Removing the Overhead from 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

The implementation presented in this paper—DSZOOM-WF—is a sequentially consistent, fine-grained 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. All interrupt- and/or poll-based asynchronous protocol processing is 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 consistently demonstrates performance comparable to hardware-based distributed shared memory 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 [26], [8], [21], [40], [41]. 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 [20].

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 the error-correcting codes (ECC) [37] or by in-line code *snippets* (small fragments of machine code) [37], [35]. 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 [36].

Today's implementations of SW-DSM systems suffer from long remote latencies and their scalability has 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 [6], [17]. To our

SC2001 November 2001, Denver

knowledge, the shortest SW-DSM read latency to date is that of Shasta [34]. The 15-microsecond round-trip read latency is roughly divided into 5 microseconds, of "real" communication and 10 microseconds of interrupt and agent overhead [12]. Most other SW-DSM implementations have substantially larger interrupt overheads, and latencies closer to 100 microseconds have been reported [37].

In this paper we suggest a new efficient approach for software-based coherence protocols. While other work has proposed elaborate schemes for cutting down on the overhead associated with interrupting and/or polling caused by the asynchronous communication between the agents [5], [29], 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 [24] fine-grain SW-DSM, between the nodes of a Sun Orange (earlier referred to as Sun WildFire) system without relying on its hardware-based coherence capabilities [15], [30]. All loads and stores are instead performed to the node's local "private" memory. We use the executable editing library (EEL) [25] 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 a coherence protocol implemented in C that copies data to the node's "private" local memory by performing loads and stores from and to remote memory.

A total of twelve unmodified SPLASH-2 applications [43], 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 [39] as well as the hardware-coherent Sun Orange DSM system. 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 Orange hardware, i.e., a perceived remote latency of 2.7 microseconds for the application. Our approach is close to what hardware cache coherence can do on the same platform. On average, our implementation demonstrates a relative difference for SPLASH-2 speedups of 31.6% compared to the hardware-based cache-coherent, memory access (CC-NUMA) 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 system [31], our initial proof-of-concept DSZOOM implementation that emulates fine-grain software-based 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 nodes' 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 are 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 bytes 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) [18]. 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.

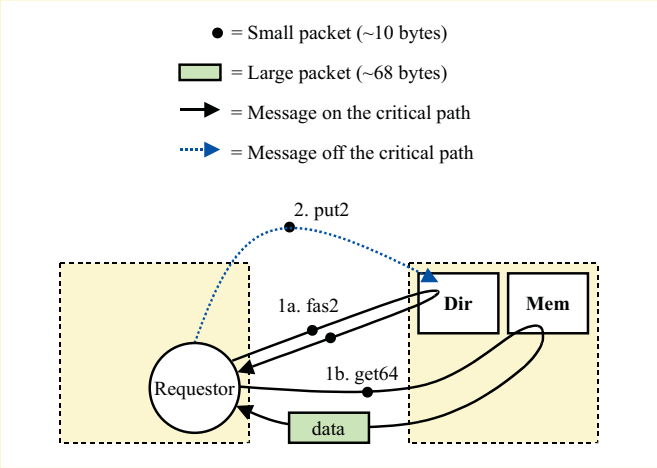


Figure 1: Read data from home node — 2-hop read.

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 a 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 [15]. 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, that 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. [29].

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

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

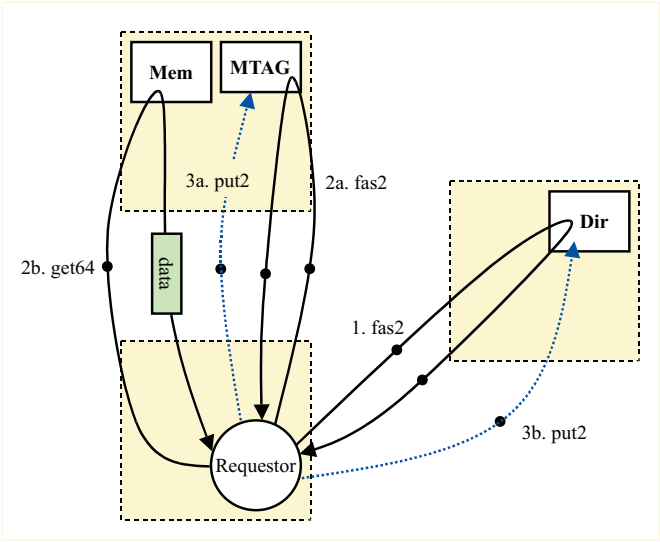


Figure 2: Read data modified in a third node — 3-hop read.

Algorithm 1 Pseudo-code for global coherence load operations. Emphasized line is implemented as in-line assembler, while the remaining protocol is implemented by a routine coded in C.

```
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_mtag)
    ELSE
      read_remote(data)
  update_release(dir)
```

"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. [37] and Scales et al. [35]. 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 state (MTAG) per global cache line (similar to the *private state table* found in Shasta [34]), 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 consulted if the MTAG indicates 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 does 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 are 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, so 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, so 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 are updated and released through two put2 write operations at the end of the handler, i.e., off

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

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.

Algorithm 2 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.

```
lock(my_mtag)
IF (my_mtag == my_mask) // Is only "my" bit set?
  IF (me != home)   // Have we already locked the dir?
    lock_test(dir) // Try once to lock the directory
    // Release our MTAG if dir is busy
    IF (busy(dir))
      release(my_mtag) // To avoid deadlocks
      lock(dir)        // Now, first lock directory
      lock(my_mtag)    // then lock 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
      store_remote(MAGIC) // Invalidate remote nodes
  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)
```

Algorithm 1 shows pseudo-code for global coherence load operations. The pseudo-code for global coherence store and load-store operations is shown in Algorithm 2. Emphasized lines in both algorithms are implemented as UltraSPARC in-line assembler, while the remaining protocol is implemented by routines coded in C.

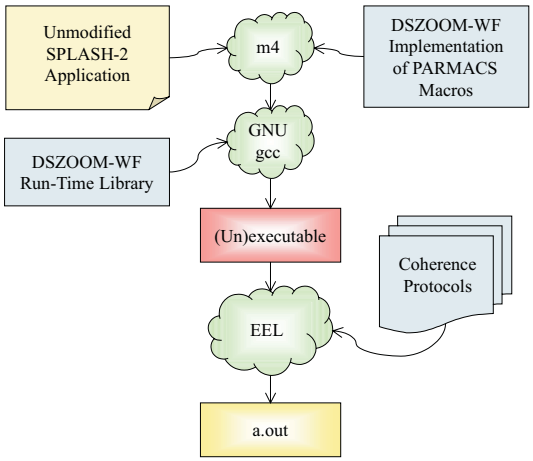


Figure 3: DSZOOM-WF compilation process.

3 Implementation Details

This section describes our implementation. DSZOOM-WF is a sequentially consistent [24] fine-grain SW-DSM implemented on top of a 2-node Sun Orange prototype SMP cluster configured as a cache-coherent, non-uniform memory access (CC-NUMA) architecture (without relying on its hardware-coherent capabilities). Our cluster is built from two Sun Enterprise E6000 SMP machines (referred to as cabinet 1 and cabinet 2). DSZOOM-WF compilation process is shown in Figure 3. The unmodified SMP application written with PARMACS macros is first preprocessed with a m4 macro preprocessor. m4 will replace all macros with DSZOOM-WF run-time library calls. A standard GNU gcc compiler is used to compile and link the preprocessed file with a DSZOOM-WF run-time library. The resulting file, the "(Un)executable," is then passed to our binary modification tool that is based on an unmodified version of the executable editing library (EEL) [25]. The binary modification tool inserts fine-grain access control checks after shared-memory loads, it inserts range checks and node-local MTAG lookups before stores, and it also adds calls to the corresponding coherence protocol routines shown in Algorithm 1 and Algorithm 2. Finally, the `a.out` is produced and can be used as if it was executed inside one SMP.

The implementation of PARMACS macros is based on the System V IPC version of the shared-memory macros developed by Artiaga et al. [3], [2]. The macro library was modified in several ways. We use user-level synchronization through physically distributed test-and-set locks instead of System V IPC semaphore library calls. Additionally, we added support for process distribution by using the Solaris system call `pset_bind`. The support for correct memory placement and directory distribution based on Sun Orange "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 pro-

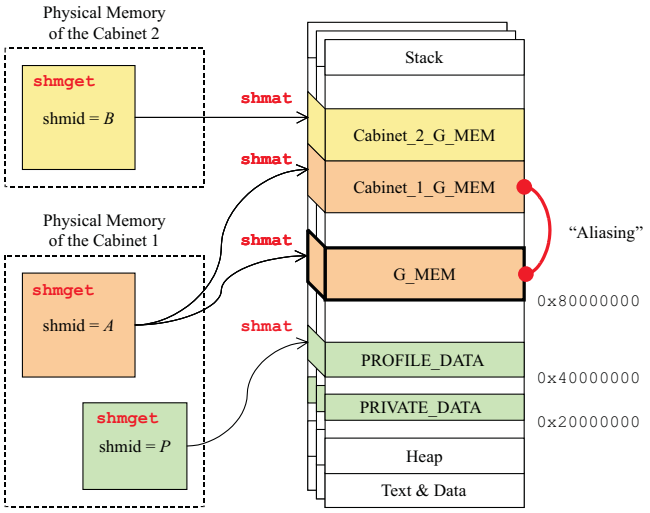


Figure 4: Address space layout and attachment of the shared memory objects for processes running in the cabinet 1.

cesses in cabinet 1 is shown in Figure 4. The compiled code makes global memory accesses to the `G_MEM` area. Shared memory objects with shared memory identifiers *A* and *B* represent the physically shared memory of every node in the cluster. The shared memory identifier *P*, which is physically allocated and placed in the cabinet 1, is attached to the `PROFILE_DATA` area with a standard Solaris system call `shmat`. This globally shared memory segment 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 mainly 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), a library that was successfully used in several similar projects based on the UltraSPARC architecture, e.g., Blizzard-S [38] and Sirocco-S [36]. The following code example shows the code snippet for one global floating-point fine-grain access control check.

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

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, i.e., the `%reg` is a non-magic value.[4] Thus, this access control check is comparable to the most efficient access check reported to date; three extra instructions adding three extra cycles for each load to global data [36]. The actual implementation of the low-level fine-grain instrumentation is still far from optimal. The DSZOOM-WF system requires in total between 7 and 8 instructions after every global load and 17 instructions before every global store/load-store. The reason why our instrumentation overhead for stores/load-stores is so high compared to some other fine-grain SW-DSM implementations (for example, Shasta [34] and Sirocco-S [36]) is because our local MTAG lookups are atomic in the current implementation, i.e., the MTAG byte is locked by a local atomic operation before the access write to the cache-line is determined. The characterization of the dynamic overheads for studied applications is presented in Section 4.3.

4 Performance Study

This section describes experimental setup, applications used in this study, sequential and parallel binary instrumentation overheads, and finally, DSZOOM-WF performance results for parallel execution.

4.1 Experimental Setup

Most experiments in this paper are performed on a Sun Enterprise E6000 SMP [39]. The server has 16 UltraSPARC II (250 MHz) processors and 4 Gbyte uniformly shared memory with an access time of 330 ns (*lmbench* latency [28]) 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 Orange built from two E6000 nodes connected through a hardware-coherent interface with a raw bandwidth of 800 Mbyte/s in each direction [15], [30]. The Orange system has been configured as a traditional cache-coherent, non-uniform memory access (CC-NUMA) architecture with its data migration capability activated while its coherent memory replication (CMR) has been kept inactive. The Sun Orange access time to local memory is the same as above, 330 ns, while accessing data located in the other E6000 node takes about 1700 ns (lmbench latency). The E6000 and the Orange DSM system are both running a slightly modified version of the Solaris 2.6 operating system.

DSZOOM-WF runs in user space on the Sun Orange system. The data migration and the CMR data replication of the Orange interconnect are kept inactive.

4.2 Applications

The benchmarks we use in this study are well-known scientific workloads from the SPLASH-2 benchmark suite [43]. The

[4] Line 3 can be eliminated if the code is executed on a SPARC-V9 architecture.

Program	Problem Size, Iterations	Uniproc Time [s]	% Load	% Store	Instrumented Uniproc Time [s]
FFT	1 048 576 points (48.1 Mbyte)	15.47	19.0	16.5	21.28 (1.38)
LU-c	1 024×1 024, block 16 (8.0 Mbyte)	69.17	15.5	9.4	109.64 (1.59)
LU-nc	1 024×1 024, block 16 (8.0 Mbyte)	82.43	16.7	11.1	123.81 (1.50)
Radix	4 194 304 items (36.5 Mbyte)	28.95	15.6	11.6	32.68 (1.13)
Barnes	16 384 bodies (32.8 Mbyte)	37.16	23.8	31.1	38.44 (1.03)
FMM	32 768 particles (8.1 Mbyte)	109.76	17.5	13.6	116.27 (1.06)
Ocean-c	514×514 (57.5 Mbyte)	43.89	27.0	23.9	58.77 (1.34)
Ocean-nc	258×258 (22.9 Mbyte)	17.04	11.6	28.0	21.14 (1.24)
Radiosity	room (29.4 Mbyte)	25.10	26.3	27.2	26.74 (1.07)
Raytrace	car (32.2 Mbyte)	9.73	19.0	18.1	11.75 (1.21)
Water-nsq	2 197 molecules, 2 steps (2.0 Mbyte)	85.01	13.4	16.2	90.06 (1.06)
Water-sp	2 197 molecules, 2 steps (1.5 Mbyte)	22.98	15.7	13.9	24.95 (1.09)
Average		**45.56**	**18.4**	**18.3**	**56.29 (1.22)**

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. Binary instrumentation overhead is given in parentheses in the last column.

data-set sizes for the applications studied are presented in Table 1. The reason why we cannot run Volrend and Cholesky (that are also part of the original SPLASH-2 distribution) is that the global variables used as shared are not correctly allocated with the `G_MALLOC` macro. 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 Binary Instrumentation

Efficient binary instrumentation (or compiler support) is very important for the overall DSZOOM performance. In this section we primarily focus on characterizing the overhead of inserted fine-grain access control checks for global loads and stores by presenting the dynamic overheads for all of the studied SPLASH-2 programs.

Sequential-execution times for non-instrumented programs, and the percentage of statically replaced loads/stores for SPLASH-2 applications are shown in Table 1. Currently, our binary modification tool statically replaces on average 18.4% loads and 18.3% stores during the instrumentation phase. We use numerous techniques to perform static and dynamic analysis of the unmodified binaries in order to recognize as many global loads and stores as possible [42], [25], [35], i.e., the binary modification tool will not replace many of the stack and static data accesses. Uniprocessor execution times in seconds for instrumented programs are also shown in Table 1. Efficiency overhead for sequential execution, presented in parentheses in the last column, is between 1.03 and 1.59 (which averages 1.22) for all of the studied applications. Thus, instrumented code takes between 3% and 59% longer time to execute the program after the global fine-grain access control checks for loads and atomic/node-local MTAG lookups for stores are added.

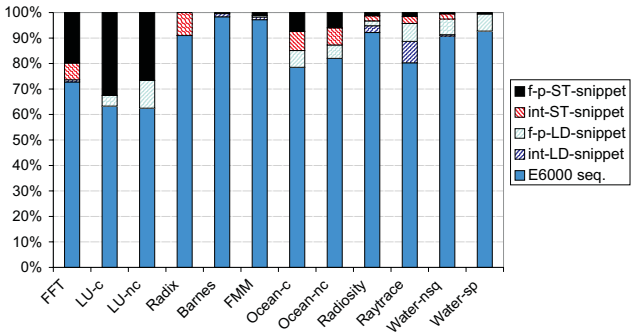


Figure 5: Normalized instrumentation overhead breakdown for sequential-execution.

State-of-the-art checking overheads (for example in Shasta [34]) are in the range of 5-35%. Unfortunately, increased software checking overhead gives us a bad starting point for some of the applications (e.g., for FFT, LU-c, and LU-nc).

Normalized instrumentation overhead breakdown for sequential-execution is shown in Figure 5. Floating-point store snippets are the major slowdown factor for FFT, LU-c, and LU-nc. LU is one of the most store-intensive SPLASH-2 applications [43] and will typically perform much better on software-based DSM systems with weaker memory models (for example on GeNIMA [5] with home-based LRC protocol).

[5] Artiaga has experienced similar problems with the original fork-exec versions of Volrend and Cholesky [1].

Program	8 Processors		16 Processors	
FFT	1.29	(-8.9%)	1.08	(-29.8%)
LU-c	1.58	(-0.2%)	1.50	(-8.3%)
LU-nc	1.60	(+9.6%)	1.44	(-6.2%)
Radix	1.15	(+2.4%)	1.07	(-6.0%)
Barnes	1.15	(+11.5%)	1.05	(+1.7%)
FMM	1.03	(-3.2%)	1.02	(-3.6%)
Ocean-c	1.25	(-8.4%)	1.14	(-20.3%)
Ocean-nc	1.56	(+32.3%)	1.52	(+28.2%)
Radiosity	1.09	(+2.4%)	1.06	(-0.8%)
Raytrace	1.20	(-1.2%)	1.10	(-10.6%)
Water-nsq	1.06	(-0.4%)	1.05	(-0.6%)
Water-sp	1.09	(+0.9%)	1.09	(+0.8%)
Average	**1.25**	**(+2.5%)**	**1.18**	**(-4.6%)**

Table 2: Instrumentation overheads for parallel execution on a single-node DSZOOM-WF system. The change in overheads for 8- and 16-processor runs compared to the uniprocessor overheads is presented in parentheses.

Program	ILC	Protocol	Distribution
FFT	1.29	1.27	1.14
LU-c	1.58	1.00	1.02
LU-nc	1.60	1.01	1.00
Radix	1.15	1.16	1.10
Barnes	1.15	1.03	1.06
FMM	1.03	1.09	1.08
Ocean-c	1.25	1.08	1.08
Ocean-nc	1.56	1.09	1.03
Radiosity	1.09	1.21	1.18
Raytrace	1.20	1.23	1.20
Water-nsq	1.06	1.01	1.01
Water-sp	1.09	1.01	1.00
Average	**1.25**	**1.10**	**1.08**

Table 3: Efficency overheads for effects of in-line checks (ILC), coherence protocol processing, and the memory and process distribution across the nodes for 8-processor runs on a 2-node DSZOOM system.

In order to examine the instrumentation overhead for parallel execution with 8 and 16 processors, we configure DSZOOM-WF as a single-node system with a cache-line-sized coherency unit of 128 bytes, i.e., a system without inter-node communication. This shows the overhead introduced by the inserted run-time in-line checks (ILCs) when there is no protocol activity. Table 2 presents the parallel binary instrumentation overheads for a single-node DSZOOM-WF configuration. The instrumentation overheads for 8-processor nodes averages 1.25 (the overhead increases by 2.5% compared to the sequential execution). For 16-processor nodes, this overhead averages 1.18 (the overhead decreases by 4.6% compared to the sequential execution).

4.4 Parallel Performance

This section presents the parallel performance of the applications for the DSZOOM-WF system. We report results for 2-node SMP clustering of 4 and 8 processors per node. We also characterize inserted overheads compared to many of the unmodified SMP applications by presenting the dynamic overheads for instrumented SPLASH-2 programs.

Figure 6 shows execution times in seconds for 8- and 16-processor runs for Sun Enterprise E6000, 2-node CC-NUMA, and three different DSZOOM configurations:

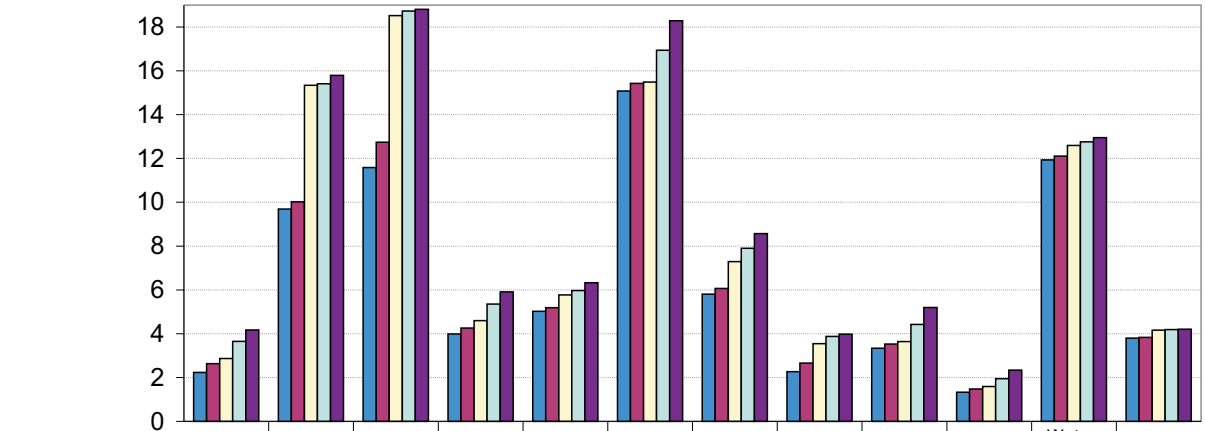
- **Single-node DSZOOM-WF**. This is a system without inter-node communication. It shows the effects of the inserted run-time in-line checks for global loads and stores as described in the previous section.
- **DSZOOM-EMU**. This is a system without any "real" physical memory and process distribution. This configuration emulates DSZOOM-WF between "virtual nodes," modeled as processes inside a single SMP multiprocessor. It shows effects of the protocol processing for a 2-node DSZOOM-WF system.
- **2-node DSZOOM-WF**. This configuration is a "real" DSZOOM-WF implementation. Both the memory and the running processes are physically distributed across the nodes. If we compare this configuration to the previous one, we can see how the Sun Orange interconnect impacts performance.

Cache-line-sized coherency unit of 128 bytes is used for all configurations. The performance of several 16-processor runs shown in Figure 6(b) is lower than expected. This is due to the contention on the SMP memory bus mainly caused by the misses from processors within that particular SMP node.

Table 3 shows efficency overheads for the effects of in-line checks, global coherence protocol processing, and the effects of the physical memory and process distribution across the nodes. The efficency overhead numbers are derived from Figure 6(a). On average, the run-time in-line checks are the largest efficiency overhead factor for 8-processor runs.

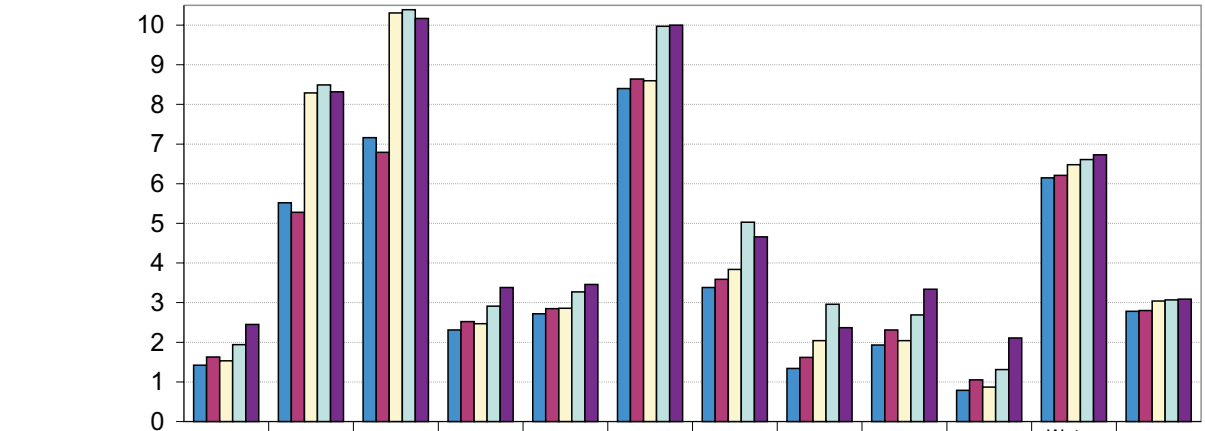
Normalized execution time breakdowns for 8- and 16-processor runs for a 2-node DSZOOM-WF with a coherency unit of 128 bytes are shown in Figure 7. The execution time is divided into *Task*, inserted run-time in-line checks (*ILC*) for global loads and stores, the synchronization cost (*Barriers* and *Locks*), and the cost of coherency protocol processing (*Store* and *Load*).

Our performance numbers presented so far are based on a constant cache-line-sized coherency unit of 128 bytes for all of the tested configurations. Choosing the different coherence granularity can potentially improve the performance for many applications (for example, see Shasta [35]). Table 4 reports



	FFT	LU-c	LU-nc	Radix	Barnes	FMM	Ocean-c	Ocean-nc	Radiosity	Raytrace	Water-nsq	Water-sp
E6000 8 CPUs	2.23	9.69	11.59	3.99	5.02	15.08	5.81	2.27	3.34	1.33	11.93	3.80
CC-NUMA 2x4	2.63	10.02	12.74	4.26	5.19	15.43	6.07	2.66	3.53	1.48	12.11	3.83
DSZOOM-WF 1x8	2.87	15.34	18.52	4.60	5.77	15.49	7.29	3.55	3.64	1.59	12.59	4.16
DSZOOM-EMU 2x4	3.65	15.41	18.73	5.35	5.97	16.94	7.90	3.88	4.42	1.95	12.76	4.19
DSZOOM-WF 2x4	4.17	15.79	18.81	5.91	6.33	18.29	8.57	3.98	5.20	2.34	12.95	4.21

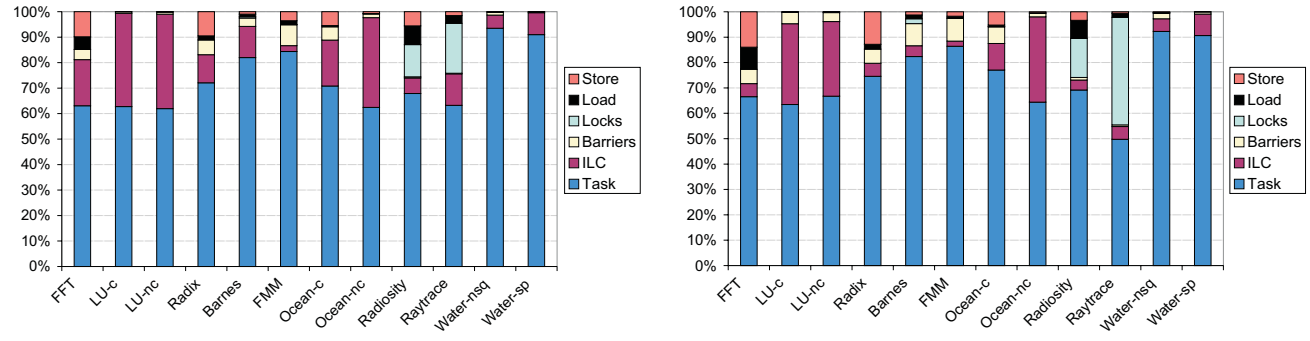
(a) 8 processors



	FFT	LU-c	LU-nc	Radix	Barnes	FMM	Ocean-c	Ocean-nc	Radiosity	Raytrace	Water-nsq	Water-sp
E6000 16 CPUs	1.42	5.52	7.16	2.31	2.72	8.40	3.38	1.34	1.93	0.79	6.15	2.78
CC-NUMA 2x8	1.63	5.28	6.79	2.52	2.85	8.64	3.59	1.62	2.31	1.05	6.21	2.80
DSZOOM-WF 1x16	1.53	8.29	10.31	2.47	2.86	8.60	3.84	2.04	2.04	0.87	6.48	3.04
DSZOOM-EMU 2x8	1.94	8.49	10.39	2.91	3.27	9.97	5.03	2.96	2.69	1.31	6.61	3.07
DSZOOM-WF 2x8	2.45	8.32	10.17	3.38	3.46	10.00	4.66	2.37	3.34	2.11	6.73	3.09

(b) 16 processors

Figure 6: Execution times in seconds for (a) 8- and (b) 16-processor runs for Sun Enterprise E6000, 2-node CC-NUMA, single-node DSZOOM-WF, 2-node DSZOOM-EMU, and 2-node DSZOOM-WF.



(a) 8 processors

(b) 16 processors

Figure 7: Normalized execution time breakdowns for (a) 8- and (b) 16-processor runs for a 2-node DSZOOM-WF with cache-line-sized coherency unit of 128 bytes.

Program	Unit Size [bytes]	Time [s]	
FFT	2 048	2.04	(+20.1%)
LU-c	1 024	8.22	(+1.2%)
Barnes	64	3.42	(+1.2%)
FMM	64	9.99	(+0.1%)
Ocean-c	256	4.35	(+7.1%)

Table 4: Effects of the coherency unit variations for a 2-node DSZOOM-WF with 8 processor nodes.

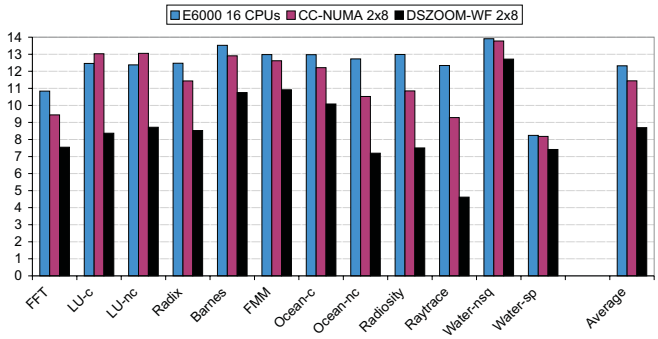


Figure 8: Application speedups for Sun Enterprise E6000, 2-node CC-NUMA, and 2-node DSZOOM-WF.

our experiments for several of the SPLASH-2 applications that have demonstrated performance improvements for a different coherency unit sizes on a 2 node DSZOOM-WF system with 8 processors per node. For example, if the FFT is executed with a cache-line-sized coherency unit of 2048 bytes, its overall performance is improved with 20.1% compared to the values presented in Figure 6(a).

The speedup values for 16-processor runs for Sun Enterprise E6000, 2-node CC-NUMA, and 2-node DSZOOM-WF with "optimal" coherency units are shown in Figure 8. The speedups shown are the ratio of the execution time of the application running on 16 processors to the execution time of the original sequential application (with no access control checks). We can see that our all-software solution is close to what hardware CC-NUMA architecture can do on the same platform. On average, our implementation demonstrates a relative difference for SPLASH-2 speedups of 31.6% compared to the hardware DSM implementation.

5 Related Work

Many different SW-DSM implementations have been proposed over the years: Blizzard-S [38], Brazos [40], Cashmere-2L [41], [10], CRL [19], GeNIMA [5], Ivy [26], [27], MGS [44], Munin [8], Shasta [35], [34], [32], [33], [10], Sirocco-S [36], SoftFLASH [11], and TreadMarks [21]. Most of them suffer from synchronous interrupt protocol processing. We belive that many of these implementations would benefit from a more efficient protocol implementation; such the one described here.

The DSZOOM-WF's basic approach is derived from several fine-grain SW-DSM systems: Shasta, Blizzard-S, and Sirocco-S. Our "magic"-value technique for fine-grain access control checks presented in Section 3 is similar to Shasta's "flag"-value and Blizzard's "sentinel"-value optimizations. This technique was independently introduced in Shasta [35] and Blizzard-S [38] for use with all types of loads. There are several other systems that use compiler-generated checks to implement a global address space (for example, Olden [7], Split-C [9], and Midway [4]).

Regarding the simple architectural support [17], the GeNIMA proposal is closest to our work [5], [14]. GeNIMA pro-

poses a protocol and a general network interface mechanism 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.

Several other papers have suggested hardware support for fine-grain remote write operations in the network interface [23], [22]. One of the recent implementations is the automatic update release consistency (AURC) home-based protocol [16]. This implementation is a page-based SW-DSM which eliminates "diffs"—the compact encoded representation of the differences between the two pages, frequently used in many page-based SW-DSM systems—by using fine-grain remote writes for both the application data and the protocol meta-data. The AURC approach usually performs better than all-software home-based LRC implementations.

6 Conclusions

In this paper we have presented the DSZOOM-WF system, an all-software (sequentially consistent) fine-grain SW-DSM implementation. 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 interconnect has been added.

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

DSZOOM-WF consistently demonstrates performance comparable to hardware-based DSM implementations. On average, the speedup difference between our implementation and the hardware CC-NUMA system is 31.6% for the studied SPLASH-2 applications.

7 Future Work

We plan to extend this work in several different directions. First, cache-coherence protocol code optimizations will improve 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 the 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 the coherence protocol routines and the run-time system.

Second, in order to improve the performance of the DSZOOM-WF system, weaker memory models, such as lazy release consistency (LRC) [20] and the release consistency model presented by Gharachorloo et al. [13], [35], can be used instead of the sequential consistency model that is currently implemented. This kind of optimization will allow many update actions to be deferred and combined into a single operation.

Third, we plan to experiment with several inter-node lock synchronization algorithms (e.g., ticket-based locks). The test-and-set locks that we are currently using work well for small-scale SMP nodes, but they are not adequate for large-scale, CC-NUMA nodes. Usually, test-and-set locks lead to poor caching performance and increased inter-node communication in many CC-NUMA systems. We believe that we can speed up many lock-intensive applications with improved synchronization algorithms.

Finally, to make this kind of system more usable it is desirable to make a POSIX-threads implementation 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, Ernest Artiaga (Technical University of Catalonia) for help with couple of PARMACS applications, Sverker Holmgren and Henrik Löf (Department of Scientific Computing, Uppsala University) for providing access to the Sun Orange system. We would also like to thank Lars Albertsson, Erik Berg, and Thiemo Voigt (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

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

[2] 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.

[3] 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.

[4] B. N. Bershad, M. J. Zekauskas, and W. A. Sawdon. The Midway Distributed Shared Memory System. In *Proceedings of the 38th IEEE Computer Society International Conference*, pages 528–537, February 1993.

[5] 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.

[6] A. Bilas and J. P. Singh. The Effects of Communication Parameters on End Performance of Shared Virtual Memory Clusters. In *Proceedings of Supercomputing '97*, November 1997.

[7] M. C. Carlisle and A. Rogers. Software Caching and Computation Migration in Olden. In *Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 29–38, July 1995.

[8] 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.

[9] D. E. Culler, A. Dusseau, S. C. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. Yelick. Parallel Programming in Split-C. In *Proceedings of Supercomputing '93*, pages 262–273, November 1993.

[10] 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.

[11] 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.

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

[13] 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.

[14] C. Gibson and A. Bilas. Performance of Shared Virtual Memory on Clusters of DSMs. In *Proceedings of the 8th International Conference on High Performance Computing (HiPC 2001)*, December 2001.

[15] 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.

[16] L. Iftode, M. Blumrich, C. Dubnicki, D. L. Oppenheimer, J. P. Singh, and K. Li. Shared Virtual Memory with Automatic Update Support. Technical Report TR-575-98, Princeton University, February 1998.

[17] L. Iftode and J. P. Singh. Shared Virtual Memory: Progress and Challenges. *Proceedings of the IEEE, Special Issue on Distributed Shared Memory*, 87(3):498–507, March 1999.

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

[19] 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.

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

[21] 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.

[22] L. Kontothanassis, G. Hunt, R. Stets, N. Hardavellas, M. Cierniak, S. Parthasarathy, W. Meira, S. Dwarkadas, and M. Scott. VM-based Shared Memory on Low-Latency, Remote-Memory-Access Networks. In *Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA'97)*, June 1997.

[23] L. Kontothanassis and M. Scott. Using Memory-Mapped Network Interfaces to Improve the Performance of Distributed Shared Memory. In *Proceedings of the 2nd IEEE Symposium on High Performance Computer Architecture*, February 1996.

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

[25] 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.

[26] 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.

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

[28] 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.

[29] 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.

[30] L. Noordergraaf and R. van der Pas. Performance Experiences on Sun's Wildfire Prototype. In *Proceedings of Supercomputing '99*, November 1999.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] 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 6th International Conference on Parallel Architectures and Compilation Techniques*, October 1998.

[37] 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.

[38] 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.

[39] 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.

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

[41] 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.

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

[43] 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.

[44] 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.