Shared Memory Computing on Clusters with Symmetric Multiprocessors and System Area Networks

Leonidas Kontothanassis, HP Labs
Robert Stets, Google, Inc.
Galen Hunt, Microsoft Research
Umit Rencuzogullari, VMware, Inc.
Gautam Altekar, University of California, Berkeley
Sandhya Dwarkadas and Michael L. Scott, University of Rochester

Cashmere is a software distributed shared memory (S-DSM) system designed for clusters of server-class machines. It is distinguished from most other S-DSM projects by (1) the effective use of fast user-level messaging, as provided by modern system-area networks, and (2) a "two-level" protocol structure that exploits hardware coherence within multiprocessor nodes. Fast user-level messages change the tradeoffs in coherence protocol design; they allow Cashmere to employ a relatively simple directory-based coherence protocol. Exploiting hardware coherence within SMP nodes improves overall performance when care is taken to avoid interference with inter-node software coherence.

We have implemented Cashmere on a Compaq AlphaServer/Memory Channel cluster, an architecture that provides fast user-level messages. Experiments indicate that a one-level version of the Cashmere protocol provides peformance comparable to, or slightly better than, that of lazy release consistency. Comparisons to Compaq's Shasta protocol also suggest that while fast user-level messages make finer-grain software DSMs competitive, VM-based systems continue to outperform software-based access control for applications without extensive fine grain sharing.

Within the family of Cashmere protocols we find that leveraging intra-node hardware coherence provides a 25% performance advantage over a more straightforward one-level implementation. Moreover, contrary to our original expectations, non-coherent hardware support for remote memory writes, total message ordering, and broadcast provide comparatively little in the way of additional benefits over just fast messaging for our application suite.

Authors' addresses: Leonidas Kontothanassis, Cambridge Research Lab, Hewlett-Packard Corporation, One Cambridge Center, Cambridge, MA 02139 (kthanasi@hp.com); Robert Stets, Google Inc., 2400 Bayshore Parkway, Mountain View, CA 94043 (stets@google.com); Galen Hunt, Microsoft Research, One Microsoft Way, Redmond, WA 98052 (galenh@microsoft.com); Umit Rencuzogullari, VMware, Inc., 3145 Porter Drive, Palo Alto, CA 94304 (umit@vmware.com); Gautam Altekar, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, Berkeley, CA 94720-1776 (galtekar@eecs.berkeley.edu); Sandhya Dwarkadas, and Michael L. Scott, Department of Computer Science, University of Rochester, Rochester, NY 14627-0226 ({sandhya,scott}@cs.rochester.edu).
This work was supported in part by NSF grants CDA–9401142, EIA–9972881, EIA–0080124, CCR–9702466, CCR–9705594, and CCR-9988361; by DARPA/AFRL contract number F29601-00-K-0182; and by a pair of external research grants from Compaq.

1. INTRODUCTION

The high performance computing community has relied on parallel processing for over twenty years. At both the architectural and program levels it is customary to categorize parallel systems as shared memory or message based. Message based systems are typically easier to construct but more difficult to program; the inverse is true for their shared memory counterparts. Small-scale symmetric multiprocessors have made shared memory popular and cost-effective for a variety of low-end parallel systems, and there is a strong incentive to provide an easy migration path to higher-performance systems.

At the very high end, where hardware remains the dominant cost, and programmers are willing to invest as much time as necessary to achieve the maximum possible performance, it is likely that message based hardware and software will continue to dominate for the indefinite future. Shared memory, however, is increasingly popular in the mid-range world. Several vendors, including SGI, Sun, IBM, and HP, sell cache-coherent, non-uniform memory access (CC-NUMA) machines that scale to dozens or even a few hundreds of processors. Unfortunately, these machines have significantly higher per-processor costs than their smaller SMP cousins. As a consequence, many organizations are opting instead to deploy *clusters* of SMPs, connected by high-bandwidth, low-latency system-area networks (SANs) when their applications allow for such substitution.

Software distributed shared memory (S-DSM) attempts to bridge the gap between the conceptual appeal of shared memory and the price-performance of message passing hardware by allowing shared memory programs to run on non-shared memory clusters. S-DSM as a research field dates from the thesis work of Kai Li in the mid 1980s [Li89b]. Li's Ivy system, designed for a network of Apollo workstations, and its successor, Shiva [Li89a], for the Intel Hypercube, used the virtual memory system to emulate a simple cache coherence protocol. Accesses to non-local data would result in read or write page faults, treated like non-exclusive or exclusive misses in a traditional invalidation-based coherence protocol [Goo87].

Counting Ivy and Shiva as the first, we can identify four different generations of S-DSM systems. The second generation sought to improve performance through protocol optimizations and relaxed memory models. The Munin project [BCZ90; CBZ91] at Rice University showed how to tune the coherence protocol to the sharing patterns of the application. Rice's subsequent TreadMarks project [ACD96; KCZ92] achieved further performance improvements by introducing the concept of lazy release consistency, in which multiple processes can modify (separate portions of) a page at the same time, sending updates only to those processes that are later determined to be causally "downstream". TreadMarks has been widely distributed to sites throughout the world. It is now commercially available, and has become the standard against which other S-DSM systems are judged.

Third-generation S-DSM systems allow coherence blocks to correspond to semantically meaningful data structures, rather than VM pages, thereby eliminating the *false sharing* that can lead to spurious coherence overhead when independent shared structures lie in a single page. In return, third-generation systems typically impose a more restrictive programming model [HLR93; JKW95; SGZ93; ZSB94].

Fourth-generation S-DSM systems attempt to improve the performance of second

generation systems and remove the programming limitations of the third generation through inexpensive hardware support, finer grain access control, or both. Example systems include the Princeton Shrimp [BLA94], the Wisconsin Typhoon [RLW94], Cashmere [KHS97], Shasta [SGT96], Blizzard [SFL94] and HLRC [SBI98]. Shrimp and Typhoon employ snooping hardware capable of reflecting loads and stores to remote locations, or enforcing access control at the granularity of cache lines. Blizzard and Shasta exploit low latency user-level messaging and couple it with software instrumentation of reads and writes to provide finer granularity access control to data. Cashmere and HLRC rely entirely on the emergence of very low latency user-level messaging and commodity SMP nodes.

The Cashmere project has its roots in the "NUMA memory management" work of the late 1980s [BFS89; BSF91; CoF89; LaE91]. This work sought to minimize the overhead of remote memory access on cacheless shared memory multiprocessors by replicating and migrating pages. In subsequent work we employed trace-based [BoS92] and execution-based [KoS95a] simulation to explore how similar techniques might be used to simplify the construction of cache-based multiprocessors. In particular, we speculated that very attractive price-performance ratios might be obtained by combining VM-based software coherence with hardware capable of (non-coherent) remote fills and write-backs of cache lines.

By 1995 technological advances had led to the commercial availability of system-area networks resembling those assumed in our simulations. One of these was the first-generation Memory Channel network [Gil96] from Digital Equipment Corporation, which we acquired as a field test site under an external research grant. Others include the Princeton Shrimp [BLA94] and HP Hamlyn [BJM96] projects, Myrinet (with appropriate software on its interface card) [PDB97], and the GSN/HIPPI [Ame96], VIA [Com97], and Infiniband [Inf02] standards. The last four of these are available as commercial products.

By separating authorization from communication, and using ordinary loads and stores to trigger the latter, modern system-area networks are able to achieve latencies more than two decimal orders of magnitude lower than is typically possible with messages implemented by the OS kernel. The second-generation Memory Channel [FiG97], for example, can modify remote memory safely in less than 3μs. It also provides sustained bandwidth (with programmed I/O) of more than 75MB/s. Finally, Memory Channel provides ordered, reliable delivery of packets (eliminating the need for acknowledgments), broadcast/multicast (possibly eliminating the need for multiple messages), and the ability to write to user-accessible remote memory without the intervention of a remote processor. All of these features are exploited by Cashmere, though simple low latency proved to be by far the most important.

The original version of Cashmere (also referred to as CSM-1L), described at *ISCA* [KHS97], treated each individual processor, even within a multiprocessor, as if it were a separate network node. It also employed an assembly language rewriting tool to "double" shared memory stores, effectively implementing write-through to the (remote) home location of the data. The next Cashmere version was a two level protocol (also known as CSM-2L) that exploits hardware coherence within multiprocessor nodes, and uses a novel "two-way diffing" mechanism to update both home and remote data copies. This version of the system was first described at *SOSP* [SDH97], and exploits Memory Channel features whenever

possible. Subsequent variants of this protocol described at *HPCA* [SDK00] vary their use of the special Memory Channel features and at one extreme use the network only for low-latency messages. We have also developed a version of Cashmere (called Cashmere-VLM) that allows the programmer to control the paging policy for out-of-core datasets [DHK99]; we do not describe that system here.

The remainder of this paper is organized as follows. We describe the Cashmere protocol in more detail in Section 2, focusing on the two-level version of the system (also referred to as CSM-2L) and its variants. We begin Section 3 with a description of our hardware platform and application suite, and then continue with performance results. Section 4 summarizes the relative strengths and weaknesses of Cashmere with respect to other state-of-the-art S-DSM systems, notably TreadMarks and Shasta [SGT96]. We show that a one-level version of the Cashmere protocol provides performance comparable to or better than that of TreadMarks-style lazy release consistency. It also outperforms Shasta-style fine-grain S-DSM for "well tuned" applications. However, Shasta is less sensitive to fine-grain temporal and spatial sharing, and has an advantage in the (generally less scalable) applications that exhibit them. We discuss related work in more detail in Section 5, and conclude with recommendations and suggestions for future research in Section 6.

2. PROTOCOL VARIANTS AND IMPLEMENTATION

Cashmere was designed for clusters connected by a high performance (low latency) system area network. It was also designed to take advantage of hardware cache coherence within symmetric multiprocessor (SMP) nodes, when available. Though the system can be configured to treat each processor of an SMP as if it were a separate network node, work on Cashmere [DGK99; SDH97] and other systems [DGK99; ENC96; SBI98; ScG97; SGA98] has shown that SMP-aware protocols can significantly improve the performance of S-DSM. In the interest of brevity we therefore focus in this paper on the SMP-aware version of the protocol. At the same time, we consider protocol variants that exploit, to varying degrees, several special features of the Memory Channel network, allowing us to evaluate the marginal utility of these features for Cashmere-style S-DSM.

We begin this section by describing basic features of the SMP-aware Cashmere protocol. This protocol exploits several special features of the Memory Channel network, including remote memory writes, broadcast, and in-order message delivery, in addition to low-latency messaging. Following a more detailed description of the network, we introduce protocol variants that make progressively less use of its features, culminating in a variant that depends on nothing more than low latency. A discussion of the SMP-oblivious version of the protocol can be found in earlier work [KHS97].

2.1 Protocol Overview

Cashmere requires all applications to follow a *data-race-free* [AdH93] programming model. Simply stated, one process must synchronize with another in order to see its modifications, and all synchronization primitives must be visible to the system. More specifically, applications must protect accesses to shared data with explicit *acquire* and *release* operations. Acquires typically correspond to critical section

entry and to departure from a barrier; releases typically correspond to critical section exit and to arrival at a barrier.

Cashmere uses the virtual memory (VM) subsystem to track data accesses. The coherence unit is an 8KB VM page (this is the minimum page size on the Alpha architecture used in our implementation). Each page of shared memory in Cashmere has a single, distinguished *home node*. The home node maintains a master copy of the page. The coherence protocol implements a "moderately lazy" version of release consistency [KCZ92]. Invalidation messages are sent at release operations, and the master copy of the page is updated to reflect changes made by the releaser, but the processing of invalidations is delayed until a subsequent acquire at the destination node, and retrieval of updates from the master copy is delayed until the next access after the invalidation. A page may be written by more than one processor concurrently, but for data-race-free applications we can be sure that the changes made by those processors will be to disjoint portions of the page, and can be merged at the home node without conflict.

The resulting consistency model lies in between those of TreadMarks [ACD96] and Munin [CBZ91]. Munin invalidates or updates (depending on protocol variant) all copies of a page at release time. TreadMarks communicates consistency information among synchronizing processes at acquire time, allowing it to invalidate copies only when there is a *causal chain* between a producer and a consumer. Updates are generated lazily, on demand, and communicated between producers and consumers point-to-point. Invalidations in Cashmere take effect at the time of the next acquire, whether it is causally related or not, and all outstanding updates are gathered from the home node at the time of a subsequent access.

A logically global *page directory* indicates, for each page, the identity of the home node and the members of the *sharing set*—the nodes that currently have copies. The main protocol entry points are page faults and synchronization operations. On a page fault, the protocol updates sharing set information in the page directory and obtains an up-to-date copy of the page from the home node. If the fault is due to a write access, the protocol also creates a pristine copy of the page (called a *twin*) and adds the page to the local *dirty list*. As an optimization in the write fault handler, a page that is shared by only one node is moved into *exclusive* mode. In this case, the twin and dirty list operations are skipped, and the page will incur no protocol overhead until another sharer emerges.

When a process reaches a release operation, it invokes protocol code which examines each page in the dirty list and compares the page to its twin (performing a *diff operation*) in order to identify modifications. These modifications are collected and either written directly into the master copy at the home node (using remote writes) or, if the page is not mapped into Memory Channel space, sent to the home node in the form of a *diff message*, for local incorporation. After propagating diffs, the protocol downgrades permissions on the dirty pages and sends *write notices* to all nodes in the sharing set. Like diffs, write notices can be sent by means of remote writes or explicit messages, depending on the protocol variant. Notices sent to a given node are accumulated in a list that each of the node's processors will peruse on its next acquire operation, invalidating any mentioned pages that have a mapping on that processor, and that have not subsequently been updated by some other local processor.

Temporal ordering of protocol operations is maintained through intra-node timestamps. Each node has a logical clock that is incremented on protocol events: page faults, acquires, and releases—the operations that result in communication with other nodes. This logical clock is used to timestamp the local copy of a page at the time of its last update, as well as to indicate when a write notice for the page was last processed. Page fetch requests can safely be eliminated if the page's last update timestamp is greater than the page's last write notice timestamp.

All processors in a given node share the same physical frame for a given data page. Software overhead is incurred only when sharing spans nodes. Moreover protocol transactions from different processors on the same node are coalesced whenever possible, reducing both computational and communication overhead.

To avoid conflicts between inter-node software coherence and intra-node hardware coherence, data from the home node cannot simply be copied into local memory during a page update operation, because the new data might overwrite modifications performed by other concurrent writers on the local node. A common solution to this problem has been to "shoot down" the other processors' mappings, and then force those processors to wait while the processor performing the page update operation flushes changes to the home node and then downloads the entire page. To avoid the expense of shootdown, CSM-2L employs a novel *incoming diff* page update operation. The processor compares incoming data from the home node to the existing twin of the page (if any) and then writes only the differences to both the working page and the twin. Since applications are required to be data-race-free, these differences are exactly the modifications made on remote nodes, and will not overlap modifications made on the local node. Updating the twin ensures that only local modifications are flushed back to the home node at the time of the next release. Full details of the SMP-aware Cashmere protocol can be found in our *SOSP* paper [SDH97].

2.2 Memory Channel

The Memory Channel is a reliable, low-latency network with a memory-mapped, programmed I/O interface. The hardware provides a *remote-write* capability, allowing processors to modify remote memory without remote processor intervention. To use remote writes, a processor must first *attach* to *transmit regions* or *receive regions* in the Memory Channel's address space, which is 512MB in size (it was 128MB in the first-generation network). Transmit regions are mapped to uncacheable I/O addresses on the Memory Channel's PCI-based network adapter. Receive regions are backed by physical memory, which must be "wired down" by the operating system. All I/O operations are fully pipelined, making it possible for the processors to use the full PCI bus bandwidth.

An application sets up a message channel by logically connecting transmit and receive regions. A store to a transmit region passes from the host processor to the Memory Channel adapter, where the data are placed into a packet and injected into the network. At the destination, the network adapter removes the data from the packet and uses DMA to write the data to the corresponding receive region in main memory.

A store to a transmit region can optionally be reflected back to a receive region on the source node by instructing the source adaptor to use *loopback* mode for a

Protocol Name	Data	Metadata	Synchronization	Home Migration
CSM-DMS	MC	MC	MC	No
CSM-MS	Explicit	MC	MC	No
CSM-S	Explicit	Explicit	MC	No
CSM-None	Explicit	Explicit	Explicit	No
CSM-MS-Mg	Explicit	MC	MC	Yes
CSM-None-Mg	Explicit	Explicit	Explicit	Yes
CSM-ADB (2L)	Explicit/ADB	MC	MC	Yes
CSM-ADB (1L)	MC/ADB	MC	MC	No

Table 1. These protocol variants have been chosen to isolate the performance impact of special network features on the areas of S-DSM communication. Use of special Memory Channel features is denoted by an "MC" under the area of communication. Otherwise, explicit messages are used. The use of Memory Channel features is also denoted in the protocol suffix (D, M, and/or S), as is the use of home node migration (Mg). ADB (Adaptive Data Broadcast) indicates the use of broadcast to communicate widely shared data modifications.

given channel. A loopback message goes out into the network and back, and is then processed as a normal message.

By connecting a transmit region to multiple receive regions, nodes can make use of hardware broadcast. The network guarantees that broadcast messages will be observed in the same order by all receivers, and that all messages from a single source will be observed in the order sent. Broadcast is more expensive than point-to-point messages, because it must "take over" the crossbar-based central network hub. Broadcast and total ordering, together with loopback, are useful in implementing cluster-wide synchronization, to be described in the following section.

2.3 Protocol Variants

Special network features like direct read/write (load/store) access to remote memory, broadcast/multicast, and total message ordering may be used for a variety of purposes in S-DSM, including data propagation, metadata (directory and write notice) maintenance, and synchronization. As noted in the previous section, the Memory Channel provides all of these other than direct remote reads, and the base Cashmere protocol depends on all of them.

In an attempt to identify the marginal utility of special network features for various protocol operations, we developed several variants of the Cashmere-2L protocol, summarized in Table 1. The first four variants, described in more detail in Section 2.3.1, all follow the outline described in Section 2.1, but vary in their reliance on special network features. The next two variants, described in more detail in Section 2.3.2, exploit the observation that when remote writes are not used for data propagation, one can change the home node of a page at very low cost, often achieving significant savings in coherence overhead. This optimization is unattractive when program data reside in the Memory Channel address space, because of the high cost of remapping transmit and receive regions. The final protocol variant, mentioned in the last two lines of the table and described in more detail in Section 2.3.3, uses broadcast not only for synchronization and page directory maintenance, but for data propagation as well. In an attempt to assess scalability, we consider it not only in CSM-2L, but in CSM-1L as well, where we can treat the processors of our hardware cluster as 32 separate nodes.

All of our protocol variants rely in some part on efficient explicit messages. To minimize delivery overhead [KHS97], we arrange for each processor to poll for messages on every loop back edge, branching to a handler if appropriate. The polling instructions are added to application binaries automatically by an assembly language rewriting tool.

2.3.1 *Reliance on Special Network Features.* For each of the areas of protocol communication (data propagation, metadata maintenance, and synchronization), we can leverage the full capabilities of the Memory Channel (remote writes, total ordering, and inexpensive broadcast), or instead send explicit messages between nodes. The combinations we consider are described in the paragraphs below. We assume in all cases that reliable message delivery is ensured by the network hardware. When using explicit messages, however, we send acknowledgments whenever we need to determine the order in which messages arrived, or to ensure that they arrive in some particular order.

2.3.1.1 *CSM-DMS: Data, Metadata, and Synchronization using Memory Channel.* The base protocol, denoted CSM-DMS, is the Cashmere-2L protocol described in Section 2.1 and in our original paper on SMP-aware S-DSM [SDH97]. This protocol exploits the Memory Channel for all communication: to propagate shared data, to maintain metadata, and for synchronization.

Data: All shared data are mapped into the Memory Channel address space. Each page is assigned a home node, namely, the first node to touch the page after initialization. The home node creates a receive mapping for the page. All other nodes create a transmit mapping as well as a local copy of the page. Shared data are fetched from the home node using messages. Fetches could be optimized by a remote read operation or by allowing the home node to write the data directly to the working address on the requesting node. Unfortunately, the first optimization is not available on the Memory Channel. The second optimization is also effectively unavailable because it would require shared data to be mapped at distinct Memory Channel addresses on each node. With only 512MB of Memory Channel address space, this would severely limit the maximum dataset size. (For eight nodes, we could share only about 64MB.)

Modifications are written back to the home node at the time of a release.[1] With home node copies kept in Memory Channel space these modifications can be applied with remote writes, avoiding the need for processor intervention at the home. Addressing constraints still limit dataset size, but the limit is reasonably high, and is not affected by system size.

To avoid race conditions, Cashmere must be sure that all diffs are completed before returning from a release operation. To avoid the need for explicit acknowledgments, CSM-DMS writes all diffs to the Memory Channel and then resets a synchronization location in Memory Channel space to complete the release. Network total ordering ensures that the diffs will be complete before the completion of the release is observed.

[1] An earlier Cashmere study [KHS97] investigated using write-through to propagate data modifications. Release-time diffs were found to use bandwidth more efficiently than write-through, and to provide better performance.

Metadata: System-wide metadata in CSM-DMS consists of the page directory and write notice lists. CSM-DMS replicates the page directory on each node and uses remote write to broadcast all changes. It also uses remote writes to deliver write notices to a list on each node using appropriate locking. The write notice buffers are sized to hold write notices for all shared pages, thereby avoiding overflow issues. At an acquire, a processor simply reads its write notices from local memory. As with diffs, CSM-DMS takes advantage of network ordering to avoid write notice acknowledgments.

Synchronization: Application locks, barriers, and flags all leverage the Memory Channel's broadcast and write ordering capabilities. Locks are represented by an 8-entry array in Memory Channel space, and by a test-and-set flag on each node. A process first acquires the local test-and-set lock and then sets, via remote-write broadcast, its node's entry in the 8-entry array. The process waits for its write to appear via loopback, and then reads the entire array. If no other entries are set, the lock is acquired; otherwise the process resets its entry, backs off, and tries again. This lock implementation allows a processor to acquire a lock without requiring any remote processor assistance. Barriers are represented by an 8-entry array, a "sense" variable in Memory Channel space, and a local counter on each node. The last processor on each node to arrive at the barrier updates the node's entry in the 8-entry array. A single master processor waits for all nodes to arrive and then toggles the sense variable, on which the other nodes are spinning. Flags are write-once notifications based on remote write and broadcast.

2.3.1.2 *CSM-MS: Metadata and Synchronization using Memory Channel.* CSM-MS does not place shared data in Memory Channel space, and so avoids network-induced limitations on dataset size. As a result, however, CSM-MS cannot use remote writes to apply diffs. Instead it sends diff messages, which require processing assistance from the home node and explicit acknowledgments to establish ordering. In CSM-MS, metadata and synchronization still leverage all Memory Channel features.

2.3.1.3 *CSM-S: Synchronization using Memory Channel.* CSM-S uses special network features only for synchronization. Explicit messages are used both to propagate shared data and to maintain metadata. Instead of broadcasting a directory change, a process must send the change to the home node in an explicit message. The home node updates the entry and acknowledges the request.

Directory updates (or reads that need to be precise) can usually be piggybacked onto an existing message. For example, a directory update is implicit in a page fetch request and so can be piggybacked. Also, write notices always follow diff operations, so the home node can simply piggyback the sharing set (needed to identify where to send write notices) onto the diff acknowledgment. In fact, an explicit directory message is needed only when a page is invalidated.

2.3.1.4 *CSM-None: No Use of Special Memory Channel Features.* The fourth protocol, CSM-None, uses explicit messages (and acknowledgments) for all communication. This protocol variant relies only on low-latency messaging, and so could easily be ported to other low-latency network architectures. Our protocol still depends on polling to efficiently detect message arrival. Other efficient mechanisms

for message arrival detection could be used instead if available on other platforms. Polling can also be implemented efficiently on non-remote write networks [DRM98; WBv97].

2.3.2 *CSM-XXX-Mg: Home Node Migration.* All of the above protocol variants use first-touch home node assignment [MKB95]. Home assignment is extremely important because processors on the home node write directly to the master copy of the data and so do not incur the overhead of twins and diffs. If a page has multiple writers during the course of execution, protocol overhead can potentially be reduced by migrating the home node to an active writer.

Due to the high cost of remapping Memory Channel regions, home node migration is unattractive when data are placed in Memory Channel space. Hence, home node migration cannot be combined with CSM-DMS. In our experiments we incorporate it into CSM-MS and CSM-None, creating CSM-MS-Mg and CSM-None-Mg. (We also experimented with CSM-S-Mg, but its performance does not differ significantly from that of CSM-S. We omit it to minimize clutter in the graphs.) When a processor incurs a write fault, these protocols check the local copy of the directory to see if any process on the home node appears to be actively writing the page (the local copy is only a hint in CSM-None-Mg). If there appears to be no other writer, a migration request is sent to the home. The request is granted if it is received when no other node is in the process of writing the page. The home changes the directory entry to point to the new home. The new home node will receive an up-to-date copy of the page as part of the normal coherence operations. The marginal cost of changing the home node identity is therefore very low.

2.3.3 *CSM-ADB: Adaptive Broadcast.* The protocol variants described in the previous sections all use invalidate-based coherence: data are updated only when accessed. CSM-ADB uses Memory Channel broadcast to efficiently communicate application data that is widely shared (read by multiple consumers). To build the protocol, we modified the CSM-MS messaging system to create a new set of buffers, each of which is mapped for transmit by any node and for receive by all nodes except the sender. Pages are written to these globally mapped buffers selectively, based on the following heuristics: multiple requests for the same page are received simultaneously; multiple requests for the same page are received within the same synchronization interval on the home node (where a new interval is defined at each release); or there were two or more requests for the page in the previous interval. These heuristics enable us to capture both repetitive and non-repetitive multiple-consumer access patterns. Pages in the broadcast buffers are invalidated at the time of a release if the page has been modified in that interval (at the time at which the directory on the home node is updated). Nodes that are about to update their copy of a page check the broadcast buffers for a valid copy before requesting one from the home node. The goal is to reduce contention and bandwidth consumption by eliminating multiple requests for the same data. In an attempt to assess the effects of using broadcast with a larger number of nodes, we also report CSM-ADB results using 32 processors on a one-level protocol [KHS97]—one that does not leverage hardware shared memory for sharing within the node—so as to emulate the communication behavior of a 32-node system.

MC property	Value
Latency	3.3 μs
Bandwidth (p2p)	70MB/s
Bandwidth (aggregate)	100MB/s

Operation	MC Direct	MC Messages
Diff (μs)	31–129	70–245
Lock (μs)	10	33
Barrier (μs)	29	53

Table 2. Basic memory channel properties and operation costs on 32 processors. Diff cost varies according to the size of the diff.

3. RESULTS

We begin this section with a brief description of our hardware platform and our application suite. Next, we compare the performance of the one-level (CSM-1L) protocol with that of the two-level DMS version (CSM-2L) that leverages intra-node hardware coherence. Finally we present our investigation of the impact on performance of Memory Channel features and of the home node migration and broadcast optimizations.

3.1 Platform and Basic Operation Costs

Our experimental platform is a set of eight AlphaServer 4100 5/600 nodes, each with four 600 MHz 21164A processors, an 8 MB direct-mapped board-level cache with a 64-byte line size, and 2 GBytes of memory. The 21164A has two levels of on-chip cache. The first level consists of 8 KB each of direct-mapped instruction and (write-through) data cache, with a 32-byte line size. The second level is a combined 3-way set associative 96 KB cache, with a 64-byte line size. The nodes are connected by a Memory Channel 2 system area network with a peak point-to-point bandwidth of 70 MB/sec and a one-way, cache-to-cache latency for a 64-bit remote-write operation of 3.3 μs.

Each AlphaServer node runs Compaq Tru64 Unix 4.0F, with TruCluster v1.6 (Memory Channel) extensions. The systems execute in multi-user mode, but with the exception of normal Unix daemons no other processes were active during the tests. In order to increase cache efficiency, application processes are pinned to a processor at startup. No other nodes are connected to the Memory Channel.

The typical round-trip latency for a null message in Cashmere is 15 μs. This time includes the transfer of the message header and the invocation of a null handler function. A page fetch operation costs 220 μs. A twin operation requires 68 μs.

As described in Section 2.3.1, remote writes, broadcast, and total ordering can be used to significantly reduce the cost of diffs, metadata maintenance (directory updates and write notice propagation), and synchronization. Table 2 shows the basic Memory Channel latency and bandwidth properties as well as costs for diff operations, lock acquires, and barriers with (*MC Direct*) and without (*MC Messages*) the use of these special features. The cost of diff operations varies according to the size of the diff. In the MC Direct case, directory updates, write notices, and flag synchronization all leverage remote writes and total ordering, and directory updates and flag synchronization also rely on inexpensive broadcast. In the MC Messages case, directory updates are small messages with simple handlers, so their cost is only slightly more than the cost of a null message. The cost of write notices depends greatly on the write notice count and number of destinations. Write notices sent to different destinations can be overlapped, thus reducing the operation's

Program	Problem Size	Time (s)
Barnes	128K bodies (26MB)	120.4
CLU	2048x2048 (33MB)	75.4
LU	2500x2500 (50MB)	143.8
EM3D	64000 nodes (52MB)	30.6
Gauss	2048x2048 (33MB)	234.8
Ilink	CLP (15MB)	212.7
SOR	3072x4096 (50MB)	36.2
TSP	17 cities (1MB)	1342.5
Water-Nsquared	9261 mols. (6MB)	332.6
Water-Spatial	9261 mols. (16MB)	20.2
Ocean	1026x1026 (242MB)	37.1
Raytrace	Balls4 (102MB)	44.9
Volrend	Head (23MB)	3.3

Table 3. Dataset sizes and sequential execution time of applications. Ocean, Raytrace, and Volrend appear only in Section 4.2.

overall latency. Flag operations must be sent to all nodes, but again messages to different destinations can be overlapped, so perceived latency should not be much more than that of a null message.

3.2 Application Suite

Most of our applications are well known benchmarks from the Splash [SWG92; WOT95] and TreadMarks [ACD96] suites.

Briefly, the applications are Barnes, an N-body TreadMarks simulation derived from the same application in the SPLASH-1 suite; CLU and LU, lower and upper triangular matrix factorization kernels (with and without contiguous allocation of a single processor's data, respectively), from the SPLASH-2 suite;[2] EM3D, a program to simulate electromagnetic wave propagation through 3D objects [CDG93]; Gauss, a locally-developed solver for a system of linear equations $Ax = B$ using Gaussian Elimination and back-substitution; Ilink, a widely used genetic linkage analysis program from the FASTLINK 2.3P [DSC94] package, which locates disease genes on chromosomes; SOR, a Red-Black Successive Over-Relaxation program, from the TreadMarks distribution; TSP, a traveling salesman problem, from the TreadMarks distribution; Water-Nsquared, a fluid flow simulation from the SPLASH-2 suite; and Water-Spatial, another SPLASH-2 fluid flow simulation that solves the same problem as Water-Nsquared, but with the data partitioned spatially. To the extent possible we have endeavored to present results for a uniform subset of these suites across all our experiments. The SPLASH-2 appliactions (Water-Spatial and CLU) were not available, however, at the time of the study reported in Section 4.1, and three additional additional SPLASH-2 applications, for which we lack complete results, are included in Section 4.2 because they illustrate particularly interesting tradeoffs. These are Ocean, which simulates large-scale ocean currents using a Gauss-Seidel multigrid equation solver; Raytrace, which renders a three-

[2]Both CLU and LU tile the input matrix and assign each column of tiles to a contiguous set of processors. Due to its different allocation strategy, LU incurs a large amount of false sharing across tiles. To improve scalability, we modified LU to assign a column of tiles to processors within the same SMP, thereby reducing false sharing across node boundaries.

dimensional scene using a hierarchical uniform grid; and Volrend, which uses ray casting and an octree structure to render a three-dimensional cube of voxels.

The dataset sizes and uniprocessor execution times for these applications are presented in Table 3. Our choice of dataset sizes was driven by a desire to allow reasonable scaling while still showcasing the various protocol overheads. Larger dataset sizes would lead to better speedups but less protocol differentiation. Execution times were measured by running each uninstrumented application sequentially without linking it to the protocol library.

3.3 Performance Impact of a Multilevel Protocol

Figure 1 presents speedup bar charts for our applications on up to 32 processors on both the 1-level and 2-level protocols. All calculations are with respect to the sequential times in Table 3. The configurations we use are written as N:M, where N is the total number of processes (each on its own processor), and M is the number of processes per cluster node. So 8:2, for example, uses two processors in each of 4 nodes, while 8:4 uses all four processors in each of 2 nodes.

Figure 2 presents a breakdown of execution time at 32 processors for each application. The breakdown is normalized with respect to total execution time for CSM-2L. The components shown represent time spent executing user code (User), time spent in protocol code (Protocol), and communication and wait time (Comm & Wait). In addition to the execution time of user code, User time also includes cache misses and time needed to enter protocol code, i.e., kernel overhead on traps and function call overhead from a successful message poll. Protocol time includes all time spent executing within the Cashmere library, while communication and wait time includes stalls waiting for page transfers, and wait time at synchronization points.

Compared to the base 1L protocol, the two-level protocol shows slight improvement for CLU, SOR, TSP, and Water-Nsquared (1–9%), good improvement for EM3D (22%), and substantial improvement for Ilink, Gauss, LU, Water-spatial, and Barnes (40–46%). A slightly optimized version of the one-level protocol that migrates the home node during program execution [SDH97] closes the gap for EM3D, but the two-level protocols still hold their performance edge for the applications overall.

SOR and EM3D both benefit from the ability of 2L to use hardware coherence within nodes to exploit application locality, in particular, nearest neighbor sharing. This locality results in significant reduction of page faults, page transfers, coherence operations, and time spent in protocol code. Performance benefits due to protocol improvements are more pronounced for EM3D since it has a low computation-to-communication ratio and protocol time is a higher percentage of overall execution time.

CLU benefits from hardware coherence within the home node and from a reduction in the amount of data transferred, due to the ability to coalesce requests. CLU's high computation-to-communication ratio limits overall improvements. Water-Nsquared also benefits from coalescing remote requests for data. In both applications, the amount of data transmitted in 2L relative to 1L is halved.

Gauss shows significant benefits from coalescing remote requests, since the access pattern for the shared data is essentially single producer/multiple consumer (ideally

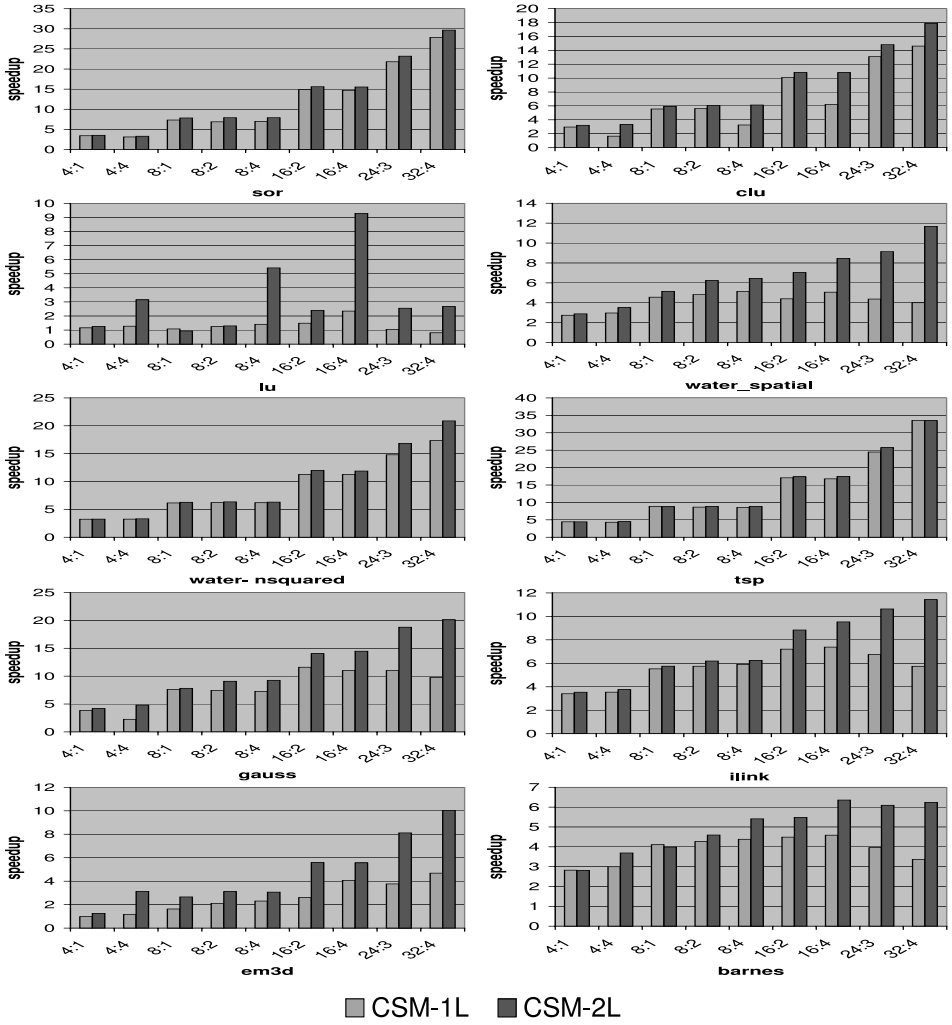

Fig. 1. Speedups for One-Level (1L) and Two-Level (2L) protocols.

implemented with broadcast). There is a four-fold reduction in the amount of data transferred, but little reduction in the number of read and write page faults. These gains come at the expense of slightly higher protocol overhead due to increased maintenance costs of protocol metadata (Figure 2). However, there is a 45% improvement in execution time, a result of the reduction in communication and wait time. Ilink's communication pattern is all-to-one in one phase, and one-to-all in the other (a master-slave style computation). Hence, its behavior is similar to that of Gauss, with a 40% performance improvement in performance. Barnes shows a 46% improvement in performance. The benefits in this case come from the ability to coalesce page fetch requests, which significantly reduces the amount of data transferred. Since the computation-to-communication ratio is low for this application, this reduction has a marked effect on performance.

LU and Water-Spatial show the largest improvements when moving from a one-

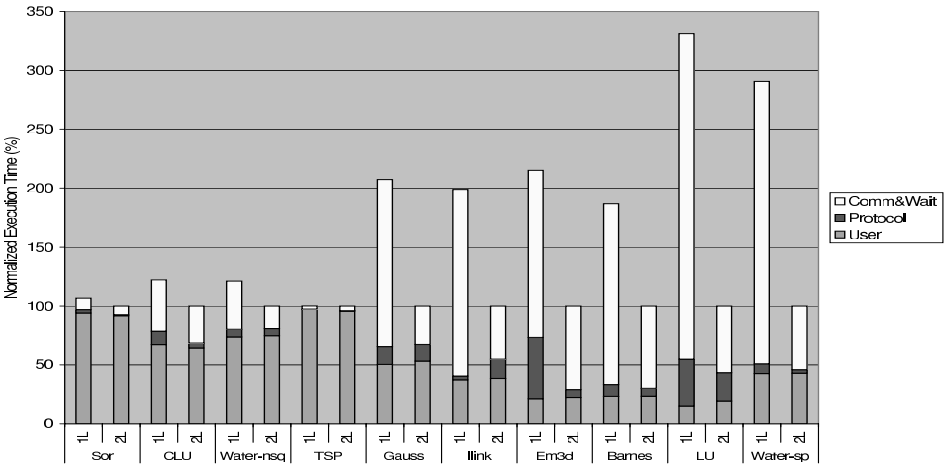

Fig. 2. Breakdown of percent normalized execution time for the Two-Level (2L) and One-Level (1L) protocols at 32 processors. The components shown represent time spent executing user code (User), time spent in protocol code (Protocol), and communication and wait time (Comm & Wait).

level to a two-level protocol. LU incurs high communication costs due to write-write false sharing at row boundaries. Leveraging hardware coherence within nodes alleviates this effect and results in a dramatic reduction in communication and wait time. This is further evidenced by the performance differences among configurations with the same number of processors. Using more processors per node and a smaller number of nodes results in much higher performance. Water-Spatial exhibits behavior similar to that of SOR and EM3D. Simulated molecules are grouped into physically proximate cells, so data sharing happens only when processors must access molecules in neighboring cells. Molecules can move among cells, however, creating a loss of locality and an increase in false sharing over time. The two-level protocol can better deal with loss of locality within an SMP node and as a result experiences significantly reduced communication and synchronization overheads.

TSP's behavior is non-deterministic, which accounts for the variations in user time. Performance of the two-level protocol is comparable to that of the one-level protocol. The high computation-to-communication ratio results in good performance in both cases.

3.3.1 *Effect of Clustering.* The performance of SOR and Gauss decreases for all protocols when the number of processors per node (degree of clustering) is increased, while keeping the number of processors fixed (see Figure 1). Both applications are matrix-based, with only a small amount of computation per element of the matrix. Their dataset sizes do not fit in the second level caches and hence a large amount of traffic is generated between the caches and main memory due to capacity misses. Increasing the number of processors per node increases the traffic on the shared node bus, thereby reducing performance. Cox et al. [CDK94] report similar results in their comparison of hardware and software shared memory systems.

CLU exhibits negative clustering effects only for the one-level protocols. The

4:4, 8:4, and 16:4 configurations experience a significant performance drop, which is not present in the two-level protocol. This can be attributed to the burstiness of CLU's page accesses. As a pivot row is factored, its pages are accessed only by their owner, and are therefore held in exclusive mode. After factoring is complete, other processors immediately access the pages, generating requests for each page to leave exclusive mode. As the number of processors per node increases, the shared network link creates a communication bottleneck. The two-level protocol alleviates this bottleneck by exploiting hardware coherence and coalescing requests sent to a remote node for the same page.

EM3D, LU, Water-spatial, and Barnes show a performance improvement when the number of processors per node is increased in the two-level protocol. The one-level protocol does not show similar performance gains. These applications have low computation-to-communication ratios, allowing the reduction in inter-node traffic due to the use of intra-node sharing in the two-level protocol to yield benefits even at 8 and 16 processor totals. At 4 processors, the two-level protocol also benefits from sharing memory in hardware thus avoiding software overheads and extra traffic on the bus.

At first glance, these results may appear to contradict those of previous studies [BIM96; CDK94; ENC96], which report that bandwidth plays a major role in the performance of clustered systems. Our results compare one and two-level protocols on the *same* clustered hardware, as opposed to two-level protocols on clustered hardware versus one-level protocols on non-clustered hardware (with consequently higher network bandwidth per processor).

The performance of Water, TSP, and Ilink is unaffected by the number of processors per node at 8 and 16 processor totals, regardless of the protocol. The reduction in inter-node communication due to the use of hardware coherence in the two-level protocol improves performance significantly at 24 and 32 processors for Gauss, Ilink, EM3D, LU, Water-spatial, and Barnes. SOR, CLU, TSP, and Water show only slight overall benefits from exploiting hardware coherence, due to their high computation-to-communication ratios.

Overall the extra synchronization and data structures in the two-level protocol have little effect on performance. This observation is supported by the similar performance of the 8:1 configuration for most of the applications studied.

3.4 Performance Impact of Protocol Variants

Throughout this section, we will refer to Figure 3 and Table 4. Figure 3 shows a breakdown of execution time, normalized to that of the CSM-DMS protocol, for the first six protocol variants from Table 1 (page 7). The breakdown indicates time spent executing application code (User), executing protocol code (Protocol), waiting on synchronization operations (Wait), and sending or receiving messages (Message). Table 4 lists the speedups and statistics on protocol communication for each of the applications running on 32 processors. The statistics include the numbers of page transfers, diff operations, and home node migrations (and attempts).

3.4.1 *The Impact of Memory Channel Features.* Five of our ten applications show measurably better performance on CSM-DMS (fully leveraging Memory Channel features) than on CSM-None (using explicit messages). Barnes runs 80% faster;

EM3D and Water-Nsquared run 20-25% faster; LU and Water-Spatial run approximately 10% faster. CLU, Gauss, Ilink, SOR, and TSP are not sensitive to the use of Memory Channel features and do not show any significant performance differences across our protocols. All of the protocols use first-touch initial home node assignment. With multiple sharers per page, however, the timing differences among protocol variants can lead to first-touch differences. To eliminate these differences and isolate Memory Channel impact, we captured the first-touch assignments from CSM-DMS and used them to explicitly assign home nodes in the other protocols.

Barnes exhibits a high degree of sharing and incurs a large Wait time on all protocol variants (see Figure 3). CSM-DMS runs roughly 40% faster than CSM-MS and 80% faster than CSM-S and CSM-None. This performance difference is due to the lower Message and Wait times in CSM-DMS. In this application, the Memory Channel features serve to optimize data propagation and metadata maintenance, thereby reducing application perturbation, and resulting in lower wait time. Due to the large amount of false sharing in Barnes, application perturbation also results in large variations in the number of pages transferred. Since synchronization time is dominated by software coherence protocol overhead the use of Memory Channel features to optimize synchronization has little impact on perofrmance.

Water-Nsquared, too, obtains its best performance on CSM-DMS. The main reason is page lock contention for the delivery of diffs and servicing of page faults. CSM-DMS does not experience this contention since it can deliver diffs using remote writes. The Memory Channel features also provide a noticeable performance advantage by optimizing synchronization operations in this application. Water-Nsquared uses per-molecule locks, and so performs a very large number of lock operations. Overall, CSM-DMS performs 13% better than CSM-MS and CSM-S and 18% better than CSM-None.

CSM-DMS also provides the best performance for EM3D: a 23% margin over the other protocols. Again, the advantage is due to the use of Memory Channel features to optimize data propagation. In contrast to Barnes and LU, the major performance differences in EM3D are due to Wait time, rather than Message time. Performance of EM3D is extremely sensitive to higher data propagation costs. The application exhibits a nearest neighbor sharing pattern, so diff operations in our SMP-aware protocol occur only between adjacent processors spanning nodes. These processors perform their diffs at barriers, placing them directly in the critical synchronization path. Any increase in diff cost will directly impact the overall Wait time. Figure 3 shows this effect, as Message time increases from 18% in CSM-DMS to 24% in CSM-MS, but Wait time increases from 41% to 65%. This application provides an excellent example of the sensitivity of synchronization Wait time to any protocol perturbation.

At the given matrix size, LU incurs a large amount of protocol communication due to write-write false sharing at row boundaries, and CSM-DMS performs 12% better than the other protocols. The advantage is due primarily to optimized data propagation, as CSM-DMS uses remote writes and total ordering to reduce the overhead of diff application.

Water-Spatial is sensitive to data propagation costs. The higher cost in CSM-MS, CSM-S, and CSM-None perturbs the synchronization Wait time and hurts overall performance. CSM-DMS outperforms the other protocols on Water-Spatial

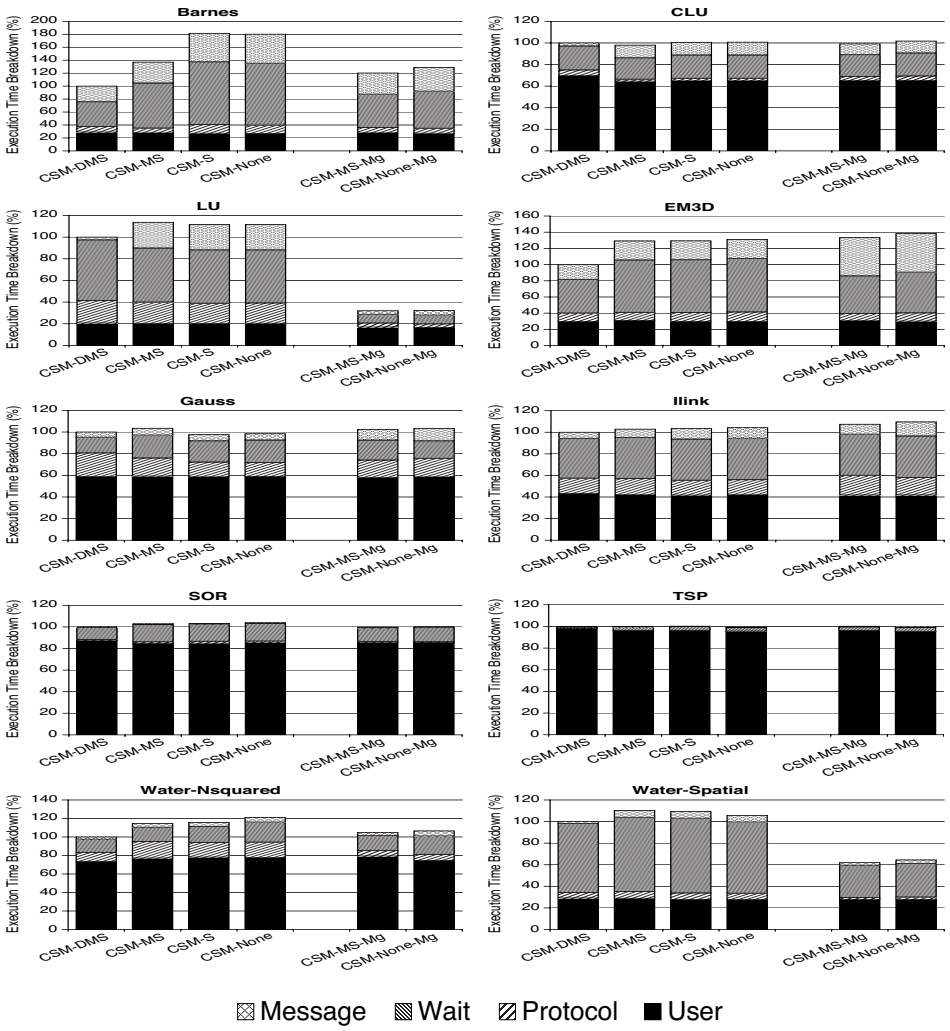

Fig. 3. Normalized execution time breakdown for the applications and protocol variants on 32 processors. The suffix on the protocol name indicates the kinds of communication using special Memory Channel features (D: shared Data propagation, M: protocol Metadata maintenance, S: Synchronization, None: No use of Memory Channel features). Mg indicates a migrating home node policy.

by 10%.

CLU shows no significant difference in overall performance across the protocols. This application has little communication that can be optimized. Any increased Message time is hidden by the existing synchronization time. Ilink performs a large number of diffs, and might be expected to benefit significantly from remote writes. However, 90% of the diffs are applied at the home node by idle processors, so the extra overhead is mostly hidden from the application. Hence, the benefits are negligible. Gauss, SOR, and TSP, likewise, are not noticeably dependent on the use of Memory Channel features.

Application		Cashmere Variant					
		DMS	MS	S	None	MS-Mg	None-Mg
Barnes	Speedup (32 procs)	7.6	5.5	4.2	4.2	6.3	5.9
	Page Transfers (K)	66.0	63.4	96.8	96.1	69.1	78.5
	Diffs (K)	60.8	50.2	66.4	61.8	45.1	47.5
	Migrations (K)	—	—	—	—	15.6 (15.6)	11.6 (67.4)
CLU	Speedup (32 procs)	18.3	18.4	18.0	18.0	18.2	17.7
	Page Transfers (K)	8.3	11.9	11.9	11.9	11.9	11.9
	Diffs (K)	0	0	0	0	0	0
	Migrations (K)	—	—	—	—	3.5 (3.5)	3.5 (3.5)
LU	Speedup (32 procs)	4.0	3.5	3.6	3.6	12.5	12.4
	Page Transfers (K)	44.1	44.4	44.6	44.4	51.1	53.1
	Diffs (K)	285.6	278.06	278.9	277.4	1.1	1.1
	Migrations (K)	—	—	—	—	5.5 (5.5)	5.5 (5.5)
EM3D	Speedup (32 procs)	13.5	10.5	10.5	10.3	10.2	9.8
	Page Transfers (K)	32.8	32.8	33.1	33.1	43.9	43.8
	Diffs (K)	7.1	7.1	7.1	7.1	0	0
	Migrations (K)	—	—	—	—	1.9 (1.9)	1.9 (1.9)
Gauss	Speedup (32 procs)	22.7	21.9	23.2	23.0	22.1	21.9
	Page Transfers (K)	38.2	42.2	40.1	40.3	43.9	44.1
	Diffs (K)	3.6	3.6	3.6	3.6	0.5	0.1
	Migrations (K)	—	—	—	—	4.5 (4.5)	4.6 (4.6)
Ilink	Speedup (32 procs)	12.5	12.1	11.1	11.1	11.6	11.4
	Page Transfers (K)	50.0	50.0	53.1	53.1	51.9	56.1
	Diffs (K)	12.0	12.2	12.4	12.4	8.7	8.6
	Migrations (K)	—	—	—	—	1.9 (2.7)	1.9 (6.2)
SOR	Speedup (32 procs)	31.2	30.1	30.1	29.9	31.2	30.9
	Page Transfers (K)	0.3	0.3	0.3	0.3	0.7	0.7
	Diffs (K)	1.4	1.4	1.4	1.4	0	0
	Migrations (K)	—	—	—	—	0	0
TSP	Speedup (32 procs)	33.9	34.0	33.8	34.2	33.9	34.0
	Page Transfers (K)	12.6	12.2	12.3	12.2	14.1	13.9
	Diffs (K)	8.0	7.8	7.8	7.8	0.1	0.1
	Migrations (K)	—	—	—	—	5.0 (5.0)	5.0 (5.0)
Water-Nsquared	Speedup (32 procs)	20.6	18.0	17.8	17.0	19.6	19.3
	Page Transfers (K)	31.5	29.8	29.4	22.9	28.3	32.9
	Diffs (K)	251.1	234.4	249.7	243.7	17.2	26.3
	Migrations (K)	—	—	—	—	9.2 (9.3)	11.0 (11.7)
Water-Spatial	Speedup (32 procs)	7.7	7.0	7.0	7.2	12.3	11.8
	Page Transfers (K)	4.0	4.5	4.8	4.9	5.2	5.6
	Diffs (K)	6.2	6.2	6.4	6.4	0.1	0.1
	Migrations (K)	—	—	—	—	0.3 (0.3)	0.3 (0.3)

Table 4. Application speedups and statistics at 32 processors. For home node migrations, the parenthesized figure indicates the number of attempts.

3.4.2 *Home Node Migration: Optimization for a Scalable Data Space.* Home node migration can reduce the number of remote memory accesses by moving the home node to active writers, thereby reducing the number of invalidations, twins and diffs, and (sometimes) the amount of data transferred across the network. Our results show that this optimization can be very effective. Six of our ten applications are affected by home node migration. Two of them (EM3D and Ilink) suffer slightly; four (LU, Water-Spatial, Barnes, and Water-Nsquared) benefit, in some cases dramatically.

Migration is particularly effective in LU and Water-Spatial, where it significantly reduces the number of diff and attendant twin operations (see Table 4). In fact, for these applications, CSM-None-Mg, which does not leverage the special Memory Channel features at all, outperforms the full Memory Channel protocol, CSM-DMS, reducing execution time by 67% in LU and 34% in Water-Spatial.

In Barnes and Water-Nsquared, there are also benefits, albeit smaller, from using migration. In both applications, CSM-MS-Mg and CSM-None-Mg outperform their

first-touch counterparts, CSM-MS and CSM-None. Both applications incur many fewer diffs when using migration (see Table 4). The smaller number of diffs (and twins) directly reduces Protocol time and, indirectly, Wait time. In Barnes, the execution time for CSM-MS-Mg and CSM-None-Mg is 12% and 27% lower, respectively, than in CSM-MS and CSM-None, bringing performance to within 30% of CSM-DMS for CSM-None-Mg. Water-Nsquared shows 8% and 12% improvements in CSM-MS-Mg and CSM-None-Mg, respectively, bringing performance to within 7% of CSM-DMS for CSM-None-Mg.

Home migration hurts performance in EM3D and Ilink. The reduction in the number of diff operations comes at the expense of increased page transfers due to requests by the consumer, which was originally the home node. Since only a subset of the data in a page is modified, the net result is a larger amount of data transferred, which negatively impacts performance. CSM-None-Mg also suffers from a large number of unsuccessful migration requests in Ilink (see Table 4). These requests are denied because the home node is actively writing the page. In CSM-MS-Mg, the home node's writing status is globally available in the replicated page directory, so a migration request can be skipped if inappropriate. In CSM-None-Mg, however, a remote node only caches a copy of a page's directory entry, and may not always have current information concerning the home node. Thus, unnecessary migration requests cannot be avoided. Barnes, like Ilink, suffers a large number of unsuccessful migration attempts with CSM-None-Mg, but the reduction in page transfers and diffs more than makes up for these.

Overall, the migration-based protocol variants deliver very good performance, while avoiding the need to map shared data into the limited amount of remotely accessible address space. The performance losses in EM3D and Ilink are fairly low (3–5%), while the improvements in other applications are comparatively large (up to 67%).

3.4.3 *Selective Broadcast for Widely Shared Data.* Selective use of broadcast for data that are accessed by multiple consumers (as in CSM-ADB) can reduce the number of messages and the amount of data sent across the network, in addition to reducing contention and protocol overhead at the producer (home node). Compared against CSM-MS-Mg (the non-broadcast variant with the best performance), performance improvements on an 8-node, 32-processor system vary from −3% in SOR (the only application hurt by broadcast) to 13% in Ilink (Figure 4). Since data are distributed to both producer and consumer during the first iteration of SOR, broadcast is used for the second iteration. Subsequent iterations correctly detect a single consumer and revert to point-to-point messaging. While the cost of predicting whether broadcast should be used is negligible, the use of broadcast when only one (due to misprediction) or a small number of nodes share the data could result in potential contention by not allowing simultaneous point-to-point communication.

In order to determine the effects on performance when using a larger cluster, we emulated a 32-node system by running a one-level (non-SMP-aware) protocol in which each processor is in effect a separate node.[3] Performance improvements at

[3]Emulation differs from a real 32-node system in that the (four) processors within a node share

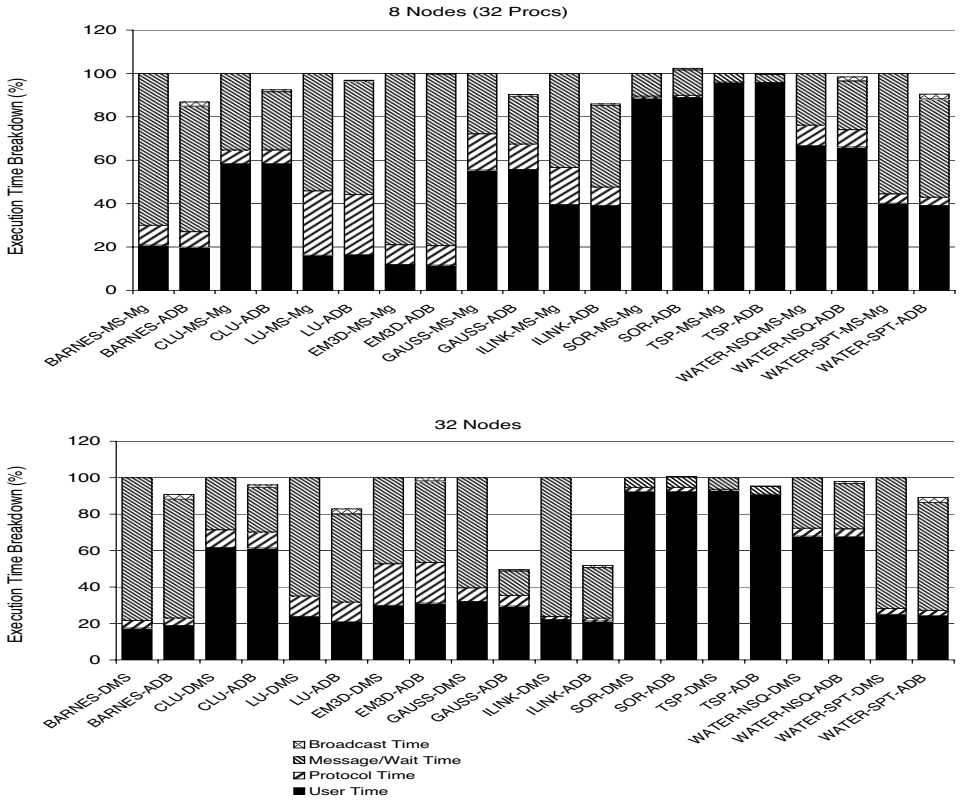

Fig. 4. Normalized execution time breakdown using adaptive broadcast of data (CSM-ADB) in comparison to CSM-MS-Mg at 8 nodes and CSM-DMS at 32 nodes.

32 nodes jump to 18, 49, and 51% for LU, Ilink and Gauss, respectively. The large gains in these applications come from a reduction in Message and Wait time. In Gauss, the protocol is able to detect and optimize the communication of each pivot row to multiple consumers: 10K pages are placed in the broadcast buffers, and satisfy 172K out of a total of 182K page updates. In Ilink, 12K pages are placed in the broadcast buffers, and satisfy 191K out of a total of 205K page updates. The number of consumers in LU is not as large: 106K pages are placed in the broadcast buffers, and satisfy 400K out of a total of 1.19M page updates. SOR is again the only application to be hurt by broadcast, and only by about 1%.

4. ALTERNATE SOFTWARE DSM APPROACHES USING FAST USER-LEVEL MESSAGES

Cashmere was developed with a Memory Channel style architecture in mind and as a consequence performs quite well on Memory Channel clusters. It is important, however, to examine how other software DSM approaches perform on this kind of

the same network interface and messages among processors within a node are exchanged through shared memory. We used CSM-DMS as the base to which ADB was added, since this was the only protocol for which we had 32-node emulation capabilities.

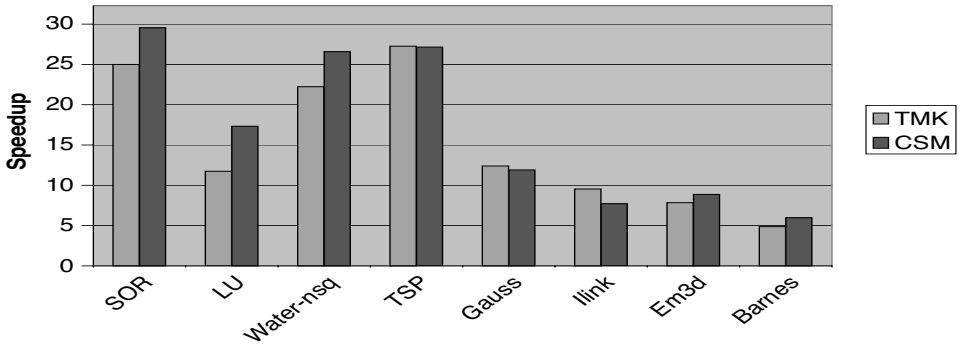

Fig. 5. Comparative speedups of TreadMarks (TMK) and Cashmere (CSM) on 32 processors (16 for Barnes).

architecture as well. To that end, we look at the relative performance of Cashmere and two other state-of-the-art DSMs: TreadMarks and Shasta. TreadMarks was developed for higher latency networks (and therefore uses a lazy release consistency model to minimize the number of messages) but is likely to benefit from the lower-latency Memory Channel messages as well. Shasta was designed with a Memory Channel architecture in mind but also attempts to reduce overheads associated with fine-grain sharing by using smaller coherence units and an all-software approach to detecting coherence events. Unfortunately, we have been unable to recollect the performance numbers for TreadMarks and Shasta on the latest version of our experimental environment and thus the performance numbers in this section are based on earlier versions of the hardware. To avoid any confusion, we describe the experimental platform on which performance numbers were collected in each of the subsections below.

4.1 Comparing Cashmere with TreadMarks

TreadMarks [ACD96; KCZ92] is widely recognized as the benchmark system for S-DSM on local-area networks. In this section, we compare the performance of one-level versions of both Cashmere and TreadMarks, with an eye toward evaluating the impact of faster system-area networks on coherence protocol design. While the two-level version of Cashmere provides substantial performance improvements over the one-level implementation, a two-level version of TreadMarks was unavailable at the time this comparison was undertaken. Therefore we limited ourselves to the comparison of the one-level versions.

The performance numbers presented in this section were collected on eight DEC AlphaServer 2100 4/233 computers connected by the first generation of the Memory Channel network. Each AlphaServer was equipped with four 21064A processors operating at 233Mhz and with 256MB of physical memory. The Memory Channel 1 interconnect provides a process-to-process message latency of 5.2 μs and per-link transfer bandwidths of 29MB/s. Aggregate peak bandwidth is about 60MB/s.

We have discussed the relative performance of Cashmere and TreadMarks in considerable detail in an earlier paper [KHS97]. Unfortunately the version of Cashmere presented in that paper used *write-doubling* to collect remote changes to a page.

In that initial design we employed an extra compilation pass to add an additional write to Memory Channel space for every write to the shared memory program area. This extra write would propagate the modified value to the home node of the page, keeping the master copy of the data continually up-to-date. The goal was to overlap computation with communication. Later experiments revealed, however, that the computational cost, network interface occupancy, and additional network contention of write doubling on the Memory Channel were detrimental to performance.

In a subsequent redesign of the system we moved away from write-doubling and adopted *twins and diffs* to collect remote changes to a page [SDH97]. Figure 5 shows speedup results with this later (but still one-level) version of the protocol. Neither Cashmere nor TreadMarks is taking advantage of intra-node hardware coherence in this experiment; both treat the hardware as 32 separate processors. An exception occurs with Barnes, in which performance becomes worse (for both protocols) with more than 16 processors. To ensure a reasonable comparison, we report results for 16 processors in this case.

For five of the applications (SOR, LU, Water-Nsquared, EM3D, and Barnes), Cashmere outperforms TreadMarks by 18% to 48%. For two applications (Gauss and TSP), performance is approximately the same. For Ilink, TreadMarks outperforms Cashmere by approximately 24%. For the five applications in which Cashmere outperforms TreadMarks, the difference stems mainly from the ability to get new versions of a page from a single home node, rather than having to collect diffs from multiple previous writers. For Ilink, in which TreadMarks performs better than Cashmere, the difference derives from the sparsity of the application's data structures. Only a small portion of each page is modified between synchronization operations, and TreadMarks benefits from the ability to fetch diffs rather than whole pages. This is also the reason for TreadMarks' slightly improved performance on Gauss.

Overall, our conclusion is that the lower latency provided by modern system-area networks favors the "less aggressively lazy" release consistency of Cashmere—a conclusion shared by the designers of Princeton's HLRC [SBI98]. The directory structure used by Cashmere and HLRC is also significantly simpler than the comparable structure in TreadMarks (distributed intervals and timestamps). Interestingly, the comparative overhead of dependence tracking in Cashmere and TreadMarks depends on the nature of the application: Cashmere may force a process to invalidate a falsely shared page at an acquire operation even when that page has been modified only by a non-causally-related process. Conversely, TreadMarks requires causally-related write notice propagation to processes at the time of an acquire, even for pages they never touch.

4.2 Tradeoffs between Cashmere and Fine-grain Software Shared Memory

Shasta [SGT96] is a software-only S-DSM system designed for the same hardware as Cashmere. Like Cashmere, Shasta exploits low-latency messages in its coherence protocol, but rather than leverage virtual memory, it relies on compiler support to insert coherence checks in-line before load and store instructions. Aggressive static analysis serves to eliminate redundant checks in many cases, and the lack of reliance on VM allows the system to associate checks with small and variable-size coherence

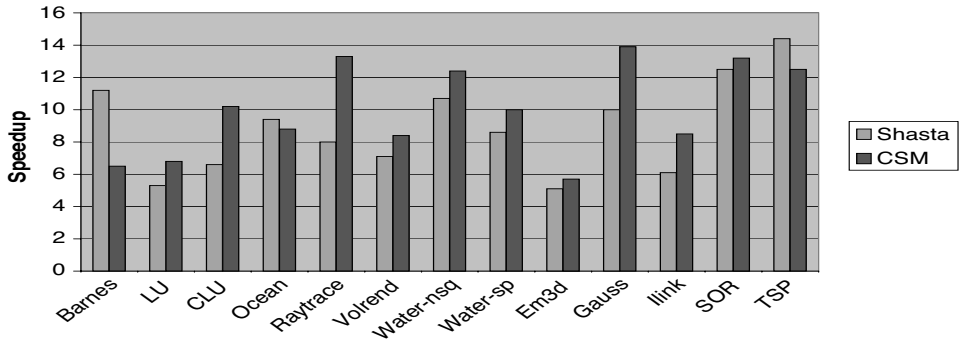

Fig. 6. Comparative speedup for Shasta and Cashmere on our application suite.

blocks. Both systems leverage intra-node hardware coherence. Software is invoked only in cases of inter-node sharing. In collaboration with the Shasta designers, we undertook a detailed comparison of the two systems [DGK99]; we summarize that comparison here. A similar study appeared in [ZIS97]. The main difference between our work and that study is our use of SMP nodes and a software-only fine-grain coherence protocol.

The performance numbers presented in this section were collected on four DEC Alpha Server 4100 multiprocessors connected by the second generation of the Memory Channel network. Each AlphaServer had four 400Mhz 21164 processors with 512MB of memory. The Memory Channel 2 provides a process-to-process message latency of 3.2 μs and per-link transfer bandwidths of 70MB/s. Aggregate peak bandwidth is about 100MB/s.

Figure 6 shows the speedup achieved on 16 processors by the two systems for our application suite. For nine of the applications (LU, CLU, Raytrace, Volrend, Water-Nsquared, Water-spatial, Em3d, Gauss, and Ilink), Cashmere outperforms Shasta by amounts varying between 12% and 66%; for two applications (Ocean and SOR) performance is approximately the same; for the remaining two (Barnes and TSP) Shasta outperforms Cashmere by 72% and 15% respectively.

These numbers are based on the best version of each application on each system. Some of the applications (notably Raytrace, Volrend, and Barnes) had to be modified before they could yield good performance on Cashmere [DGK99]. These modifications were all quite minor, but did require an understanding of the program source. They yielded performance improvements for Shasta as well, but nowhere near as much as they did for Cashmere.

In general, we found that Shasta's ability to enforce coherence on blocks smaller than the virtual memory pages used by Cashmere provides significant advantages for programs that exhibit fine-grain temporal and spatial sharing. This advantage, however, comes at a price in more coarse-grain applications, due to the overhead of in-line coherence checks. Furthermore, in many cases smaller coherence blocks increase the total number of inter-node coherence operations: larger coherence blocks are more natural in coarse-grain applications. Conversely, we have discovered that for systems like Cashmere, which detect access faults using the virtual memory hardware, the main sources of overhead are critical section dilation and the commu-

nication of unneeded data in computations with spatially fine-grain access. Certain programming idioms that are commonly used in the context of hardware supported shared memory, such as work queues, parallel reductions, and atomic counters, can cause excessive communication overhead unless they are tuned for coarser-grain systems. It is our belief that for programmers developing shared-memory applications for clusters "from scratch", VM-based coherence can be expected to provide higher performance than software-only alternatives. Shasta on the other hand provides an easier migration path for shared memory applications developed on hardware-coherent SMPs.

5. RELATED WORK

The original idea of using virtual memory to implement coherence on networks dates from Kai Li's thesis work [Li89b]. Nitzberg and Lo [NiL91] provide a survey of early VM-based systems. Several groups employed similar techniques to migrate and replicate pages in early, cache-less shared-memory multiprocessors [BSF91; LaE91]. Lazy, multi-writer protocols were pioneered by Keleher et al. [KCZ92], and later adopted by several other groups. Several of the ideas in Cashmere were based on Petersen's coherence algorithms for small-scale, non-hardware-coherent multiprocessors [PeL93].

Wisconsin's Blizzard system [SFL94] maintains coherence for cache-line-size blocks, either in software or by using artificially-induced ECC faults to catch references to invalid blocks. It runs on the Thinking Machines CM-5 and provides a sequentially-consistent programming model. The Shasta system [SGT96], described in Section 4.2, extends the software-based Blizzard approach with a relaxed consistency model and variable-size coherence blocks.

AURC [IDF96] is a multi-writer protocol designed for the Shrimp network interface [BLA94]. Like Cashmere, AURC relies on remote memory access to write shared data updates to home nodes. Because the Shrimp interface connects to the memory bus of its ia486-based nodes, AURC is able to double writes in hardware, avoiding a major source of overhead in Cashmere-1L. On the protocol side, AURC uses distributed information in the form of timestamps and write notices to maintain sharing information. As a result, AURC has no need for directory, lock, or write notice metadata: remote-mapped memory is used only for fast messages and doubled writes. Finally, where the Cashmere protocol was originally designed to read lines from remote memory on a cache miss [KoS95b; KoS96], the AURC protocol was designed from the outset with the assumption that whole pages would be copied to local memory.

Inspired by AURC, Samanta et al. [SBI98] developed an SMP-aware Home-based LRC protocol (HLRC) that detects writes and propagates diffs entirely in software. They implemented three versions of their protocol: one in which processes within an SMP do not share any state (similar to CSM-1L); one in which processes within an SMP share data but have independent page tables and thus can have independent views of a page's state (similar to CSM-2L); and one in which all processes within an SMP share data *and* page tables, so that coherence operations on behalf of a process affect all processes in the SMP. Reported results indicate that the ability to share state improves performance, but that this state is best accessed through separate page tables. These results are consistent with our findings in Cashmere. The most

significant difference between CSM-2L and the best-performing HLRC variant is the latter's continued use of vector timestamps, passed through synchronization objects, to avoid page update operations on non-causally-related execution paths. Like hardware coherence protocols, Cashmere opts for simplicity by ensuring that all of a process's data are up-to-date at synchronization time.

A variety of systems implement coherence entirely in software, without VM support, but require programmers to adopt a special programming model. In some systems, such as Split-C [CDG93] and Shrimp's Deliberate Update [BLA94], the programmer must use special primitives to read and write remote data. In others, including Shared Regions [SGZ93], Cid [Nik94], and CRL [JKW95], remote data are accessed with the same notation used for local data, but only in regions of code that have been bracketed by special operations. The Midway system [ZSB94] requires the programmer to associate shared data with synchronization objects, allowing ordinary synchronization acquires and releases to play the role of the bracketing operations. Several other systems use the member functions of an object-oriented programming model to trigger coherence operations [CAL89; FCN94; TKB92]. The extra information provided by such systems to the coherence system can lead to superior performance at the cost of extra programming effort. In some cases, it may be possible for an optimizing compiler to obtain the performance of the special programming model without the special syntax [DCZ96].

Cox et al. were among the first to study layered hardware/software coherence protocols [CDK94]. They simulate a system of 8-way, bus-based multiprocessors connected by an ATM network, using a protocol derived from TreadMarks [ACD96], and show that for clustering to provide significant benefits, reduction in inter-node messages and bandwidth requirements must be proportional to the degree of clustering. Karlsson and Stenstrom [KaS96] examined a similar system in simulation and found that the limiting factor in performance was the latency rather than the bandwidth of the message-level interconnect. Bilas et al. [BIM96] present a simulation study of the automatic update release consistent (AURC) protocol on SMP nodes. They find that the write-through traffic of AURC, coupled with the fact that processors within an SMP have to share the same path to the top-level network, can result in TreadMarks-style lazy release consistency performing better than AURC with clustering.

Yeung et al. [YKA96], in their MGS system, were the first to actually implement a layered coherence protocol. They use a Munin-like multi-writer protocol to maintain coherence across nodes. The protocol also includes an optimization that avoids unnecessary diff computation in the case of a single writer. The MGS results indicate that performance improves with larger numbers of processors per node in the cluster. MGS is implemented on the Alewife [ABC95] hardware shared memory multiprocessor with a mesh interconnect, allowing per-node network bandwidth to scale up with the number of processors. In contrast, commercial multiprocessors typically provide a single connection (usually through the I/O bus) that all processors within a node have to share, thus limiting the benefits of larger nodes.

SoftFLASH [ENC96] is a kernel-level implementation of a two-level coherence protocol on a cluster of SMPs. The protocol is based on the hardware coherent FLASH protocol. Shared data are tracked via TLB faults, and page tables are shared among processes, necessitating the frequent use of inter-processor TLB

shootdown. Experimental results with SoftFLASH indicate that any beneficial effects of clustering are offset by increased communication costs. These results are largely explained, we believe, by the heavy overhead of shootdown, especially for larger nodes.

Bilas et al. [BLS99] use their GeNIMA S-DSM to examine the impact of special network features on S-DSM performance. Their network has remote write, remote read, and specialized lock support, but no broadcast or total ordering. The GeNIMA results show that a combination of remote write, remote read, and synchronization support help avoid the need for interrupts or polling and provide moderate improvements in S-DSM performance. However, the base protocol against which they perform their comparisons uses inter-processor interrupts to signal message arrival. Interrupts on commodity machines are typically on the order of a hundred microseconds, and so largely erase the benefits of a low-latency network [KHS97]. Our evaluation assumes that messages can be detected through a much more efficient polling mechanism, as is found with other SANs [DRM98; vBB95], and so each of our protocols benefits from the same low messaging latency. Our evaluation also goes beyond the GeNIMA work by examining protocol optimizations that are closely tied to the use (or non-use) of special network features. One of the protocol optimizations, home node migration, cannot be used when shared data are remotely accessible, while another optimization, adaptive data broadcast, relies on a very efficient mapping of remotely accessible memory.

Speight and Bennett [SpB98] evaluate the use of multicast and multithreading in the context of S-DSM on high-latency unreliable networks. In their environment, remote processors must be interrupted to process multicast messages, thereby resulting in higher penalties when updates are unnecessary. In addition, while their adaptive protocol is purely history-based, Cashmere-ADB relies on information about the current synchronization interval to predict requests for the same data by multiple processors. This allows us to capture multiple-consumer access patterns that do not repeat.

Our home node migration policy is conceptually similar to the page migration policy found in some CC-NUMA multiprocessors [LaL97; VDG96]. Both policies attempt to migrate pages to active writers. The respective mechanisms are very different, however. In the CC-NUMA multiprocessors, the system will attempt to migrate the page only after remote write misses exceed a threshold. The hardware will then invoke the OS to transfer the page to the new home node. In Cashmere, the migration occurs on the first write to a page and also usually requires only an inexpensive directory change. Since the migration mechanism is so lightweight, Cashmere can afford to be very aggressive.

Amza et al. [ACD97] describe adaptive extensions to the TreadMarks [ACD96] protocol that avoid twin/diff operations on shared pages with only a single writer (pages with multiple writers still use twins and diffs). In Cashmere, if a page has only a single writer, the home always migrates to that writer, and so twin/diff operations are avoided. Even in the presence of multiple concurrent writers, our scheme will always migrate to one of the multiple concurrent writers, thereby avoiding twin/diff overhead at one node. Cashmere is also able to take advantage of the replicated directory when making migration decisions (to determine if the home is currently writing the page). Adaptive DSM (ADSM) [MoB98] also incorporates a

history-based sharing pattern characterization technique to adapt between single and multi-writer modes, and between invalidate and update-based coherence. Our adaptive update mechanism uses the initial request to detect sharing, and then uses broadcast to minimize overhead on the processor responding to the request.

6. CONCLUSIONS

This paper has traced the progression of a state-of-the-art software distributed shared memory system over time. The Cashmere system was originally developed to take advantage of new system-area network (SAN) technology that offered fast user level communication, broadcast, the ability to write remote memory, and ordering and reliability guarantees. These features allowed us to develop a simpler protocol that matched or exceeded the performance of lazy release consistency on this type of network. The simplicity of the protocol allowed us to extend it relatively easily to take advantage of intra-node cache coherence, improving performance by an additional 25%.

We also examined tradeoffs among alternative ways of implementing S-DSM on SANs. We considered a software-only approach developed by colleagues at Compaq's Western Research Lab, and also examined multiple variants of the Cashmere protocol that utilize the special Memory Channel features to different degrees. We discovered that while the software-only approach is less sensitive to fine-grain sharing, a VM-based system continues to provide a noticeable performance advantage for "well tuned" applications. We also discovered, to our surprise, that most of the performance benefits realized by Cashmere are due to fast user-level messaging; the protocol is much less dependent on other network features (ordering, broadcast, total ordering) than initially expected. Nonetheless, the special network features improve performance for certain applications, with improvements as high as 44%.

High performance technical computing remains an important segment of the computer industry, with increased usage expected in such fields as biotechnology and life sciences. Moreover, clusters are becoming the de facto hardware platform in these fields, with basic building blocks ranging from uniprocessor machines to medium-size multiprocessors. As of this writing, two of the top three supercomputers in the world [top500] are SAN-connected clusters. While hand-tuned message-passing code is likely to remain the dominant programming paradigm at the high end, S-DSM systems will continue to provide an attractive solution for medium-scale systems, particularly for programmers "migrating up" from small-scale SMPs, where shared memory is the programming model of choice due to its ease of use and programmer familiarity.

ACKNOWLEDGMENTS

DeQing Chen, Michal Cierniak, Nikolaos Hardavellas, Sotirios Ioannidis, Grigorios Magklis, Wagner Meira, Srinivasan Parthasarathy, Alex Poulos, and Mohammed Zaki contributed to one or more of the Cashmere implementations. The Shasta results reported in Section 4.2 were obtained with the generous assistance of Dan Scales and Kourosh Gharachorloo. The authors would like to thank Ricardo Bianchini and Alan L. Cox for many helpful discussions concerning this paper.

REFERENCES

[AdH93] ADVE, S. V. and HILL, M. D. A Unified Formulation of Four Shared-Memory Models. *IEEE Trans. on Parallel and Distributed Systems*, 4(6):613–624, June 1993.

[ABC95] AGARWAL, A., BIANCHINI, R., CHAIKEN, D., JOHNSON, K., KRANZ, D., KUBIATOWICZ, J., LIM, B.-H., MACKENZIE, K., and YEUNG, D. The MIT Alewife Machine: Architecture and Performance. In *Proc. of the 22nd Intl. Symp. on Computer Architecture*, Santa Margherita Ligure, Italy, June 1995.

[Ame96] AMERICAN NATIONAL STANDARDS INSTITUTE. Information Systems — High-Performance Parallel Interface — Mechanical, Electrical, and Signalling Protocol Specification (HIPPI-PH). ANSI X3.183-1991 (R1996), New York, NY, 1996. Also ISO/IEC 11518-1:1995.

[ACD96] AMZA, C., COX, A. L., DWARKADAS, S., KELEHER, P., LU, H., RAJAMONY, R., YU, W., and ZWAENEPOEL, W. TreadMarks: Shared Memory Computing on Networks of Workstations. *Computer*, 29(2):18–28, Feb. 1996.

[ACD97] AMZA, C., COX, A., DWARKADAS, S., and ZWAENEPOEL, W. Software DSM Protocols that Adapt between Single Writer and Multiple Writer. In *Proc. of the 3rd Intl. Symp. on High Performance Computer Architecture*, San Antonio, TX, Feb. 1997.

[BCZ90] BENNETT, J. K., CARTER, J. B., and ZWAENEPOEL, W. Adaptive Software Cache Management for Distributed Shared Memory Architectures. In *Proc. of the 17th Intl. Symp. on Computer Architecture*, Seattle, WA, May 1990.

[BIM96] BILAS, A., IFTODE, L., MARTIN, D., and SINGH, J. P. Shared Virtual Memory Across SMP Nodes Using Automatic Update: Protocols and Performance. Technical Report TR-517-96, Dept. of Computer Science, Princeton Univ., Oct. 1996.

[BLS99] BILAS, A., LIAO, C., and SINGH, J. P. Using Network Interface Support to Avoid Asynchronous Protocol Processing in Shared Virtual Memory Systems. In *Proc. of the 26th Intl. Symp. on Computer Architecture*, Atlanta, GA, May 1999.

[BLA94] BLUMRICH, M., LI, K., ALPERT, R., DUBNICKI, C., FELTEN, E., and SANDBERG, J. Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer. In *Proc. of the 21st Intl. Symp. on Computer Architecture*, Chicago, IL, Apr. 1994.

[BFS89] BOLOSKY, W. J., FITZGERALD, R. P., and SCOTT, M. L. Simple But Effective Techniques for NUMA Memory Management. In *Proc. of the 12th ACM Symp. on Operating Systems Principles*, Litchfield Park, AZ, Dec. 1989.

[BSF91] BOLOSKY, W. J., SCOTT, M. L., FITZGERALD, R. P., FOWLER, R. J., and COX, A. L. NUMA Policies and Their Relation to Memory Architecture. In *Proc. of the 4th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, Santa Clara, CA, Apr. 1991.

[BoS92] BOLOSKY, W. J. and SCOTT, M. L. Evaluation of Multiprocessor Memory Systems Using Off-Line Optimal Behavior. *Journal of Parallel and Distributed Computing*, 15(4):382–398, Aug. 1992.

[BJM96] BUZZARD, G., JACOBSON, D., MACKEY, M., MAROVICH, S., and WILKES, J. An Implementation of the Hamlyn Sender-Managed Interface Architecture. In *Proc. of the 2nd Symp. on Operating Systems Design and Implementation*, Seattle, WA, Oct. 1996.

[CBZ91] CARTER, J. B., BENNETT, J. K., and ZWAENEPOEL, W. Implementation and Performance of Munin. In *Proc. of the 13th ACM Symp. on Operating Systems Principles*, Pacific Grove, CA, Oct. 1991.

[CAL89] CHASE, J. S., AMADOR, F. G., LAZOWSKA, E. D., LEVY, H. M., and LITTLEFIELD, R. J. The Amber System: Parallel Programming on a Network of Multiprocessors. In *Proc. of the 12th ACM Symp. on Operating Systems Principles*, Litchfield Park, AZ, Dec. 1989.

[Com97] COMPAQ, INTEL, AND MICROSOFT. Virtual Interface Architecture Specification. Draft Revision 1.0, Dec. 1997. Available at ftp://download.intel.com/design/servers/vi/san_10.pdf.

[CoF89] COX, A. L. and FOWLER, R. J. The Implementation of a Coherent Memory Abstraction on a NUMA Multiprocessor: Experiences with PLATINUM. In *Proc. of the 12th ACM Symp. on Operating Systems Principles*, Litchfield Park, AZ, Dec. 1989.

[CDK94] COX, A., DWARKADAS, S., KELEHER, P., LU, H., RAJAMONY, R., and ZWAENEPOEL, W. Software Versus Hardware Shared-Memory Implementation: A Case Study. In *Proc. of the 21st Intl. Symp. on Computer Architecture*, Chicago, IL, Apr. 1994.

[CDG93] CULLER, D., DUSSEAU, A., GOLDSTEIN, S., KRISHNAMURTHY, A., LUMETTA, S., VON EICKEN, T., and YELICK, K. Parallel Programming in Split-C. In *Proc., Supercomputing '93*, Portland, OR, Nov. 1993.

[DRM98] DUNNING, D., REGNIER, G., MCALPINE, G., CAMERON, D., SHUBERT, B., BERRY, F., MERRITT, A. M., GRONKE, E., and DODD, C. The Virtual Interface Architecture. *IEEE Micro*, 18(2):66–76, Mar. 1998.

[DSC94] DWARKADAS, S., SCHÄFFER, A. A., COTTINGHAM JR., R. W., COX, A. L., KELEHER, P., and ZWAENEPOEL, W. Parallelization of General Linkage Analysis Problems. *Human Heredity*, 44:127–141, 1994.

[DCZ96] DWARKADAS, S., COX, A. L., and ZWAENEPOEL, W. An Integrated Compile-Time/Run-Time Software Distributed Shared Memory System. In *Proc. of the 7th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, Cambridge, MA, Oct. 1996.

[DHK99] DWARKADAS, S., HARDAVELLAS, N., KONTOTHANASSIS, L. I., NIKHIL, R., and STETS, R. Cashmere-VLM: Remote Memory Paging for Software Distributed Shared Memory. In *Proc. of the 13th Intl. Parallel Processing Symp.*, San Juan, Puerto Rico, Apr. 1999.

[DGK99] DWARKADAS, S., GHARACHORLOO, K., KONTOTHANASSIS, L. I., SCALES, D. J., SCOTT, M. L., and STETS, R. Comparative Evaluation of Fine- and Coarse-Grain Approaches for Software Distributed Shared Memory. In *Proc. of the 5th Intl. Symp. on High Performance Computer Architecture*, Orlando, FL, Jan. 1999.

[ENC96] ERLICHSON, A., NUCKOLLS, N., CHESSON, G., and HENNESSY, J. SoftFLASH: Analyzing the Performance of Clustered Distributed Virtual Shared Memory. In *Proc. of the 7th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, Cambridge, MA, Oct. 1996.

[FCN94] FEELEY, M. J., CHASE, J. S., NARASAYYA, V. R., and LEVY, H. M. Integrating Coherency and Recovery in Distributed Systems. In *Proc. of the 1st Symp. on Operating Systems Design and Implementation*, Monterey, CA, Nov. 1994.

[FiG97] FILLO, M. and GILLETT, R. B. Architecture and Implementation of Memory Channel 2. *Digital Technical Journal*, 9(1):27–41, 1997.

[Gil96] GILLETT, R. Memory Channel: An Optimized Cluster Interconnect. *IEEE Micro*, 16(2):12–18, Feb. 1996.

[Goo87] GOODMAN, J. R. Coherency for Multiprocessor Virtual Address Caches. In *Proc. of the 2nd Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, Palo Alto, CA, Oct. 1987.

[HLR93] HILL, M. D., LARUS, J. R., REINHARDT, S. K., and WOOD, D. A. Cooperative Shared Memory: Software and Hardware for Scalable Multiprocessors. *ACM Trans. on Computer Systems*, 11(4):300–318, Nov. 1993.

[IDF96] IFTODE, L., DUBNICKI, C., FELTEN, E. W., and LI, K. Improving Release-Consistent Shared Virtual Memory Using Automatic Update. In *Proc. of the 2nd Intl. Symp. on High Performance Computer Architecture*, San Jose, CA, Feb. 1996.

[Inf02] INFINIBAND TRADE ASSOCIATION. InfiniBand Architecture Specification. Release 1.1, Nov. 2002. Available at www.infinibandta.org/specs.

[JKW95] JOHNSON, K. L., KAASHOEK, M. F., and WALLACH, D. A. CRL: High-Performance All-Software Distributed Shared Memory. In *Proc. of the 15th ACM Symp. on Operating Systems Principles*, Copper Mountain, CO, Dec. 1995.

[KaS96] KARLSSON, M. and STENSTRO¨M, P. Performance Evaluation of a Cluster-Based Multiprocessor Built from ATM Switches and Bus-Based Multiprocessor Servers. In *Proc. of the 2nd Intl. Symp. on High Performance Computer Architecture*, San Jose, CA, Feb. 1996.

[KCZ92] KELEHER, P., COX, A. L., and ZWAENEPOEL, W. Lazy Release Consistency for Software Distributed Shared Memory. In *Proc. of the 19th Intl. Symp. on Computer Architecture*, Gold Coast, Australia, May 1992.

[KoS95a] KONTOTHANASSIS, L. I. and SCOTT, M. L. High Performance Software Coherence for Current and Future Architectures. *Journal of Parallel and Distributed Computing*, 29(2):179–195, Nov. 1995.

[KoS95b] KONTOTHANASSIS, L. I. and SCOTT, M. L. Software Cache Coherence for Large Scale Multiprocessors. In *Proc. of the 1st Intl. Symp. on High Performance Computer Architecture*, Raleigh, NC, Jan. 1995.

[KoS96] KONTOTHANASSIS, L. I. and SCOTT, M. L. Using Memory-Mapped Network Interfaces to Improve the Performance of Distributed Shared Memory. In *Proc. of the 2nd Intl. Symp. on High Performance Computer Architecture*, San Jose, CA, Feb. 1996.

[KHS97] KONTOTHANASSIS, L. I., HUNT, G. C., STETS, R., HARDAVELLAS, N., CIERNIAK, M., PARTHASARATHY, S., MEIRA, W., DWARKADAS, S., and SCOTT, M. L. VM-Based Shared Memory on Low-Latency, Remote-Memory-Access Networks. In *Proc. of the 24th Intl. Symp. on Computer Architecture*, Denver, CO, June 1997.

[LaE91] LAROWE JR., R. P. and ELLIS, C. S. Experimental Comparison of Memory Management Policies for NUMA Multiprocessors. *ACM Trans. on Computer Systems*, 9(4):319–363, Nov. 1991.

[LaL97] LAUDON, J. and LENOSKI, D. The SGI Origin: A ccNUMA Highly Scalable Server. In *Proc. of the 24th Intl. Symp. on Computer Architecture*, Denver, CO, June 1997.

[Li89a] LI, K. and SCHAEFER, R. A Hypercube Shared Virtual Memory System. In *Proc. of the 1989 Intl. Conf. on Parallel Processing*, St. Charles, IL, Aug. 1989. Penn. State Univ. Press.

[Li89b] LI, K. and HUDAK, P. Memory Coherence in Shared Virtual Memory Systems. *ACM Trans. on Computer Systems*, 7(4):321–359, Nov. 1989.

[MKB95] MARCHETTI, M., KONTOTHANASSIS, L. I., BIANCHINI, R., and SCOTT, M. L. Using Simple Page Placement Policies to Reduce the Cost of Cache Fills in Coherent Shared-Memory Systems. In *Proc. of the 9th Intl. Parallel Processing Symp.*, Santa Barbara, CA, Apr. 1995.

[MoB98] MONNERAT, L. R. and BIANCHINI, R. Efficiently Adapting to Sharing Patterns in Software DSMs. In *Proc. of the 4th Intl. Symp. on High Performance Computer Architecture*, Las Vegas, NV, Feb. 1998.

[Nik94] NIKHIL, R. S. Cid: A Parallel, "Shared-memory" C for Distributed-Memory Machines. In *Proc. of the 7th Annual Workshop on Languages and Compilers for Parallel Computing*, Aug. 1994.

[NiL91] NITZBERG, B. and LO, V. Distributed Shared Memory: A Survey of Issues and Algorithms. *Computer*, 24(8):52–60, Aug. 1991.

[PeL93] PETERSEN, K. and LI, K. Cache Coherence for Shared Memory Multiprocessors Based on Virtual Memory Support. In *Proc. of the 7th Intl. Parallel Processing Symp.*, Newport Beach, CA, Apr. 1993.

[PDB97] PHILBIN, J. F., DUBNICKI, C., BILAS, A., and LI, K. Design and Implementation of Virtual Memory-Mapped Communication on Myrinet. In *Proc. of the 11th Intl. Parallel Processing Symp.*, Geneva, Switzerland, Apr. 1997.

[RLW94] REINHARDT, S. K., LARUS, J. R., and WOOD, D. A. Tempest and Typhoon: User-level Shared-Memory. In *Proc. of the 21st Intl. Symp. on Computer Architecture*, Chicago, IL, Apr. 1994.

[SBI98] SAMANTA, R., BILAS, A., IFTODE, L., and SINGH, J. P. Home-based SVM Protocols for SMP clusters: Design and Performance. In *Proc. of the 4th Intl. Symp. on High Performance Computer Architecture*, Las Vegas, NV, Feb. 1998.

[SGZ93] SANDHU, H. S., GAMSA, B., and ZHOU, S. The Shared Regions Approach to Software Cache Coherence on Multiprocessors. In *Proc. of the 4th ACM Symp. on Principles and Practice of Parallel Programming*, San Diego, CA, May 1993.

[SGT96] SCALES, D. J., GHARACHORLOO, K., and THEKKATH, C. A. Shasta: A Low Overhead, Software-Only Approach for Supporting Fine-Grain Shared Memory. In *Proc. of the 7th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, Cambridge, MA, Oct. 1996.

[ScG97] SCALES, D. J. and GHARACHORLOO, K. Towards Transparent and Efficient Software Distributed Shared Memory. In *Proc. of the 16th ACM Symp. on Operating Systems Principles*, St. Malo, France, Oct. 1997.

[SGA98] SCALES, D. J., GHARACHORLOO, K., and AGGARWAL, A. Fine-Grain Software Distributed Shared Memory on SMP Clusters. In *Proc. of the 4th Intl. Symp. on High Performance Computer Architecture*, Las Vegas, NV, Feb. 1998.

[SFL94] SCHOINAS, I., FALSAFI, B., LEBECK, A. R., REINHARDT, S. K., LARUS, J. R., and WOOD, D. A. Fine-grain Access Control for Distributed Shared Memory. In *Proc. of the 6th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, Oct. 1994.

[SWG92] SINGH, J. P., WEBER, W.-D., and GUPTA, A. SPLASH: Stanford Parallel Applications for Shared-Memory. *ACM SIGARCH Computer Architecture News*, 20(1):5–44, Mar. 1992.

[SpB98] SPEIGHT, E. and BENNETT, J. K. Using Multicast and Multithreading to Reduce Communication in Software DSM Systems. In *Proc. of the 4th Intl. Symp. on High Performance Computer Architecture*, Las Vegas, NV, Feb. 1998.

[SDK00] STETS, R., DWARKADAS, S., KONTOTHANASSIS, L. I., RENCUZOGULLARI, U., and SCOTT, M. L. The Effect of Network Total Order, Broadcast, and Remote-Write Capability on Network-Based Shared Memory Computing. In *Proc. of the 6th Intl. Symp. on High Performance Computer Architecture*, Toulouse, France, Jan. 2000.

[SDH97] STETS, R., DWARKADAS, S., HARDAVELLAS, N., HUNT, G. C., KONTOTHANASSIS, L. I., PARTHASARATHY, S., and SCOTT, M. L. Cashmere-2L: Software Coherent Shared Memory on a Clustered Remote-Write Network. In *Proc. of the 16th ACM Symp. on Operating Systems Principles*, St. Malo, France, Oct. 1997.

[TKB92] TANENBAUM, A. S., KAASHOEK, M. F., and BAL, H. E. Parallel Programming Using Shared Objects and Broadcasting. *Computer*, 25(8):10–19, Aug. 1992.

[top500] *Top 500 Supercomputer Sites*. Univ. of Manheim, Univ. of Tennessee, and NERSC/LBNL, June 2003. http://www.top500.org/lists/2003/06/.

[VDG96] VERGHESE, B., DEVINE, S., GUPTA, A., and ROSENBLUM, M. Operating System Support for Improving Data Locality on CC-NUMA Compute Servers. In *Proc. of the 7th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, Cambridge, MA, Oct. 1996.

[vBB95] VON EICKEN, T., BASU, A., BUCH, V., and VOGELS, W. U-Net: A User-Level Network Interface for Parallel and Distributed Computing. In *Proc. of the 15th ACM Symp. on Operating Systems Principles*, Copper Mountain, CO, Dec. 1995.

[WBv97] WELSH, M., BASU, A., and VON EICKEN, T. Incorporating Memory Management into User-Level Network Interfaces. Technical Report TR97-1620, Cornell Univ., Aug. 1997.

[WOT95] WOO, S. C., OHARA, M., TORRIE, E., SINGH, J. P., and GUPTA, A. Methodological Considerations and Characterization of the SPLASH-2 Parallel Application Suite. In *Proc. of the 22nd Intl. Symp. on Computer Architecture*, Santa Margherita Ligure, Italy, June 1995.

[YKA96] YEUNG, D., KUBIATOWITCZ, J., and AGARWAL, A. MGS: A Multigrain Shared Memory System. In *Proc. of the 23rd Intl. Symp. on Computer Architecture*, Philadelphia, PA, May, 1996.

[ZSB94] ZEKAUSKAS, M. J., SAWDON, W. A., and BERSHAD, B. N. Software Write Detection for Distributed Shared Memory. In *Proc. of the 1st Symp. on Operating Systems Design and Implementation*, Monterey, CA, Nov. 1994.

[ZIS97] ZHOU, Y., IFTODE, L., SINGH, J. P., LI, K., TOONEN, B. R., SCHOINAS, I., HILL, M. D., and WOOD, D. A. Relaxed Consistency and Coherence Granularity in DSM Systems: A Performance Evaluation. In *Proc. of the 6th ACM Symp. on Principles and Practice of Parallel Programming*, Las Vegas, NV, June 1997.