Flexibility Implies Performance

Håkan Zeffer, Zoran Radović, and Erik Hagersten
Department of Information Technology, Uppsala University
P.O. Box 337, SE-751 05, Uppsala, Sweden
{hakan.zeffer, zoran.radovic, erik.hagersten}@it.uu.se

Abstract

No single coherence strategy suits all applications well. Many promising adaptive protocols and coherence predictors, capable of dynamically modifying the coherence strategy, have been suggested over the years.

While most dynamic detection schemes rely on plentiful of dedicated hardware, the customization technique suggested in this paper requires no extra hardware support for its per-application coherence strategy. Instead, each application is profiled using a low-overhead profiling tool. The appropriate coherence flag *setting, suggested by the profiling, is specified when the application is launched.*

We have compared the performance of a hardware DSM (Sun WildFire) to a software DSM built with identical interconnect hardware and coherence strategy. With no support for flexibility, the software DSM runs on average 45 percent slower than the hardware DSM on the 12 studied applications, while the flexibility can get the software DSM within 11 percent. Our all-software system outperforms the hardware DSM on four applications.

1 Introduction

While hardware-based shared-memory systems have been successfully built for many years, the cost in terms of design and verification for each new generation is ever increasing. Meanwhile, the advance in semiconductor technology have set the shared-memory server trend towards multiple cores per die (CMP) and multiple threads per core (SMT) [21]. For example, next generation CMPs promise to include as much as 32 hardware threads per chip [20]. We believe that this technology shift forces a reevaluation of the way to interconnect multiple such chips to form larger systems.

This paper presents a highly flexible all-software shared-memory proposal with a very low system design cost and short time-to-market. Even though this system could be used across a large application domain, we believe that it is especially well suited for system designs targeting the high-performance computing (HPC) market. Partly because the recent trend towards cluster-based HPC computing, and partly because the data access pattern regularity, often displayed by scientific codes, can be exploited by tailor made coherence schemes accustomed to each applications.

In this paper, we extend the DSZOOM system [29] with several novel optimization options and add a low-overhead profiling mode that suggests appropriate *coherence flags* for the 12 applications studied. When compared with a hardware DSM system built from identical node hardware, interconnect and coherence strategy, the base system is trailing by 45 percent on average (slowdown-factor range is 0.98–3.02) for these 12 applications. The profiled coherence flags brought the numbers down to 17 percent on average (0.69–1.85) and hand-tuned coherence flags down to 11 percent (0.69–1.66). It was a bit surprising, but very encouraging, to note that the profile-based coherence flags propelled the software DSM to outperform the hardware DSM for four of the applications.

The technology presented in this paper can easily be incorporated in many parallel execution environments, such as OpenMP [7] or UPC compilers [2], providing a low cost but high performance execution platform for HPC applications. That way, the low-overhead profiling could be integrated in the compiler infrastructure and more advanced optimizations based on compiler analysis be implemented in the system.

The next section gives an overview of the basic DSZOOM system. Section 3 presents new coherence and bandwidth optimizations. A detailed performance evaluation is presented in Section 4, while Section 5 describes the low-overhead profiling tool. Detailed implementation issues are addressed in Section 6. Finally, we present related work and conclude in Sections 7 and 8.

2 Basic DSZOOM System

This section gives an overview of the basic DSZOOM system [29], a sequentially consistent [23] software-based

DSM implementation that is inspired by three fine-grained software coherence proposals: Blizzard-S [37], Shasta [33, 30, 31, 32], and Sirocco-S [35]. DSZOOM relies on code instrumentation to maintain fine-grain coherence (load and store operations to shared memory are augmented with coherence checks). The provided protocols assume a high bandwidth, low latency cluster interconnect, supporting fast user level mechanisms for *put*, *get*, and *atomic* operations to remote nodes' memories, such as InfiniBand [18] or Sun Fire Link [39]. The system further assumes that the write order between any two endpoints in the network is preserved. These network assumptions make is possible to remove interrupt- and/or poll-based asynchronous protocol processing found in the majority of software DSM implementations [29, 1]. A processor that has detected the need for global coherence activity will first acquire a lock associated with the coherence unit before starting the coherence activity. A requesting processor can independently lock a remote directory entry and obtain read/write permissions.

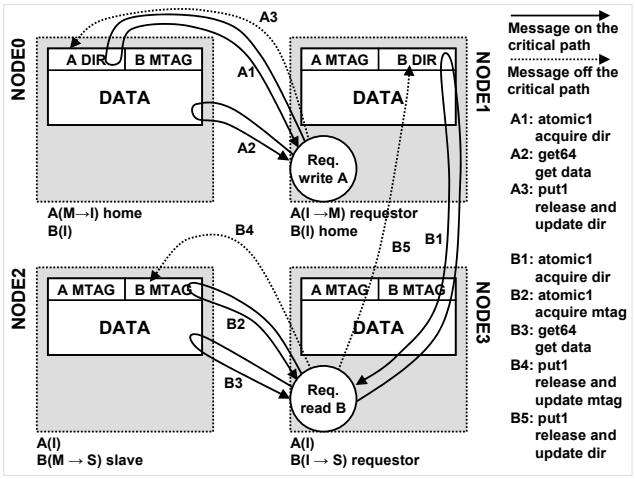


Figure 1. 2-hop write miss and 3-hop read miss examples for the invalidation-based protocol.

2.1 The Invalidation-Based Protocol

The invalidation-based protocol states, modified, shared and invalid (MSI), are explicitly represented by global data structures in the nodes' memories. Bits of a memory operation's effective address determine the location of a coherence unit's directory location, i.e., its "home node." All coherence units in invalid state store a "magic" data value, as independently suggested by Scales et al [33] and Chiou et al [6] (Schoinas et al [36] use the same technique in the Blizzard-S system). This significantly reduces the number of directory accesses caused by load operations, since the directory only has to be consulted on a read miss.[1]

To reduce the number of accesses to remote directory entries caused by global store operations, each node has one byte of local state (MTAG) per global coherence unit (similar to Shasta's *private state table* [32]), indicating if the coherence unit is locally writable. Before each global store operation, the MTAG byte is checked. The directory only has to be consulted if the MTAG indicates that the node currently does not have write permission to the coherence unit. The directory will assume the role of MTAG in home nodes, and hence, no extra MTAG state is needed for home nodes. To avoid race conditions, the corresponding MTAG entry has to be locked before a write permission check is carried out. Otherwise, a coherence unit can be downgraded between the consultation and the point in time where the store is performed.

Figure 1 illustrates the protocol activity caused by a 2-hop write miss (transactions A1-A3) and a 3-hop read miss (transactions B1-B5). The state transitions for coherence units *A* and *B* can be found below each node. *atomic1*, *get64* and *put1* correspond to a 1 byte remote atomic operation, a remote 64 byte get and a remote 1 byte put.

2-hop Write Miss Example: The requestor in node1, *reqA*, checks its local MTAG for write permission of coherence unit *A*. Since node1 does not has write permission, the store protocol is called and the coherence activity is started. *reqA* acquires exclusive access to *A*'s directory located in node0's memory (A1). When the directory is locked, the data is retrieved from the home node with a remote *get64* operation (A2). To end the coherence activity, *reqA* releases and updates the directory with a single remote *put1* operation (A3), which is off the critical path.

3-hop Read Miss Example: The requestor in node3, *reqB*, tries to read coherence unit *B*. However, the load returns the "magic" value and the load protocol is called. *reqB* locks the directory entry and determines the identity of the node holding the data (B1). The data happens to reside in node2, in a modified state. A second remote atomic operation (B2) to node2's MTAG structure disables write permission on that node. The data is fetched with a remote *get64* operation (B3). Node2's MTAG and the home's directory are then released and updated. These two *put1* operations (B4 and B5) are also off the critical path.

2.2 Write Permission Cache

DSZOOM's access control checks for stores represent the largest part of the total instrumentation cost [45]. Most of this overhead comes from the fact that the locally cached directory entry (MTAG) must be checked atomically for each global store operation. This section describes *write permission cache* (WPC) that hides some of the instrumen-

[1]The directory also has to be consulted in the rare case when the real data value is equal to the magic value [36, 33].

```
01: original_store:
02:   ST Rx, addr

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

21: wpc_fast_path_snippet:
22:   if (WPC != CU_id[addr])
23:       call wpc_slow_path_snippet
24:   ST Rx, addr

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

Figure 2. Original and WPC store snippets.

tation cost for stores [45]. While Shasta reduces its instrumentation overhead by statically merging coherence actions at instrumentation time [33] (*batching*), a WPC dynamically merges store coherence checks at runtime. 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. The identity of the coherence unit is stored in a dedicated register, which is consulted before the next store is performed. (DSZOOM reserves UltraSPARC's application registers [41] for fast WPC checks.) 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.

Lines 01 to 17 of Figure 2 show how an original store instruction expands into a *store snippet*. `Ry` is a temporary register, `Rx` contains the value to be stored and `addr` is the effective address of this particular store operation. (`CU_id[addr]` refers to `addr`'s coherence unit identifier.) Lines 12 and 17 acquire and release the MTAG lock. Lines 13 and 14 load and check the MTAG value for permission. If the processor does not has write permission, the store protocol is called at line 15. Finally, at line 16, the original store is performed.

Lines 21 to 37 of Figure 2 show a WPC snippet. The snippet consists of a fast- and a slow-path. The slow-path snippet is called when a WPC miss occurs. The lock of the currently cached coherence unit identifier is then released (line 32). Lines 34, 35 and 36, lock, load and check the MTAG for permission. Again, if the processor does not has write permission, the store protocol is called. Note that the lock is kept at the end of the WPC snippet. Holding on to the MTAG lock raises WPC related deadlock issues. We address these in Section 6.

Figure 3 shows that the WPC hit rate for SPLASH-2 benchmarks [42] varies greatly depending on the application, the number of WPC entries and the coherence unit size. (GCC 3.3.4, optimization level 3, 16-processor runs.) For example, two entries demonstrate much better hit rate, which is most significant for `fft` and `ocean` applications.

3 Extending DSZOOM's Flexibility

One of the key observations of this paper is that the WPC technology can be used as an efficient software-based *store buffer* in an update-based system. To be more specific, multiple stores could be merged before the MTAG lock is released and the data is distributed to other nodes.

In this section, we extend DSZOOM's flexibility with a new update-based protocol based on store-buffer filtering that outperforms the base protocol for some applications, typically when the number of read misses is high. In a fine-grained software DSM system, the gain is two-fold because only store operations must be instrumented. Section 3.2 presents additional bandwidth reduction techniques.

3.1 The Update-Based Protocol

The update-based protocol is based on write permission. All nodes have read permission to all data whereas only one has read-write permission for each coherence unit. The states read-write (W) and read (R or !W) are explicitly represented in the nodes' memories. Remote directory traffic is tamed with an update version of the MTAG optimization described in Section 2.1. Also the update protocol is race-free, i.e., the corresponding MTAG entry has to be locked before a write permission check is carried out. Moreover, to guarantee data integrity, data must be distributed to other nodes before a MTAG is released.

3-hop Write Miss Example: Figure 4 shows coherence activity caused by an update 3-hop write miss. *atomic1* and *put1* correspond to a 1 byte remote atomic operation and a remote 1 byte put. The state transitions for coherence unit *D* can be found below the nodes. The requestor, *reqD*, first checks its local MTAG for write permission of coherence unit *D*. Since node2 does not has write permission, the store protocol is called and the coherence activity is started. *reqD* locks the directory located at the home node (`D1`). The directory indicates that the write permission is located on node0. Hence, `D2` locks node0's MTAG. This removes write permission on that node. `D3` and `D4` updates the MTAG (!W) and the directory (node2's id) respectively.

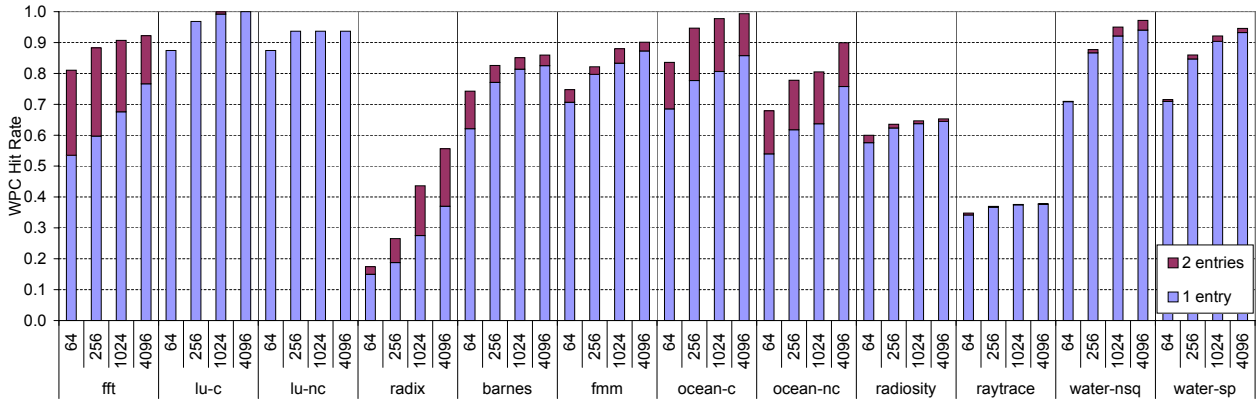


Figure 3. WPC hit rate for different applications and coherence unit sizes (64-4096 bytes).

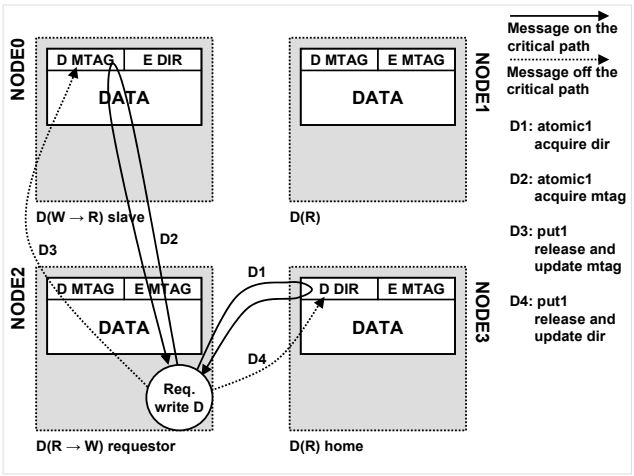


Figure 4. 3-hop write miss example for the update-based protocol.

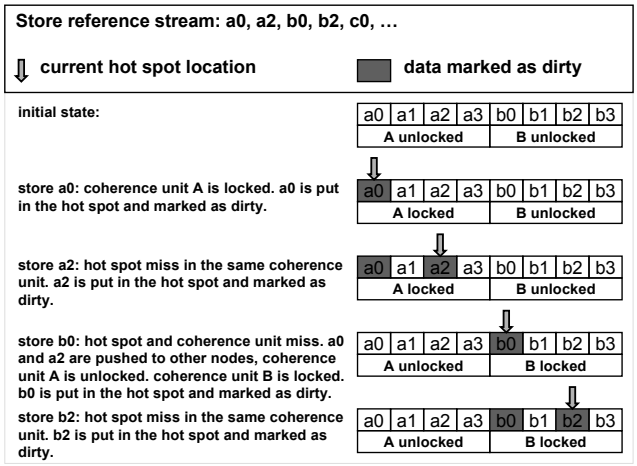


Figure 5. Dirty-data WPC example.

Node2 is now in state read-write and has correct data (data has to be distributed before a lock is released, the network order is preserved).

3.2 Bandwidth Reduction Techniques

Update-based coherence protocols have to deal with the potential bandwidth problem introduced by excessive data pushing. In this section, we present two mechanisms (filtering strategies) that address the bandwidth problem: *dirty-data* and *private-data* filtering.

3.2.1 Dirty-Data Filtering

Scaling the coherence unit size has two potential benefits: (1) a large coherence unit size can reduce the number of coherence misses, and (2) the WPC hit rate is improved. Hence, the number of locks taken and the instrumentation overhead are reduced. However, update-based coherence protocols can be very sensitive to coherence unit size scaling. A large coherence unit wastes bandwidth when exposed to write-write false sharing or if processors only write parts of the coherence unit before distributing the data. We address this problem with a *dirty-data* WPC that tracks modifications of a current cache line. Hence, only modifications (dirty data) are distributed to other nodes. A coherence unit is divided into smaller parts. The lock is obtained per coherence unit whereas the WPC points to one part of the coherence unit, the "hot spot." Data are marked dirty when the hot spot is moved.

Figure 5 shows how the dirty-data WPC handles a stream of store references. The coherence unit size is 512 bytes and the "hot spot" size is 64 bytes. When the store to `a0` appears, the snippet code locks coherence unit *A*. `a0` is put

in the hot spot and marked as dirty. The next store to `a2` misses in the hot spot, but is to the same coherence unit. The hot spot is moved to `a2`, which is marked as dirty. The store to `b0` triggers both a hot spot and a coherence-unit miss. `a0` and `a2` are pushed to the other nodes and coherence unit *A* is unlocked. Coherence unit *B* is locked and `b0` is both put in the hot spot and marked dirty. The store to `b2` is handled just as the one to `a2`.

3.2.2 Private-Data Filtering

Benchmarks often show a large amount of stores that are to node-private data. For some applications, this is true even at page granularity. For example, 89 (77) percent of the global stores in `lu-c` (`ocean-c`) are to private pages. We exploit this application property to reduce the global update bandwidth consumed with a *private-data* filter. The virtual-memory system is used to keep track of private-to-shared state changes and the page state is used to omit updates to node-private pages.

Our private-data filter is implemented with a page-permission check from a locally-cached page directory before each global update. In addition, at read/write page faults to the shared memory segment, our signal handler updates the page directory with new permission information through a remote-atomic and a remote-put operation and uses `mprotect(2)` to set up local memory mappings. If the node does not has data, a page fetch might be needed. While multiple schemes are possible, our system only allows page-permission upgrades.

4 Performance Evaluation

Table 1 shows data set sizes for all of the SPLASH-2 applications studied [42]. The reason why we cannot run `volrend` is that shared variables are not correctly allocated with the *G_MALLOC* macro. `cholesky` is not run because we were not able to find large enough working sets.

4.1 Compiler and Instrumentation Software

All experiments in this paper use the GCC 3.3.4 compiler. To simplify instrumentation, we use GCC's `-fno-delayed-branch` flag that avoids loads and stores in delay slots, and `-mno-app-regs` that reserves UltraSPARC's thread-private registers [41] for our snippets. These two flags slow down SPLASH-2 applications with less than 3 percent (avg.). Note that only the DSZOOM system uses those flags. All benchmarks are compiled with optimization level 3.

We extend DSZOOM's instrumentation tool with a simplified version of Shasta's batching technique [33, 45]. The tool implements a *read-modify-write* batching, which merges load and store coherence checks (to the same effective address) by replacing the load check with the store's WPC check. We also schedule application instructions into coherence snippets to increase instruction-level parallelism (inspired by EEL [24]).

Program	Large (Small) Problem Size
`fft`	4M (64k) points
`lu-c`	2048×2048 (512×512) matrices, 16×16 blocks
`lu-nc`	2048×2048 (512×512) matrices, 16×16 blocks
`radix`	32M (2M) integers, radix 1024
`barnes`	128k (16k) particles
`fmm`	128k (32k) particles
`ocean-c`	1026×1026 (258×258)
`ocean-nc`	1026×1026 (258×258)
`radiosity`	largeroom (room), -ae 5000.0 -en 0.050 -bf 0.10
`raytrace`	car (teapot)
`water-nsq`	4913 (2197) molecules, 2 time steps
`water-sp`	32768 (2197) molecules, 2 time steps

Table 1. SPLASH-2 benchmarks. The small working set is used together with the profiling mode described in Section 5. All performance results are based on the large data set sizes.

4.2 Hardware Setup

Most of the experiments are measured on a Sun Enterprise E6000 server [38]. The server has 16 UltraSPARC II (250 MHz) processors and 4 Gbyte uniformly shared memory with an access time of 330 ns (*lmbench* latency [26]) 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 sequential experiments run on two processor types: a 250 MHz UltraSPARC II (USII) and a 900 MHz UltraSPARC III (USIII). The USIII processor has a 32 kbyte instruction cache, a 64 kbyte data cache, a 2 kbyte write cache and a 2 kbyte prefetch cache. The second-level cache is 8 Mbyte and off-chip.

The hardware DSM results have been measured on a 2-node Sun WildFire system built from two E6000 nodes connected through a hardware-coherent interface with a raw bandwidth of 800 Mbyte/s in each direction [15, 16]. 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 disabled. 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). WildFire runs the Solaris 2.6 operating system.

All software DSM implementations run in user space on

Program	Time [s] USII (III)	inv USII (III)	inv-swpc USII (III)	inv-dwpc USII (III)
fft	*17.4 (9.7)*	2.67 (2.89)	2.18 (2.08)	1.65 (1.52)
lu-c	*132.1 (42.9)*	3.56 (5.83)	1.48 (1.79)	1.51 (1.84)
lu-nc	*270.4 (87.4)*	2.15 (3.30)	1.53 (1.45)	1.60 (1.49)
radix	*57.7 (22.8)*	1.52 (1.75)	1.70 (1.79)	1.66 (1.69)
barnes	*161.0 (52.5)*	1.09 (1.15)	1.07 (1.13)	1.10 (1.13)
fmm	*155.0 (48.0)*	1.15 (1.28)	1.10 (1.17)	1.10 (1.18)
ocean-c	*84.5 (56.5)*	1.71 (1.72)	1.49 (1.30)	1.33 (1.18)
ocean-nc	*132.2 (91.4)*	1.45 (1.42)	1.38 (1.26)	1.31 (1.19)
radiosity	*39.8 (14.7)*	1.08 (1.19)	1.11 (1.18)	1.11 (1.18)
raytrace	*80.5 (23.9)*	1.24 (1.36)	1.24 (1.36)	1.24 (1.35)
water-nsq	*162.8 (68.2)*	1.18 (1.28)	1.16 (1.27)	1.12 (1.27)
water-sp	*115.7 (48.3)*	1.16 (1.28)	1.15 (1.23)	1.17 (1.24)
Avg.		1.66 (2.04)	1.38 (1.42)	1.33 (1.35)

Table 2. Sequential instrumentation overhead for 250 MHz UltraSPARC II and 900 MHz UltraSPARC III processor runs.

the Sun WildFire system. The WildFire interconnect is in that case used as a "non-coherent" cluster interconnect between E6000 nodes. Non-cacheable block load, block store and regular SPARC atomic memory operations (`ldstub`) are used as remote put, get and atomic operations.

4.3 Instrumentation Overhead

Table 2 shows sequential-execution time in seconds for non-instrumented programs (second column). It also reports the factor increase in execution time when both load and store instrumentation is inserted for three invalidation-based configurations: the invalidation-based protocol without WPC (`inv`), the invalidation-based protocol with a 1-entry WPC (`inv-swpc`) and the invalidation-based protocol with a 2-entry WPC (`inv-dwpc`) (see Table 3 for abbreviations). All experiments run on both USII and USIII processors with a coherence unit size of 512 bytes. On average, instrumentation overhead for the slower processor (USII) is lowered from 66 percent for the `inv` protocol to 33 percent when a 2-entry WPC is used (`inv-dwpc`). For the faster processor, this reduction is even more significant (from 104 percent to 35 percent). The store instrumentation overhead for WPC implementations can be reduced even further if larger coherence unit sizes are used. For example, the store instrumentation overhead for `fft` is reduced from 178 to 27 percent for the USIII target when a 2-entry WPC is added and the coherence unit is scaled from 64 to 8192 bytes. Note that the instrumentation techniques based on WPC (such as `inv-swpc` and `inv-dwpc`) only add ALU instructions to the fast path of the execution when instrumenting loads and stores. This results in an instrumentation overhead which is fairly independent of processor technology, as can be seen in Table 2. Traditional instrumentation techniques (such as `inv`), that also add memory operations for store instrumentation, experience a much higher overhead for UltraSPARC III than for UltraSPARC II.

Abbreviation	DSZOOM Configuration
inv	invalidation-based protocol
upd	update-based protocol
swpc	single (1-entry) WPC
dwpc	double (2-entry) WPC
df	dirty-data filtering
pf	private-data filtering

Table 3. Protocol abbreviations.

We believe that the instrumentation overhead can be further tamed with compiler support for instrumentation, as been demonstrated by Niwa et al [27]. This could potentially close the performance gap to hardware DSM even further. However, this would require a recompilation when a shared-memory application is moved to software DSM.

4.4 Parallel Performance

Figure 6 shows the performance impact of various coherence protocols and optimizations for 16-processor runs when compared to DSZOOM's base protocol (`inv-64`). Selecting the most optimal coherence unit size improves `fft`'s performance with 25 percent. Adding the most appropriate WPC strategy amounts for an additional 0.4 speedup and instrumentation scheduling adds another 0.05. The base update protocol (`upd-swpc-64`) results in a slowdown compared with the `inv-64` protocol. However, tuning the WPC setting and enabling the private-data filtering improves `fft`'s update performance with more than 100 percent. Only performance improvements are shown in Figure 6, which is why some of the optimizations are not visible for all applications.

There is a large variation among the applications as to which class of optimization is the most important. `ocean-c`'s update protocol is greatly improved by the private-data filtering, which makes it outperform the best invalidate-based protocol, while `lu-c` enjoys a great boost to its invalidate protocol from its most optimal WPC setting. While Figure 6 can help understanding the importance of the different optimizations (further explained in Section 4.5), it should be pointed out that the different performance improvements reported are somewhat dependent on each other, why the orders in which they are presented do effect their individual contributions.

Figure 7 shows parallel performance for 16-processor

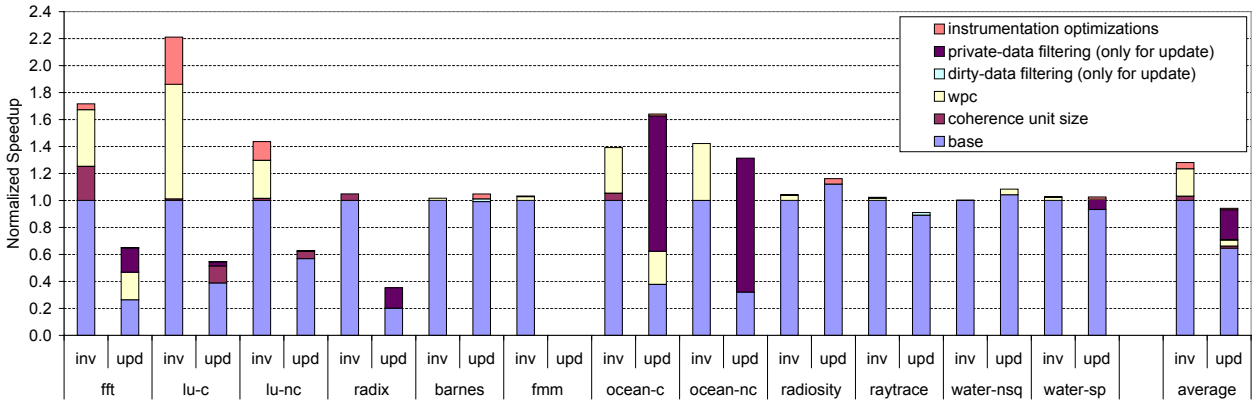


Figure 6. Invalidation- (`inv`) and update-based (`upd`) speedup when normalized to DSZOOM's invalidation-based system with a coherence unit size of 64 bytes (`inv-64`).

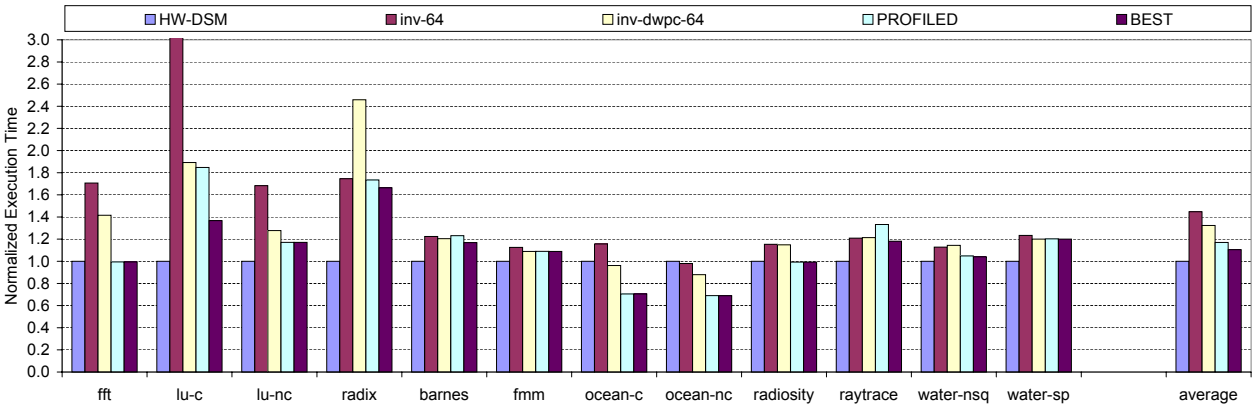


Figure 7. Hardware DSM vs. four software DSM experiments. (16-processor runs.)

runs. Here, the software DSM performance can be compared to the execution time of a 2-node Sun WildFire (HW-DSM) and DSZOOM's base configuration (`inv-64`). As a comparison, the execution time for `inv-dwpc-64` as well as the configuration suggested by the profiling tool (PROFILED), described in Section 5, are also reported. The rightmost bar (BEST) shows the best performance obtained by testing all coherence flag settings. (See Table 4 for (PROFILED) and (BEST) configurations.) To ensure that our results are not affected by application scaling characteristics, we also run all applications with four processors per node (eight in total). These results are almost identical to the ones presented in this paper (when compared to the hardware DSM) and are omitted because of space.

On average the `inv-64` protocol is 45 percent slower than the hardware DSM system. This overhead is reduced to 32 percent when the `inv-dwpc-64` protocol is used. Optimal coherence unit size, number of WPC en-

Program	PROFILED	BEST
`fft`	`inv-dwpc-2048`	`inv-dwpc-2048`
`lu-c`	`inv-swpc-2048`	`inv-dwpc-2048`
`lu-nc`	`inv-swpc-128`	`inv-swpc-128`
`radix`	`inv-64`	`inv-128`
`barnes`	`upd-swpc-64`	`upd-df-swpc-512`
`fmm`	`inv-swpc-64`	`inv-swpc-64`
`ocean-c`	`upd-pf-dwpc-512`	`upd-pf-dwpc-1024`
`ocean-nc`	`inv-dwpc-2048`	`inv-dwpc-1024`
`radiosity`	`upd-swpc-64`	`upd-swpc-64`
`raytrace`	`upd-swpc-64`	`inv-swpc-64`
`water-nsq`	`upd-pf-swpc-64`	`upd-pf-swpc-256`
`water-sp`	`upd-pf-swpc-64`	`upd-pf-swpc-64`

Table 4. Classification results from the profile feedback run and the best coherence setting.

tries and instrumentation optimizations further improve the invalidation-based DSZOOM performance with almost 30 percent. The invalidation-based protocol actually outperforms the hardware DSM system when run with `fft`, `ocean-c` and `ocean-nc`. While an invalidation-based coherence protocol together with "the best" coherence unit size offers stable performance, some applications show peak performance when run in an update-based environment as long as bandwidth usage is kept low. The update-based protocol is able to outperform the hardware DSM system for `ocean-c`, `ocean-nc`[2] and `radiosity`. Number of WPC entries and the private-data filter are the most important optimizations while in update mode.

Maybe the most notable performance feature is the similarity between the performance of the best DSZOOM protocol (BEST) and the Sun WildFire system (HW-DSM). The performance of the two systems is within 30 percent of each other for all applications except `radix` and `lu-c`. Note also that the continuous and non-continuous versions of `lu`, `ocean` and `water` all achieve a similar performance compared with the hardware DSM. This is typically not the case for traditional software DSMs. On average, DSZOOM is 11 percent slower than the hardware DSM. When `radix`, the application with the worst locality, is omitted, this slowdown is reduced to only 5 percent.

4.5 FFT Case Study

To be able to further show the impact of the different optimizations, we provide a case study of a representative application. Figure 8 shows DSZOOM performance for (a) the invalidation- and (b) the update-based protocols when coherence unit size is scaled. Execution time is normalized against the hardware DSM system whereas bandwidth (collected with WildFire interconnect counters) is normalized against the `inv-64` configuration. Figure 8 (a) shows that `fft` scales with coherence unit size in an invalidation-based environment. The `inv-dwpc` configuration performs best and outperforms the hardware DSM system when a coherence unit size of 2048 bytes is used. This is because parts of the `fft` application use two write streams, and hence, shows much better WPC hit rate with a 2-entry WPC than with a 1-entry WPC, see Figure 3.

Figure 8 (b) shows the update-based protocol with its filters and combinations of them. The performance of update is poor when used with a single WPC entry (`upd-swpc`). Scaling the coherence unit size makes it worse. However, performance is improved when the private-data filter is added (`upd-pf-swpc`). This is because the private-data filter exploits the fact that more than 50 percent of the global stores in `fft` are to private pages, and hence, manages to reduce the consumed bandwidth with more than 50 percent. Again, when coherence unit size is increased the performance goes down. The dirty-data filter configuration (`upd-df-swpc`) starts out where the `upd-swpc` system started, but improves as coherence unit size scales. This is because the single write stream part improves as coherence unit size scales but a 1-entry WPC and a large coherence unit size is a bad match for the part with two write streams. However, since the dirty-data filter removes updates of non-dirty data, coherence unit size scaling can be used. The `upd-df-pf-swpc` configuration shows that the dirty-data and the private-data filter target different kinds of bandwidth. The performance starts out where `upd-pf-swpc` starts but improves when coherence unit size is scaled.

Since `fft` has high WPC hit rate for a 2-entry WPC, the `upd-dwpc` configuration shows much better performance than the 1-entry WPC configuration. Again, adding the dirty-data filter both improves the performance and reduces the bandwidth. It is interesting to see that the parallel performance of `fft` directly follows consumed bandwidth.

[2]The reason why `inv-64` is better than the hardware DSM when run on `ocean-nc` is that the `-fno-delayed-branch` actually improves performance on this particular application.

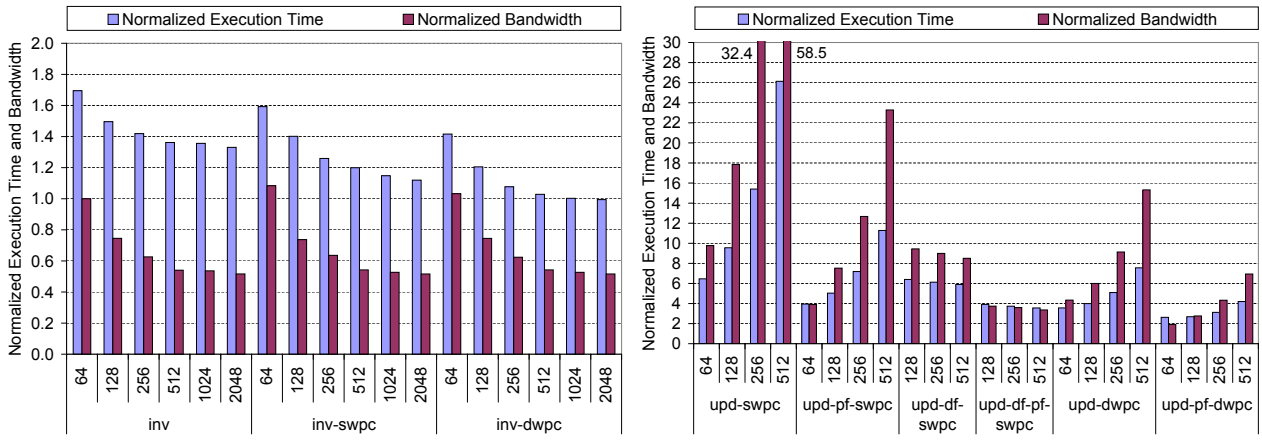
5 Profiling and Classification of Applications

Previous section demonstrates that a flexible software DSM system can deliver hardware DSM competitive performance. However, choosing optimal coherence flags may be a cumbersome task. This section presents a simple classification algorithm for fast finding of appropriate coherence settings. The classification heuristic is based on feedback from a low-overhead profile run. Our classification algorithm is not general, it is intended to show that also a simple heuristic can be used to achieve appropriate coherence settings, and hence, high performance.

We have tested our profiling mode and classification algorithm with both small and large working set sizes. Our results are almost identical. Thus, for the applications studied, it is possible to use the small working set size during the profile run and reuse the same coherence strategy for result runs! While scaling down the workload size on a uniprocessor system can heavily affect the cache performance, the profile mode tracks the entire shared memory and especially coherence traffic. Hence, it is not heavily dependent on machine parameters, such as cache sizes. (Table 1 shows the small working set size used for classification and the large working set size used for result runs.)

5.1 Low-Overhead Profiling

Our low-overhead profiling is capable to collect 1- and 2-entry WPC hit rate, global update bandwidth and coherence



(a) Performance of the invalidation-based protocol.

(b) Performance of the update-based protocol.

Figure 8. DSZOOM performance for `fft`. The execution time is normalized to that of the HW-DSM whereas the bandwidth is normalized to `inv-64`.

Program	**Base Mode** Large Set	**Profile Mode** Large Set	**Profile Mode** Training Set
`fft`	5.99 s	6.13 s (1.02)	0.18 s (0.03)
`lu-c`	28.79 s	22.90 s (0.80)	0.55 s (0.02)
`lu-nc`	46.58 s	96.49 s (2.07)	2.90 s (0.06)
`radix`	13.09 s	16.86 s (1.29)	3.56 s (0.27)
`barnes`	14.62 s	19.64 s (1.34)	2.77 s (0.19)
`fmm`	16.26 s	26.73 s (1.64)	7.66 s (0.47)
`ocean-c`	9.37 s	9.04 s (0.96)	1.51 s (0.16)
`ocean-nc`	17.71 s	17.44 s (0.98)	2.18 s (0.12)
`radiosity`	4.89 s	9.16 s (1.87)	0.23 s (0.05)
`raytrace`	15.27 s	18.69 s (1.22)	14.02 s (0.92)
`water-nsq`	12.85 s	13.74 s (1.07)	3.41 s (0.27)
`water-sp`	9.12 s	10.62 s (1.16)	1.65 s (0.18)
Avg.		(1.29)	(0.23)

Table 5. Performance of the profiling mode for large and training (small) input data sets. Normalized execution time is shown inside parenthesis.

unit size information in a single run with less than 30 percent overhead (avg.). The estimation of coherence unit size uses *virtual coherence units*. We slice the global memory space into different segments (currently, 2048 bytes each). These segments use different virtual coherence unit sizes. When a coherence miss occurs, permission for the entire virtual coherence unit size is acquired. DSZOOM's runtime system collects the number of misses for all different coherence sizes in a single run. Of course, it is important to divide the memory space in a representative way. We have tried multiple schemes (omitted because of space) and found that a simple modulo scheme works satisfactorily for our conservative classification algorithm and the applications studied. The virtual-memory system collects non-private store information. When a WPC miss occurs, a local page directory lookup classifies the store as private or non-private. The number of non-private store misses is proportional to the global update bandwidth used in the system when run in update mode. Hence, this information lets one decide if the update-based coherence protocol is a good candidate or not, and if the private-data filter should be used.

Table 5 shows performance of the profiling mode for 16-processor runs. Numbers for both training (small) and large data input sets are shown. On average, the profiling mode runs 29 percent slower than the base mode for large input sets. It is interesting to see that three applications run faster when the profiling mode is turned on! All three applications gain speedup because of the virtual-coherence technique described above! The large coherence unit size used in some of its segments are very beneficial for these appli-

cations. The performance of a profile mode is significantly increased for training input sets. For example, the most extreme case (`lu-c`) runs about 50 times faster than the base mode with a large input set.

5.2 Simple Classification Method

This section describes the proposed classification algorithm. Because the DSZOOM system implements multiple memory consistency models (further discussed in Section 6), and some applications require a stricter memory model than others, the memory consistency model has to be taken into account when choosing coherence flags. If the memory consistency model requirement for an application is not known, a conservative choice has to be made. We use the memory consistency model and the global update bandwidth to select between invalidate/update. We have used the global update bandwidth data and the low-overhead profile run execution time to estimate bandwidth/sec. Applications that consume less than 100 Mbyte/s are classified as update candidates.

The global update bandwidth with and without the private-data filter is used to make a choice for update filtering techniques. For example, `ocean-c` consumes more than 400 Mbyte/s without private-data filter. This number is reduced to less than 10 Mbyte/s when the filter is enabled (we enable the private-data filter if it reduces the bandwidth with more than 10 percent). Currently, we do not have a good metric for the dirty-data filter. However, it is reasonable to use this filter when a large coherence unit size is used with the `upd-swpc` configuration.

We use the number of coherence misses to different virtual coherence unit sizes to select coherence unit size for an application. Applications with a significant amount of spatial locality (e.g., `fft` and `lu-c`) are easily recognized because the number of misses is reduced by 50 percent each time the coherence unit size is doubled. Applications that expose false sharing are also easily recognized since the number of misses increase. However, applications that exploit some locality and at the same time introduce some false sharing are harder to classify. We use a conservative approach for these applications by choosing a small coherence unit size. In addition, an upper limit of 512 bytes for update-based protocols is applied.

Finally, the number of WPC entries has to be selected. Since 1- and 2-entry WPC hit rate is contained in the profile data, this seems like a simple task. However, the WPC hit rate is collected on a system running with a coherence unit size of 64 bytes. This is the reason why the classification of `lu-c` does not show peak performance (see Table 4 and Figure 7).

6 Memory Consistency, Deadlock, Scalability and Hardware Issues

The introduction of the WPC technique raises multiple questions regarding deadlocks and memory consistency models, which are addressed in this section. We also discuss the protocol scalability and the hardware DSM platform used for comparison.

6.1 Memory Consistency

The invalidation-based protocol of the base architecture (without a WPC implementation) maintains sequential consistency (SC) [23] by requiring all acknowledges from the sharing nodes to be received before a global store request is granted. Introducing the WPC in an invalidation-based environment 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 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 dictate the memory model of the system. The invalidation-based system implements total store order (TSO) [41] since E6000 nodes are used.

For an update-based system without load instrumentation, such as the one we present in Section 3.1, the sequential consistency property is sacrificed. Our update-based software DSM system with a 1-entry WPC and a 64 bytes coherence unit size implements processor consistency (PC) [13, 12]. Writes from a processor can not be observed out of issue order by another processor since node's hardware keeps write ordering correct per coherence unit. When a processor decides to write to a new coherence unit, the old coherence unit is made available to all other nodes, and hence, the processor store order is preserved. However, the order in which writes from two processors are seen by others may differ. Thus, PC and not TSO is implemented. The memory consistency model gets more relaxed if more than one WPC entry or a coherence unit size larger than 64 bytes is used. Such a system needs multiple updates to push all data to other nodes, and hence, store issue order can get lost. This consistency model is similar to weak-ordering (WO) [8]. Our PC and WO systems do not implement causal correctness [34].

6.2 Deadlock Avoidance Mechanisms

To avoid WPC related deadlocks, our runtime system releases a processor's WPC entries at synchronization points, at failures to acquire MTAG/directory entries and at thread termination. However, since SC (TSO) and PC are supported, flag synchronization not visible to the runtime sys-

```
01: /* P0's code */      11: /* P1's code */
02: a = 1;               12: b = 1;
03: while (flag != 1)    13: flag = 1;
04:     ; /* wait */     14: ...
05: ...
```

Figure 9. WPC-deadlock code. *a*, *b* and *flag* are all global variables initially assigned the value zero. *a* and *b* are located on the same coherence unit whereas *flag* is located on another.

tem can occur. The code in Figure 9 can for example lead to deadlock. *a*, *b* and *flag* are all global variables initially assigned the value zero. *a* and *b* are both located on the same coherence unit *u*. *flag* is located on coherence unit $v \neq u$. Let two processors, *P0* and *P1*, execute the code shown in Figure 9. *P0* enters the code first (executes line 02) and assigns *a* the value one. This implies that *P0* puts *a*'s coherence unit *u* in its WPC. When *P0* reaches line 03, it starts to spin on the shared variable *flag*, waiting for *P1*. *P1* enters the code (line 12) and tries to obtain write permission for *b*. However, since the directory state for *b*'s coherence unit *u* is locked and cached by *P0*'s WPC, we have a deadlock! *P0* is waiting for *P1* to update the *flag* variable, and *P1* is waiting for *P0* to release the caching of *u*.

These WPC related deadlocks are easily avoided with extra runtime system support. We have in an earlier study discussed three possible mechanisms: (1) a processor's WPC entries can be flushed periodically by the runtime system, (2) the processor waiting for coherence unit *u* can signal a WPC release to the processor caching write permission for *u* with a remote interrupt, and (3) *P0* can detect its lack of forward progress and flush its WPC entries.[3] For more information see [44]. However, we are convinced that the simplest and best solution is to implement WPC deadlock avoidance in the instrumentation tool. A WPC FIFO replacement policy together with simple basic-block analysis can be used to guarantee that all MTAG locks are released before flag synchronizations (not currently implemented). For simplicity, our instrumentation tool is manually guided in the two applications (*barnes* and *fmm*) that use flag synchronization.

6.3 Protocol Scalability

This paper only presents data for a 2-node system since our WildFire machine only contains two E6000 nodes. We have used "virtual clustering" [43] to show that our invalidation-based protocol scales with number of nodes [28]. This data is omitted since this paper is focused on the hardware comparison and because of limited space. However, we do not believe that our update-based protocol will scale for a large number of nodes. The reason why we have not used virtual clustering emulation while testing the update-based protocols is because it is very difficult to model bandwidth in an accurate way. It would be very interesting to test how a WPC-based store buffer and bandwidth filters will affect update scalability. We consider this evaluation as future work.

[3] A countdown register updated by the runtime system can be used.

6.4 Hardware DSM Considerations

There are several considerations that have to be taken into account when comparing two systems against each other. For example, will new technology trends change the findings?

This paper shows that our WPC technique makes instrumentation overhead scale while moving to a new processor generation with a significantly higher clock frequency. Longer remote and local memory latencies will decrease the instrumentation overhead and increase the impact of coherence unit size scaling and application specific coherence flags. The instrumentation overhead can also be reduced by moving the instrumentation stage to a optimizing compiler (e.g., Niwa et al [27]). Simple loop and basic block analysis can be used to select the best WPC strategy for each part of the program, and as discussed above, completely remove all WPC related deadlocks.

We find our results representative since we compare our system against a hardware DSM running on exactly the same cluster interconnect and node hardware. Especially for a system with a remote to local memory latency ratio of about 6 running the benchmarks tested.

7 Related Work

Traditional implementations of software-based shared memory rely on virtual memory hardware to detect when coherence activity is needed. Early page-based systems [25] suffer from false sharing that arises from fine-grain sharing of data within a page. Two main research directions have evolved to improve the performance of software shared memory implementations: relaxing consistency models [3, 19, 46, 33] and providing fine-grained access control [37, 33].

Page-based systems often rely on week memory consistency models and multiple writer protocol to manage the false sharing introduced by their large coherence unit [10, 40, 1]. Carter et al [3] introduce the release consistency (RC) model in shared virtual memory. Lazy release consistency (LRC) was introduced by Keleher et al [19] and home based lazy released consistency (HLRC) by Zhou et al [46]. The majority of systems implement numerous coherence strategies/protocols. For example, Munin [3] im-

plements both invalidate- and update-based protocols (including delayed-update and write-shared protocols).

First of all, our DSM proposal differs from these systems because it is a fine-grain approach that shows much more predictable performance for applications with fine-grain synchronization. In addition, our system can run multiple memory consistency models, including SC (TSO), PC and WO, with reasonable performance while page-based systems often rely on RC, LRC or HLRC protocols. Our private-data bandwidth filter is similar to Munin's time-out mechanism that makes it possible to only update nodes that actually use data. However, contrary to Munin, our private-data filter is completely synchronous, and hence, removes all asynchronous protocol messaging. Moreover, it is designed to be used with a fine-grain system and can be run in processor consistency mode whereas Munin relies on release consistency. Our dirty-data filter can be compared to the multiple writer protocols' twin and diff strategies. However, it is much more "light weight" and considerably faster to manage. Where twin and diff strategies have to compare the entire page, we simply check 2-4 bits located in a register before an update. Moreover, the dirty-data filter is a bandwidth reduction technique, while twin and diff strategies maintain coherence and memory consistency. Furthermore, we compare our system with a hardware DSM while running unmodified applications with and without fine-grain sharing and synchronization patterns.

Fine-grained software DSMs maintain coherence by instrumenting memory operations in the programs [35, 33, 32]. These systems usually provide stable and predictable performance for the majority of parallel benchmarks originally developed for hardware multiprocessors. On the other hand, the instrumentation cost for most of the systems is not negligible. An interesting comparative study of two mature software-based systems from the late 90s shows that the performance gap between fine- and coarse-grain software DSMs can be bridged by adjusting coherence unit size, program restructuring and relaxing memory consistency models [9].

In a early version of Blizzard [11], application specific software-based protocols, which provided very high performance, were implemented and evaluated. Shasta [33, 32] implements support for multiple coherence granularities within a single application. This mechanism is exposed to the programmer through multiple memory allocation functions. Zhou et al [47] presents performance tradeoffs for relaxed consistency and coherence granularity on a platform that provides access control in hardware but runs coherence protocols in software. Their study focuses on coherent shared memory systems with a fixed coherence granularity (64, 256, 1,024, and 4,096 bytes). The results show that no single combination of protocol and granularity performs the best for all SPLASH-2 [42] applications studied.

Our DSZOOM system differs from all these systems because it uses a synchronous directory protocol and since the virtual memory system is used to enhance performance. Where the other fine-grain systems use user-level hand optimized coherence protocols or application rewrite to enhance coherence protocol performance, we propose the use of coherence profiling and coherence flags. This is the first comparison (to our knowledge) with a real hardware DSM machine. It is also the first study in which an all-software system is able to outperform an all-hardware DSM!

The Stanford FLASH [22] project addresses concerns with hardwired protocols by migrating the entire protocol-engine to software handlers executed on a separate processor. SMTp is a more recent proposal [5] in which the coherence protocol is run by one SMT thread. Multiple systems implement a simple hardware directory protocol backed up with software handlers. The protocol described by Hill et al [17] uses a single hardware pointer. In addition, the programmer or compiler can annotate programs with Check-In/Check-Out (CICO) directives to minimize the number of software traps. Chaiken and Agarwal [4] describe performance and cost of software extended coherence shared memory as implemented in Alewife. Grahn and Stenström [14] extends Chaiken and Agarwal's work. While most of the findings in this paper can be implemented in such systems, they rely on modified memory controllers [5], protocol processors [22] and/or hardware support for *n* pointers in hardware [4, 14, 17].

8 Conclusions

This paper presents a highly flexible all-software distributed shared memory system that combines code instrumentation and page protection mechanisms. Fine-grain access control checks applied at shared loads and stores avoid false sharing without any application rewriting or memory model weakening. The paper also presents two protocol classes that are based on classical invalidate/update schemes. The page protection mechanism is applicable in both cases to minimize unnecessary global memory replication and, in particular, as an efficient bandwidth-reduction technique for update-based protocols. Several other reduction techniques are presented, such as all-software store buffering, dirty- and private-data filtering.

The paper demonstrates the flexibility of this approach with two simple invalidate/update synchronous protocols that support more then 50 different combinations of coherence flags. A very simplistic low-overhead profiling mode of the system is capable to find appropriate coherence flags for the studied applications. This is an automatic single-run process based on the profile run feedback.

The system demonstrates stable and predictable performance for all applications studied. In fact, several appli-

cations run faster with this system than with a much more expensive hardware-based DSM with an identical interconnect. On average, our software DSM is 11 percent slower than the hardware DSM. When `radix`, the application with the worst locality, is omitted, this slowdown is reduced to only 5 percent.

References

[1] A. Bilas, C. Liao, and J. P. Singh. Using Network Interface Support to Avoid Asynchronous Protocol Processing in Shared Virtual Memory Systems. In *Proceedings of the 26th Annual International Symposium on Computer Architecture (ISCA'99)*, pages 282–293, May 1999.

[2] W. W. Carlson, J. M. Draper, D. E. Culler, K. Yelick, E. Brooks, and K. Warren. Introduction to UPC and Language Specification. Technical Report CCS-TR-99-157, The George Washington University, May 1999.

[3] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and Performance of Munin. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP '91)*, pages 152–164, Oct. 1991.

[4] D. Chaiken and A. Agarwal. Software-Extended Coherent Shared Memory: Performance and Cost. In *Proceedings of the 21st Annual International Symposium on Computer Architecture (ISCA'94)*, pages 314–324, Apr. 1994.

[5] M. Chaudhuri and M. Heinrich. SMTp: An Architecture for Next-generation Scalable Multi-threading. In *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA'04)*, pages 124–135, June 2004.

[6] D. Chiou, B. S. Ang, R. Greiner, Arvind, J. C. Hoe, M. J. Beckerle, J. E. Hicks, and G. A. Boughton. StarT-NG: Delivering Seamless Parallel Computing. In *Proceedings of the 1st International Euro-Par Conference (Euro-Par 1995)*, pages 101–116, Aug. 1995.

[7] L. Dagum and R. Menon. OpenMP: An Industry-Standard API for Shared Memory Programming. *IEEE Computational Science and Engineering*, 5(1):46–55, Jan.-Mar. 1998.

[8] M. Dubois, C. Scheurich, and F. Briggs. Memory Access Buffering in Multiprocessors. In *Proceedings of the 13th Annual International Symposium on Computer Architecture (ISCA'86)*, pages 434–442, June 1986.

[9] S. Dwarkadas, K. Gharachorloo, L. I. Kontothanassis, D. J. Scales, M. L. Scott, and R. Stets. Comparative Evaluation of Fine- and Coarse-Grain Approaches for Software Distributed Shared Memory. In *Proceedings of the 5th International Symposium on High-Performance Computer Architeture (HPCA-5)*, pages 260–269, Jan. 1999.

[10] A. Erlichson, N. Nuckolls, G. Chesson, and J. L. Hennessy. SoftFLASH: Analyzing the Performance of Clustered Distributed Virtual Shared Memory. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, pages 210–220, Oct. 1996.

[11] B. Falsafi, A. R. Lebeck, S. K. Reinhardt, I. Schoinas, M. D. Hill, J. R. Larus, A. Rogers, and D. A. Wood. Application-Specific Protocols for User-Level Shared Memory. In *Proceedings of the 1994 ACM/IEEE Conference on Supercomputing (SC'94)*, pages 380–389, Nov. 1994.

[12] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory Consistency and Event Ordering in Scalable Shared-memory Multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture (ISCA'90)*, pages 15–26, May 1990.

[13] J. R. Goodman. Cache Consistency and Sequential Consistency. Technical Report 61, SCI Committee, Mar. 1989.

[14] H. Grahn and P. Stenström. Efficient Strategies for Software-Only Protocols in Shared-Memory Multiprocessors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA'95)*, pages 38–47, June 1995.

[15] E. Hagersten and M. Koster. WildFire: A Scalable Path for SMPs. In *Proceedings of the 5th International Symposium on High-Performance Computer Architeture (HPCA-5)*, pages 172–181, Jan. 1999.

[16] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 3rd edition, 2003.

[17] M. D. Hill, J. R. Larus, S. K. Reinhardt, and D. A. Wood. Cooperative Shared Memory: Software and Hardware for Scalable Multiprocessors. *ACM Transactions on Computer Systems*, 11(4):300–318, Nov. 1993.

[18] InfiniBand Trade Association, InfiniBand Architecture Specification, Release 1.2, Oct. 2004. Available from `http://www.infinibandta.org`.

[19] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy Release Consistency for Software Distributed Shared Memory. In *Proceedings of the 19th Annual International Symposium on Computer Architecture (ISCA'92)*, pages 13–21, May 1992.

[20] K. Krewell. Sun's Niagara Pours on the Cores: Early Details Revealed at Hot Chips 16. In *Microprocessor Report*, Sept. 2004.

[21] K. Krewell. Best Servers of 2004: Where Multicore Is the Norm. In *Microprocessor Report*, Jan. 2005.

[22] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. L. Hennessy. The Stanford FLASH Multiprocessor. In *Proceedings of the 21st Annual International Symposium on Computer Architecture (ISCA'94)*, pages 302–313, Apr. 1994.

[23] L. Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, C-28(9):690–691, Sept. 1979.

[24] J. R. Larus and E. Schnarr. EEL: Machine-Independent Executable Editing. In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 291–300, June 1995.

[25] K. Li. *Shared Virtual Memory on Loosely Coupled Multiprocessors*. PhD thesis, Department of Computer Science, Yale University, Sept. 1986.

[26] L. W. McVoy and C. Staelin. lmbench: Portable Tools for Performance Analysis. In *Proceedings of the 1996 USENIX Annual Technical Conference*, pages 279–294, Jan. 1996.

[27] J. Niwa, T. Matsumoto, and K. Hiraki. Comparative Study of Page-based and Segment-based Software DSM through

Compiler Optimization. In *Proceedings of the 14th International Conference on Supercomputing (ICS'00)*, pages 284–295, May 2000.

[28] Z. Radović and E. Hagersten. DSZOOM – Low Latency Software-Based Shared Memory. Technical Report 2001:03, Parallel and Scientific Computing Institute (PSCI), Sweden, Apr. 2001.

[29] Z. Radović and E. Hagersten. Removing the Overhead from Software-Based Shared Memory. In *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing (SC'01)*, Nov. 2001.

[30] D. J. Scales and K. Gharachorloo. Design and Performance of the Shasta Distributed Shared Memory Protocol. In *Proceedings of the 11th International Conference on Supercomputing (ICS'97)*, pages 245–252, July 1997.

[31] D. J. Scales and K. Gharachorloo. Towards Transparent and Efficient Software Distributed Shared Memory. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)*, pages 157–169, Oct. 1997.

[32] D. J. Scales, K. Gharachorloo, and A. Aggarwal. Fine-Grain Software Distributed Shared Memory on SMP Clusters. In *Proceedings of the 4th International Symposium on High-Performance Computer Architeture (HPCA-4)*, pages 125–136, Feb. 1998.

[33] D. J. Scales, K. Gharachorloo, and C. A. Thekkath. Shasta: A Low-Overhead Software-Only Approach to Fine-Grain Shared Memory. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, pages 174–185, Oct. 1996.

[34] C. Scheurich. *Access Ordering and Coherence in Shared Memory Multiprocessors*. PhD thesis, University of Southern California, May 1989.

[35] I. Schoinas, B. Falsafi, M. Hill, J. R. Larus, and D. A. Wood. Sirocco: Cost-Effective Fine-Grain Distributed Shared Memory. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT'98)*, pages 40–49, Oct. 1998.

[36] I. Schoinas, B. Falsafi, M. D. Hill, J. R. Larus, C. E. Lucas, S. S. Mukherjee, S. K. Reinhardt, E. Schnarr, and D. A. Wood. Implementing Fine-Grain Distributed Shared Memory On Commodity SMP Workstations. Technical Report #1307, Computer Sciences Department, University of Wisconsin–Madison, Mar. 1996.

[37] I. Schoinas, B. Falsafi, A. R. Lebeck, S. K. Reinhardt, J. R. Larus, and D. A. Wood. Fine-grain Access Control for Distributed Shared Memory. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI)*, pages 297–306, Oct. 1994.

[38] A. Singhal, D. Broniarczyk, F. Cerauskis, J. Price, L. Yuan, C. Cheng, D. Doblar, S. Fosth, N. Agarwal, K. Harvey, E. Hagersten, and B. Liencres. Gigaplane: A High Performance Bus for Large SMPs. In *Proceedings of IEEE Hot Interconnects IV*, pages 41–52, Aug. 1996.

[39] S. J. Sistare and C. J. Jackson. Ultra-High Performance Communication with MPI and the Sun Fire Link Interconnect. In *Proceedings of the 2002 ACM/IEEE Conference on Supercomputing (SC'02)*, Nov. 2002.

[40] R. Stets, S. Dwarkadas, N. Hardavellas, G. Hunt, L. I. Kontothanassis, S. Parthasarathy, and M. Scott. Cashmere-2L: Software Coherent Shared Memory on a Clustered Remote-Write Network. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)*, pages 170–183, Oct. 1997.

[41] D. L. Weaver and T. Germond, editors. *The SPARC Architecture Manual, Version 9*. PTR Prentice Hall, 2000.

[42] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA'95)*, pages 24–36, June 1995.

[43] D. Yeung, J. Kubiatowicz, and A. Agarwal. Multigrain Shared Memory. *ACM Transactions on Computer Systems*, 18(2):154–196, May 2000.

[44] H. Zeffer, Z. Radović, O. Grenholm, and E. Hagersten. Evaluation, Implementation and Performance of Write Permission Caching in the DSZOOM System. Technical Report 2004-005, Department of Information Technology, Uppsala University, Feb. 2004.

[45] H. Zeffer, Z. Radović, O. Grenholm, and E. Hagersten. Exploiting Spatial Store Locality through Permission Caching in Software DSMs. In *Proceedings of the 10th International Euro-Par Conference (Euro-Par 2004)*, pages 551–560, Aug. 2004.

[46] Y. Zhou, L. Iftode, and K. Li. Performance Evaluation of Two Home-Based Lazy Release Consistency Protocols for Shared Virtual Memory. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation (OSDI '96)*, pages 75–88, Oct. 1996.

[47] Y. Zhou, L. Iftode, J. P. Singh, K. Li, B. R. Toonen, I. Schoinas, M. D. Hill, and D. A. Wood. Relaxed Consistency and Coherence Granularity in DSM Systems: A Performance Evaluation. In *Proceedings of the Sixth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP '97)*, pages 193–205, June 1997.