Appendix A

Additional Case Studies

This appendix presents two more case studies in addition to those presented in Chapter 3.

The first case study shows MemSpy's use on LocusRoute, a parallel application that is part of the SPLASH [SWG92] benchmark suite. This study shows MemSpy pointing out false sharing in a number of program variables. A simple restructuring of the code leads to significant performance improvements.

The second case study shows the tuning process for Vrender, a volume rendering program [LL93, Agr93]. This study shows MemSpy's use to detect a problem of poor spatial locality in the sequential version of the code, as well as correctness and performance bugs in the parallel version of the code.

In both the LocusRoute case study and the parallel portion of the Vrender case study, we use a more complex memory simulator than the one described in Chapter 4. This simulator is described in Section A.1.

A.1 Simulator Description

For the parallel case studies discussed in this appendix, we present results measured using a slightly more complex simulator than the one described in the body of the thesis. There are several main changes from the simpler simulator. First, while the previous simulator rescheduled between simulated parallel threads at the granularity of synchronization events, this simulator interleaves threads at a much finer granularity – once per memory

reference. For applications which synchronize relatively frequently, this has little effect. However, for applications which synchronize less frequently, like LocusRoute, this method more realistically interleaves references from different processors, which brings out effects like the false sharing shown here.

A second key difference between this simulator and the original one is that an infinite write buffer is simulated by setting the latency of all write operations to be exactly one processor cycle. In the original simulator, reads and writes had the same latency.

Finally, in this simulator, cache misses have variable latency depending on which processor's memory services them, and how distant it is from the referencer. A cache miss serviced locally incurs an overhead of 34 processor cycles. For cache misses serviced remotely, transactions must travel across the two dimensional mesh network, which uses a wormhole routing method. For a 32 byte line, a message traveling one hop on the network take 85 cycles, but each additional hop requires only an extra 5 cycles.

For LocusRoute, the key attribute of this simulator that led to its use was the more frequent thread reschedules. Since LocusRoute is an application which performs very little synchronization, rescheduling only at synchronization points leads to thread interleavings with significantly less false sharing. While the simple simulator was useful for detecting early performance bugs in LocusRoute, the bug of false sharing described here becomes more apparent with tighter, more realistic, thread interleavings.

For Vrender, this simulator was chosen to primarily to display the additional penalties for accesses to memory in remote clusters, as compared to accesses to memory in local clusters. The performance bugs discussed in this case study were also apparent when using the simple simulator, but the simpler simulator was much more optimistic in the degree to which fixing these bugs would improve the performance of the parallel Vrender.

A.2 LocusRoute

The LocusRoute program is one of the parallel benchmarks from the SPLASH benchmark suite [SWG92]. LocusRoute performs automatic routing of the wires in VLSI standard cell circuits.

A.2.1 Initial MemSpy Output

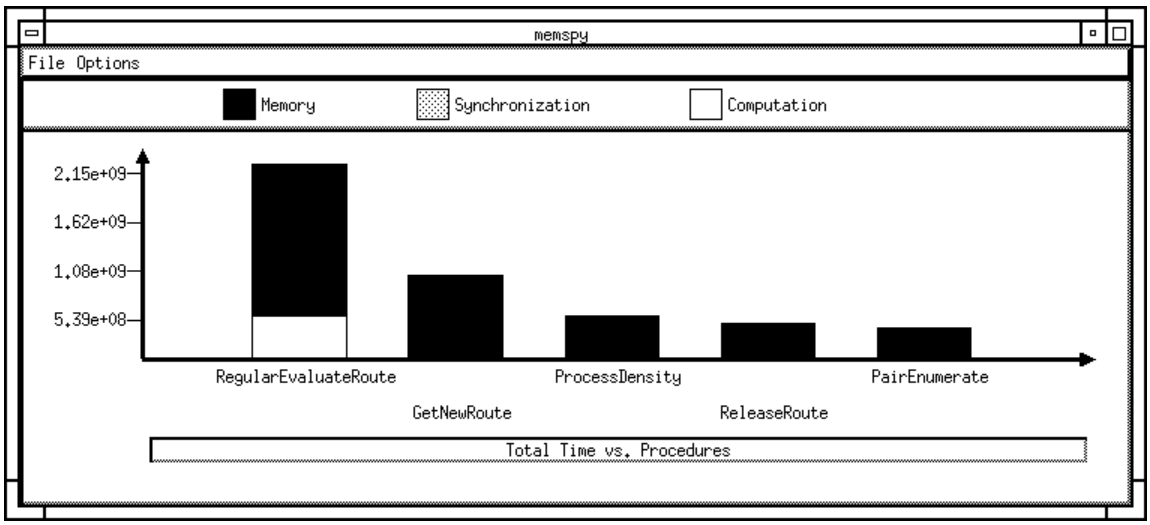

Figure A.1: LocusRoute: Initial MemSpy output display.

Figure A.1 shows MemSpy's initial output for LocusRoute. This output indicates that a 16 processor run of LocusRoute on the input file Primary2.grin requires roughly 2.15 billion cycles totalled over all processors. The execution time for the parallel run is roughly 14.5 seconds. This display also highlights significant memory bottlenecks in all five of the routines shown. If we were to click on the bars for each of these routines, we would bring up five different data breakdowns. To summarize these breakdowns, which are not pictured here, they show that program's main shared data structure, `Global`, is the bottleneck in the `RegularEvaluateRoute` and `ProcessDensity` routines. However, in `GetNewRoute`, `ReleaseRoute`, and `PairEnumerate`, the memory bottleneck is, surprisingly, the program's static data. Overall, it is responsible for roughly 45% of the program's memory stall time. To understand why this is, we look at the detailed statistics display for the static data (Figure A.2), and see a large number of invalidation misses – over 98%.

Figure A.3 shows the code for one of the routines with many invalidation misses to static data. Here, the only static data is the `RouteFreeListHead` array. This array holds per-processor free lists for `Route` data objects. Each of the array elements is a

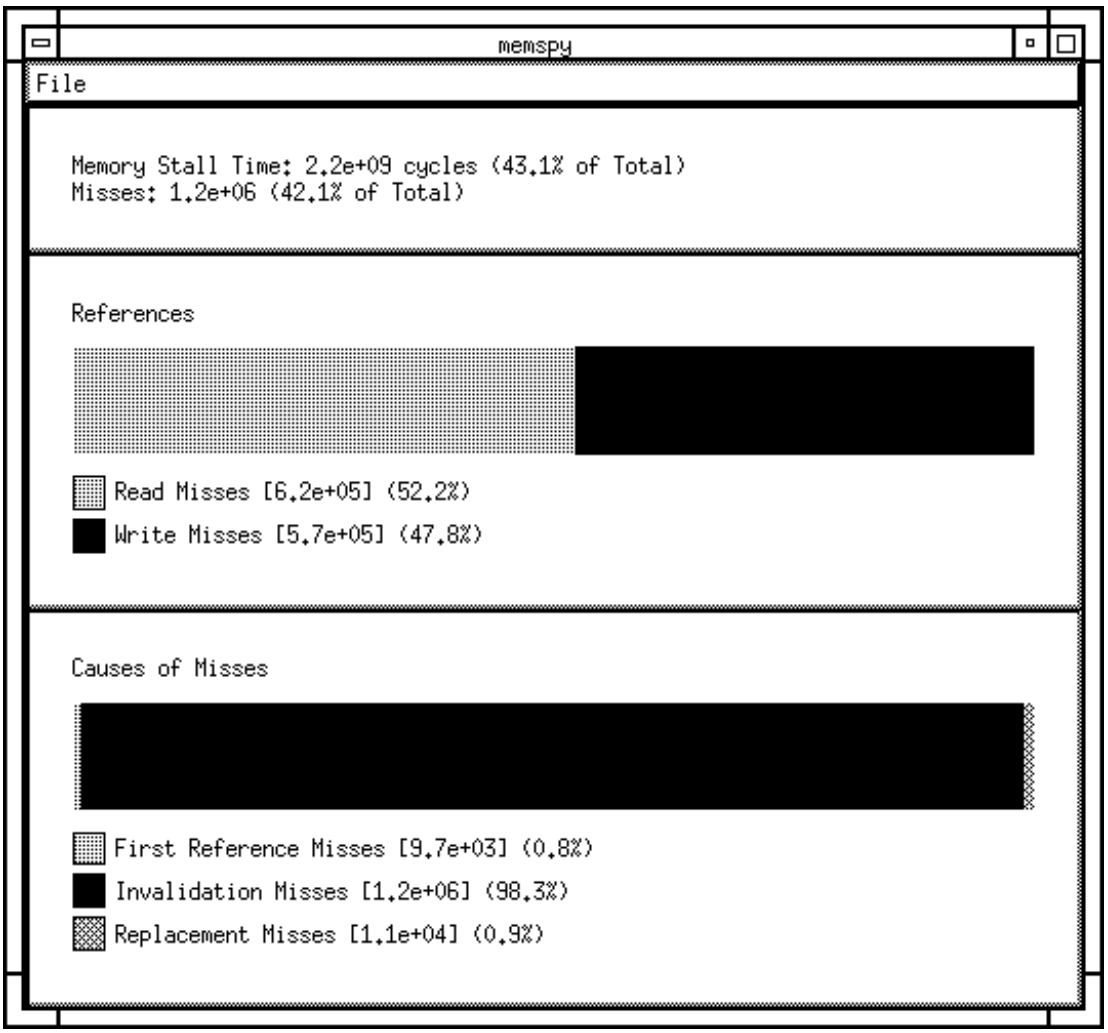

Figure A.2: LocusRoute: Detailed MemSpy output for `static data`.

```
Route GetNewRoute(NumberOfBytes)
int NumberOfBytes;
{
    Route NewRoute;
    int SlaveNumber;

    GET_PID(SlaveNumber)

    if (RouteFreeListHead[SlaveNumber] == RouteNil) {
        NewRoute = (Route) G_MALLOC(sizeof(RouteCornerType));
    }
    else {
        NewRoute = RouteFreeListHead[SlaveNumber];
        RouteFreeListHead[SlaveNumber] =
             RouteFreeListHead[SlaveNumber]->Link;
    }

    return(NewRoute);
}
```

Figure A.3: LocusRoute: `GetNewRoute` code.

4 byte pointer, so the pointers for 8 different processors fit into a single 32 byte cache line. Thus, the potential for false sharing of these cache lines is quite significant. Similar instances of false sharing appear in several other procedures as well.

A.2.2 Reducing False Sharing

Figure A.4 shows the definitions of several frequently used per-processor static variables, including `RouteFreeListHead`, the one accessed in `GetNewRoute`. Each of these variables is defined as a one-dimensional vector of length `MAXPROCS`. Throughout the program, they are indexed using the particular thread index. Such array-based definitions of fairly small per-processor variables are prone to false sharing, because several elements, assigned to different processors, are contained in a single cache line.

To reduce the false sharing, we can restructure the definitions as shown in Figure A.5 to coalesce per-processor variables into a single structure definition, and then define an

```
Route RouteFreeListHead[MAXPROCS];
struct SegmentHeadSyncRecord *SegmentHeadSync[MAXPROCS];
int CurrentRouteListEntry[MAXPROCS];
Wire *CurrentWire[MAXPROCS];
struct SegmentHeadSyncRecord *SegmentHeadSyncFreeList[MAXPROCS];
struct EnumerateSyncRecord *EnumerateSyncFreeList[MAXPROCS];
RoutedWire *RoutedWireFreeList[MAXPROCS];
SegmentRouteType *SegmentRouteFreeList[MAXPROCS];
Route RouteSets[MAXPROCS][MAXROUTESETS];
Route BestRoute[MAXPROCS];
```

Figure A.4: Static variable definitions prone to false sharing.

```
typedef struct perprocvars {
  Route RouteFreeListHead;
  struct SegmentHeadSyncRecord *SegmentHeadSync;
  int CurrentRouteListEntry;
  Wire *CurrentWire;
  struct SegmentHeadSyncRecord *SegmentHeadSyncFreeList;
  struct EnumerateSyncRecord *EnumerateSyncFreeList;
  RoutedWire *RoutedWireFreeList;
  SegmentRouteType *SegmentRouteFreeList;
  Route RouteSets[MAXROUTESETS];
  Route BestRoute;
} perproc;

perproc pp[MAXPROCS];
```

Figure A.5: Static variable definitions restructured to reduce false sharing.

array of that structure. In this way, data items are grouped by the processor using them. This tends to (i) reduce false sharing and (ii) improve per-processor spatial locality. It reduces false sharing because most cache lines contain only data used by one process. It improves spatial locality because each cache line contains items useful to a particular processor. In the previous approach using per-processor arrays, cache lines contained only one item of use to a particular processor.

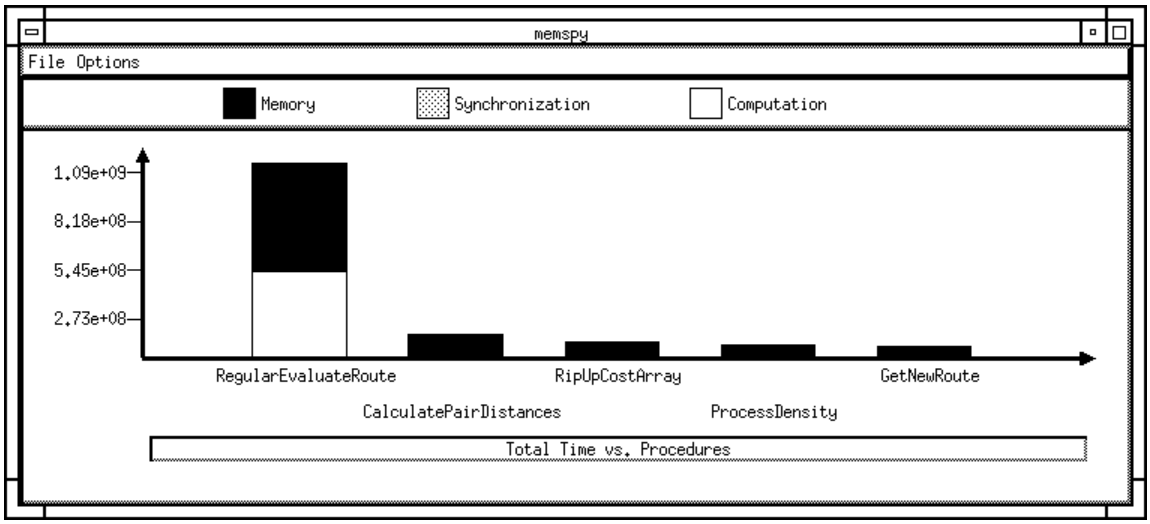

Figure A.6: LocusRoute: MemSpy output display after restructuring.

Following this optimization, simulated performance improved by more than a factor of two. Simulated runtime for the code is now just 4.76 seconds. Figure A.6 shows the new execution time breakdown for LocusRoute after the restructuring.

A.3 Vrender

The second case study, Vrender, shows the tuning process for a fast volume rendering program that uses a novel shear-warp algorithm [LL93]. Volume rendering is a technique for producing two dimensional images from three dimensional sampled data. This visualization problem is important in many domains, including medical imaging, graphical visualization for science and engineering, and the entertainment industry.

The input for volume rendering is a large three dimensional array of scalar values called voxels. Using models for computing opacity, color and shading, the volume renderer processes this voxel data into two-dimensional color images.

This case study is divided into two parts. First, we discuss changes made to the sequential version of the code, to improve its memory behavior. The changes result in an overall performance improvement of 11.2% in rendering an image. This performance

improvement is especially significant since the volume rendering code was already highly optimized prior to this change. (It is currently the fastest existing sequential volume rendering implementation.) Next, we discuss the process of using MemSpy as an aid to parallelizing this code. We show MemSpy's usefulness in detecting both memory correctness and performance bugs.

A.3.1 Sequential Vrender Code

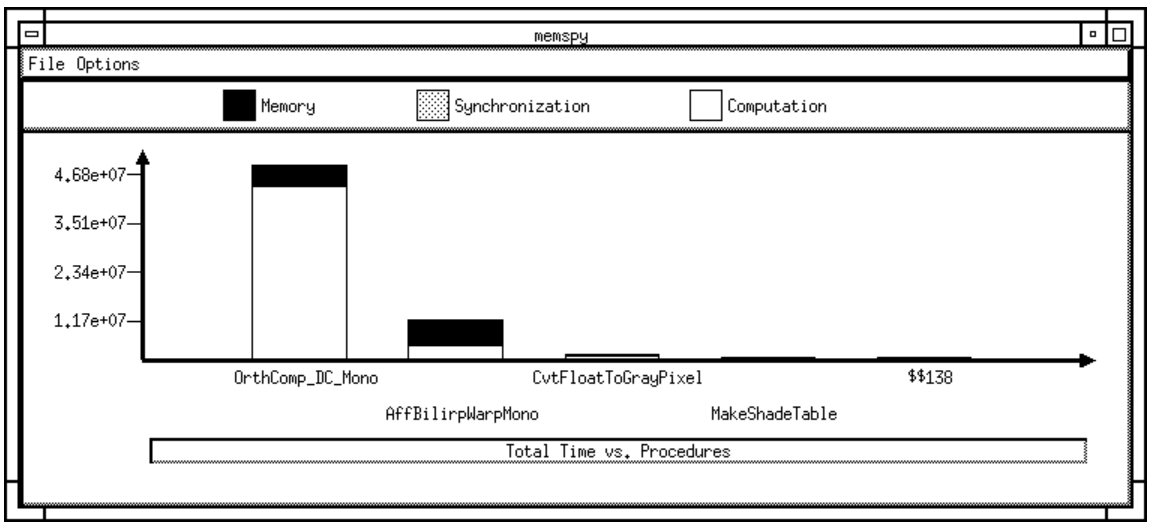

Figure A.7: Vrender: Initial MemSpy output display.

The initial MemSpy simulation indicates that the original sequential code renders an image from a 256 x 256 x 225 volume data set in 1.5 seconds. A single rendering takes roughly 1.2 seconds, while the rest of the time is spent in initialization. Figure A.7 shows an overview of the memory bottlenecks in this original version of the code.

Figure A.8 shows the data bottlenecks within `OrthComp_DC_Mono`, the main rendering routine. The bulk of the stall time is devoted to the run-length-encoded volume data stored in `run_data` and `run_lens`. The volume data is much too large to fit in the cache, so it is not surprising to see that it incurs much of the stall time. However, the programmer at this point was surprised to see the other three data structures: `cm_opcflt`, `cm_clrflt`, and `cm_lnk` that also appeared near the top of the bottleneck

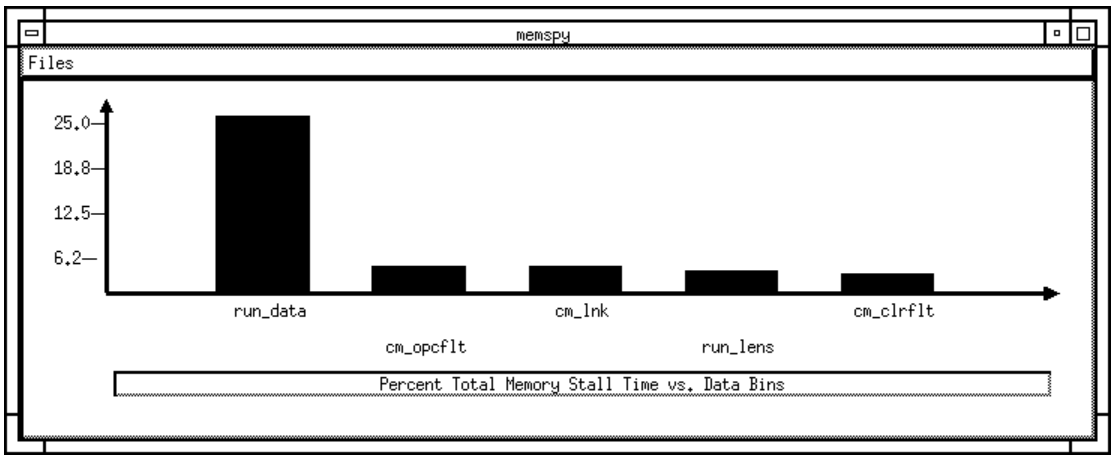

Figure A.8: Vrender: MemSpy data display for initial sequential code.

ranking. These data structures are much smaller, and were not expected to be significant bottlenecks. The arrays `cm_opcflt`, `cm_clrflt` contain the opacity and color information of the composited image being calculated. The third array `cm_lnk` indicates which parts of the image have been fully computed so far.

As Figure A.9 shows for `cm_opcflt`, the misses for these data structures are split between roughly one third first reference misses, and two thirds replacement misses. Most of the elements in these arrays are accessed on each processing of a two dimensional 256 x 256 "slice" of voxel data. However, each element is read only once per slice of voxel data. With this referencing pattern, there is little temporal locality in the three arrays. Since so much volume data sweeps through the cache on each iteration, elements of `cm_opcflt`, `cm_lnk`, and `cm_clrflt` are unlikely to remain in the cache until their next usage. Thus, the code must optimize the spatial locality of the three arrays *within each iteration*, to best take advantage of cache line prefetching opportunities.

To do this, the programmer merged the three arrays into a single data structure so that corresponding elements of the arrays are likely to be on the same cache line. The programmer redefined the array elements as three elements of a structure, and then allocated an array of that structure. This takes advantage of locality: corresponding

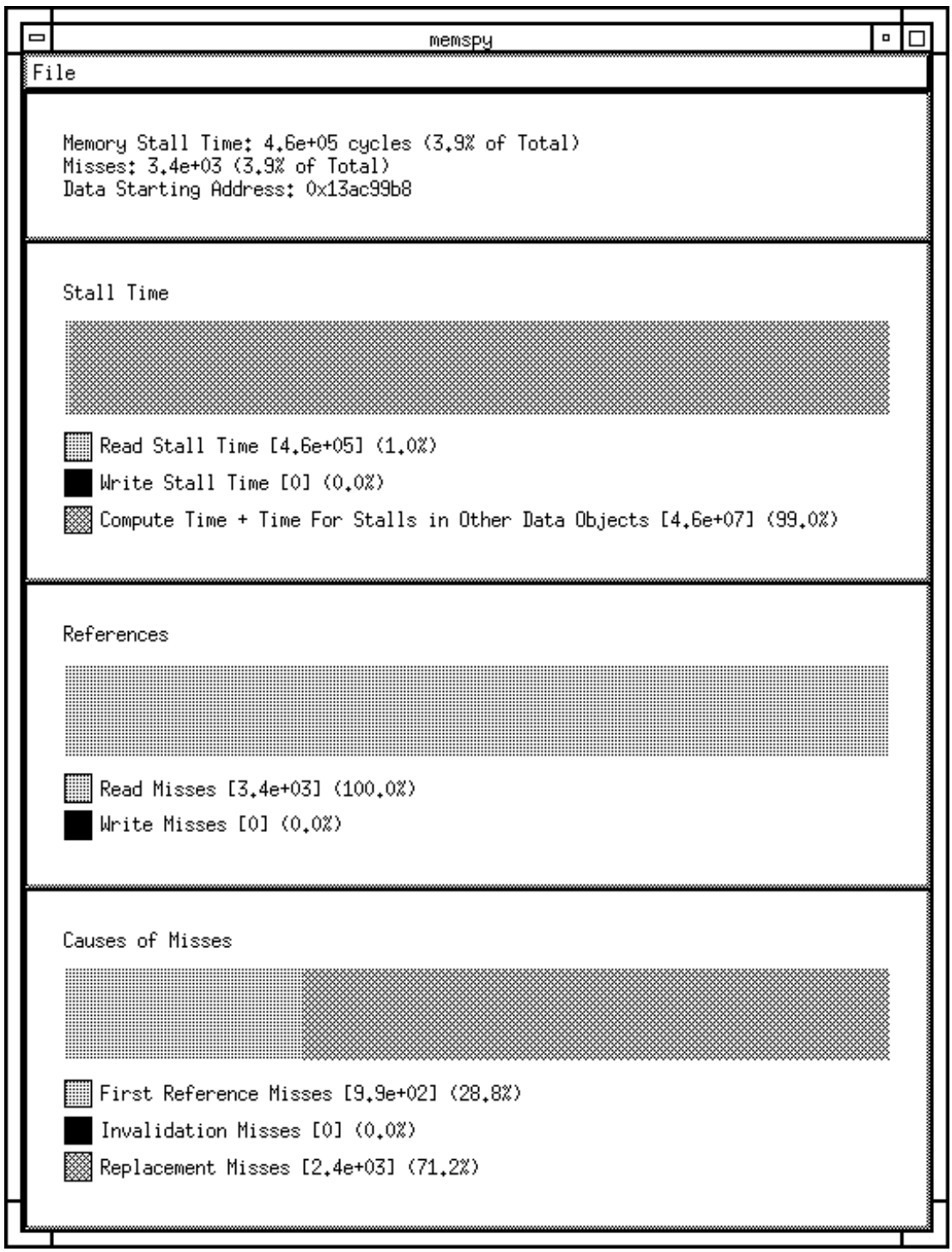

Figure A.9: Vrender: Detailed MemSpy output for `cm_opcflt`.

elements of the three arrays are expected to be read in close succession. When the first variable of the triplet is referenced, the other two variables are likely to be pulled into the cache on the same cache line.

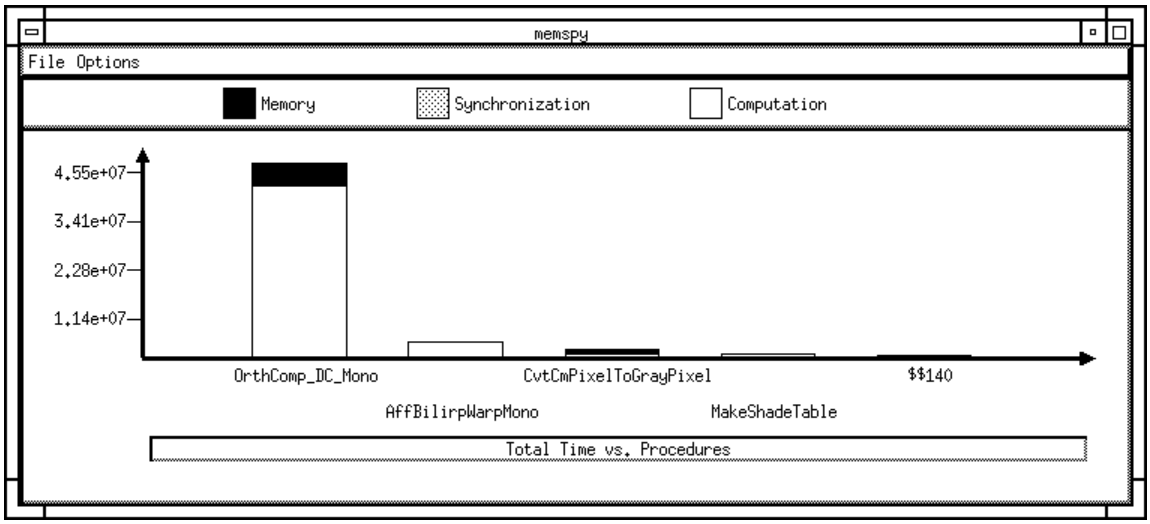

Figure A.10: Vrender: MemSpy output display for optimized sequential code.

Figure A.10 shows the new MemSpy output following this optimization. The overall performance of the code has been improved by about 12%. It now runs in roughly 1.32 simulated seconds. The percentage of stall time attributed to the three interleaved variables has dropped to less than 5%. (Note that this performance improvement applies to the first rendering performed for a volume. If subsequent renderings are performed from slightly different viewing angles, the three arrays would be likely to remain in the 1MB secondary cache anyway, so the locality improvement has little effect.) At this point, the bulk of the memory overhead stems from accesses to the volume data itself, which are difficult to optimize further.

A.3.2 Parallel Vrender Code

In order to obtain substantially faster execution time, the sequential Vrender code was parallelized [Agr93] to run on medium-scale shared memory multiprocessors such as the Stanford DASH machine [LLG+90, LLJ+93]. In the decomposition, each processor is

assigned a contiguous set of rows from the volume data, to calculate their contribution to the image. By statically assigning tasks, the programmers hoped to minimize synchronization overhead in the parallel decomposition. Furthermore by assigning contiguous lines to the same processor, the programmers hoped to take advantage of locality in the data set. The locality comes about because a single voxel scanline contributes to two scanlines of the composited image. Thus, one can reuse the common voxel data.

Unfortunately, the MemSpy simulated runtime for this version with 1 processor is 2.4 seconds, or roughly 1.8 times larger than the previous sequential version. With 16 processors, the execution time drops to 1.65 seconds, a runtime which is still larger than the best sequential code and which represents only a factor of 1.4 speedup over the 1 processor parallel code. This subsection details the use of MemSpy to improve the parallelism and execution time of the parallel code.

Initial Parallel Decomposition

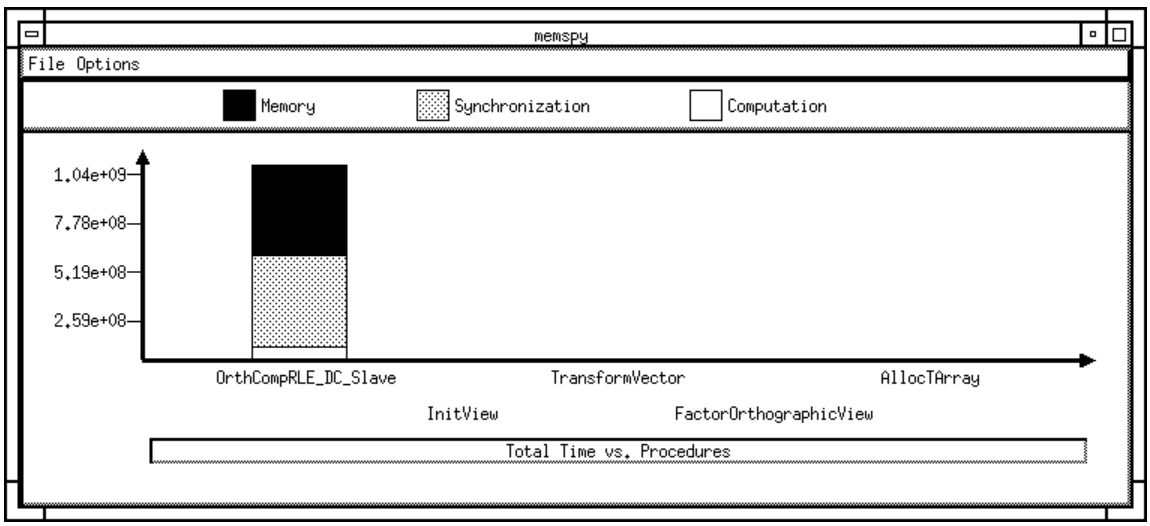

Figure A.11: Vrender: MemSpy output display for initial parallel code.

Figure A.11 shows the MemSpy output for the initial parallel decomposition. The output is shown only for the core volume rendering routine itself, and not for the (still sequential) initialization routines. The output shows that only a tiny fraction of the total time, less

than 10%, is spent in useful computation. The rest of the time is split roughly equally between memory stall time and waiting at synchronization objects.

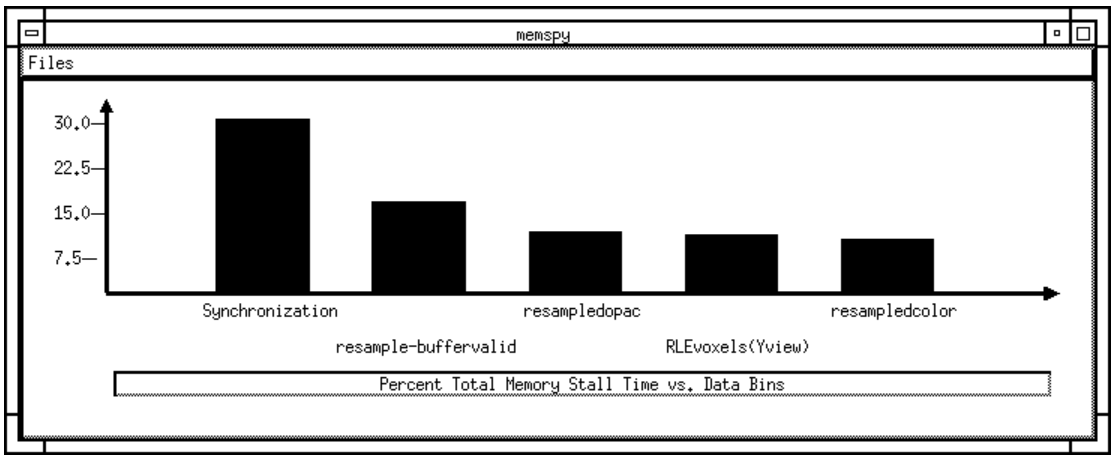

Figure A.12: Vrender: MemSpy data display for initial parallel code.

Figure A.12 shows the data breakdown for memory stall time in this routine. The programmer noticed a significant fraction of stall time (15%) was devoted to the variable called `resample_buffervalid`. This data structure is associated with the current scanline of voxels. It is used to indicate whether these voxels have been visited before or not. Figure A.13 shows the detailed statistics for this data structure. Almost half the misses in this data structure are due to invalidations. Since voxel scanlines are associated with particular processors, each processor should have its own copy of this buffer. That is, this data structure was intended to be an unshared structure with *no* invalidations at all! At this point, the programmers examined the code and realized that due to a *correctness* bug in the code, the data had been misallocated; all the processors were sharing a single buffer, rather than each allocating its own.

This bug is an interesting example of how MemSpy's detailed information on the causes of a data structure's misses can point both performance and correctness behavior that does not match the programmer's expectations.

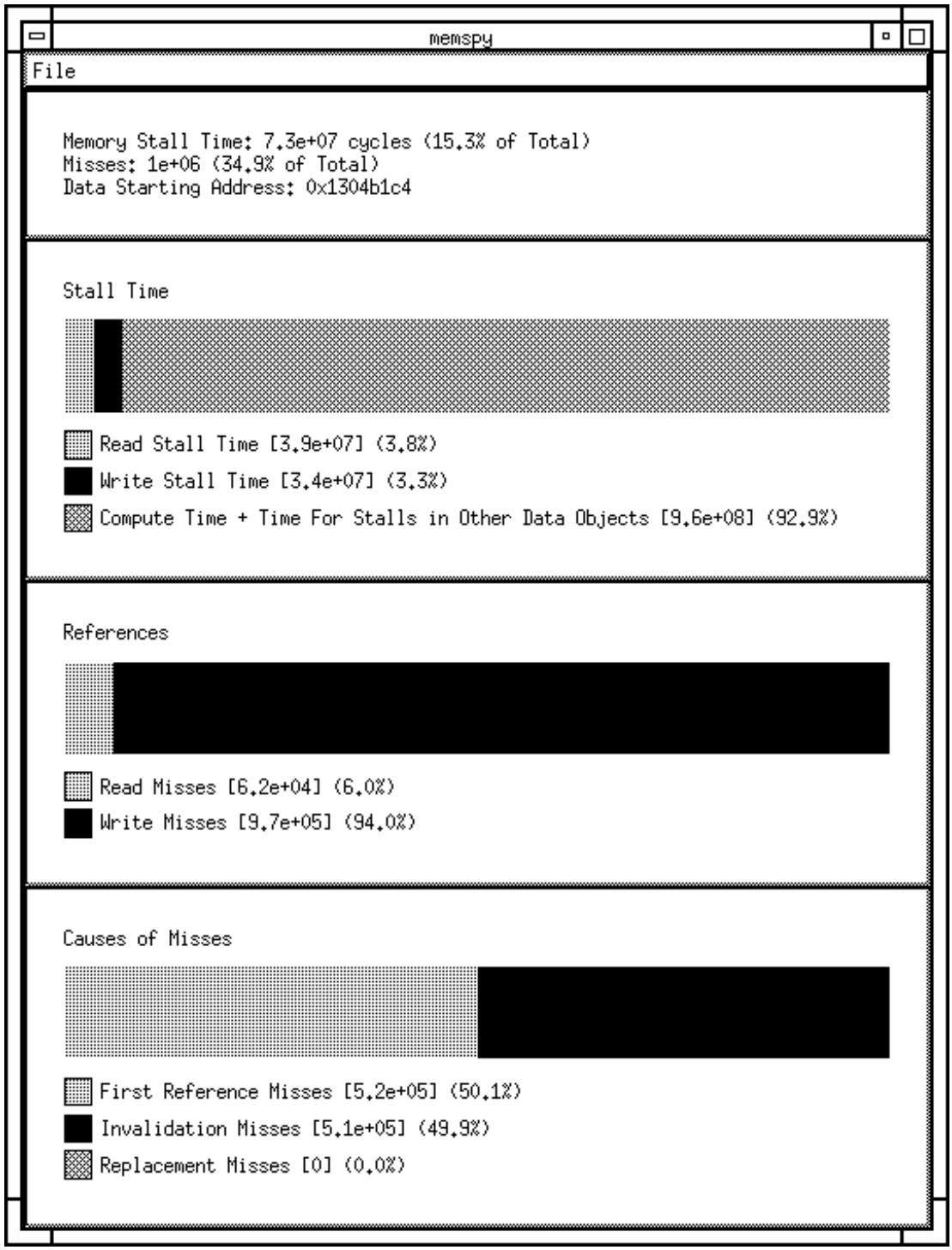

Figure A.13: Vrender: Detailed MemSpy output for `resample_buffervalid`.

At the same time, the programmers also noticed a significant amount of time spent in the `resampledopac` and `resampledcolor` data structures. In studying this, they noticed that the initialization of several shared data structures had been included within a parallel loop, rather than outside it. That is, *all* processors were initializing these arrays to 0, rather than just one. Since the correct value was placed in the arrays regardless, this is not a correctness bug per se. However, it is clearly an unintentional error which led to significant performance degradation.

Parallel Decomposition Following Tuning

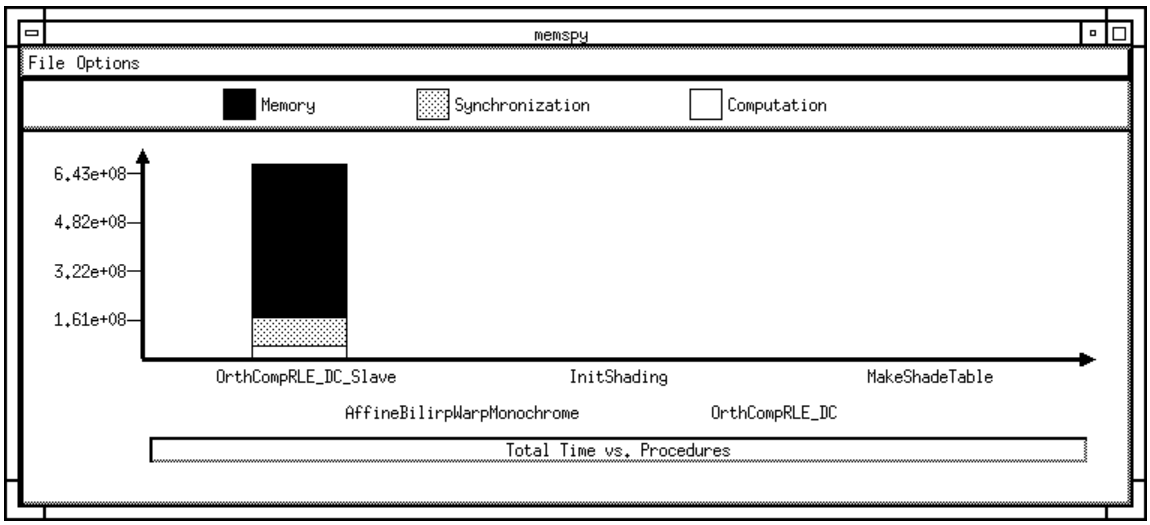

Figure A.14: Vrender: MemSpy output display after tuning step 1.

Figure A.14 shows the the new MemSpy output after the two major bugs from the initial implementation were fixed. At this point, the memory bottleneck has become less significant. In addition, removing the extraneous initialization code has improved performance from 1.65 seconds to 1.01 seconds. However, memory is still a major bottleneck. References to the input volume data are the main factor here, and are difficult to optimize away.

The work presented here is part of an ongoing study to improve the parallel performance of shear-warp volume rendering. Work on different parallel decompositions

is currently continuing. Future versions of Vrender could focus on (i) prefetching the volume data to reduce its contribution to memory stall time, and (ii) assigning volume data to local memories in order to reduce the stall times of cache misses when they do occur.

Appendix B

MemSpy User Interface

The MemSpy user interface is roughly 7000 lines of object oriented C++ code written using the InterViews toolkit [LVC89]. The InterViews toolkit, also in object oriented C++, offers graphical programming abstractions (such as scroll bars, buttons, and drawing primitives) built on top of X11 [SG86].

Using this toolkit, the MemSpy interface was built to provide to the user the series of histograms and displays shown in the examples of Chapter 3 and Appendix A. This appendix provides a complete description of each of the displays presented in MemSpy.

B.1 Initial Statistics

MemSpy initially provides two displays. The first is a per-procedure breakdown of the application time spent in computation, memory stalls, and synchronization. An example of this initial breakdown is shown in Figure B.1. This corresponds to the displays shown in Figures 3.3 and 3.10.

The second display is shown in Figure B.2. This display is a summary of overall memory statistics for the program. This is identical in form to the displays shown in Figures 3.5 and 3.12. The distinction here is that this initial display gives statistics for *all* references in the program, rather than for particular procedures or data structures. This is useful for getting an overall view of memory behavior, and for comparing overall results from multiple runs.

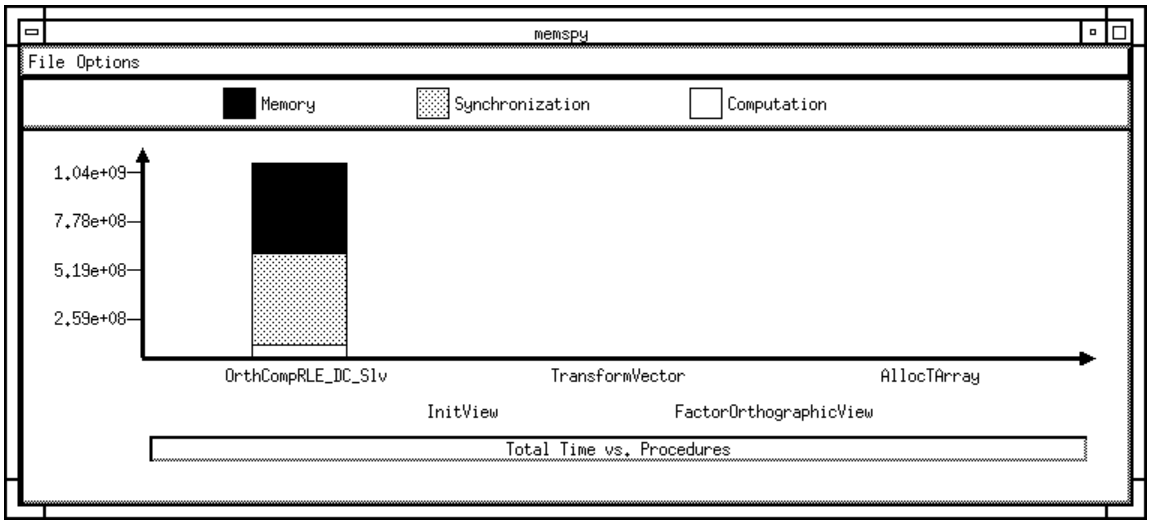

Figure B.1: Initial MemSpy output display.

From here, the user can follow one of two courses for obtaining more program statistics. The first option is to request the "Full Memory Stall Time Breakdown". This brings up the display shown in Figure B.3 and discussed in Section B.2. Second, to find out more about the memory behavior of a particular procedure, the user can use the mouse to click on the memory portion of the procedure's breakdown bar. This brings up the display shown in Figure B.4 and discussed in Section B.3.

B.2 Full Stall Time Breakdown

Selecting the "Full Stall Time Breakdown" button brings up the display shown in Figure B.3. This display gives memory stall time statistics in three different ways. The topmost display gives an ordering of memory stall time attributed to different procedure-data pairs. The second display orders memory stall times attributed to data bins in the code. The third display orders stall times by procedures. These different displays offer the user three orthogonal views of program behavior. As discussed in Section 2.2.3, these different views can give unique insights about program behavior. Ranking bottlenecks by data bins can be helpful when a particular data bin causes a bottleneck, but references to it

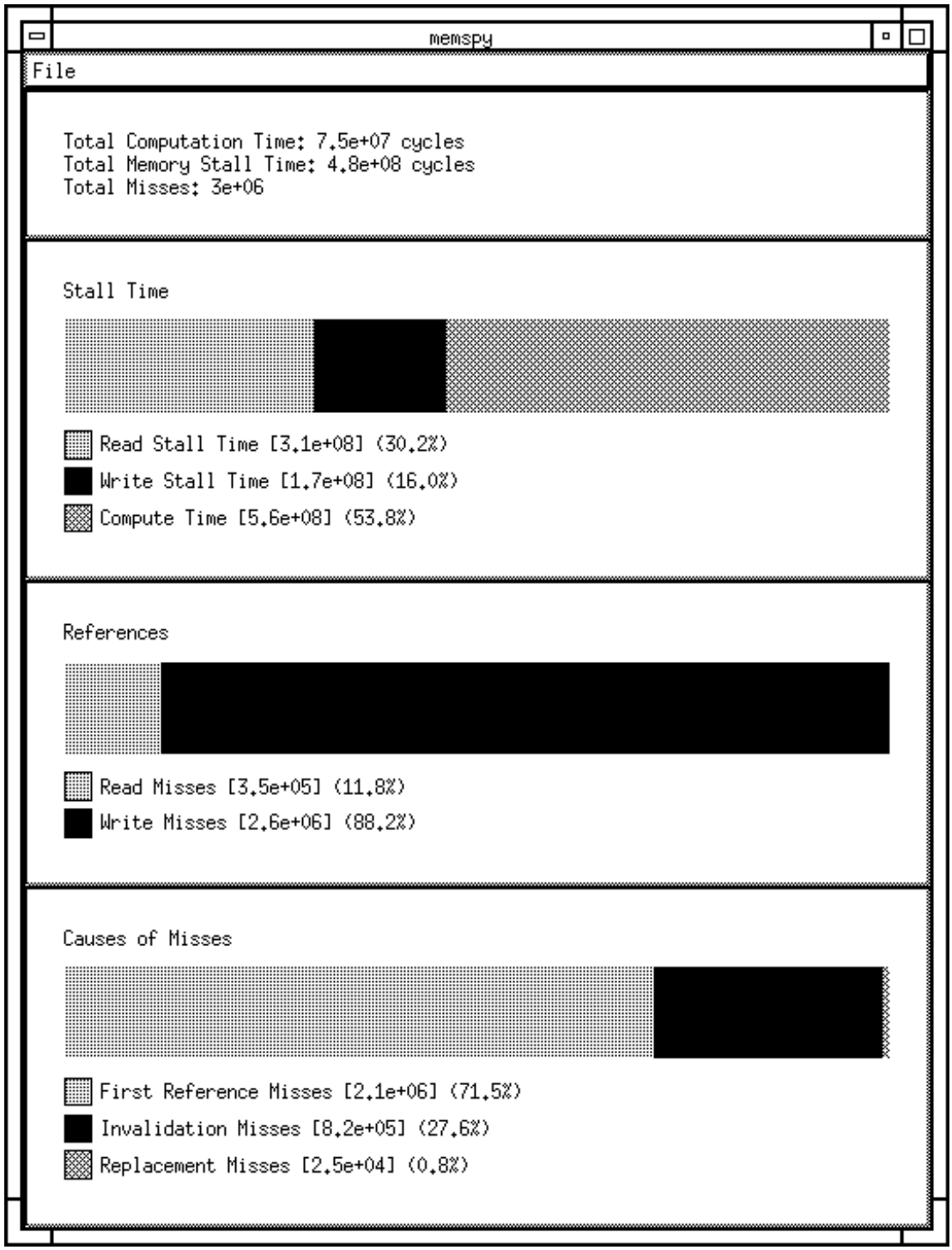

Figure B.2: Detailed display of overall statistics.

are spread over several procedures. Similarly, a per-procedure or per-data/per-procedure ranking may be more useful when bottlenecks are localized to a particular procedure. From here, the user can click on any of the individual stall time bars, and bring up detailed statistics for an individual bin. Section B.4 describes these.

B.3 Data Breakdowns

Clicking on the memory portions of any procedure's time breakdown in the MemSpy overview display (Figure B.1) brings up a data breakdown, as shown in Figure B.4. This display shows how the program's memory stall time is broken down among the different data bins accessed in this procedure. This breakdown has been very useful in localizing memory bottlenecks to particular data objects, within a particular procedure. From here the user can view more detailed information about a particular procedure-data bin by clicking on the memory bar for the bin of interest. This brings up the detailed display discussed in the next section.

B.4 Detailed Statistics Displays

One of the main thrusts of MemSpy was to provide detailed statistics for programmers on the frequency and causes of cache misses. Our focusing mechanisms, already described, work well to guide the user towards problems. At this point more detailed information about the cache misses for a particular bin can give important guidance about what is causing problems and how to fix them. MemSpy's detailed display is broken into sub-parts described below.

B.4.1 Read, Write Breakdowns

MemSpy gives a breakdown of how many of the misses occurred on read references versus how many of the misses occurred on write references. This can be useful for the programmer to understand the reference patterns around the memory bottleneck.

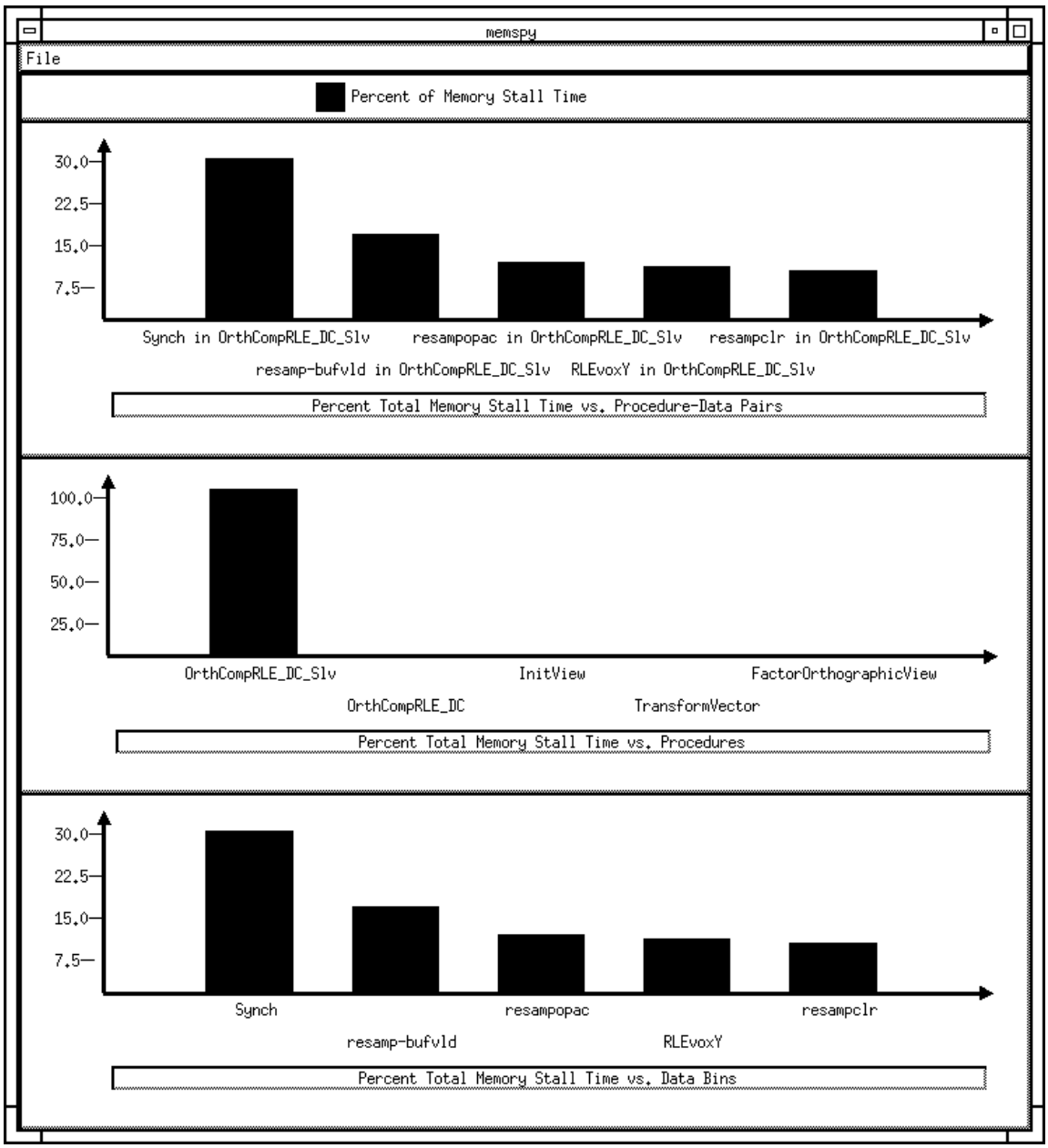

Figure B.3: Full stall time breakdown display.

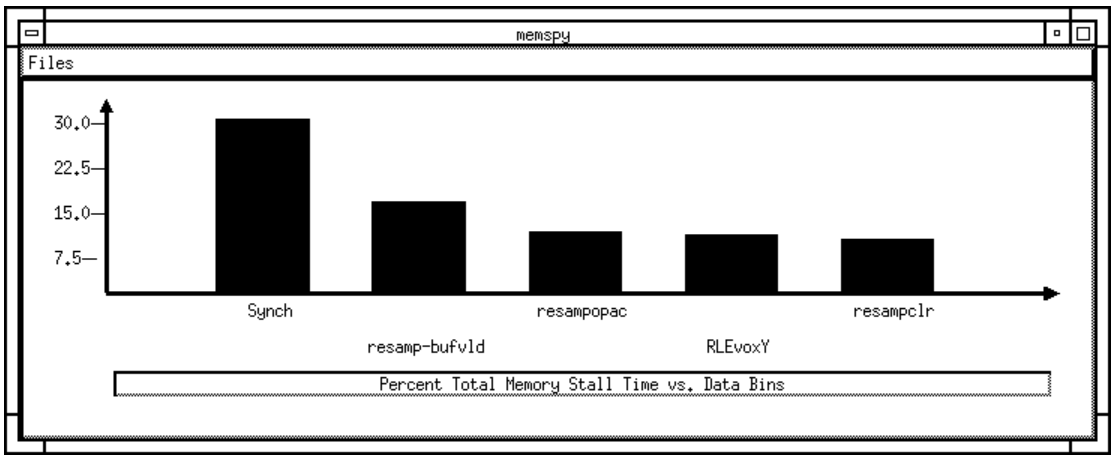

Figure B.4: Data breakdown display.

B.4.2 Causes of Cache misses

One of the most helpful statistics provided by MemSpy has been a breakdown of the causes of cache misses for each statistical bin. This is included as part of the detailed statistics box shown in Figure B.5. This breakdown indicates what fraction of the misses to this bin were (i) first-reference (cold) misses, (ii) invalidation misses, or (iii) replacement misses. These statistics help the user understand whether a memory bottleneck is due to interference (high replacement misses) or due to excessive sharing in parallel code (high invalidation misses). To further understand cases of interference, the user can click on the replacement portion of the bar. MemSpy will then provide statistics on the causes of replacements.

B.5 Causes of Replacements

As shown in Figure B.6, this display breaks down the causes of replacement misses for a particular statistical bin. It is used to show which data structures are primarily responsible for causing items to be pushed out of the cache, for the references that

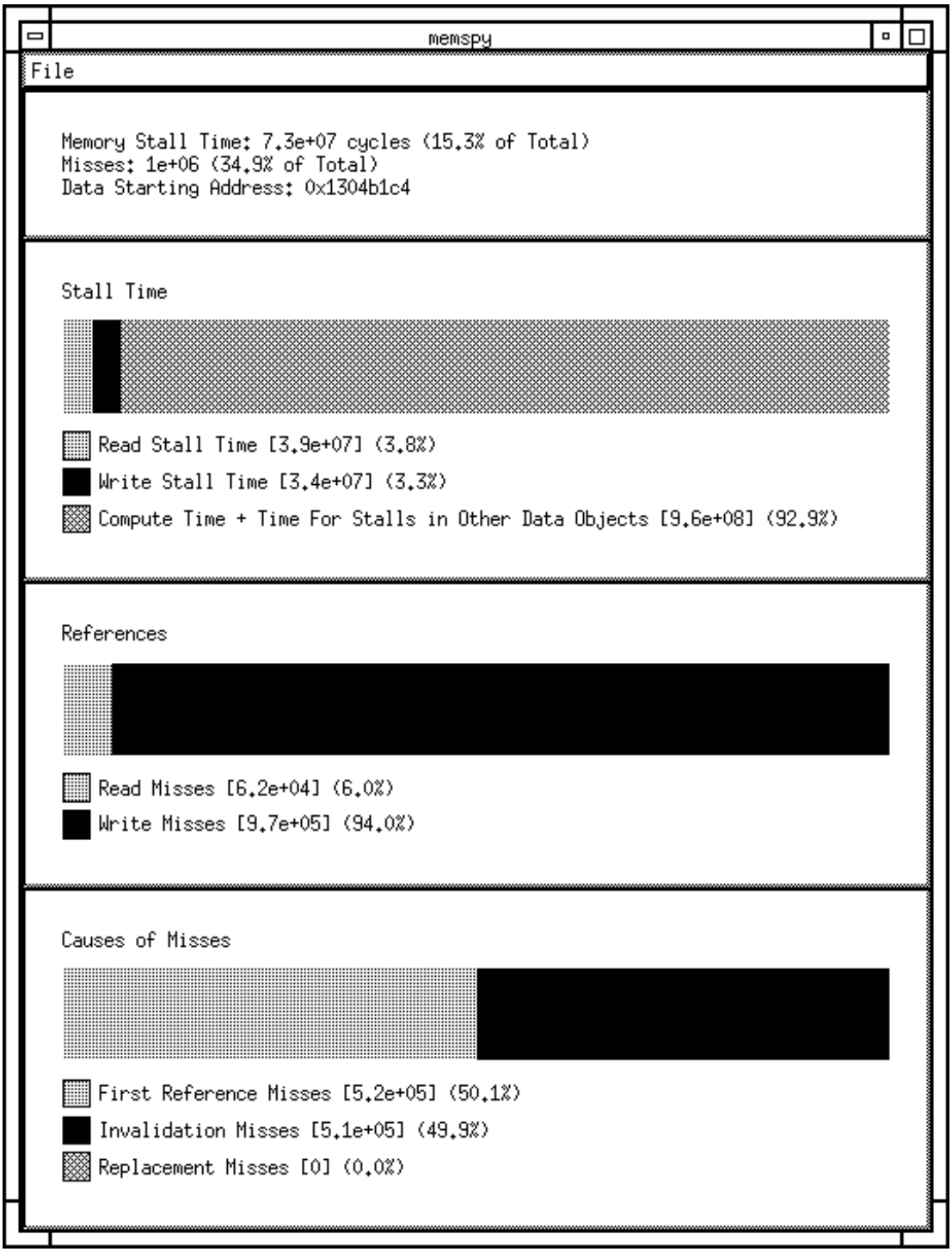

Figure B.5: Detailed display.

pertain to this statistical bin. This is useful for detecting instances of interference, in which data structures are competing for some or all of the cache space.

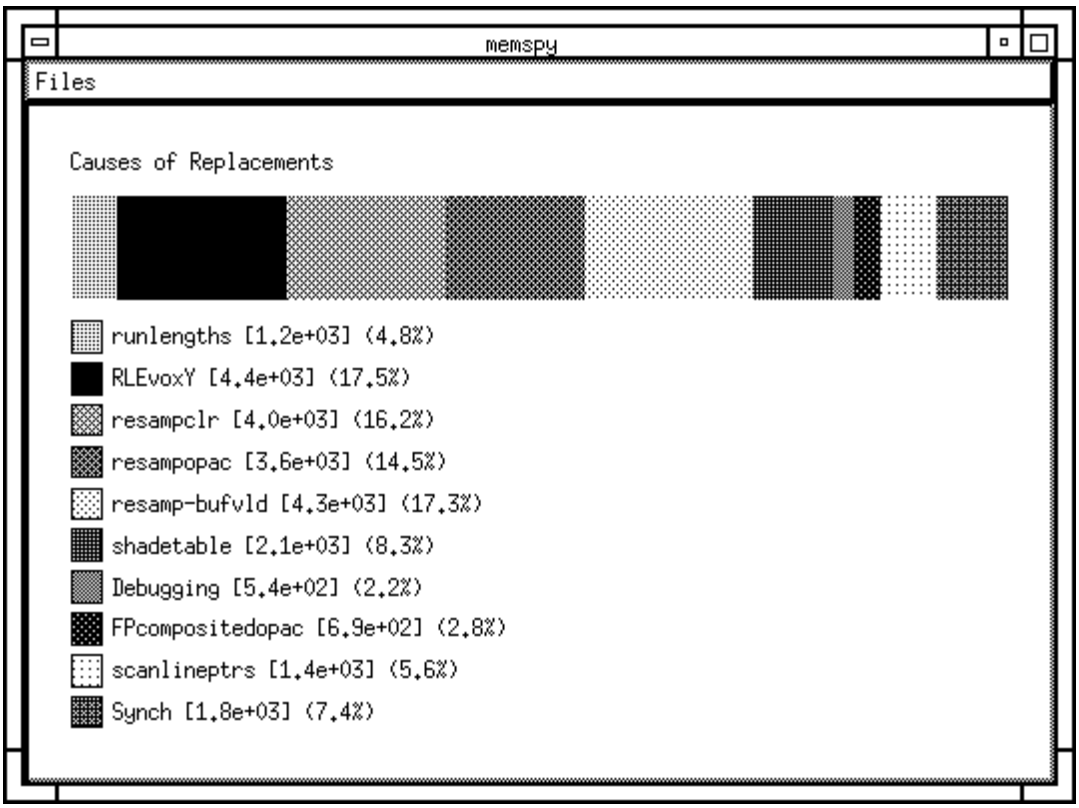

Figure B.6: Causes of replacements display.

Bibliography

[AG88] Ziya Aral and Ilya 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.

[Agr93] Maneesh Agrawala. Parallelizing the Shear Warp Volume Rendering Algorithm. CS390 Project Report, Stanford University, September 1993.

[AGS90] Ziya Aral, Ilya Gertner, and Greg Schaffer. Efficient Debugging Primitives for Multiprocessors. In *Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS III)*, pages 87–95, April 1990.

[AL90] Thomas E. Anderson and Edward D. Lazowska. Quartz: A Tool for Tuning Parallel Program Performance. In *Proc. ACM SIGMETRICS Conf. on the Measurement and Modeling of Computer Systems*, pages 115–125, May 1990.

[BM89] Helmar Burkhart and Roland Millen. Performance-Measurement Tools in a Multiprocessor Environment. *IEEE Transactions on Computers*, 38(5):725–737, May 1989.

[CB93] J. Bradley Chen and Brian N. Bershad. The Impact of Operating System Structure on Memory System Performance. In *Proc. Fourteenth Symposium on Operating System Principles*, November 1993.

[CD93] Jacqueline Chame and Michel Dubois. Cache Inclusion and Processor Sampling in Multiprocessor Simulations. In *Proc. ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems*, May 1993.

[Che93a] J. Bradley Chen. Overheads of Epoxie-based Monitoring. Personal Communication, 1993.

[Che93b] J. Bradley Chen. Software Methods for System Address Tracing. In *Proc. Fourth Workshop on Workstation Operating Systems*, October 1993.

[Coc53] W. G. Cochran. *Sampling Techniques*. John Wiley and sons, New York, NY, 1953.

[CWN92] Richard Comerford, George F. Watson, and Ray Ng. Special Report: Memory. *IEEE Spectrum*, 29(10):34–57, October 1992.

[DBKF90] Jack Dongarra, Orlie Brewer, James Arthur Kohl, and Samuel 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.

[DGL89] I. Duff, R. Grimes, and J. Lewis. Sparse Matrix Test Problems. *ACM Transactions on Mathematical Software*, 15:1–14, 1989.

[GGJ+89] V. A. Guarna, Dennis Gannon, David Jablonowski, Allen D. Malony, and Yogesh Gaur. Faust: An Integrated Environment for Parallel Programming. *IEEE Software*, pages 20 – 26, July 1989.

[GH90] Aaron Goldberg and John Hennessy. MTOOL: A Method for Detecting Memory Bottlenecks. Technical Report WRL-TN-17/90, DEC Western Research Laboratory, 1990.

[GH91a] Aaron J. Goldberg and John Hennessy. MTOOL: A Method for Isolating Memory Bottlenecks in Shared Memory Multiprocessor Programs. In *Proc. Intl. Conf. on Parallel Processing*, pages 251–257, August 1991.

[GH91b] Aaron J. Goldberg and John Hennessy. Performance Debugging Shared Memory Multiprocessor Programs with MTOOL. In *Proc. Supercomputing*, pages 481–490, November 1991.

[GKM83] Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick. An Execution Profiler for Modular Programs. *Software– Practice and Experience*, 13:671–685, August 1983.

[GNS+91] Dirk Grunwald, G. Nutt, A. Sloane, D. Wagner, W. Waite, and B. Zorn. Execution Architecture Independent Program Tracing. Technical Report CU-CS-525-91, University of Colorado, April 1991.

[Gol92] Aaron J. Goldberg. *Multiprocessor Performance Debugging and Memory Bottlenecks*. PhD thesis, Stanford University, May 1992. Also Computer Systems Laboratory Tech. Report CSL-TR-92-542.

[Gol93] Stephen R. Goldschmidt. *Simulation of Multiprocessors, Speed and Accuracy*. PhD thesis, Stanford University, June 1993.

[HJ91] John Hennessy and Norman Jouppi. Computer Technology and Architecture: An Evolving Interaction. *IEEE Computer*, pages 18 – 29, September 1991.

[HP90] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufman Publishers, Inc., San Mateo, California, 1990.

[KHW91] R. E. Kessler, Mark D. Hill, and David A. Wood. A Comparison of Trace-Sampling Techniques for Multi-Megabyte Caches. Technical Report 1048, Univ. of Wisconsin Computer Sciences Department, September 1991.

[Lar93] James R. Larus. Efficient Program Tracing. *IEEE Computer*, pages 52–61, May 1993.

[LL93] Philippe Lacroute and Marc Levoy. Fast Volume Rendering Using a Shear-Warp Factorization of the Viewing Transformation. Submitted for publication. Stanford University, December 1993.

[LLG+90] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. The Directory-Based Protocol for the DASH Multiprocessor. In *Proc. Seventeenth Annual International Conference on Computer Architecture*, May 1990.

[LLJ+93] Dan Lenoski, James Laudon, Truman Joe, David Nakahira, Luis Stevens, Anoop Gupta, and John L. Hennessy. The DASH Prototype: Logic Overhead and Performance. *IEEE Trans. on Parallel and Distributed Systems*, pages 41–61, January 1993.

[LO+87] Ewing Lusk, Ross Overbeek, et al. *Portable Programs for Parallel Processors*. Holt, Rinehart and Winston, Inc., 1987.

[LPI88] Subhasis Laha, Janak H. Patel, and Ravishankar K. Iyer. Accurate Low-Cost Methods for Performance Evaluation of Cache Memory Systems. *IEEE Trans. on Computers*, pages 1325–1336, November 1988.

[LRW91] Monica Lam, Edward Rothberg, and Michael Wolf. The Cache Performance and Optimizations of Blocked Algorithms. In *Proc. Fourth Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 63–74, April 1991.

[LSV+89] Ted Lehr, Zary Segall, Dalibor F. Vrsalovic, Eddie Caplan, Alan L. Chung, and Charles E. Fineman. Visualizing Performance Debugging. *IEEE Computer*, pages 38–51, October 1989.

[LVC89] Mark A. Linton, John M. Vlissides, and Paul R. Calder. Composing User Interfaces with InterViews. *IEEE Computer*, 22(2):8–22, Feb 1989.

[LW92] Alvin R. Lebeck and David A. Wood. Cache Profiling and the SPEC Benchmarks: A Case Study. Technical report, Univ. of Wisconsin Computer Sciences Department, March 1992.

[MCH+90] Barton P. Miller, Morgan Clark, Jeff Hollingsworth, Steven Kierstead, Sek-See Lim, and Timothy Torzewski. IPS-2: The Second Generation of a

Parallel Program Measurement System. *IEEE Transactions on Parallel and Distributed Systems*, 1(2), April 1990.

[MG89] Margaret Martonosi and Anoop Gupta. Tradeoffs in Message Passing and Shared Memory Implementations of a Standard Cell Router. In *Proc. 1989 International Conference on Parallel Processing*, pages 88–96, August 1989.

[MGA92] Margaret Martonosi, Anoop Gupta, and Thomas Anderson. MemSpy: Analyzing Memory System Bottlenecks in Programs. In *Proc. ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems*, pages 1–12, June 1992.

[MGA93] Margaret Martonosi, Anoop Gupta, and Thomas Anderson. Effectiveness of Trace Sampling for Performance Debugging Tools. In *Proc. ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems*, May 1993.

[Mil88] Barton P. Miller. DPM: A Measurement System for Distributed Programs. *IEEE Transactions on Computers*, 37(2):243–248, February 1988.

[MRA+89] Allen D. Malony, Daniel Reed, James W. Arendt, Ruth A. Aydt, Dominique Grabas, and Brian K. Totty. An Integrated Performance Data Collection, Analysis, and Visualization System. Technical Report TTR11, University of Illinois Department of Computer Science, March 1989.

[MRW92] A. D. Malony, D. A. Reed, and H. A. G. Wijshoff. Performance Measurement Intrusion and Perturbation Analysis. *IEEE Transactions on Parallel and Distributed Systems*, 3(4):433–450, July 1992.

[NSS+88] D. Notkin, L. Snyder, D. Socha, M. L. Bailey, B. Forstall, K. Gates, R. Greenlaw, W. G. Griswold, T. J. Holman, R. Korry, G. Lasswell, R. Mitchell, and P. A. Nelson. Experiences with Poker. In *Proc. ACM SIGPLAN Parallel Programming: Experience with Applications, Languages and Systems (PPEALS)*, July 1988.

[Rei90] M. H. Reilly. *A Performance Monitor for Parallel Programs*. Academic Press, 1990.

[RG92] Edward Rothberg and Anoop Gupta. Parallel ICCG on a Hierarchical Memory Multiprocessor— Addressing the Triangular Solve Bottleneck. *Parallel Computing*, 18(7):719–41, July 1992.

[SG86] Robert W. Scheifler and Jim Gettys. The X Window System. *ACM Transactions on Graphics*, 5(2):79–109, April 1986.

[SM93] Sekhar R. Sarukkai and Allen D. Malony. Perturbation Analysis of High Level Instrumentation for SPMD Programs. In *Proc. Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 44–53, May 1993.

[SMDO88] T. Sterling, A. Musciano, B. Donald, and R. Osborne. Multiprocessor Performance Measurement Using Embedded Instrumentation. In *Proc. International Conference on Parallel Processing*, August 1988.

[Smi91] Michael D. Smith. Tracing with Pixie. Technical Report CSL-TR-91-497, Stanford University, November 1991.

[SPE89] SPEC Benchmark Suite Release 1.0, October 1989.

[SR85] Zary Segall and Larry Rudolph. PIE: A Programming and Instrumentation Environment for Parallel Processing. *IEEE Software*, pages 22–37, November 1985.

[Sto90] H. S. Stone. *High-Performance Computer Architecture*. Addison-Wesley, Reading, MA, second edition, 1990.

[SWG92] Jaswinder Pal Singh, Wolf-Dietrich Weber, and Anoop Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. *Computer Architecture News*, 20(1):5–44, March 1992.

[Tha90] S. S. Thakkar. Performance of Parallel Applications on a Shared-Memory Multiprocessor System. In M. Simmons and R. Koskela, editors, *Performance Instrumentation and Visualization*, pages 235–258. Addison-Wesley, 1990.

[Tor92] Josep Torrellas. *Multiprocessor Cache Memory Performance: Characterization and Optimization*. PhD thesis, Stanford University, Aug 1992. Stanford CSL Tech. Report CSL-TR-92-545.

[Wal92] David W. Wall. Systems for Late Code Modification. In *Code Generation – Concepts, Tools, Techniques*, pages 275–293. Springer-Verlag, 1992.

[WHK91] David A. Wood, Mark D. Hill, and R. E. Kessler. A Model for Estimating Trace-Sample Miss Ratios. In *Proc. ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems*, pages 79–89, June 1991.

[ZH88] Benjamin Zorn and Paul N. Hilfinger. A Memory Allocation Profiler for C and Lisp. Technical Report UCB/CSD 88/404, University of California, Berkeley, February 1988.