SIP: Performance Tuning through Source Code Interdependence

Erik Berg and Erik Hagersten

Uppsala University, Information Technology, Deparment of Computer Systems
P.O. Box 337, SE-751 05 Uppsala, Sweden
{erikberg,eh}@docs.uu.se

Abstract. The gap between CPU peak performance and achieved application performance widens as CPU complexity, as well as the gap between CPU cycle time and DRAM access time, increases. While advanced compilers can perform many optimizations to better utilize the cache system, the application programmer is still required to do some of the optimizations needed for efficient execution. Therefore, profiling should be performed on optimized binary code and performance problems reported to the programmer in an intuitive way. Existing performance tools do not have adequate functionality to address these needs. Here we introduce *source interdependence profiling*, SIP, as a paradigm to collect and present performance data to the programmer. SIP identifies the performance problems that remain after the compiler optimization and gives intuitive hints at the source-code level as to how they can be avoided. Instead of just collecting information about the events directly caused by each source-code statement, SIP also presents data about events from some interdependent statements of source code.
A first SIP prototype tool has been implemented. It supports both C and Fortran programs. We describe how the tool was used to improve the performance of the SPEC CPU2000 183.equake application by 59 percent.

1 Introduction

The peak performance of modern microprocessors is increasing rapidly. Modern processors are able to execute two or more operations per cycle at a high rate. Unfortunately, many other system properties, such as DRAM access times and cache sizes, have not kept pace. Cache misses are becoming more and more expensive. Fortunately, compilers are getting more advanced and are today capable of doing many of the optimizations required by the programmer some years ago, such as blocking. Meanwhile, the software technology has matured, and good programming practices have been developed. Today, a programmer will most likely aim at, first, getting the correct functionality and good maintainability; then, profile to find out where in the code the time is spent; and, finally optimizing that fraction of the code. Still, many applications spend much of their execution time waiting for slow DRAMs.

B. Monien and R. Feldmann (Eds.): Euro-Par 2002, LNCS 2400, pp. 177–186.

Although compilers evolve, they sometimes fail to produce efficient code. Performance tuning and debugging are needed in order to identify where an application can be further optimized as well as how it should be done. Most existing profiling tools do not provide the information the programmer needs in a straightforward way. Often the programmer must have deep insights into the cache system and spend a lot of time interpreting the output to identify and solve possible problems.

Profiling tools are needed to explain the low-level effects of an application's cache behavior in the context of the high level language. This paper describes a new paradigm that gives straightforward aid to identify and remove performance bottlenecks. A prototype tool, implementing a subset of the paradigm, has proven itself useful to understand home-brewed applications at the Department of Scientific Computing at Uppsala University. In this paper we have chosen the SPEC CPU2000 183.equake benchmark as an example.

The paper is outlined as follows. Section 2 discusses the ideas behind the tool and general design considerations. Section 3 gives the application writer's view of our first prototype SIP implementation; Section 4 demonstrates how it is used for tuning of equake. Section 5 describes the tool implementation in more detail, section 6 compares SIP to other related tools, before the final conclusion.

2 SIP Design Considerations

The semantic gap between hardware and source code is a problem of application tuning. Code-centric profilers, which for example present the cache miss rate per source-code statement, reduces this gap, but the result can be difficult to interpret. We have no hints as to why the misses occurred. High cache miss ratios are often not due to one single source code statement, but depend on the way different statements interact, and how well they take advantage of the particular data layout used. Data-centric profilers instead collect information about the cache utilization for different data structures in the program. This can be useful to identify a poorly laid out, or misused, data structure. However, it provides little guidance to exactly where the code should be changed.

We propose using a profiler paradigm that presents data based on the interdependence between source code statements: *Source Interdependence Profiler*, SIP. SIP is both code-centric, in that statistics are mapped back on the source code, and data-centric, in that the collected statistics can be subdivided for each data structure accessed by a statement. The interdependence information for individual data structures accessed by a statement tells the programmer which data structures that may be restructured or accessed in a different way to improve performance.

The interdependence between different memory accesses can be either positive or negative. *Positive cache interdependence*, i.e., a previously executed statement has touched the same cache line, can cause a cache hit; *negative cache interdependence*, i.e., a more recent executed statement has touched a different cache line indexing to the same cache set and causing it to be replaced, may cause a cache miss. A statement may be interdependent with itself because of loop constructs or because it contains more than one access to the same memory location.

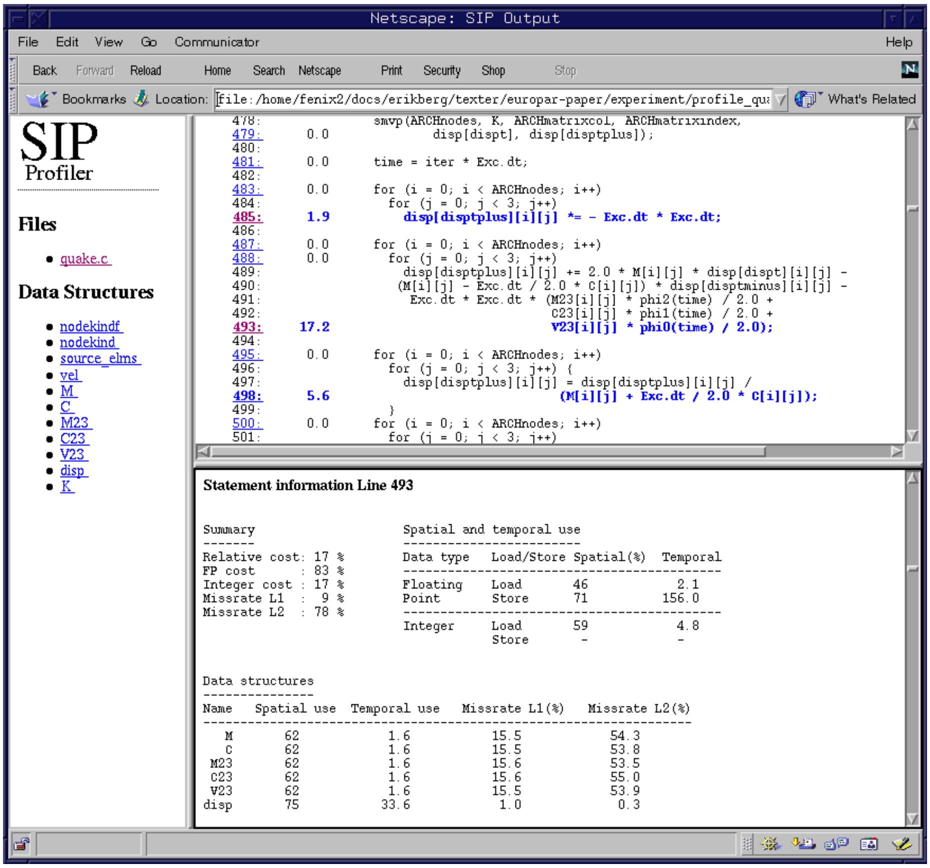
To further help the programmer, the positive cache interdependence collected during a cache line's tenure in the cache is subdivided into spatial and temporal locality. The spatial locality tells how large fraction of the cache line was used before eviction, while the temporal locality tells how many times each piece of data was used on average.

3 SIP Prototype Overview

The prototype implementation works in two phases. In the first phase, the studied application is run on the Simics [10] simulator. A cache simulator and a statistics collector is connected to the simulator. During the execution of the studied application, cache events are recorded and associated with load and store instructions in the binary executable. In the second phase, an analyzer summarizes the gathered information and correlates it to the studied source code. The output from the analyzer consists of a set of HTML files viewable by a standard browser. They contain the source code and the associated cache utilization.

Figure 1 shows a sample output from the tool. The browser shows three panes. To the left is an index pane where the source file names and the data structures are presented, the upper right pane shows the source code, and the lower right contains the results of the source-interdependence analysis. A click on a source file name in the index pane will show the content of the original source file with line numbers in the source pane. It will also show estimated relative execution costs to the left of the last line of every statement in the file. Source statements with high miss rates or execution times are colored and boldfaced. The source-interdependence analysis results for a statement can be viewed by clicking on the line number of the last line of the statement. It will show in the left lower pane in the following three tables:

- **Summary**
  A summary of the complete statement. It shows the estimated relative cost of the statement as a fraction of the total execution time of the application, the fraction of load/store cost caused by floating point and integer accesses, and miss rates for first- and second-level caches.
- **Spatial and Temporal Use**
  Spatial and temporal use is presented for integer and floating point loads and stores. The Spatial and Temporal use measures are chosen to be independent from each other to simplify the interpretation.
  - **Spatial use**
    Indicates how large fraction of the data brought into cache that is ever used. It is the percentage, on average, of the number of bytes allocated into cache by this statement that are ever used before evicted. This includes used by this same statement again, e.g. in next iteration of a loop, or used by another statement elsewhere in the program.


Netscape: SIP Output
File Edit View Go Communicator Help
Back Forward Reload Home Search Netscape Print Security Shop Stop
Bookmarks Location: file:/home/fenix2/docs/erikberg/texter/europar-paper/experiment/profile_qua
What's Related

SIP Profiler

Files

- quake.c

Data Structures

- nodekindf
- nodekind
- source_elms
- vel
- M
- C
- M23
- C23
- V23
- disp
- K

```
478:                smvp(ARCHnodes, K, ARCHmatrixcol, ARCHmatrixindex,
479:       0.0              disp[dispt], disp[disptplus]);
480:
481:       0.0          time = iter * Exc.dt;
482:
483:       0.0          for (i = 0; i < ARCHnodes; i++)
484:                      for (j = 0; j < 3; j++)
485:       1.9              disp[disptplus][i][j] *= - Exc.dt * Exc.dt;
486:
487:       0.0          for (i = 0; i < ARCHnodes; i++)
488:       0.0            for (j = 0; j < 3; j++)
489:                        disp[disptplus][i][j] += 2.0 * M[i][j] * disp[dispt][i][j] -
490:                       (M[i][j] - Exc.dt / 2.0 * C[i][j]) * disp[disptminus][i][j] -
491:                         Exc.dt * Exc.dt * (M23[i][j] * phi2(time) / 2.0 +
492:                                            C23[i][j] * phi1(time) / 2.0 +
493:      17.2                                  V23[i][j] * phi0(time) / 2.0);
494:
495:       0.0          for (i = 0; i < ARCHnodes; i++)
496:                      for (j = 0; j < 3; j++) {
497:                        disp[disptplus][i][j] = disp[disptplus][i][j] /
498:       5.6                                      (M[i][j] + Exc.dt / 2.0 * C[i][j]);
499:                      }
500:       0.0          for (i = 0; i < ARCHnodes; i++)
501:                      for (j = 0; j < 3; j++)
```

Statement information Line 493

```
Summary                       Spatial and temporal use
-------                       ------------------------
Relative cost: 17 %           Data type   Load/Store Spatial(%)  Temporal
FP cost      : 83 %           ----------------------------------------------
Integer cost : 17 %           Floating    Load       46             2.1
Missrate L1  :  9 %           Point       Store      71           156.0
Missrate L2  : 78 %           ----------------------------------------------
                              Integer     Load       59             4.8
                                          Store       -              -


Data structures
---------------
Name   Spatial use  Temporal use   Missrate L1(%)   Missrate L2(%)
-------------------------------------------------------------------
   M      62            1.6           15.5             54.3
   C      62            1.6           15.5             53.8
 M23      62            1.6           15.6             53.5
 C23      62            1.6           15.6             55.0
 V23      62            1.6           15.5             53.9
disp      75           33.6            1.0              0.3
```

Fig. 1. A screen dump from experiments with SPEC CPU2000 183.equake, 32 bit binary on UltraSPARCII. It shows the index pane (left), source pane (right) and profile information pane (bottom). It shows that the application exhibits poor spatial locality (46 percent) and temporal locality (2.1 times) for floating point loads.

  - **Temporal use**
    The average number of times data is reused during its tenure in the cache. First touch is not counted, i.e. a temporal use equal to zero indicates that none of the data is not touched more than once before it is evicted. Data that is never touched is disregarded, and therefore this measure does not depend on the spatial use.

– **Data Structures:**
  Miss ratios, spatial use and temporal use are presented for the individual data structures, or arrays, accessed by the statement.

This prototype SIP implementation does not implement the explicit pointers to other statements where data is reused, but only the implicit interdependence in spatial and temporal use. We anticipate that future enhancements of the tool will include the explicit interdependencies.

4 Case Study: SPEC 183.equake

A case study shows how SIP can be used to identify and help understanding of performance problems. We have chosen the 183.equake benchmark from the SPEC [15] CPU2000 suite. It is an earthquake-simulator written in C. First, SIP was used to identify the performance bottlenecks in the original[1] application and examine their characteristics. Figure 1 shows a screen dump of the result. The statement on lines 489-493 accounts for slightly more than 17 percent of the total execution time. Click on “493”, and the browser will show the statement information in the lower pane as in the figure. As can be seen under Summary, the cost of floating-point loads and stores is large. Miss rates are also large, especially in the Level 2 cache.

4.1 Identifying Spatial Locality Problems

The spatial use shows poor utilization of cached data. Floating-point loads show the worst behavior. As can be seen in the lower right pane under “Spatial and temporal use”, not more than 46 percent of the floating-point data fetched into cache by loads in this statement are ever used. Floating-point store and integer loads behave better, 71 and 59 percent respectively. The information about individual data structures, in bottom table of the same pane, points in the same direction. All but one, the array disp, have only 62 percent spatial use. When examining the code, the inner-most loop, beginning on line 488, corresponds to the last index of the data accesses on lines 489 - 492. This should result in good spatial behavior and contradicts the poor spatial percentage reported by the tool.

These results caused us to take a closer look at the memory layout. We found a problem in the memory allocation function. The data structure in the original code is a tree, where the leafs are vectors containing three doubles each. The memory allocation function does not allocate these vectors adjacent to each other, but leaves small gaps between them. Therefore not all of the data brought into the cache are ever used, causing the poor cache utilization.

A simple modification of the original memory-allocation function substantially increases performance. The new function allocates all leaf vectors adjacent to each other and the SIP tool shows that the spatial use of data improves. The speedups caused by the memory-allocation optimization are 43 percent on a 64-bit (execution time reduced from 1446s to 1008s) and 10 percent on a 32-bit executable. The probable reason of the much higher speedup on the 64-bit

[1] In the prototype, the main function must be instrumented with a start call to tell SIP that the application has started. Recognizable data structures must also be instrumented. For heap-allocated data structures, this can be done automatically.

binary is that the larger pointers cause larger gaps between the leafs in the original memory allocation. The SIP tool also revealed other code spots that benefit from this optimization. Therefore the speedup of the application is larger than the 17 percent execution cost of the statement on lines 489-493. A matrix-vector multiplication especially benefits by the above optimization. All speedup measurements were conducted with a Sun Forte version 6.1 C compiler and a Sun E450 server with 16KB level 1 data cache, 4MB unified level 2 cache and 4GB of memory, running SunOS 5.7. Both 64- and 32-bit executables were created with the *-fast* optimization flag. All speed gains were measured on real hardware.

4.2 Identifying Temporal Problems

The temporal use of data is also poor. For example, Figure 1 shows that floating-point data fetched into the cache from the statement are only reused 2.1 times on average. The code contains four other loop nests that access almost the same data structures as the loop nest on lines 487-493. They are all executed repeatedly in a sequence. Because the data have not been reused more, the working sets of the loops are too large to be contained in the cache. Code inspection reveals that loop merging is possible. Profiling an optimized version of the program with the loops merged shows that the data reuse is much improved. The total speedups with both this and the previous memory allocation optimizations are 59 percent on a 64-bit and 25 percent on a 32-bit executable.

5 Implementation Details

The prototype implementation of SIP is based on the Simics full-system simulator. Simics[10] simulates the hardware in enough detail to run an unmodified operating system and, on top of that, the application to be studied. This enables SIP to collect data non-intrusively and to take operating-system effects, such as memory-allocation and virtual memory system policies, into account. SIP is built as a module of the simulator, so large trace files are not needed. The tool can profile both Fortran and C code compiled with Sun Forte compilers and can handle highly optimized code. As described earlier, the tool works in two phases, the collecting phase and the analyzing phase.

5.1 SIP Collecting Phase

During the collecting phase, the studied application is run on Simics to collect cache behavior data. A memory-hierarchy simulator is connected to Simics. It simulates a multilevel data-cache hierarchy. The memory-hierarchy simulator can be configured for different cache parameters to reflect the characteristics of the computer, for which the studied application is to be optimized. The parameters are cache sizes, cache line sizes, access times, etc. The slowdown of the prototype tool's analyzing phase is around 450 times, mostly caused by the simulator, Simics.

The memory hierarchy reports every cache miss and evicted data to a statistics collector. Whenever some data is brought to a higher level of the cache hierarchy, the collector starts to record the studied application's use of it. When data are evicted from a cache, the recorded information is associated with the instruction that originally caused the data to be allocated into the cache. All except execution count and symbol reference are kept per cache level. The information stored for each load or store machine instruction includes the following:

- **Execution count**
  The total number of times the instruction is executed.
- **Cache misses**
  The total number of cache misses caused by the instruction.
- **Reuse count**
  The reuse count of one cache-line-sized piece of data is the number of times it is touched from the time it is allocated in the cache until it is evicted. Reuse count is the sum of the reuse counts of all cache-line-sized pieces of data allocated in the cache.
- **Total spatial use**
  The sum of the spatial use of all cache-line-sized pieces of data allocated in cache. The spatial use of one cache line-sized-piece of data is the number of different bytes that have been touched from the time it is allocated in cache until it is evicted.
- **Symbol reference**
  Each time a load or store instruction accesses memory, the address is compared to the address ranges of known data structures. The addresses of the data structures comes from instrumenting the source code. If a memory-access address matches any known data structure, a reference to that data structure is associated with the instruction PC. This enables the tool to relate caching information with specific data structures.

5.2 SIP Analyzing Phase

The analyzer uses the information from the statistics collector and produces the output. First, a mapping from machine instructions to source statements is built. This is done for every source file of the application. Second, for each source code statement, every machine instruction that is related to it is identified. Then, the detailed cache behavior information can be calculated for every source statement; and finally, the result is output as HTML files.

SIP uses compiler information to relate the profiling data to the original source code. To map each machine instruction to a source-code statement, the analyzer reads the debugging information [16] from the executable file and builds a translation table between machine-instruction addresses and source-code line numbers. The machine instructions are then grouped together per source statement. This is necessary since the compiler reorganizes many instructions from different source statements during optimization and the tool must know which load and store instructions that belongs to any source statement. The accurate machine-to-source-code mapping generated by Sun Forte C and F90 compilers makes this grouping possible. It can often be a problem to map optimized machine code to source code, but in this case it turned out to work quite well.

Derived measures are calculated at source-statement level. The information collected for individual machine instructions are summarized over their respective source-code statements, i.e. total spatial use for one statement is the sum of the total spatial uses of every load and store instruction that belongs to that statement. Reuse count is summarized analogous. To calculate the information that is presented in the table "Spatial and temporal use" in Figure 1, instructions are further subdivided into integer load, integer store, floating point load and floating point store for each source statement. For example, the total spatial use for floating-point load of one statement is the sum of the total spatial uses of every floating-point load instruction that belongs to that statement. The spatial use for a statement is calculated as:

$$Spatial\ use(\%) = 100 \cdot \frac{total\ spatial\ use\ of\ the\ statement}{\#cache\ misses\ of\ the\ statement \cdot cache\ line\ size}$$

Temporal use is calculated as:

$$Temporal\ use = \frac{reuse\ count\ of\ the\ statement}{total\ spatial\ use\ of\ the\ statement} - 1$$

The output is generated automatically in HTML format. It is easy to use and it does not need any specialized viewer. SIP creates two output files for each source file, one that contains the source code with line numbers, and one that contains the detailed cache information. It also produces a main file that sets up frames and links to the other files.

6 Related Work

Source-code interdependence can be investigated at different levels. Tools that simply map cache event counts to the source code do not give enough insights in how different parts of the code interact. Though useful, they fail to fully explain some performance problems. Cacheprof [14] is a tool that annotates source-code statements with the number of cache misses and the hit-and-miss ratios. It is based on assembly code instrumentation of all memory access instructions. For every memory access, a call to a cache simulator is inserted.

MemSpy [11] is based on the tango [6] simulator. For every reference to dynamically allocated data, the address is fed to a cache simulator. It can be used for both sequential and parallel applications. The result is presented at the procedure and data-structure level and indicates whether the misses were caused by communication or not. The FlashPoint tool [12] gathers similar information using the programmable cache-coherence controllers in the FLASH multiprocessor computer. CPROF [8] uses a binary executable editor to insert calls to a cache simulator for every load and store instruction. It annotates source code with cache-miss ratios divided into the categories of compulsory, conflict and capacity. It also gives similar information for data-structures. It does not investigate how different source statements relate to each other through data use, except for the implicit information given by the division into conflict and capacity. The full system simulator SimOS[9] has also been used to collect similar data and to optimize code. MTOOL[5] is a tool that compares estimated cycles due to pipeline stalls with measurements of actual performance. The difference is assumed to be due to cache miss stalls. Buck and Hollingsworth [2] present two methods for finding memory bottlenecks; counter overflow and n-way search based on the number of cache misses to different memory regions.

DCPI [1] is a method to get systemwide profiles. It collects information about such things as cache misses and pipeline stalls and maps this information to machine or source code. It uses the ProfileMe[3] hardware mechanism in the Alpha processor to accurately annotate machine instructions with different event counters, such as cache misses and pipeline stalls. The elaborate hardware support and sampling of nearby machine instructions can find dependencies between different machine instructions, but the emphasis is on detailed pipeline dependencies rather than memory-system interaction. SvPablo [4] is a graphical viewer for profiling information. Data can be collected from different hardware counters and mapped to source code. The information is collected by instrumenting source-code with calls to functions that read hardware counters and records there values. Summaries are produced for procedures and loop constructs.

MHSIM[7] is the tool that is most similar to SIP. It is based on source-code instrumentation of Fortran programs. A call to a memory-hierarchy simulator is inserted for every data access in the code. It gives spatial and temporal information at loop, statement and array-reference levels. It also gives conflict information between different arrays. The major difference is that it operates at source-code level and therefore gives no information as to whether the compiler managed to remove any performance problems. The temporal measure in MHSIM is also less elaborate. For each array reference, it counts the fraction of accesses that hit previously used data.

7 Conclusions and Future Work

We have found that source-code interdependence profiling is useful to optimize software. In a case study we have shown how the information collected by SIP, Source code Interdependence Profiling, can be used to substantially improve an application's performance. The mechanism to detect code interdependencies increases the understanding of an application's cache behavior. The comprehensive measures of spatial and temporal use presented in the paper also proved useful. It shows that further investigation should prove profitable.

Future work includes adding support to relate different pieces of code to each other through their use of data. Further, we intend to reduce the tool overhead by collecting the information by assembly code instrumentation and analysis.

We also plan to incorporate this tool into DSZOOM [13], a software distributed shared memory system.

References

1. J. Anderson, L. Berc, J. Dean, S. Ghemawat, M. Henzinger, S. Leung, D. Sites, M. Vandevoorde, C. Waldspurger, and W. Weihl. Continuous profiling: Where have all the cycles gone? *ACM Transactions on Computer Systems*, 1997.
2. B. Buck and J. Hollingsworth. Using hardware performance monitors to isolate memory bottlenecks. In *Proceedings of Supercomputing*, 2000.
3. J. Dean, J. Hicks, C. Waldspurger, W. Weihl, and G. Chrysos. ProfileMe: Hardware support for instruction-level profiling on out-of-order processors. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, 1997.
4. L. DeRose and D. Reed. Svpablo: A multi-language architecture-independent performance analysis system. In *10th International Conference on Performance Tools*, pages 352–355, 1999.
5. A. Goldberg and J. Hennessy. MTOOL: A method for isolating memory bottlenecks in shared memory multiprocessor programs. In *Proceedings of the International Conference on Parallel Processing*, pages 251–257, 1991.
6. S. Goldschmidt H. Davis and J. Hennessy. Tango: A multiprocessor simulation and tracing system. In *Proceedings of the International Conference on Parallel Processing*, 1991.
7. R. Fowler J. Mellor-Crummey and D. Whalley. Tools for application-oriented performance tuning. In *Proceedings of the 2001 ACM International Conference on Supercomputing*, 2001.
8. Alvin R. Lebeck and David A. Wood. Cache profiling and the SPEC benchmarks: A case study. *IEEE Computer*, 27(10):15–26, 1994.
9. S. Devine M. Rosenblum, E. Bugnion and S. Herrod. Using the simos machine simulator to study complex systems. *ACM Transactions on Modelling and Computer Simulation*, 7:78–103, 1997.
10. P. Magnusson, F. Larsson, A. Moestedt, B. Werner, F. Dahlgren, M. Karlsson, F. Lundholm, J. Nilsson, P. Stenström, and H. Grahn. SimICS/sun4m: A virtual workstation. In *Proceedings of the Usenix Annual Technical Conference*, pages 119–130, 1998.
11. M. Martonosi, A. Gupta, and T. Anderson. Memspy: Analyzing memory system bottlenecks in programs. In *ACM SIGMETRICS International Conference on Modeling of Computer Systems*, pages 1–12, 1992.
12. M. Martonosi, D. Ofelt, and M. Heinrich. Integrating performance monitoring and communication in parallel computers. In *Measurement and Modeling of Computer Systems*, pages 138–147, 1996.
13. Z. Radovic and E. Hagersten. Removing the overhead from software-based shared memory. In *Proceedings of Supercomputing 2001*, November 2001.
14. J. Seward. The cacheprof home page `http://www.cacheprof.org/`.
15. SPEC. Standard performance evaluation corporation `http://www.spec.org/`.
16. Sun. *Stabs Interface Manual, ver.4.0.* Sun Microsystems, Inc, Palo Alto, California, U.S.A., 1999.