as *self*-interference, rather than either cross-interference with other variables, or perhaps other possible performance bugs.

In the parallel application Tri, we stepped through a case study improving the memory behavior of each of the primary data structures in the algorithm. The tool was instrumental in isolating the initial contributions of different variables to the total memory stall time, and quantifying the effects of the optimizations on different variables. This easily identified and separated instances of poor spatial locality, cross-interference, and interprocessor sharing.

MemSpy has also been used to tune several other programs. For example, it has identified performance bugs due to: (i) false sharing and a "vestigial" (incremented but unused) variable in `LocusRoute`, a SPLASH benchmark [SWG92], (ii) self-interference in the `ElementArray` in `Pthor`, another SPLASH benchmark, (iii) poor spatial locality in a sequential volume rendering program, `Vrender`, and (iv) shared accesses to a private variable in a parallel version of `Vrender`.

Overall, these experiences have demonstrated the importance of data oriented statistics in identifying memory performance bugs and offering insights on program behavior. Their combination with detailed information on the causes of cache misses allows users to discern types of performance bugs present for each data structure. Furthermore, the detailed, data oriented information on the causes of replacements allows users to easily identify causes of self-interference and cross-interference, and devise solutions for them.