Exploiting Spatial Store Locality through Permission Caching in Software DSMs

Håkan Zeffer, Zoran Radović, Oskar Grenholm, and Erik Hagersten

Uppsala University, Dept. of Information Technology,
P.O. Box 337, SE-751 05 Uppsala, Sweden
http://www.it.uu.se/research/group/uart

Abstract. Fine-grained software-based distributed shared memory (SW-DSM) systems typically maintain coherence with in-line checking code at load and store operations to shared memory. The instrumentation overhead of this added checking code can be severe. This paper (1) shows that most of the instrumentation overhead in the fine-grained SW-DSM system DSZOOM is store-related, (2) introduces a new write permission cache (WPC) technique that exploits spatial store locality and batches coherence actions at runtime, (3) evaluates WPC and (4) presents WPC results when implemented in a real SW-DSM system. On average, the WPC reduces the store instrumentation overhead in DSZOOM with 42 (67) percent for benchmarks compiled with maximum (minimum) compiler optimizations.

1 Introduction

The idea of implementing a shared address space in software across clusters of workstations, blades or servers was first proposed almost two decades ago [1]. The most common approach, often called shared virtual memory (SVM), uses the virtual memory system to maintain coherence. In this paper, we concentrate on fine-grain systems, which maintain coherence by instrumenting memory operations in the programs [2–4]. An advantage with these systems is that they avoid the high degree of false sharing, which is common in SVMs. Hence, they can implement stricter memory consistency models and run applications originally written for hardware-coherent multiprocessors. However, fine-grain systems suffer from relatively high instrumentation overhead [2–4]. Multiple schemes to reduce this overhead have been proposed. For example, Shasta [3] statically merges coherence actions at instrumentation time.

In this paper, we show that most of the instrumentation overhead in the sequentially consistent [5] fine-grained software DSM system, DSZOOM [4], comes from store instrumentation. We propose a dynamic *write permission cache* (WPC) technique that exploits spatial store locality. This technique dynamically merges coherence actions at runtime. We evaluate the proposed WPC technique in a parallel test bench. In addition, we present and evaluate two real WPC implementations as part of the DSZOOM system.

The WPC reduces DSZOOM's average store instrumentation overhead with 42 percent for SPLASH-2 benchmarks compiled with the highest optimization level and 67 percent for non-optimized applications. The parallel execution time for 16-processor runs for a 2-node configuration is reduced as much as 27 (32) percent and, on average, 7 (11) percent for benchmarks compiled with maximum (minimum) compiler optimization.

2 Write Permission Cache (WPC)

Blocking directory coherence protocols have been suggested to simplify the design and verification of hardware DSM (HW-DSM) systems [6]. DSZOOM's protocol is a distributed software-version of a blocking directory protocol. Only one thread at a time can have the exclusive right to produce global coherence activity at each piece of data. The blocking protocol avoids all corner cases of traditional coherence protocols. More protocol details can be found in [4, 7].

The applications studied have more loads than stores to shared memory. Yet, the store-related coherence checks stand for the largest part of the instrumentation overhead, as we will show in section 5.2. Most of this overhead comes from the fact that a locally cached directory entry (called "MTAG") must be locked before the write permission check is performed (blocking protocol).

The idea with the *write permission cache* (WPC) is to reduce locking and consulting/checking of MTAGs by exploiting spatial store locality. Instead of releasing the MTAG lock after a store is performed, a thread holds on to the write permission and the MTAG lock, hoping that the next store will be to the same coherence unit. If indeed the next store is to the same coherence unit, the store overhead is reduced to a few ALU operations and a conditional branch instruction. When a store to another coherence unit appears, a *WPC miss* occurs. Only then, a new lock release followed by a lock acquire must be performed.

The upper part of Figure 2 shows how an original store instruction expands into a *store snippet* when the DSZOOM system is used. `Ry` is a temporary register, `Rx` contains the value to be stored and `addr` is the effective address of this particular store operation. Lines 12 and 17 acquire and release the MTAG lock. Moreover, lines 13 and 14 load and check the MTAG value for permission. If the processor does not have write permission, the store protocol is called at line 15. Finally, at line 16, the original store is performed.

The lower part of Figure 2 shows how a WPC snippet is designed. The snippet consists of two parts: a fast- and a slow-path. Line 22 checks if the current coherence unit is the same as the one cached in the WPC (a WPC entry contains a coherence unit identifier, refered to as `CU_id[addr]` in Figure 2). If that is the case, then the processor has write permission and can continue its execution. The slow-path code is entered only if a WPC miss occurs. In that case, the processor actually checks for write permission in the MTAG structure. The slow path has much in common with the ordinary store snippet. However, one major difference is that the old lock, whose coherence unit identifier is cached in the WPC, has to be released (line 32). Moreover, at the end of the snippet,

```
01: original_store:
02:     ST Rx, addr;
03:
04:
05:
06:
07:

21: wpc_fast_path_snippet:
22:     if (WPC != CU_id[addr])
23:         call slow_path
24:     ST Rx, addr
25:
26:
27:
```

```
11: original_store_snippet:
12:     LOCK(MTAG_lock[addr]);
13:     LD Ry, MTAG_value[addr];
14:     if (Ry != WRITE_PERMISSION)
15:         call st_protocol;
16:     ST Rx, addr;
17:     UNLOCK(MTAG_lock[addr]);

31: wpc_slow_path_snippet:
32:     UNLOCK(MTAG_lock[WPC]);
33:     WPC = CU_id[addr];
34:     LOCK(MTAG_lock[addr]);
35:     LD Ry, MTAG_value[addr];
36:     if (Ry != WRITE_PERMISSION)
37:         call st_protocol;
```

Fig. 1. Original and WPC-based store snippets.

the processor keeps the lock. At line 33, the processor inserts the new coherence unit identifier in its WPC. Memory mappings are created in such a way that the `CU_id[addr]` reference at lines 22 and 33 easily can be done with arithmetic instructions, i.e., a shift. Thus, the fast path contains no extra memory references since thread-private registers are used as WPC entries. In other words, an n-entry WPC system with t threads contains $n \times t$ WPC entries in total.

3 Experimental Setup

The benchmarks that are used in this paper are well-known workloads from the SPLASH-2 benchmark suite [8]. Data set sizes for the applications studied can be found in [7]. The reason why we cannot run `volrend` is that shared variables are not correctly allocated with the *G_MALLOC* macro. Moreover, all experiments in this paper use GCC 3.3.3 and a simple custom-made assembler instrumentation tool for UltraSPARC targets. To simplify instrumentation, we use GCC's `-fno-delayed-branch` flag that avoids loads and stores in delay slots. We also use `-mno-app-regs` flag that reserves UltraSPARC's thread private registers for our snippets. These two flags slow down SPLASH-2 applications with less than 3 percent (avg.). Compiler optimization levels are `-O0` and `-O3`.

3.1 Hardware and DSZOOM Setup

All sequential and SMP experiments in this paper are measured on a Sun Enterprise E6000 server [9]. The server has 16 UltraSPARC II (250 MHz) processors and 4 Gbyte uniformly shared memory with an access time of 330 ns (*lmbench* latency [10]) and a total bandwidth of 2.7 Gbyte/s. Each processor has a 16 kbyte

on-chip instruction cache, a 16 kbyte on-chip data cache, and a 4 Mbyte second-level off-chip data cache.

The HW-DSM numbers have been measured on a 2-node Sun WildFire built from two E6000 nodes connected through a hardware-coherent interface with a raw bandwidth of 800 Mbyte/s in each direction [6, 11]. The WildFire system has been configured as a traditional cache-coherent, non-uniform memory access (CC-NUMA) architecture with its data migration capability activated while its coherent memory replication (CMR) has been kept inactive. The Sun WildFire access time to local memory is the same as above, 330 ns, while accessing data located in the other E6000 node takes about 1700 ns (lmbench latency). The E6000 and the WildFire DSM system are both running a slightly modified version of the Solaris 2.6 operating system.

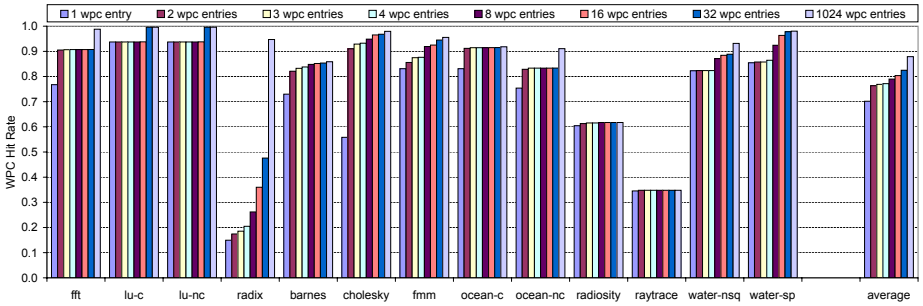
All DSZOOM implementations presented in this paper run in user space on the Sun WildFire system. The WildFire interconnect is used as a cluster interconnect between the two DSZOOM nodes. Non-cachable block load, block store and ordinary SPARC atomic memory operations (`ldstub`) are used as remote put, get and atomic operations. Each node accesses a "private copy" of the shared memory. The DSZOOM system maintains coherence between these private segments, i.e., the hardware coherence is not used. Moreover, the data migration and the CMR data replication of the WildFire interconnect are inactive when DSZOOM runs.

3.2 Test Bench Setup

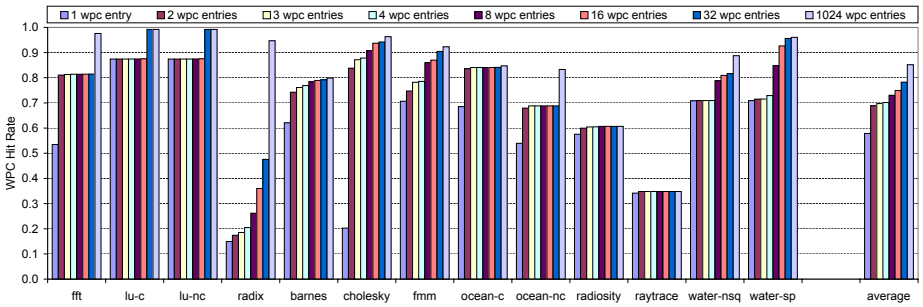
We have developed a parallel test bench environment to analyze new protocol optimizations such as the WPC. The test bench is called *protocol analyzer* (PA) and is designed for rapid prototyping and simulation of realistic workloads on parallel machines.

Our system has much in common with the Wisconsin Windtunnel II [12] simulator. It uses *direct execution* [13] and *parallel simulation* [12] to gain performance. The output from instrumented load and store operations of the studied benchmarks is used as input to PA. Moreover, an SMP (E6000) is used as host system during simulations. The SMP hardware guarantees coherence, memory consistency and correctness during the parallel execution of the program. PA implements a configuration system that makes it possible to model different memory systems. A PA model can simulate caches, cache coherence protocols and much more by using shared memory and simple counters. However, it is not possible to simulate target system's execution time.

Instrumentation overhead and calls to PA models can introduce timing errors or skewness in the simulation. It is important to consider these timing issues when analyzing data produced by the simulator.



(a) WPC hit rate for benchmarks compiled with optimization level `-O0`.



(b) WPC hit rate for benchmarks compiled with optimization level `-O3`.

Fig. 2. WPC hit rate for SPLASH-2 benchmarks compiled with `-O0` (a) and `-O3` (b) using a coherence unit size of 64 bytes.

4 Evaluating WPC

4.1 Simulated WPC Hit Rate

In this section, we investigate if it is possible to achieve high WPC hit rate with a few WPC entries. This is especially important for an efficient software WPC implementation. We use a multiple-entry WPC model to simulate WPC hit rate in PA. WPC hit rate is measured as hits in the WPC divided by the number of stores to shared memory. All data are collected during the parallel execution phase of the applications when run with 16 processors. Because each processor has its own set of WPC entries, and each processor simulates its own WPC hit rate, a timing skewing introduced by PA is not a problem. This is especially true for the applications that only uses synchronization primitives visible to the runtime system. Moreover, our simulated 1-entry WPC hit rate numbers have been verified with a slightly modified DSZOOM implementation. The numbers are almost identical (maximum difference is less than 0.03 percent).

Figure 2 shows hit rate for thirteen applications when 1, 2, 3, 4, 8, 16, 32 and 1024 WPC entries and a coherence unit size of 64 bytes is used. Figure 2 contains WPC hit rate for applications compiled with (a) minimum and (b) maximum optimization levels. Applications compiled with a low optimization level seem

to have higher WPC hit rates than fully optimized binaries. Still, almost all applications compiled with `-O3` have a WPC hit rate above 0.7. In particular, this is true when two or more WPC entries are used. If the number of WPC entries is increased from one to two, applications such as `barnes`, `cholesky` and `fft` significantly improve their hit rate numbers. This is due to multiple simultaneous write streams. Increasing the number of entries from two to three or from three to four does not give such a large WPC hit rate improvement. Thus, increasing the number of WPC entries above two might not be justified. `radix` and `raytrace` show poor WPC hit rate. WPC hit rate numbers for other coherence unit sizes (32-8192 bytes) and individual applications have been studied in a technical report [7].

4.2 WPC Impact on Directory Collisions

Data sharing, such as multiple simultaneous requests to a directory/MTAG entry, might lead to processor stall time in a blocking protocol. If a requesting processor fails to acquire a directory/MTAG lock, a *directory collision* occurs. We have simulated the DSZOOM protocol in PA to estimate what impact a WPC has on the number of directory collisions in DSZOOM. We run simulations with 1, 2, 3, 4, 8, 16, 32 and 1024 WPC entries, using 16 processors. Results show that the number of collisions does not increase when the number of WPC entries is small (less or equal to 32) [7]. For larger coherence unit sizes, the amount of directory collisions (as well as false sharing) might increase. For example, `lu-nc` performs poorly with a coherence unit size larger than 128 bytes.

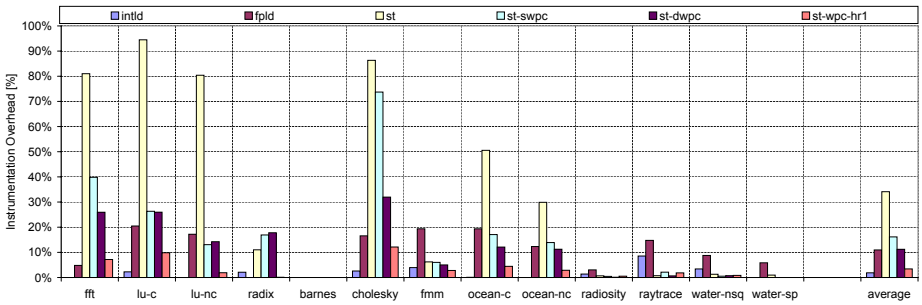
PA might introduce timing skewness during a simulation. However, because memory operations take longer time when PA is used, we believe that our collision numbers are unnecessarily negative, i.e., that the number of directory collisions will be even lower when run in DSZOOM than the simulation results currently show.

5 WPC Implementation and Performance

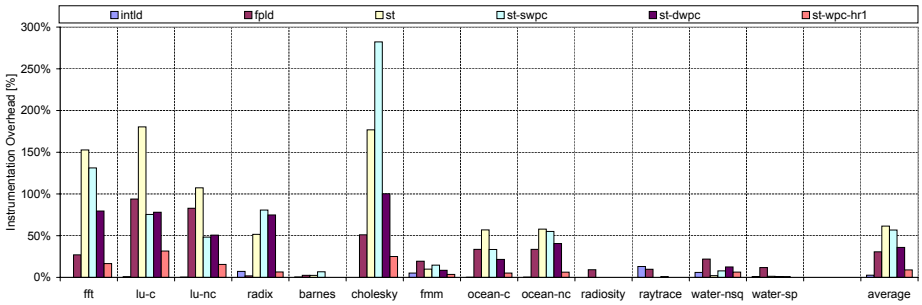
5.1 The DSZOOM-WPC Implementation

For an efficient DSZOOM-WPC implementation, it is necessary to reserve processor registers for WPC entries to avoid additional memory references in store snippets. With multiple WPC entries, the register pressure as well as the WPC checking code increases. As indicated in section 4.1, a 2-entry WPC may be a good design choice. Thus, in this paper, we implement and evaluate 1- and 2-entry WPC systems. We use SPARC's application registers as WPC entries.

The protocol of the base architecture maintains sequential consistency [5] by requiring all the acknowledges from the sharing nodes to be received before a global store request is granted. Introducing the WPC will not weaken the memory model. The WPC protocol still requires all the remotely shared copies to be destroyed before granting the write permission. WPC just extends the



(a) Sequential instrumentation overhead breakdown for non-optimized binaries.



(b) Sequential instrumentation overhead breakdown for fully optimized binaries.

Fig. 3. Sequential instrumentation overhead breakdown for integer loads (*intld*), floating-point loads (*fpld*), the original store snippet (*st*), a 1-entry WPC store snippet (*st-swpc*), a 2-entry WPC store snippet (*st-dwpc*) and a store snippet with WPC hit rate 1.0 (*st-wpc-hr1*).

duration of the permission tenure before the write permission is given up. Of course, if the memory model of each node is weaker than sequential consistency, it will decide the memory model of the system. Our system implements TSO since E6000 nodes are used.

The WPC technique also raises dead- and livelock concerns. Most of the dead- and livelock issues are solved by the DSZOOM runtime system. A processor's WPC entries have to be released at (1) synchronization points, at (2) failures to acquire directory/MTAG entries and at (3) thread termination. However, user-level flag synchronization can still introduce WPC related deadlocks. The WPC deadlock problem and three suggested solutions are discussed in [7]. In this study, applications that use flag synchronization (`barnes` and `fmm`) are manually modified with WPC release code.

5.2 Instrumentation Overhead

In this section, we characterize the overhead of inserted fine-grain access control checks for all of the studied SPLASH-2 programs. The write permission checking

code (store snippets) is the focus of this section since the WPC technology is a store optimization technique. To obtain a sequential instrumentation breakdown for different snippets, we ran the applications with just one processor and with only one kind of memory instruction instrumented at a time. This way, the code will never need to perform any coherency work and will therefore never enter the protocol code (written in C).

Sequential instrumentation overhead breakdown for the benchmarks is shown in Figure 3. The store overhead is the single largest source of the total instrumentation overhead: 61 (34) percent for optimized (non-optimized) code. The single-WPC checking code (*st-swpc*) reduces this overhead to 57 (16) percent. Double-WPC checking code (*st-dwpc*) further reduces the store overhead to 36 (11) percent. As expected, the reduction is most significant for `lu-c` and `lu-nc` because they have the highest WPC hit rate, see Figure 2, and low shared load/store ratio [8]. `fft` and `cholesky` perform much better when a 2-entry WPC is used. For `radix`, the instrumentation overhead slightly increases for the *st-swpc* and *st-dwpc* implementations. The low WPC hit rate (see Figure 2) is directly reflected in this particular instrumentation breakdown.

Finally, the "perfect" WPC checking code (*st-wpc-hr1*) (a single-WPC snippet modified to always hit in the WPC) demonstrates very low instrumentation overheads: 9 percent for optimized and 3 percent for non-optimized code.

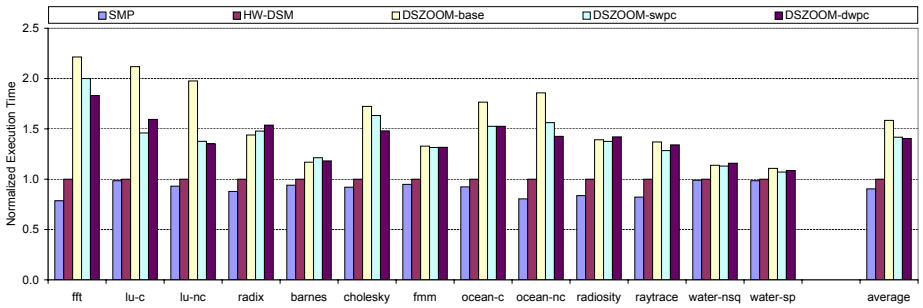
To summarize, we have seen that applications with low load/store ratio tend to have high store related instrumentation overhead. This store related overhead could be significantly reduced if the application has a high WPC hit rate and one or two WPC entries are used. On the other hand, if an application does not have a high WPC hit rate nor a low load/store ratio the ordinary store snippet might be a better alternative.

5.3 Parallel Performance

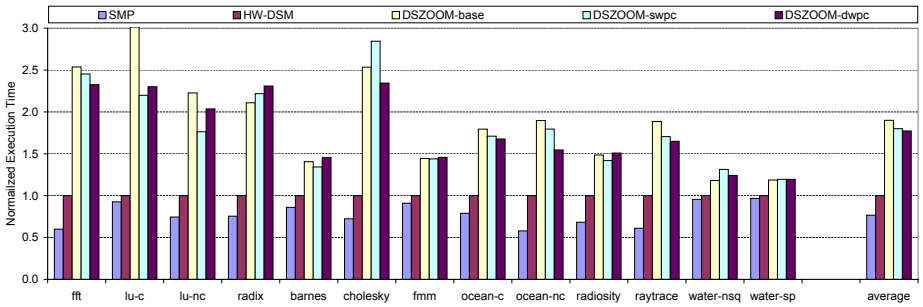
In this section, the parallel performance of two WPC-based DSZOOM systems is studied. Figure 4 shows normalized execution time for Sun Enterprise E6000 (SMP), 2-node Sun WildFire (HW-DSM) and three DSZOOM configurations:

1. DSZOOM-base: the original DSZOOM implementation.
2. DSZOOM-swpc: the DSZOOM implementation with a 1-entry WPC.
3. DSZOOM-dwpc: the DSZOOM implementation with a 2-entry WPC.

All DSZOOM configurations use a coherence unit size of 64 bytes. Both the HW-DSM configuration and the DSZOOM configurations run on two nodes and with eight processors per node (16 in total). The SMP configuration run on a single E6000 node with 16 processors and is used as an upper bound. The WPC technique improves the parallel DSZOOM performance with 7 (11) percent for benchmarks compiled with maximum (minimum) compiler optimization levels. The performance gap between the hardware-based DSM and the DSZOOM system is reduced with 14 (31) percent. Thus, the DSZOOM slowdown is in the range of 77 (40) percent compared to an expensive hardware implementation of shared memory, both running optimized (non-optimized) applications.



(a) 16-processor runs and non-optimized binaries.



(b) 16-processor runs and fully optimized binaries.

Fig. 4. Parallel performance for 16-processor configurations.

6 Related Work

The Check-In/Check-Out (CICO) cooperative shared memory implementation presented by Hill et. al. [14] uses similar ideas as the WPC technique. CICO is a programming model where a programmer can reason about access time to memory and give simple hardware coherence protocol performance hints. A *check out* annotation marks the expected first use and a *check in* annotation terminates the expected use of the data. Whereas CICO annotations are inserted as hints, a WPC entry actually "checks out" write permission since the directory/MTAG lock is not released until the next synchronization point or the next WPC miss.

Shasta [3] uses *batching* of miss checks, that is a "static merge" of coherence actions at instrumentation time. For a sequence of shared loads and stores, that touches the same coherence unit, the Shasta system combines/eliminates some of the access control checks (if possible). This way, all of the loads and stores in this sequence can proceed with only one check. The current WPC implementation works as a dynamic version of Shasta's batching technique.

7 Conclusions

In this paper, we introduce and evaluate a new *write permission cache* (WPC) technique that exploits spatial store locality. We demonstrate that the instrumentation overhead of the fine-grained software DSM system, DSZOOM [4], can be reduced with both 1- and 2-entry WPC implementations. On average, the original store instrumentation overhead, the single largest source of the total intrumentation cost, is reduced with 42 (67) percent for highly optimized (non-optimized) code. The parallel performance of the DSZOOM system for 16-processor runs (2-node configuration) of SPLASH-2 benchmarks is increased by 7 (11) percent. We believe that instrumentation-time batching (Shasta's approach [3]) of coherence actions combined with our new WPC technique might improve performance even further.

References

1. Li, K.: Shared Virtual Memory on Loosely Coupled Multiprocessors. PhD thesis, Department of Computer Science, Yale University (1986)
2. Schoinas, I., Falsafi, B., Lebeck, A.R., Reinhardt, S.K., Larus, J.R., Wood, D.A.: Fine-grain Access Control for Distributed Shared Memory. In: ASPLOS-VI. (1994)
3. Scales, D.J., Gharachorloo, K., Thekkath, C.A.: Shasta: A Low-Overhead Software-Only Approach to Fine-Grain Shared Memory. In: ASPLOS-VII. (1996) 174–185
4. Radović, Z., Hagersten, E.: Removing the Overhead from Software-Based Shared Memory. In: Proceedings of Supercomputing 2001, Denver, Colorado, USA (2001)
5. Lamport, L.: How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. IEEE Transactions on Computers **C-28** (1979) 690–691
6. Hagersten, E., Koster, M.: WildFire: A Scalable Path for SMPs. In: HPCA-5. (1999) 172–181
7. Zeffer, H., Radović, Z., Grenholm, O., Hagersten, E.: Evaluation, Implementation and Performance of Write Permission Caching in the DSZOOM System. Technical Report 2004-005, Dept. of Information Technology, Uppsala University (2004)
8. Woo, S.C., Ohara, M., Torrie, E., Singh, J.P., Gupta, A.: The SPLASH-2 Programs: Characterization and Methodological Considerations. In: ISCA-22. (1995) 24–36
9. Singhal, A., Broniarczyk, D., Cerauskis, F., Price, J., Yuan, L., Cheng, C., Doblar, D., Fosth, S., Agarwal, N., Harvey, K., Hagersten, E., Liencres, B.: Gigaplane: A High Performance Bus for Large SMPs. In: IEEE Hot Interconnects IV. (1996)
10. McVoy, L.W., Staelin, C.: lmbench: Portable Tools for Performance Analysis. In: Proceedings of the 1996 USENIX Annual Technical Conference. (1996) 279–294
11. Hennessy, J.L., Patterson, D.A.: Computer Architecture: A Quantitative Approach. 3rd edn. Morgan Kaufmann (2003)
12. Mukherjee, S.S., Reinhardt, S.K., Falsafi, B., Litzkow, M., Huss-Lederman, S., Hill, M.D., Larus, J.R., Wood, D.A.: Fast and Portable Parallel Architecture Simulators: Wisconsin Wind Tunnel II. IEEE Concurrency (2000)
13. Covington, R.C., Madala, S., Mehta, V., Jump, J.R., Sinclair, J.B.: The Rice Parallel Processing Testbed. In: ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, ACM Press (1988)
14. Hill, M.D., Larus, J.R., Reinhardt, S.K., Wood, D.A.: Cooperative Shared Memory: Software and Hardware for Scalable Multiprocessor. In: ASPLOS-V. (1992)