Implementing Low Latency Distributed Software-Based Shared Memory

ZORAN RADOVIĆ and ERIK HAGERSTEN
Uppsala University
Information Technology
Department of Computer Systems
P.O. Box 325, SE-751 05 Uppsala, Sweden
email: {zoranr,eh}@it.uu.se

Abstract

Software-implementations of shared memory are still far behind the performance of hardware-based shared memory implementations (HW-DSM) and are not viable options for most fine-grain shared memory applications. The major source for their inefficiency comes from the cost of interrupt-based asynchronous protocol processing, not from the actual network latency. As the raw hardware latency of inter-node communication decreases, the asynchronous overhead in the communication becomes more dominant.

We describe how all the interrupt- and/or poll-based asynchronous protocol processing can be completely removed by running the entire coherence protocol in the requesting processor. This not only removes the asynchronous overhead, but also makes use of a processor that otherwise would stall. The technique is applicable to both page-based and fine-grain software-based shared memory.

DSZOOM-WF—the implementation presented in this paper—is a sequentially consistent, fine-grain distributed software-based shared memory. It demonstrates a protocol-handling overhead below a microsecond for all the actions involved in a remote load operation, to be compared to the fastest implementation to date of around ten microseconds. The all-software protocol is implemented assuming some basic low-level primitives in the cluster interconnect and an operating system bypass functionality, similar to the emerging InfiniBand standard.

DSZOOM-WF demonstrates consistently comparable performance to HW-DSM implementations.

1 Introduction

Clusters of symmetric multiprocessors (SMPs) provide a powerful platform for executing parallel applications. To allow for shared-memory applications to run on such clusters, software distributed shared memory (SW-DSM) systems support the illusion of shared memory across the cluster via a software run-time layer between the application and the hardware. This approach can potentially provide a cost-effective alternative to hardware shared memory systems for executing certain classes of workloads. SW-DSM technology can also be used to connect several large hardware distributed shared memory (HW-DSM) systems and thereby extend their upper scalability limit.

Most SW-DSM systems keep coherence between page-sized coherence units [Li88], [CBZ91], [KCDZ94], [SB97], [SDH+97]. The normal per-page access privilege of the memory-management unit offers a cheap access control mechanism for these SW-DSM systems. The large page-size coherence units in the earlier SW-DSM systems created extra false sharing and caused frequent page transfers of large pages between nodes. In order to avoid most of the false sharing, weaker memory models have been used to allow many update actions to be lumped to a specific point in time, such as the lazy release consistency (LRC) protocol [Kel95].

Fine-grain SW-DSM systems with a more traditional cache-line-sized coherence unit have also been implemented. Here, the access control check is either done by altering of the error-correcting codes (ECC) [SFH+96] or by in-line code *snippets* (small fragments of machine code) [SFH$^+$96], [SGT96]. The small cache-line size reduces the false sharing for these systems, but the explicit access-control check adds extra latency for each load or store operation to global data. The most efficient access check reported to date is three extra instructions adding three extra cycles for each load to global data [SFH$^+$98].

Today's implementations of SW-DSM systems suffer from long remote latencies and their scalability have never reached acceptable levels for general SMP shared-memory applications. The coherence protocol is often implemented as communicating software agents running in the different nodes sending requests and replies to each other. Each agent is responsible for accessing its local memory and for keeping a directory structure for "its part" of the shared address space. The agent where the directory structure for a specific coherence unit resides is called its home node. The interrupt cost, associated with receiving a message, for asynchronous protocol processing is the single largest component of the slow remote latency, not the actual wire delay in the network or the software actually implementing the protocol. To our knowledge, the shortest SW-DSM read latency to date is that of Shasta [SGA97]. The 15-microsecond round-trip read latency is roughly divided into 5 microseconds, of "real" communication and 10 microseconds of interrupt and agent overhead [Gha00]. Most other SW-DSM implementations have substantially larger interrupt overheads, and latencies closer to 100 microseconds have been reported [SFH$^+$96].

In this paper we suggest a new efficient approach for software-based coherence protocols. While other work have proposed elaborate schemes for cutting down on the overhead associated with interrupting and/or polling caused by the asynchronous communication between the agents [BLS99], [MFHW96], our implementation has completely eliminated the protocol-agent interactions. In DSZOOM the entire coherence protocol is implemented in the protocol handler running in the requesting processor. This also makes use of a processor that otherwise would have been idle. Rather than relying on a "directory agent" located in the home node, as the synchronization point for the coherence of a cache line, we use a remote atomic fetch-and-set operation to allow for protocol handlers running in any node, not just the home node, to temporarily acquire atomic access to the directory structure of the cache line. We believe that the solution presented here would be beneficial both for page-sized and fine-grain SW-DSM systems, even though we will only concentrate on fine-grain SW-DSM in this paper.

We have implemented the DSZOOM-WF system, a sequentially consistent fine-grain SW-DSM, between the nodes of a Sun-WildFire [HK99] system without relying on its hardware-based coherence capabilities. All loads and stores are instead performed to the node's local "private" memory. We use the executable editing library (EEL) [LS95] to insert fine-grain access control checks before shared-memory loads and stores in a fully compiled and linked executable. Global coherence is resolved by coherence protocol implemented in C that copies data to the nodes "private" local memory by performing loads and stores from and to remote memory.

We have measured the actual protocol overhead to be less than one microsecond for a remote load operation, in addition to the 1.7 microseconds remote latency of the Sun-WildFire hardware, i.e. a perceived remote latency of 2.7 microseconds for the application. A total of twelve unmodified SPLASH-2 programs [WOT$^+$95], developed for fine-grain hardware SMP multiprocessors, are studied. We compare the performance of a DSZOOM-WF system with that of a Sun Enterprise E6000 SMP server [SBC$^+$96] as well as the hardware-coherent Sun-WildFire DSM system.

The remainder of this paper is organized as follows. Section 2 presents our basic idea and an introduction to a general DSZOOM system. The DSZOOM-WF implementation is described in Section 3. Section 4 presents the experimental environment, applications used in this study, and results of our performance study. Finally, we present related work and conclude.

2 DSZOOM Overview

This section contains an overview of the general DSZOOM system. More protocol details are reported for the related DSZOOM-EMU protocol [RH01], our initial proof-of-concept DSZOOM implementation that emulates fine-grain SW-DSM between "virtual nodes," modeled as processes inside a single SMP.

2.1 Cluster Networks Model

DSZOOM assumes a cluster interconnect with an inexpensive user-level mechanism to access memory located in other nodes, similar to the remote put/get semantics found in the cluster version of the Scalable Coherence Interface (SCI) implementation by Dolphin.[1] A ping-pong round-trip latency of 5 microseconds, including MPI protocol overhead, has been demonstrated on a SCI network with a 2 microsecond raw read latency. Some of the memory in the other nodes is mapped into a node's I/O space and can be accessed using ordinary load and store operations. The different cluster nodes run different kernel instances and do not share memory with each other in a coherent way; in other words, no invalidation messages are sent between the nodes to maintain coherence when replicated data are altered in one node. This removes the needs for the complicated coherence scheme implemented in hardware and allows the NIC to be connected to the I/O bus, e.g. PCI or SBUS, rather than to the memory bus. In order to prevent a "wild node" from destroying crucial parts of other node's memories, the incoming transactions are sent through a network MMU (IOMMU). Each kernel needs to set up appropriate IOMMU mapping to the remotely accessible part of its memory before the other nodes are accessed. Given the correct initialization of the IOMMU, user-level accesses to remote memory have been enabled. SCI-Cluster is widely used as the high-performance cluster interconnect by Sun Microsystems for large commercial and technical systems.

We further assume support for two new remote-access operations not supported by the SCI-Cluster: the half-word-wide *put2* and *fetch-and-set2* (fas2). The fas2 operation is launched by a "normal" half-word load operation and the put2 is launched by a half-word store to the remotely mapped I/O space. The network interface detects the half-word load and converts it into a fetch-and-set. The fas2 operation will return the 2 byte of data that was stored in the remote memory and also atomically set the most significant byte of the data in the remote memory. The fas2 primitive is used to aquire a lock and retrieve a corresponding small data-structure in a single operation.

[1]SCI is better known for its implementation of coherent shared memory than its non-coherent internode cluster communication. In this paper we only refer to its usage as a cluster interconnect.

There are strong indications that interconnects fulfilling our assumptions will soon be widely available. The emerging InfiniBand interconnect proposal supports efficient user-level accesses to remote memory as well as atomic operations to smaller pieces of data, e.g. `CmpSwap` (Compare and Swap) and `FetchAdd` (Fetch and Add) [Inf00]. InfiniBand's `FetchAdd` can effectively implement a function similar to the fas2 functionality for a system with up to 128 nodes. The least significant byte (LSB) of the data entity accessed is the "lock" and the remaining part of the data entity is the payload data. A `FetchAdd` returning data with a zero LSB means that the lock was acquired. The lock is released and the payload data is updated in a single operation by writing the new payload value with a zero byte concatenated at the LSB end to the data entity. In order to avoid mangling the payload data for contended locks, a `FetchAdd` returning a LSB with a value above 128 will require the contenders to poll the data-structure using ordinary `Fetch` operations until the LSB with a value below 128 has been observed.

2.2 Node Model

Each DSZOOM node consists of an SMP multiprocessor, e.g. the Sun Enterprise E6000 SMP with up to 30 processors or the Pentium Pro Quad with up to four processors. The SMP hardware keeps coherence among the caches and the memory within each SMP node. The InfiniBand-like interconnect, as described above, connects the nodes. We further assume that the write order between any two endpoints in the network is preserved.

2.3 Blocking Directory Protocol Overview

Most of the complexity of coherence protocol is related to the race conditions caused by multiple simultaneous requests for the same cache line. Blocking directory coherence protocols have been suggested to simplify the design and verification of hardware DSM systems [HK99]. The directory blocks new requests to a cache line until all previous coherence activity to the cache line has ceased. The requesting node sends a completion signal upon completion of the activity, which releases the block for the cache line. This eliminates all the race conditions, since each cache line can only be involved in one ongoing coherence activity at any specific time.

The DSZOOM protocol implements a distributed version of a blocking protocol. A processor that has detected the need for global coherence activity will first acquire a lock associated with the cache line before starting the coherence activity. A remote fas2 operation to the corresponding directory entry in the home node will bring the directory entry to the processor and also atomically acquire the cache line's "lock." If the most significant byte of the directory entry returned is set, the cache line is "busy" by some other coherence activity. The fas2 operation is repeated until the most significant byte is zero.[2] Now, the processor has acquired the exclusive right to perform coherence activities on the cache line and has also retrieved the necessary information in the directory entry using a single operation. The processor now has the same information as, and can assume the role of, the "directory agent" in the home node of a more traditional SW-DSM implementation. Once the coherence activity is completed, the lock is released and the directory is updated by a single put2 transaction. No memory barrier is needed after the put2 operation since any other processor will wait for the most significant byte of the directory entry to become zero before the directory entry can be used. Thus, the latency of the remote write will not be visible to the processor.

To summarize, we have enabled the requesting processor to momentarily assume the role of a traditional "directory agent," including access to the directory data, at the cost of one remote latency and the transfer of two small network packets. This has the advantage of removing the need for asynchronous interrupts in foreign nodes and also allows us to execute the protocol in the requesting processor that most likely would be idle waiting for the data. A further advantage is that the protocol execution is divided between all the processors in the node, not just one processor at a time as suggested in some other proposals, for example by Mukherjee et al. [MFHW96].

[2]A random back-off scheme can be used to avoid a live-lock situation, but has not been employed in DSZOOM yet.

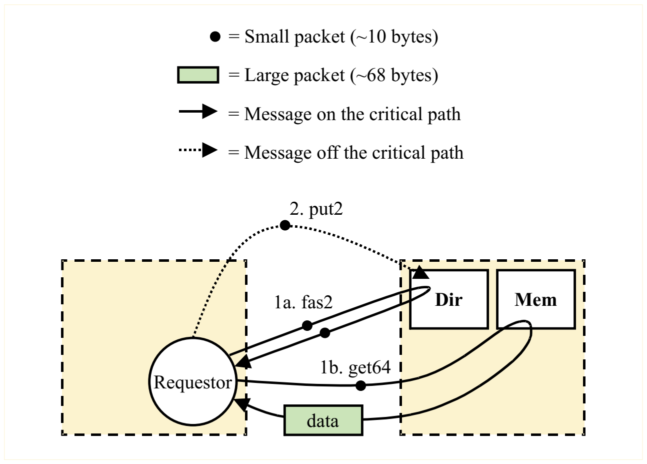

● = Small packet (~10 bytes)
= Large packet (~68 bytes)
= Message on the critical path
= Message off the critical path
2. put2
1a. fas2
1b. get64
Requestor
Dir
Mem
data

Figure 1: Read data from home node — 2-hop read.

2.4 Protocol Details

The SMP hardware keeps the coherence within the node, on top of which the global DSZOOM protocol has been added. All the coherence activities and state names discussed in this paper apply to the DSZOOM protocol.

The DSZOOM protocol states, MODIFIED, SHARED and INVALID (MSI), are explicitly represented by data structures in the node memory. The DSZOOM directory entry has eight presence bits per cache line, i.e. can support up to eight SMP nodes. The location of a cache line's directory location, i.e. its "home node", is determined by looking at some of its address bits. To avoid most of the accesses to the directory caused by global load operations, all cache lines in state INVALID store by convention a "magic" data value as independently suggested by Schoinas et al. [SFH+96] and Scales et al. [SGT96]. The directory only has to be consulted if the register contains the magic value after the load. Whenever our selected magic value is indeed the intended data value, the directory state must be examined at the cost of some unnecessary global activities. This has, however, proven to be a very rare event in all our studied applications.

To also avoid most of the accesses to the directory caused by global store operations, each node has two bytes of local MTAG state per global cache line, indicating if the cache line is locally writable. Before each global store operation, the MTAG byte is locked by a local atomic operation, before the access write to the cache line is determined. The directory only has to be

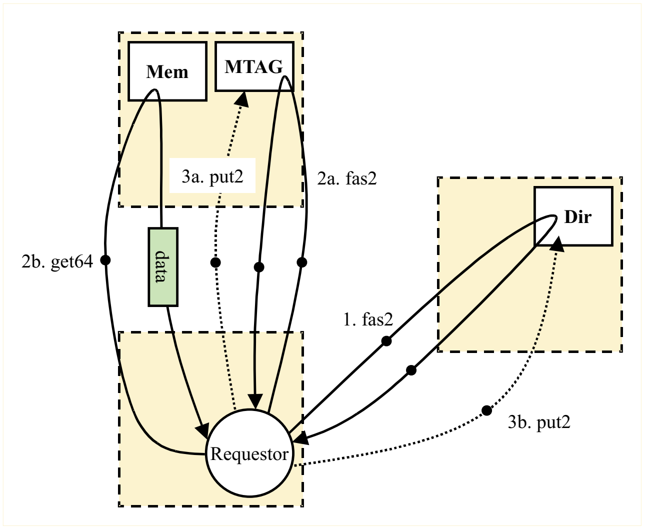


Figure 2: Read data modified in a third node — 3-hop read.

```
IF (register == MAGIC) {
  lock(dir)
  IF (presence_bits(me) == 0) {
    IF ((number(presence_bits) == 1) &&
        (remote_node != home)) {
      lock(remote_mtag)
      // Data can not be altered in the remote node now
      read_remote(data)
      update_release(remote_data)
    }
    ELSE read_remote(data)
  }
  update_release(dir)
}
```

Figure 3: Pseudo-code for global coherence load operations. Emphasized lines are implemented as in-line assembler, while the remaining protocol is implemented by a routine coded in C.

consulted if the MTAG indicate that the node currently does not have write permission to the cache line. The home node can access the cache line's directory entry by a local memory access and do not need any extra MTAG state.

The traditional SW-DSM's software handler running in the current processor sends a message to the home node and busy-waits for the reply. A new software handler is invoked in the home node upon the arrival of the request. The home handler retrieves the requested data from its local memory and modifies the corresponding directory structure before returning the data reply to the requesting handler. The two major drawbacks of this approach is the latency from asynchronously invoking a handler in the home node and the simultaneous occupancy of two handlers during most of the protocol handling, i.e. occupying two processors.

Figure 1 illustrates the DSZOOM activity caused by a read miss in a two-node system. The software protocol handler in the DSZOOM example will acquire exclusive access right to the directory entry through a single remote fas2 operation to the home node. In parallel it also speculatively retrieves the data from the home node through a remote *get64* operation. The directory entry is updated and released by a single remote put2 operation at the end of the handler. The protocol handler is completed as soon as the put2 write operation is issued to the write buffer, why the latency of this operation is not on the critical latency path of the application. While the DSZOOM approach will drastically cut the latency for retrieving remote data and will avoid using any processor time in the home node, its major drawback is the global bandwidth consumed. To illustrate the excess bandwidth consumed by DSZOOM, each global packet has been marked as either a "small packet," with a payload of less than 6 bytes, or a "large packet," with a payload of 64 bytes.[3] Each packet type is assumed to also carry 2 bytes of cyclic redundancy code (CRC) and 2 bytes of routing/header information, why the total number of bytes are 10 bytes and 68 bytes respectively. Based on these assumptions, DSZOOM's four small and one large packet will transfer 108 bytes compared with the 78 bytes used by the traditional approach, i.e. 38% more bandwidth is used in DSZOOM.

A similar bandwidth overhead can be seen in the example in Figure 2 showing a three-node system performing a 3-hop read operation, i.e. a read request to data which resides in a modified state in a node different than the home node, called the slave node. DSZOOM will need one fas2 message to lock and acquire the directory and determine the identity of the node holding the modified data. A second fas2 to that node's MTAG structure will temporarily disable write accesses to the data. Right after the fas2 has been issued a get64 is issued to speculatively bring the data to the requesting node. The directory entry and the MTAG is updated and released through two put2 write operations at the end of

[3]We have not included the implicit acknowledge packets that may be used by the lower level network implementation.

```
lock(my_mtag)
// Is only "my" bit set?
IF (my_mtag == my_mask) {
  // Have we already locked the dir?
  IF (me != home) {
    // Try once to lock the directory
    lock_test(dir)
    // Release our MTAG if dir is busy
    IF (busy(dir)) {
      // to avoid deadlocks
      release(my_mtag)
      // Now, first lock directory
      lock(dir)
      // then lock MTAG
      lock(my_mtag)
    }
  }
  // Now we have locked the dir for sure!
  IF (number(presence_bits) != 1) {
    // The data is shared by many nodes and is not writable
    IF (presence_bits(me) == 0)
      // my data is not valid
      read_data_from_one_node
    FOREACH sharer
      // invalidate remote nodes
      store_remote(MAGIC)
  }
  ELSE IF (presence_bits(me) == 0) {
    // There is a single node with a writable copy,
    // and it is not me
    IF (me == home) {
      // The dir is already locked
      read_remote(data)
      store_remote(MAGIC)
    }
    ELSE {
      lock(remote_mtag)
      read_remote(data)
      store_remote(MAGIC)
      update_release(remote_mtag)
    }
  }
  IF (me != home) update_release(dir)
  update_release(my_mtag)
}
```

Figure 4: Pseudo-code for global coherence store and load-store operations. Emphasized lines are implemented as in-line assembler, while the remaining protocol is implemented by a routine coded in C.

the handler, i.e. off the critical path. Again, DSZOOM will need more messages to complete its task: seven small and one large packet compared with the three small and one large used by the traditional approach. The traditional SW-DSM approach will need two asynchronous interrupts on the critical path before the data is forwarded to the requesting node. Thus, DSZOOM will require 41% more bandwidth for this particular operation. This is the worst-case protocol example for DSZOOM which, fortunately, is not that common in the studied examples.

Figure 3 shows pseudo-code for global load operations and Figure 4 shows pseudo-code for global store and load-store operations.

3 Implementation Details

This section describes our DSZOOM-WF implementation. DSZOOM-WF is a sequentially consistent fine-grain SW-DSM implemented on top of the 2-node Sun-WildFire prototype SMP cluster (without relaying on its hardware-coherent capabilities). Our cluster is built from two Sun Enterprise E6000 SMP machines (we call them here for cabinet 1 and cabinet 2). DSZOOM-WF compilation process is shown in Figure 5. The unmodified SMP application written with PARMACS macros is first preprocessed with a m4 macro preprocessor. m4 will replace all PARMACS macros with DSZOOM-WF run-time library calls. Standard GNU gcc compiler is used to compile and link the preprocessed file with a DSZOOM-WF run-time library.[4] The resulting file, the "(Un)executable," is then passed to our binary modification tool that will insert fine-grain access control checks before shared-memory loads and stores into binaries by statically analyzing and modifying executables. Binary modification tool will also add coherence protocol code (shown in Figure 3 and Figure 4) to the binaries. Finally, the `a.out` is produced and can be used as if it was executed inside one SMP.

The DSZOOM-WF implementation of PARMACS macros is based on the System V IPC version of the PARMACS shared-memory macros developed by Artiaga et al. [ANMB97], [AMBN98]. The macro library was modified in several ways, e.g. we use user-level

[4]Current version of EEL, i.e. version 4.0.1, can only be compiled with GNU gcc 2.8.1 compiler, and can only modify binaries that are compiled with that particular compiler.

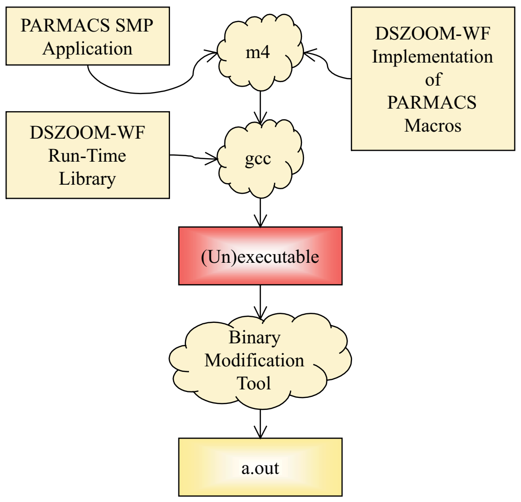


Figure 5: DSZOOM-WF compilation process.

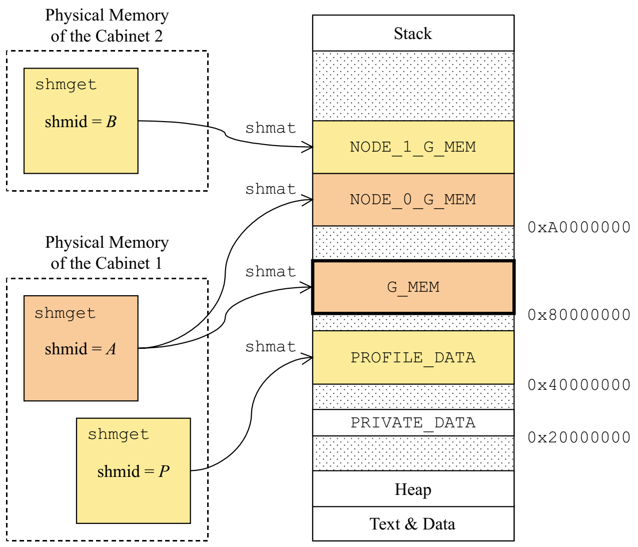


Figure 6: Address space layout and attachment of the shared memory objects for processes running in the cabinet 1.

```
    1: ld [addr], %f-p reg  // original LD
    2: fcmps %fcc0, %f-p reg, %f-p reg
    3: nop
    4: fbe,pt %fcc0, hit
    5: nop
    6: // call global coherence load routine
hit:
```

Figure 7: Fine-grain access control check for one floating-point load instruction.

synchronization through test-and-set locks instead of System V IPC semaphore library calls, we added support for process distribution by using the Solaris system call `pset_bind`, support for correct memory placement and directory distribution based on WildFire's "first-touch" policy was added as well. We also added support for memory-mapped communication between the processes. Address space layout and attachment of the shared memory objects for every process running in the cabinet 1 is shown in Figure 6. Shared memory objects with shared memory identifiers *A* and *B* represent the physical shared/global memory of every node in the cluster. Shared memory identifier *P*, which is attached to the `PROFILE_DATA` area with a standard Solaris system call `shmat`, is used for profiling purposes only. Local run-time system data for every process (e.g. DSZOOM-WF process identifier, UNIX process identifier, etc.) is stored in a privately mapped `PRIVATE_DATA` area in a current implementation and is also used only for debugging and profiling purposes. Distributed directory is placed at the beginning of the `G_MEM` area.

Binary *instrumentation* is a technique usually described as a low-cost, medium-effort approach of inserting sequences of machine instructions into a program in executable or object format. We decided to use executable editing library (EEL) [LS95], a library that was successfully used in several similar projects based on the UltraSPARC architecture, e.g. Blizzard-S [SFL+94] and Sirocco-S [SFH+98]. The actual implementation of the low-level fine-grain instrumentation is still far from optimal. Examples of more efficient instrumentation can be found in both the Shasta [SGA97] and the Sirocco-S [SFH+98]. Our binary modification tool does not instrument accesses to non-shared stack data, but it still do instrument a huge number of static data accesses. Figure 7 shows the code snippet for one global floating-point fine-grain access control check. The "magic" value in this case is a small integer corresponding to an IEEE floating-point NaN. Only instructions 1–4 are executed if the loaded data is valid.

4 Performance Study

In this section we describe experimental setup, applications used in this study, and finally, we present DSZOOM-WF performance overview.

4.1 Experimental Setup

Most experiments in this paper are performed on a Sun Enterprise E6000 SMP [SBC+96]. The server has 16 UltraSPARC II (250 MHz) processors and 4 Gbyte uniformly shared memory with an access time of 330 ns (lmbench latency [MS96]) 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 hardware 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 [HK99]. The Sun-WildFire has been configured as a traditional non-uniform memory architecture (NUMA) 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 1700 ns (lmbench latency). The E6000 and the Sun-WildFire are both running a slightly modified version of the Solaris 2.6 operating system.

DSZOOM-WF runs in user space on the Sun-WildFire system with its data migration and the CMR data replication kept inactive.

4.2 Applications

The benchmarks we use in this study are well-known scientific workloads from the SPLASH-2 benchmark suite [WOT+95]. The data-set sizes and uniprocessor-execution times are presented in Table 1. The reason why we cannot run Volrend and Cholesky is only because of the global variables used as shared. It should be possible to manually modify those applications to solve this problem.[5] We began all measurements at the start of the parallel phase to exclude DSZOOM-WF's run-time system initialization time.

4.3 DSZOOM-WF Performance Overview

Efficient binary instrumentation (or compiler support) is very important for the overall DSZOOM performance. Sequential-execution times for the instrumented SPLASH-2 programs are shown in Table 1. Efficiency overhead is between 1.11 and 1.68 (which averages 1.30) for all of the studied applications, i.e. instrumented code takes between 11% and 68% longer time to execute the program after the DSZOOM-WF global access checks are added. State-of-the-art checking overheads (for example in Shasta [SGA97]) are in the range of 5-35%. Increased software checking overhead gives us a bad starting point. Normalized instrumentation overhead breakdown for sequential-execution is shown in Figure 8. Floating-point store snippets are the major slowdown factor for both the LU-Cont and the LU-Non-Cont. LU will typically perform much better on systems with weaker memory models (for example on GeNIMA [BLS99] with home-based LRC protocol). The total number of Raytrace's static memory accesses (that are currently also instrumented) is much bigger then the number of global memory accesses. That is one of the reasons why the efficiency overhead for Raytrace is so high, i.e. 1.53. It should be possible to cut down this kind of overhead with more detailed backward slice algorithm [Wei84] to avoid instrumenting static loads and stores.

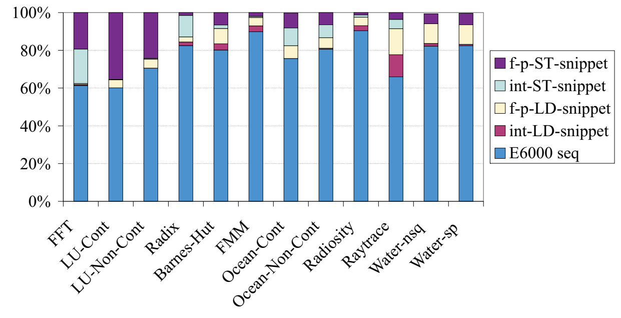


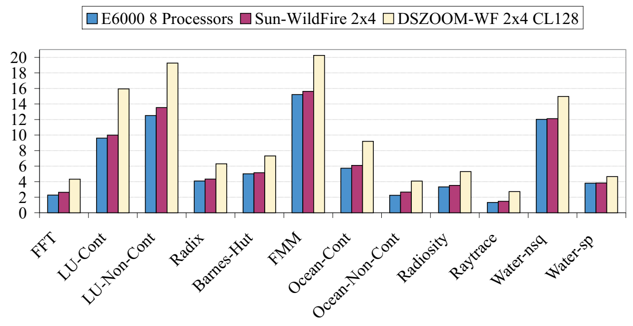
Figure 8: Normalized instrumentation overhead breakdown for sequential-execution.

Execution times for 8- and 16-processor runs for Sun Enterprise E6000, 2-node Sun-WildFire, and 2-node DSZOOM-WF with cache-line-sized coherency unit of 128 bytes are shown in Figure 9. Normalized execution time breakdowns for 8- and 16-processor runs for a 2-node DSZOOM-WF with the same coherency unit are shown in Figure 10. In our current implementation, synchronization variables (for example, locks, barriers, events, etc.) are allocated and physically placed in the cabinet 1 only. That can explain the big synchronization time increment for Ocean-Non-Cont, Raytrace,

[5] Ernest Artiaga have experienced similar problems with the original fork-exec versions of Volrend and Cholesky [Art01].

Program	Problem Size, Iterations	Non-Instrumented Seq. Time [s]	% LD	% ST	Instrumented Seq. Time [s]	Instrumentation Overhead
FFT	1,048,576 points (48.1 MB)	15.39	26.1	22.2	21.99	1.43
LU-Cont	1024×1024, block 16 (8.0 MB)	68.78	22.7	14.5	115.49	1.68
LU-Non-Cont	1024×1024, block 16 (8.0 MB)	88.63	23.9	16.6	125.50	1.42
Radix	4,194,304 items (36.5 MB)	28.81	24.1	14.9	33.17	1.15
Barnes-Hut	16,384 bodies (32.8 MB)	36.78	37.5	50.5	45.98	1.25
FMM	32,768 particles (8.1 MB)	109.03	25.5	22.9	121.80	1.12
Ocean-Cont	514×514 (57.5 MB)	43.85	28.6	26.2	58.54	1.34
Ocean-Non-Cont	258×258 (22.9 MB)	17.05	15.5	31.6	20.71	1.21
Radiosity	room (29.4 MB)	25.06	31.1	35.0	27.91	1.11
Raytrace	car (50.2 MB)	9.75	28.8	31.5	14.93	1.53
Water-nsq	2197 mols., 2 steps (2.0 MB)	85.55	24.5	32.4	103.30	1.21
Water-sp	2197 mols., 2 steps (1.5 MB)	22.91	25.5	27.6	27.65	1.21

Table 1: Data-set sizes and sequential-execution times for non-instrumented and instrumented SPLASH-2 applications. Fourth and fifth column show percentage of statically replaced loads and stores. Instrumentation overhead is shown in the last column.



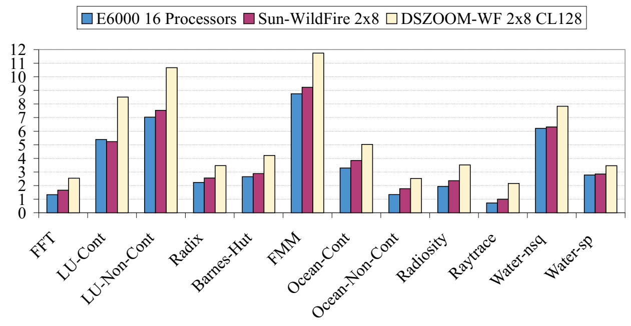
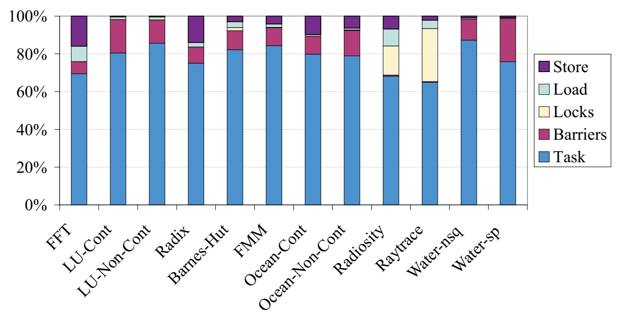


Figure 9: Execution times for 8- (left) and 16-processor runs (right) for Sun Enterprise E6000, 2-node Sun-WildFire, and 2-node DSZOOM-WF with cache-line-sized coherency unit of 128 bytes.



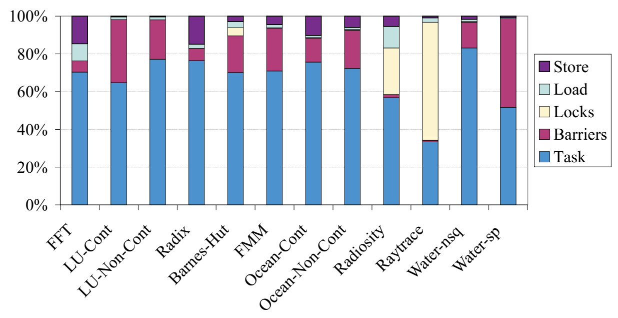


Figure 10: Normalized execution time breakdowns for 8- (left) and 16-processor runs (right) for a 2-node DSZOOM-WF with cache-line-sized coherency unit of 128 bytes.

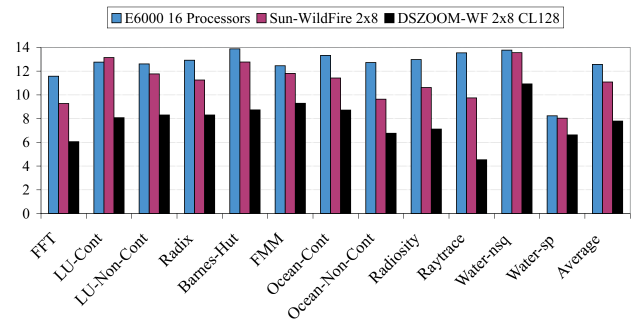


Figure 11: Application speedups for Sun Enterprise E6000, 2-node Sun-WildFire, and 2-node DSZOOM-WF.

and Water-sp for 16-processor runs compared with 8-processor experiments.

Speedup values for 16-processor runs for Sun Enterprise E6000, 2-node Sun-WildFire, and 2-node DSZOOM-WF with cache-line-sized coherency unit of 128 bytes are shown in Figure 11. Note that the DSZOOM-WF speedup values are calculated relatively Table 1, column three, i.e. non-instrumented sequential times. Performance of FFT and LU-Cont can be improved with larger coherency units [WOT$^+$95], for example we have measured that the coherency unit of 2048 bytes can speedup FFT with at least 15%.

5 Related Work

Many different SW-DSM implementations have been proposed over the years: Blizzard-S [SFL$^+$94], Brazos [SB97], Cashmere-2L [SDH$^+$97], [DGK$^+$99], CRL [JKW95], GeNIMA [BLS99], Ivy [Li88], [LH89], MGS [YKA96], Munin [CBZ91], Shasta [SGT96], [SGA97], [SG97a], [SG97b], [DGK$^+$99], Sirocco-S [SFH$^+$98], SoftFLASH [ENCH96], and TreadMarks [KCDZ94]. Most of them suffer from synchronous interrupt protocol processing. We see our work as a complement to these activities and believe that most of these implementations would benefit from a more efficient protocol implementation.

The GeNIMA proposal is closest to our work. GeNIMA proposes a protocol and network solution to avoid some of the asynchronous overhead. A processor starting a synchronous communication event, e.g. the requesting processor initiating some coherence actions, checks for incoming messages at the same time. This avoids some of the asynchronous overhead in the home node, but will also add some extra delay while waiting for a synchronous event to happen in the node. The protocol is still implemented as communicating protocol agents.

6 Conclusions

We have demonstrated how asynchronous protocol processing can be completely avoided at the cost of some extra remote transactions—trading bandwidth for efficiency. We believe that the total round-trip SW-DSM latency can be kept below three microseconds once the raw latency of a modern interconnects has been added.

The protocol technique described in this paper is applicable to the emerging InfiniBand I/O interconnect proposal. We believe a protocol, such as the one we describe, could speed up many of the existing SW-DSM implementations on such interconnect.

DSZOOM-WF demonstrates consistently comparable performance to hardware DSM implementations.

7 Future Work

We plan to continue this work in several different directions. First, cache-coherence protocol code optimizations will give direct impact on the performance of the DSZOOM-WF system. Because EEL has problems with hand-written in-line assembly in combination with high optimization levels during the compilation (our protocol routines written in C, and synchronization part of our run-time system that is also written in C, use quite a lot of in-line assembly gcc constructs) we do not use any optimizations during the compiling phase of both the coherence protocol routines and the run-time system.

Second, instrumentation technique can be improved even more. For example, instrumentation of the huge number of static loads and stores could be minimized by using a more detailed backward slice algorithm [Wei84] as in systems such as the Blizzard-S [SFL$^+$94] and the Sirocco-S [SFH$^+$98].

Third, in order to improve the performance of the DSZOOM-WF system a weaker memory models, such as the lazy release consistency (LRC) [Kel95] and the release consistency model presented by Gharachorloo et al. [GLL+90], can be used instead of the sequential consistency model that is currently implemented. This kind of optimization will allow many update actions to be lumped to a specific point in time.

Fourth, the synchronization part of our implementation should be distributed to get more balanced execution between the cabinets.

Finally, to make this kind of system more usable it is desirable to make a POSIX-threads implementation as well because most of the commercial workloads are implemented with that programming model rather than PARMACS.

Acknowledgments

We would like to thank Glen Ammons (Computer Sciences Department, University of Wisconsin–Madison) for excellent support and quick EEL updates, Sverker Holmgren and Henrik Löf (Department of Scientific Computing, Uppsala University) for providing access to machines. We would also like to thank Lars Albertsson and Erik Berg (Department of Computer Systems, Uppsala University), Anders Landin and Larry Meadows (Sun Microsystems), and the anonymous reviewers for comments on earlier drafts of the paper.

This work is supported in part by the Sun Microsystems, Inc., and the Parallel and Scientific Computing Institute (PSCI), Sweden.

References

[AMBN98] E. Artiaga, X. Martorell, Y. Becerra, and N. Navarro. Experiences on Implementing PARMACS Macros to Run the SPLASH-2 Suite on Multiprocessors. In *Proceedings of the 6th Euromicro Workshop on Parallel and Distributed Processing*, January 1998.

[ANMB97] E. Artiaga, N. Navarro, X. Martorell, and Y. Becerra. Implementing PARMACS Macros for Shared-Memory Multiprocessor Environments. Technical Report UPC-DAC-1997-07, Department of Computer Architecture, Polytechnic University of Catalunya, January 1997.

[Art01] E. Artiaga. Personal communication, April 2001.

[BLS99] 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)*, May 1999.

[CBZ91] 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, October 1991.

[DGK+99] S. Dwarkadas, K. Gharachorloo, L. 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 Architecture*, pages 260–269, January 1999.

[ENCH96] 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, October 1996.

[Gha00] K. Gharachorloo. Personal communication, October 2000.

[GLL+90] 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.

[HK99] E. Hagersten and M. Koster. WildFire: A Scalable Path for SMPs. In *Proceedings of the 5th IEEE Symposium on High-Performance Computer Architecture*, pages 172–181, February 1999.

[Inf00] InfiniBand(SM) Trade Association, InfiniBand Architecture Specification, Release 1.0, October 2000. Available from: http://www.infinibandta.org.

[JKW95] K. Johnson, M. F. Kaashoek, and D. A. Wallach. CRL: High-Performance All-Software Distributed Shared Memory. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, December 1995.

[KCDZ94] P. Keleher, A. L. Cox, S. Dwarkadas, and W. Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *Proceedings of the Winter 1994 USENIX Conference*, pages 115–131, January 1994.

[Kel95] P. Keleher. *Lazy Release Consistency for Distributed Shared Memory*. PhD thesis, Department of Computer Science, Rice University, January 1995.

[LH89] K. Li and P. Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[Li88] K. Li. IVY: A Shared Virtual Memory System for Parallel Computing. In *Proceedings of the 1988 International Conference on Parallel Processing (ICPP'88)*, volume II, pages 94–101, August 1988.

[LS95] 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.

[MFHW96] S. S. Mukherjee, B. Falsafi, M. D. Hill, and D. A. Wood. Coherent Network Interfaces for Fine-Grain Communication. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture (ISCA'96)*, pages 247–258, April 1996.

[MS96] L. W. McVoy and Carl Staelin. lmbench: Portable Tools for Performance Analysis. In *Proceedings of the 1996 USENIX Annual Technical Conference*, pages 279–294, January 1996.

[RH01] Z. Radović and E. Hagersten. DSZOOM – Low Latency Software-Based Shared Memory. Technical Report 2001:03, Parallel and Scientific Computing Institute (PSCI), Sweden, April 2001. Available from: http://www.psci.kth.se.

[SB97] E. Speight and J. Bennett. Brazos: A Third Generation DSM System. In *Proceedings of the 1st USENIX Windows NT Symposium*, August 1997.

[SBC+96] 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, August 1996.

[SDH+97] R. Stets, S. Dwarkadas, N. Hardavellas, G. Hunt, L. 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 System Principle*, October 1997.

[SFH+96] 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, March 1996.

[SFH+98] I. Schoinas, B. Falsafi, M. Hill, J. R. Larus, and D. A. Wood. Sirocco: Cost-Effective Fine-Grain Distributed Shared Memory. In *Proceedings of 6th International Conference on Parallel Architectures and Compilation Techniques*, October 1998.

[SFL+94] 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, October 1994.

[SG97a] D. J. Scales and K. Gharachorloo. Design and Performance of the Shasta Distributed Shared Memory Protocol. In *Proceedings of the 11th ACM International Conference on Supercomputing*, July 1997. Extended version available as Technical Report 97/2, Western Research Laboratory, Digital Equipment Corporation, February 1997.

[SG97b] D. J. Scales and K. Gharachorloo. Towards Transparent and Efficient Software Distributed Shared Memory. In *Proceedings of the 16th ACM Symposium on Operating System Principles, Saint-Malo, France*, October 1997.

[SGA97] D. J. Scales, K. Gharachorloo, and A. Aggarwal. Fine-Grain Software Distributed Shared Memory on SMP Clusters. Technical Report 97/3, Western Research Laboratory, Digital Equipment Corporation, February 1997.

[SGT96] 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, October 1996.

[Wei84] M. Weiser. Program Slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, July 1984.

[WOT+95] 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.

[YKA96] D. Yeung, J. Kubiatowicz, and A. Agarwal. MGS: A Multigrain Shared Memory System. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture (ISCA'96)*, pages 44–56, May 1996.