DSZOOM – Low Latency Software-Based Shared Memory

by

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

Parallel and Scientific Computing Institute

Royal Institute of Technology
and
Uppsala University

Report No. 2001:03

April, 2001

DSZOOM – Low Latency 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

April, 2001

Abstract

Software-implementations of shared memory are still far behind the performance of hardware-based shared memory implementations 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. Elaborate schemes, involving dedicated hardware and/or dedicated protocol processors, have been suggested to cut the overhead.

This paper describes how all the asynchronous overhead can be completely removed by instead 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 shared memory.

Our proof-of-concept implementation—DSZOOM-EMU—is a fine-grained 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 only some basic low-level primitives in the cluster interconnect. Based on a remote atomic and simple remote put/get operations the requesting processor can assume the role of the directory agent, traditionally assumed by a remote protocol agent in the home node in other implementations. The implementation is thread-safe and allows all processors in a node to simultaneously perform remote operations.

Contents

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-sized coherence units in the earlier SW-DSM systems created extra false sharing and caused frequent 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 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 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 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 [BS97], [IS99]. 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 concentrate on fine-grain SW-DSM in this paper.

We have implemented the DSZOOM-EMU system, our initial proof-of-concept DSZOOM implementation that emulates fine-grain software-based DSM between "virtual nodes," modeled as processes inside a single SMP. 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 a coherence protocol implemented in C that copies data to the node's local memory by performing loads and stores from and to remote memory. We have measured the actual protocol overhead to be less than a microsecond for a "remote load." Latency loops have been inserted into our protocol in order to model the latency of realistic networks. We have also modeled latencies of more traditional interrupt-driven SW-DSM implementations. A total of nine unmodified SPLASH-2 applications [WOT+95], developed for fine-grain hardware SMP multiprocessors, are studied. We compare the performance of a DSZOOM-EMU system with that of a Sun Enterprise E6000 SMP server [SBC+96] as well as the hardware-coherent Sun Orange (earlier referred to as Sun WildFire) DSM system [HK99], [NvdP99].

The remainder of this paper is organized as follows. Section 2 presents our basic idea and an introduction to a general DSZOOM system. The proof-of-concept implementation—DSZOOM-EMU—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 Basic Idea

2.1 Cluster Networks with Put/Get Semantics

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

[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.

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 a node. This removes the needs for the complicated coherence scheme implemented in hardware and allows the NIC to be connected to an 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.

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.

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 DSZOOM 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 DSZOOM 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 [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, 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. [MFHW96].

2.4 Protocol Details

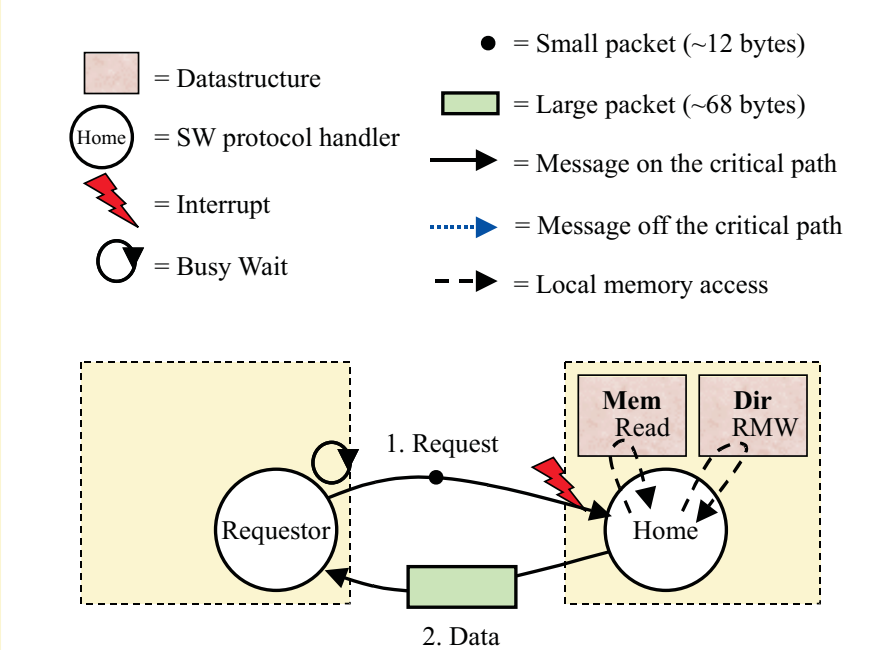
The SMP hardware keeps the coherence within the node, on top of which the global DSZOOM protocol has been added. All the coherence activities and state names discussed in this paper apply to the DSZOOM protocol.

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.

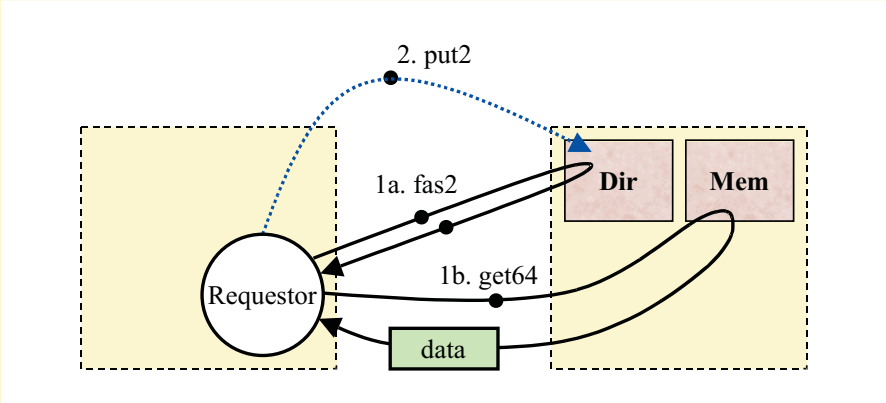
To also avoid most of the accesses to the directory caused by global store operations, each node has two bytes of local state (MTAG) per global cache line (similar to the *private state table* found in Shasta [SGA97]), indicating if the cache line is locally writable. Before each global store operation, 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.

Figure 1a illustrates the difference between DSZOOM and a more traditional SW-DSM by showing the the activity caused by a read miss in a two-node system. The traditional software DSM's

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



(a) Traditional SW-DSM



(b) DSZOOM

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

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.

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)
}
```

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, as shown in Figure 1b. 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 CPU time in the home node, its major drawback is the global bandwidth consumed.

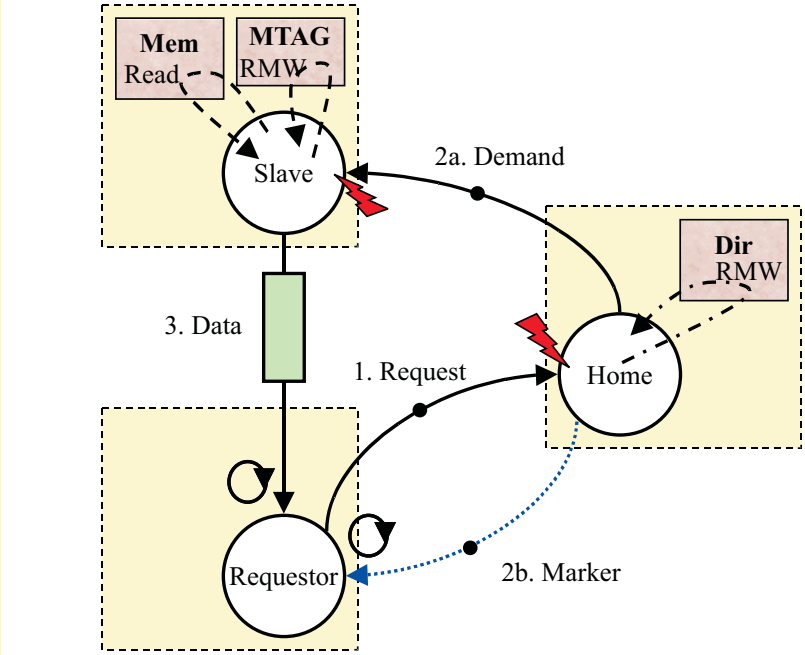
To illustrate the excess bandwidth consumed by DSZOOM, each global packet has been marked as either a "small packet," with a payload of less than 6 bytes, or a "large packet," with a payload of 64 bytes.[3] Each packet type is assumed to also carry 2 bytes of cyclic redundancy code (CRC) and 2 bytes 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. The traditional SW-DSM approach will need two asynchronous interrupts on the critical path before the data is forwarded to the requesting node. DSZOOM will need one fas2 message to lock and acquire the directory and

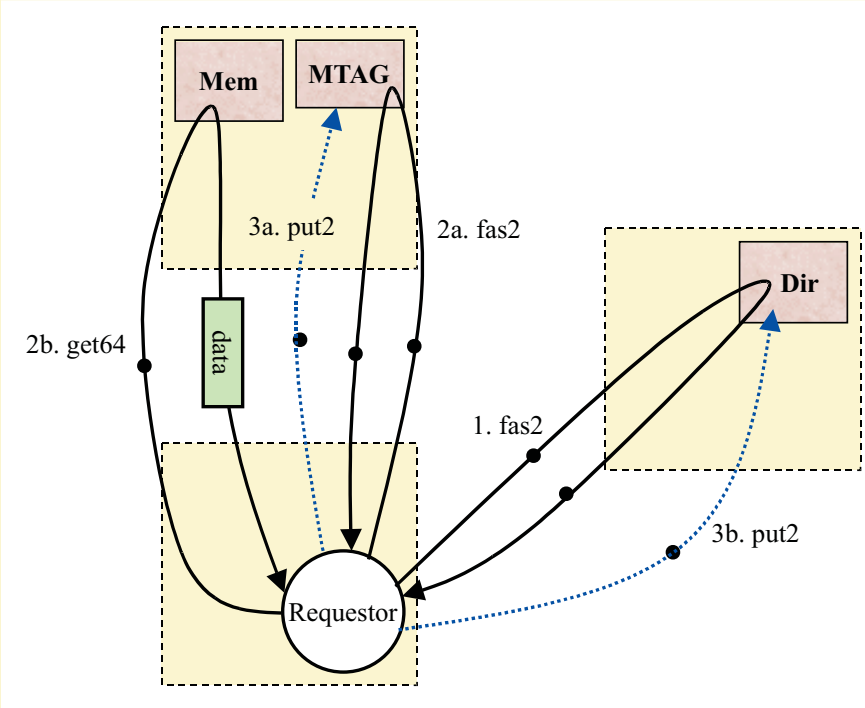
[3] We have not included the implicit acknowledge packets that may be used by the lower level network implementation.

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)
}
```



(a) Traditional SW-DSM



(b) DSZOOM

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

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 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 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.

3 DSZOOM-EMU: Proof-of-Concept Implementation

This section describes our proof-of-concept implementation. DSZOOM-EMU is a sequentially consistent [Lam79] fine-grain SW-DSM. This implementation emulates fine-grain software-based DSM between "virtual nodes," modeled as processes inside a single Sun Enterprise E6000 SMP. Thus, we logically divided one SMP into several "imaginary" SMPs, each one with its own address space, and rely on the DSZOOM protocols to keep the coherence between the address space. To make this kind of implementation more realistic we model the network delays for our virtual cluster as well by inserting some extra "dummy-loops" into the cache coherence protocol routines and the synchronization part of a DSZOOM-EMU run-time system. DSZOOM-EMU 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 PARMACS macros with DSZOOM-EMU run-time system calls. A standard GNU gcc compiler is used to compile and link the preprocessed file with a DSZOOM-EMU 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) [LS95]. 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.

3.1 Setting Up the Memory-Mapped Communication

Modified/Extended PARMACS macros are responsible for, among many other things, setting up the memory-mapped communication between the processes inside one SMP. Address space layout and attachment of the shared memory objects for every process running in the home node (`NODE_ID` = 0) is shown in Figure 4. Shared memory objects with shared memory identifiers *A*, *B*, *C*, and *D* represents the physical shared/global memory of every node in the cluster. Shared memory identifier *E*, 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-EMU process identifier, UNIX process identifier, etc.) is stored in a privately mapped `L_NODE_DATA` area in a current implementation, and is also used mainly for debugging and profiling purposes. Directory is placed at the beginning of the `G_MEM` area.

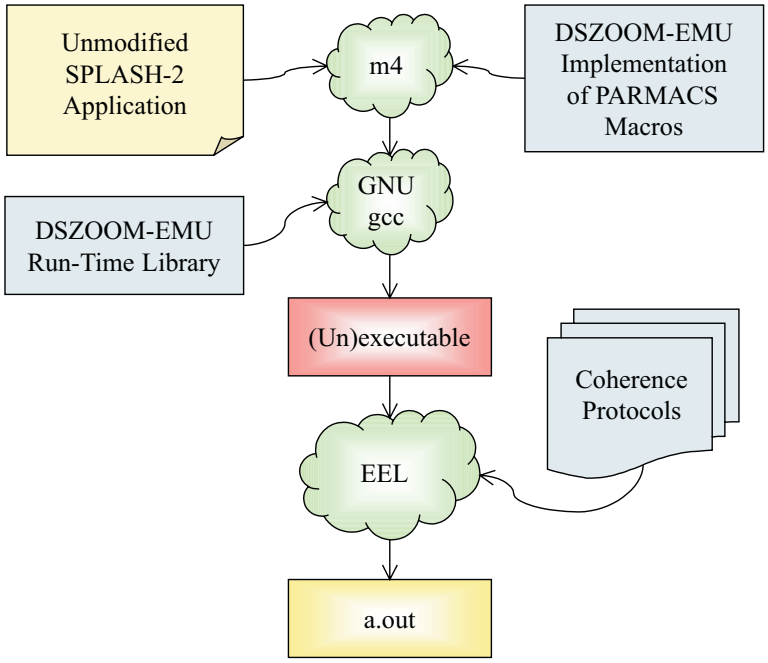


Figure 3: DSZOOM-EMU compilation process.

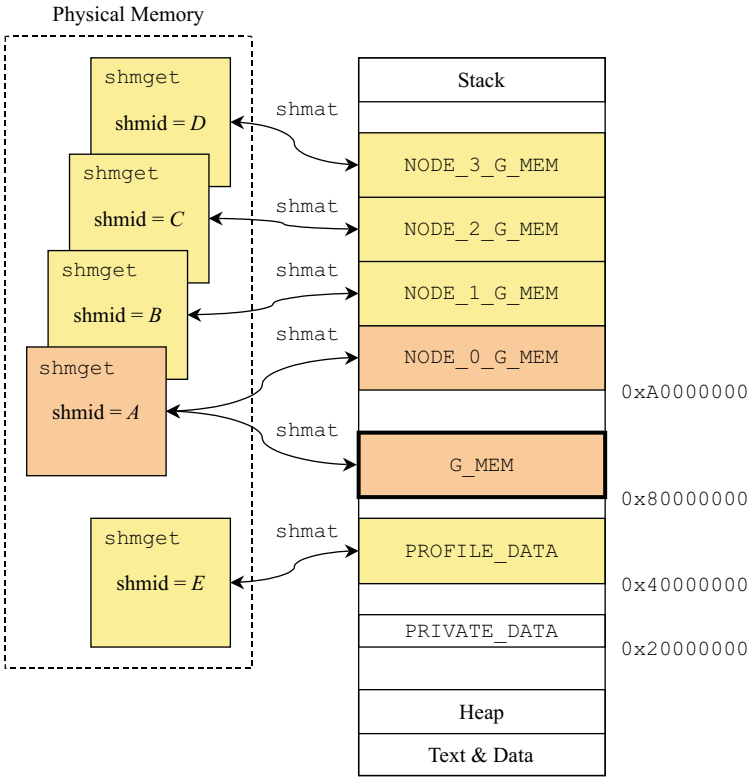


Figure 4: Address space layout and attachment of the shared memory objects for `NODE_ID` = 0. This figure assumes that the virtual SMP cluster consists of four nodes.

```
  1: ld      [%o1 + 64], %DEST_REG       //original LD
  2: sethi   %hi(._start_FFT.EXE), %temp //prepare for jmpl
  3: srl     %DEST_REG, 24, %g5          //mask "magic"
  4: cmp     %g5, 0xAA                   //check if "magic"
  5: bne     hit:                        //if not, it is a hit
  6: mov     %DEST_REG, %g6              //prepare jmpl
  7: jmpl    %temp, %o7                  //jump link to C routine
  8: add     %o1 + 64, %g5               //pass addr to C routine
  9: mov     %g6, %DEST_REG              //move LD value from C
hit:
```

Figure 5: Replacing one `lduw` instruction (partly-optimized). The `%temp` register is allocated with our binary modification tool from dead or freed registers at the insertion point. Only instructions 1–5 are executed in cases when the loaded data is valid (i.e., the `%DEST_REG` is a non-magic value). In cases when the data is a magic-value, first then the cache coherence handler routine will be invoked (with instruction 7). False misses almost never occur in practice.

3.2 Inserting the Cache-Coherence Protocol Code into Binaries

There are at least three ways of inserting cache-coherence protocol code into benchmark binaries. The most classical way of doing this is to make a compiler modification to perform that particular task, e.g., create a compiler backend that is capable of exchanging all relevant loads and stores with corresponding code snippets for those loads and stores. There are several systems that use compiler-generated checks, for example, Olden [CR95], Split-C [CDG+93], and Midway [BZS93]. Dynamic code instrumentation (e.g., *JiTI*: a Robust Just in Time Instrumentation Technique [RB00]) is another technique that also is capable of performing this task. The third alternative, 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 purpose of the proof-of-concept DSZOOM implementation is to demonstrate the implementation of a low-overhead global SW-DSM protocol, which is applicable to both page-based software-based DSMs and fine-grain SW-DSMs. The code in Figure 5 shows the code snippet replacing one particular global load instruction. 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.

3.2.1 Modeling the Network

SW-DSM is run very efficiently in a single SMP node. In order to model a more realistic set-up with real network delay, the protocol implemented in C code have extra latency loops inserted. A "remote" access has about 2 microseconds of extra latency added to model the expected latency of a remote network.

We also wanted to compare our SW-DSM implementation to one with a more common, still short, remote latency caused by the extra protocol overhead. We have used the shortest latency reported to date as our benchmark number: 15 microseconds (Shasta [SGA97], [Gha00]). This is modeled as extra network delay. The extra CPUs occupancy by the protocol agent in the remote end have not been taken into account, nor have we modeled any contention effects from single threaded agent in that scheme.

4 Performance Study

In this section we describe experimental setup, applications used in this study, and finally, we present DSZOOM-EMU 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 Orange built from two E6000 nodes connected through a hardware-coherent interface with a raw bandwidth of 800 Mbyte/s in each direction [HK99], [NvdP99]. 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.

4.2 Applications

The benchmarks we use in this study are well-known scientific workloads from the SPLASH-2 benchmark suite [WOT+95]. We study a total of nine SPLASH-2 applications (that do not require any modifications) from the original Stanford University distribution, which were originally developed for hardware multiprocessors. The applications are: *Barnes-Hut* (hierarchical N-body method), *FFT* (complex 1-D version of the radix-$\sqrt{n}$ six-step FFT algorithm [Bai90]), *LU* (blocked LU decomposition, see [WSH94] for more details), *CLU* (blocked LU decomposition with contiguous allocation of data, more optimized version of LU), *Radix* (integer radix sort kernel), *Radiosity* (iterative hierarchical diffuse radiosity method [HSA91]), *Raytrace* (rendering of a three-dimensional scene using ray tracing), *Water-nsq* (water simulation without spatial data structure), and *Water-sp* (water simulation with spatial data structure, this application solves the same problem as Water-nsq, but uses a more efficient algorithm). The benchmarks were compiled with System V IPC version of the PARMACS shared-memory macros used by Artiaga et al. [ANMB97], [AMBN98]. The macro library was modified in several ways, for example we use user-level synchronization through test-and-set locks instead of System V IPC semaphore library calls. We also began all measurements at the start of the parallel phase to exclude DSZOOM-EMU's run-time system initialization time.

The reason why we cannot run the entire SPLASH-2 application suite 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.[4]

The data-set sizes and uniprocessor-execution times for the studied SPLASH-2 applications are presented in Table 1.

[4] Artiaga has experienced similar problems with the original fork-exec versions of several applications [Art01].

Program	Problem Size	Non-Instrumented Sequential Time [s]
FFT	1,048,576 points (48.1 MB)	15.51
CLU	1024×1024, block 16 (8.0 MB)	75.45
LU	1024×1024, block 16 (8.0 MB)	85.85
Radix	4,194,304 items (36.5 MB)	28.88
Barnes-Hut	16,384 bodies (32.8 MB)	37.12
Radiosity	Test (29.4 MB)	8.62
Raytrace	Teapot (32.2 MB)	6.28
Water-nsq	2197 molecules, 2 steps (2.0 MB)	86.73
Water-sp	2197 molecules, 2 steps (1.5 MB)	23.08

Table 1: Data-set sizes and sequential-execution times for the studied SPLASH-2 applications.

Program	% LD	% ST	Instrumented Sequential Time [s]	Efficiency Overhead Total (LD%) (ST%)
FFT	40.5	23.5	22.92	1.48 (17%) (83%)
CLU	40.2	14.8	140.05	1.86 (41%) (59%)
LU	38.8	17.1	154.08	1.79 (49%) (51%)
Radix	46.1	17.3	41.16	1.43 (63%) (37%)
Barnes-Hut	52.1	51.9	65.59	1.77 (87%) (13%)
Radiosity	41.4	35.2	14.36	1.67 (90%) (10%)
Raytrace	41.4	35.2	6.28	1.67 (90%) (10%)
Water-nsq	45.0	32.5	166.21	1.92 (96%) (4%)
Water-sp	43.7	27.7	44.54	1.93 (90%) (10%)

Table 2: Data-set sizes and sequential-execution times for instrumented SPLASH-2 applications with an efficiency overhead shown in the last column. Efficiency overhead is divided into dynamic values produced by load and store snippets during the run-time. Second and third columns show percentage of statically instrumented loads, and stores respectively.

4.3 DSZOOM-EMU Performance Overview

Sequential-execution times for the instrumented SPLASH-2 programs are shown in Table 2. Efficiency overhead is between 1.43 and 1.93 for all of the studied applications, i.e., instrumented code takes between 43% and 93% longer time to execute the programs. State-of-the-art checking overheads (for example in Shasta [SGA97]) are in the range of 5-35%. Unfortunately, increased software checking overhead gives us a bad starting point for many of the applications.

All experiments presented here are performed with 8 and 16 processors. Figure 6 shows the results of our performance study for the four different configurations; (*i*) 8-processor runs on E6000 SMP server; (*ii*) CC-NUMA 2-node configuration, 4 processors per node; (*iii*) DSZOOM-EMU 2×4 (i.e., 2 nodes, 4 processors per node) with 3 microseconds network delay; and (*iv*) DSZOOM-EMU 2×4 with 15 microseconds network delay. Speedup values are calculated relatively execution times from Table 1.

Figure 7 shows the results for 16-processor configurations; (*i*) 16-processor SMP; (*ii*) CC-NUMA 2×8 (i.e., 2 nodes, 8 processors per node); (*iii*) DSZOOM-EMU 2×8, 3 microseconds delay; and (*iv*) DSZOOM-EMU 2×8, 15 microseconds delay. Also here, speedup values are calculated relatively execution times from Table 1.

The DSZOOM-EMU simulations with 1, 2, 4, and 8 nodes are also performed on a Sun Enterprise E6000 SMP server and the results are shown in Figure 8 (8-processor runs) and Figure 9 (16-processor runs) respectively. Note that the speedup values for DSZOOM-EMU systems in both Figure 8 and Figure 9 are calculated relatively values from Table 2. Network delay is about 3 microseconds for all DSZOOM-EMU configurations.

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 belive that many of these implementations would benefit from a more efficient protocol implementation; such the one described here.

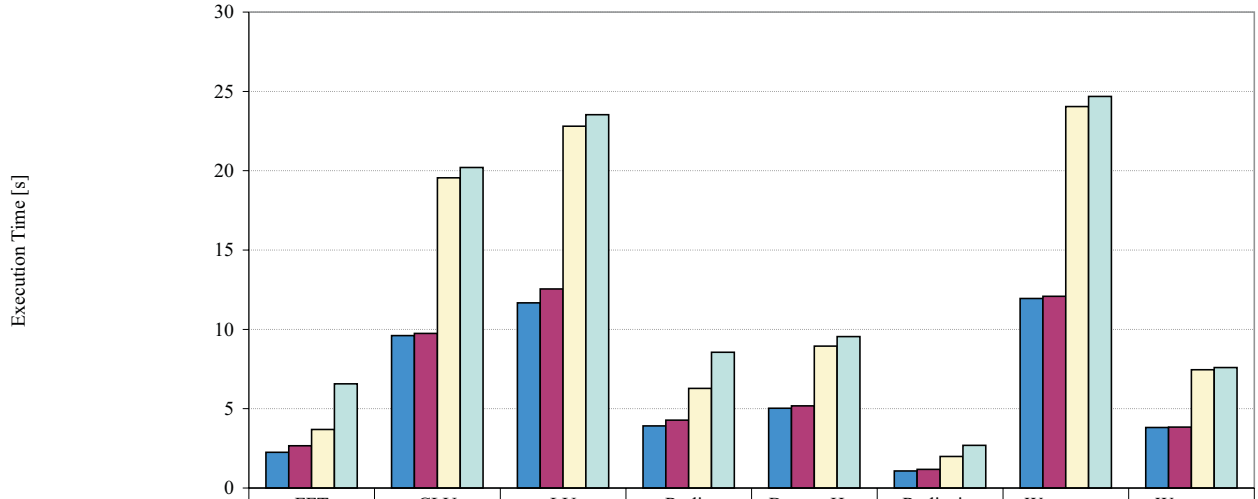
Regarding the simple architectural support [IS99], the GeNIMA proposal is closest to our work. GeNIMA proposes a protocol and a general network interface mechanism to avoid some of the asynchronous overhead. A processor starting a synchronous communication event, e.g., the requesting processor initiating 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 [KS96], [KHS+97]. One of the recent implementations is the automatic update release consistency (AURC) home-based protocol [IBD+98]. 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

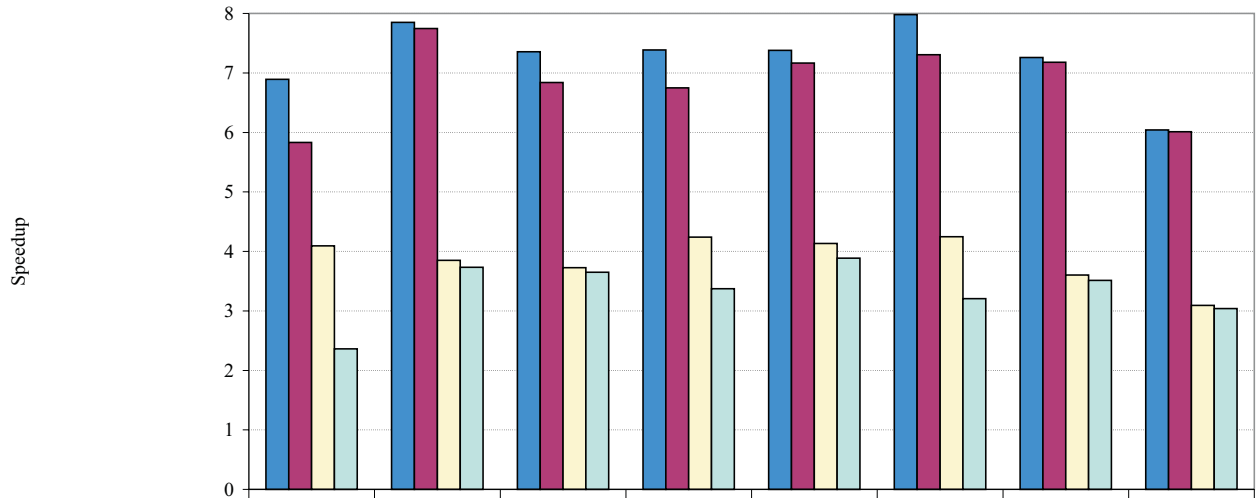
We have demonstrated how asynchronous protocol processing can be completely avoided at the cost of some extra remote transactions—trading bandwidth for efficiency. The entire protocol processing for remote SW-DSM load operation on our initial DSZOOM implementation has been measured to be below 800 nanoseconds on a 4-way 400 MHz UltraSPARC II SMP system. 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. We demonstrate a substantial improvement in speedup for many of the SPLASH-2 applications when we compare a modeled three microsecond SDSM system with the current state-of-the-art 15-microsecond.

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.



	FFT	CLU	LU	Radix	Barnes-Hut	Radiosity	Water-nsq	Water-sp
SMP 8 CPUs	2.25	9.61	11.67	3.91	5.03	1.08	11.95	3.82
CC-NUMA 2x4	2.66	9.74	12.55	4.28	5.18	1.18	12.08	3.84
DSZOOM-EMU 2x4, 3us	3.69	19.56	22.81	6.28	8.95	1.99	24.04	7.45
DSZOOM-EMU 2x4, 15us	6.57	20.21	23.53	8.56	9.55	2.69	24.68	7.59

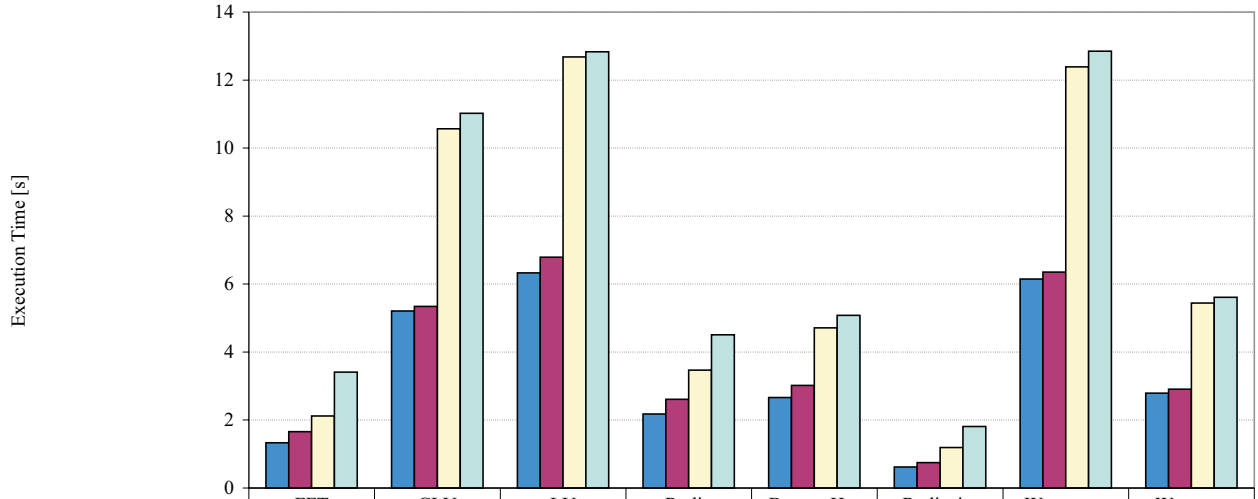
(a) Execution Time



	FFT	CLU	LU	Radix	Barnes-Hut	Radiosity	Water-nsq	Water-sp
SMP 8 CPUs	6.89	7.85	7.36	7.39	7.38	7.98	7.26	6.04
CC-NUMA 2x4	5.83	7.75	6.84	6.75	7.17	7.31	7.18	6.01
DSZOOM-EMU 2x4, 3us	4.09	3.85	3.73	4.24	4.13	4.25	3.60	3.09
DSZOOM-EMU 2x4, 15us	2.36	3.73	3.65	3.37	3.89	3.20	3.51	3.04

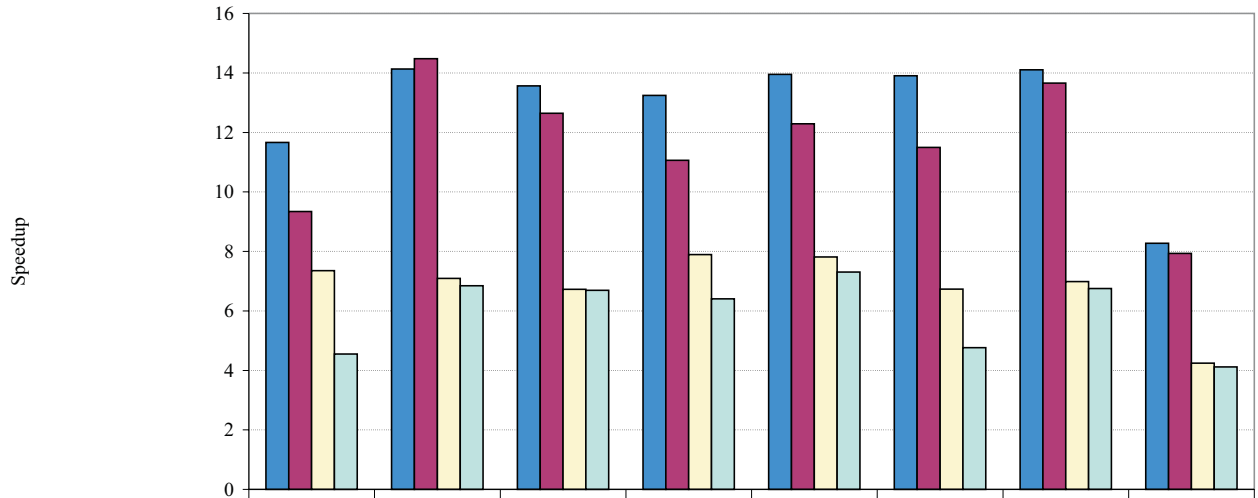
(b) Speedup

Figure 6: Execution times and speedup values for SMP 8-processor runs, CC-NUMA 2×4, DSZOOM-EMU 2×4 (3 μs network delay), and DSZOOM-EMU 2×4 (15 μs network delay). Note that all speedup values are calculated relatively execution times from Table 1.



	FFT	CLU	LU	Radix	Barnes-Hut	Radiosity	Water-nsq	Water-sp
SMP 16 CPUs	1.33	5.21	6.33	2.18	2.66	0.62	6.15	2.79
CC-NUMA 2x8	1.66	5.34	6.79	2.61	3.02	0.75	6.35	2.91
DSZOOM-EMU 2x8, 3us	2.12	10.57	12.68	3.47	4.71	1.19	12.39	5.44
DSZOOM-EMU 2x8, 15us	3.41	11.02	12.83	4.51	5.08	1.81	12.85	5.61

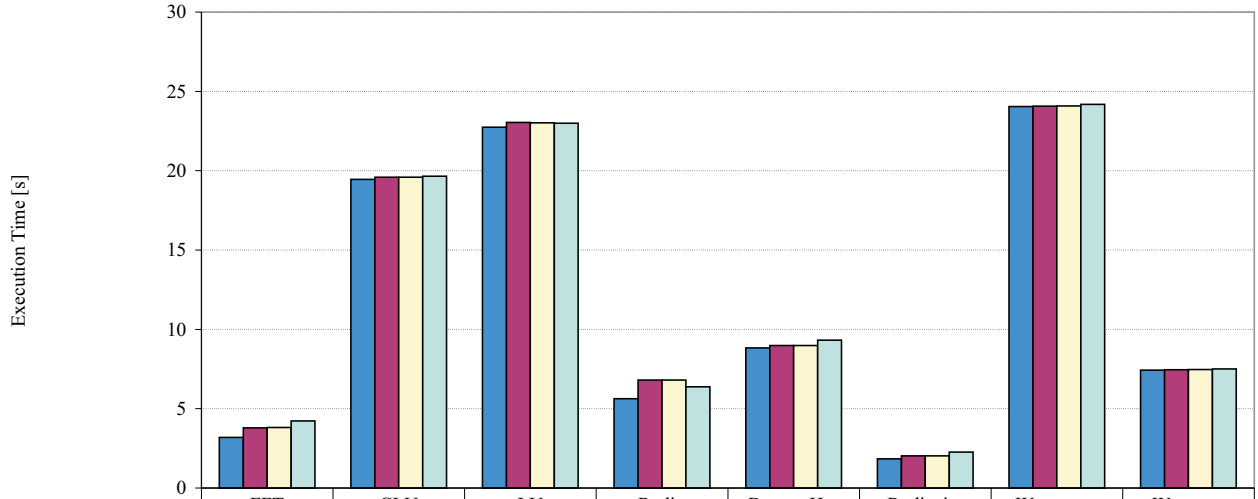
(a) Execution Time



	FFT	CLU	LU	Radix	Barnes-Hut	Radiosity	Water-nsq	Water-sp
SMP 16 CPUs	11.66	14.13	13.56	13.25	13.95	13.90	14.10	8.27
CC-NUMA 2x8	9.34	14.48	12.64	11.07	12.29	11.49	13.66	7.93
DSZOOM-EMU 2x8, 3us	7.35	7.09	6.72	7.89	7.81	6.73	6.98	4.24
DSZOOM-EMU 2x8, 15us	4.55	6.85	6.69	6.40	7.31	4.76	6.75	4.11

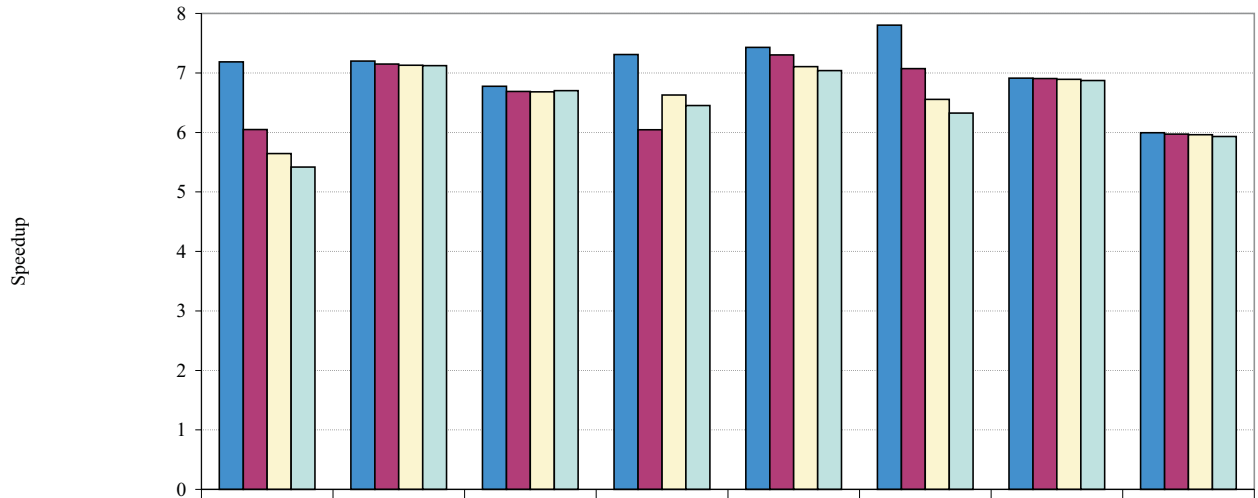
(b) Speedup

Figure 7: Execution times and speedup values for SMP 16-processor runs, CC-NUMA 2×8, DSZOOM-EMU 2×8 (3 μs network delay), and DSZOOM-EMU 2×8 (15 μs network delay). Note that all speedup values are calculated relatively execution times from Table 1.



	FFT	CLU	LU	Radix	Barnes-Hut	Radiosity	Water-nsq	Water-sp
DSZOOM-EMU 1x8	3.19	19.45	22.74	5.63	8.83	1.84	24.05	7.43
DSZOOM-EMU 2x4	3.79	19.59	23.04	6.81	8.98	2.03	24.07	7.46
DSZOOM-EMU 4x2	3.81	19.59	23.02	6.81	8.98	2.03	24.08	7.47
DSZOOM-EMU 8x1	4.23	19.66	22.99	6.38	9.32	2.27	24.18	7.51

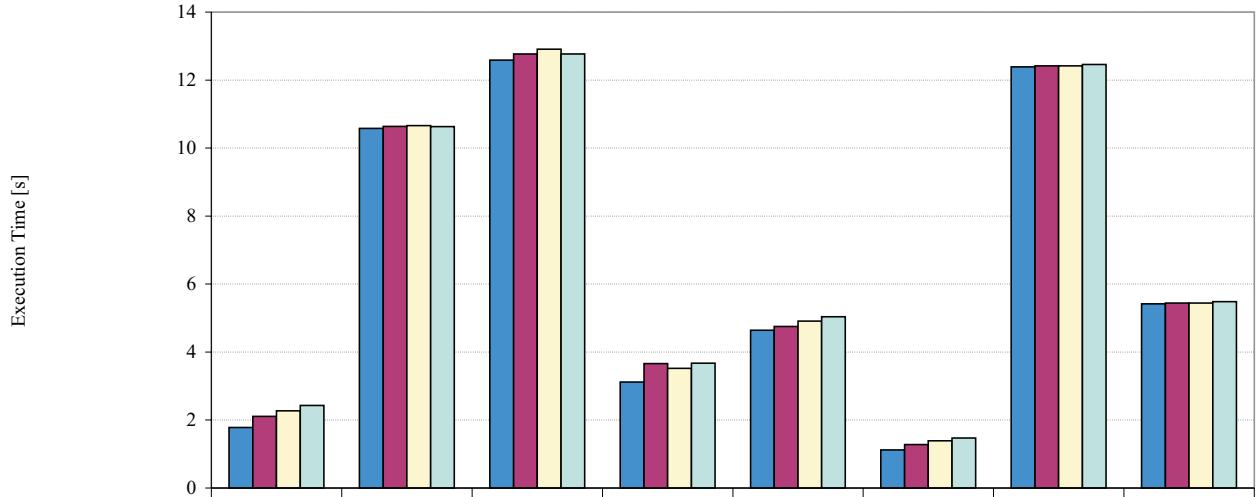
(a) Execution Time



	FFT	CLU	LU	Radix	Barnes-Hut	Radiosity	Water-nsq	Water-sp
DSZOOM-EMU 1x8	7.18	7.20	6.78	7.31	7.43	7.80	6.91	5.99
DSZOOM-EMU 2x4	6.05	7.15	6.69	6.04	7.30	7.07	6.91	5.97
DSZOOM-EMU 4x2	5.65	7.13	6.68	6.63	7.11	6.56	6.89	5.96
DSZOOM-EMU 8x1	5.42	7.12	6.70	6.45	7.04	6.33	6.87	5.93

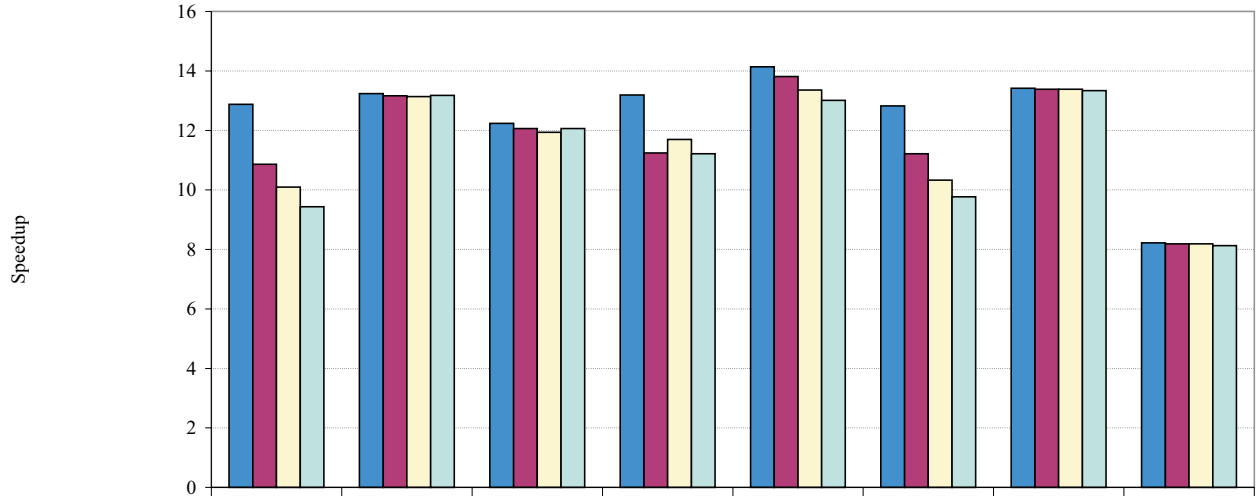
(b) Speedup

Figure 8: DSZOOM-EMU with 8 processors; 1, 2, 4, and 8 node simulations. Note that speedup values for DSZOOM-EMU systems are calculated relatively execution times from Table 2. Network delay is about 3 μs for all configurations presented here.



	FFT	CLU	LU	Radix	Barnes-Hut	Radiosity	Water-nsq	Water-sp
DSZOOM-EMU 1x16	1.78	10.58	12.59	3.12	4.64	1.12	12.39	5.42
DSZOOM-EMU 2x8	2.11	10.64	12.77	3.66	4.75	1.28	12.42	5.44
DSZOOM-EMU 4x4	2.27	10.66	12.91	3.52	4.91	1.39	12.42	5.44
DSZOOM-EMU 8x2	2.43	10.63	12.77	3.67	5.04	1.47	12.46	5.48

(a) Execution Time



	FFT	CLU	LU	Radix	Barnes-Hut	Radiosity	Water-nsq	Water-sp
DSZOOM-EMU 1x16	12.88	13.24	12.24	13.19	14.14	12.82	13.41	8.22
DSZOOM-EMU 2x8	10.86	13.16	12.07	11.25	13.81	11.22	13.38	8.19
DSZOOM-EMU 4x4	10.10	13.14	11.93	11.69	13.36	10.33	13.38	8.19
DSZOOM-EMU 8x2	9.43	13.17	12.07	11.22	13.01	9.77	13.34	8.13

(b) Speedup

Figure 9: DSZOOM-EMU with 16 processors; 1, 2, 4, and 8 node simulations. Note that speedup values for DSZOOM-EMU systems are calculated relatively execution times from Table 2. Network delay is about 3 μs for all configurations presented here.

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, we plan to port DSZOOM-EMU to a real cluster interconnect with remote load/store semantics and remote fetch-and-set and/or fetch-and-add capabilities (e.g., to begin with, we can more accurately model up to four-node configuration on a Sun Orange [HK99] prototype SMP cluster by simply binding the processes/threads to arbitrary node with Solaris system call `pset_bind`).

Third, in order to improve the performance of the DSZOOM-WF system, weaker memory models, such as lazy release consistency (LRC) [Kel95] and the release consistency model presented by Gharachorloo et al. [GLL+90], [SGT96], 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.

Fourth, 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, 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 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.

[Bai90] D. H. Bailey. FFT's in External or Hierarchical Memory. *Journal of Supercomputing*, 4(1):23–35, March 1990.

[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.

[BS97] 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.

[BZS93] 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.

[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.

[CDG+93] 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.

[CR95] 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.

[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.

[HSA91] P. Hanrahan, D. Salzman, and L. Aupperle. Rapid Hierarchical Radiosity Algorithm. In *Proceedings of SIGGRAPH*, 1991.

[IBD+98] 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.

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

[IS99] 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.

[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.

[KHS+97] 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.

[KS96] 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.

[Lam79] L. Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.

[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.

[NvdP99] L. Noordergraaf and R. van der Pas. Performance Experiences on Sun's Wildfire Prototype. In *Proceedings of Supercomputing '99*, November 1999.

[RB00] M. Ronsse and K. De Bosschere. JiTI: A Robust Just in Time Instrumentation Technique. In *Proceedings of the Workshop on Binary Translation*, October 2000.

[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 the 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.

[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.

[WSH94] S. C. Woo, J. P. Singh, and J. L. Hennessy. The Performance Advantages of Integrating Block Data Transfer in Cache-Coherent Multiprocessors. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI)*, pages 219–229, October 1994.

[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.