The Effect of Network Total Order, Broadcast, and Remote-Write Capability on Network-Based Shared Memory Computing*

Robert Stets[†], Sandhya Dwarkadas, Leonidas Kontothanassis[‡], Umit Rencuzogullari, and Michael L. Scott

Dept. of Comp. Science, University of Rochester, Rochester, NY 14627

[†]Western Research Lab, Compaq Computer Corp., Palo Alto, CA 94301

[‡]Cambridge Research Lab, Compaq Computer Corp., Cambridge, MA 02139

Abstract

Emerging system-area networks provide a variety of features that can dramatically reduce network communication overhead. In this paper, we evaluate the impact of such features on the implementation of Software Distributed Shared Memory (SDSM), and on the Cashmere system in particular. Cashmere has been implemented on the Compaq Memory Channel network, which supports low-latency messages, protected remote memory writes, inexpensive broadcast, and total ordering of network packets.

Our evaluation is based on several Cashmere protocol variants, ranging from a protocol that fully leverages the Memory Channel's special features to one that uses the network only for fast messaging. We find that the special features improve performance by 18–44% for three of our applications, but less than 12% for our other seven applications. We also find that home node migration, an optimization available only in the message-based protocol, can improve performance by as much as 67%.

These results suggest that for systems of modest size, low latency is much more important for SDSM performance than are remote writes, broadcast, or total ordering. At the same time, results on an emulated 32-node system indicate that broadcast based on remote writes of widely-shared data may improve performance by up to 51% for some applications. If hardware broadcast or multicast facilities can be made to scale, they can be beneficial in future system-area networks.

*This work was supported in part by NSF grants CDA–9401142, EIA–9972881, CCR–9702466, and CCR–9705594; and an external research grant from Compaq. Leonidas Kontothanassis is now with Akamai Technologies, Inc., 201 Broadway, Cambridge, MA 02139.

1. Introduction

Recent technological advances have led to the commercial availability of inexpensive system area networks (SANs) on which a processor can access the memory of a remote node safely from user space [5, 6, 15]. These memory-mapped network interfaces provide users with high bandwidth (>75MB/s), low latency (2–3μs) communication. This latency is two to three decimal orders of magnitude lower than that of traditional networks. In addition, these SANs sometimes also provide reliable, inexpensive broadcast and total ordering of packets [10, 15, 16].

In comparison to the traditional network of (uniprocessor) workstations, a cluster of symmetric multiprocessor (SMP) nodes on a high-performance SAN can see much lower communication overhead. Communication within the same node can occur through shared memory, while cross-SMP communication overhead can be ameliorated by the high performance network. Several groups have developed software distributed shared memory (SDSM) protocols that exploit low-latency networks [18, 22, 24, 28].

In this paper, we examine the impact of advanced networking features on the performance of the state-of-the-art Cashmere-2L [28] protocol. The Cashmere protocol uses the virtual memory subsystem to track data accesses, allows multiple concurrent writers, employs home nodes (*i.e.* maintains one master copy of each shared data page), maintains a global page directory, and leverages shared memory within SMPs to reduce protocol overhead. In practice, Cashmere-2L has been shown to have very good performance [12, 28].

Cashmere was originally designed to maximize performance by placing shared data directly in remotely writable memory, using remote writes and broadcast to replicate the page directory among nodes, and relying on network total order and reliability to avoid acknowledging the receipt of metadata information. This paper evaluates the performance implications of each of these design decisions.

Our investigation builds on earlier results from the GeNIMA SDSM [4]. GeNIMA's creators examined the performance impact of remote reads, remote writes, and specialized locking suport in the network interface. In our investigation, we examine remote writes, inexpensive broadcast, and network total order. In subsequent sections, we will explain how these features are used by Cashmere and could be or are used by GeNIMA. We also examine two effective protocol optimizations, both of which have ramifications for the use of special hardware.

In general, an SDSM protocol incurs three kinds of communication: the propagation of shared *data*, the maintenance of *metadata* (internal protocol data structures), and *synchronization*. We have constructed several variants of the Cashmere protocol that allow us to isolate the impact of Memory Channel features on each of these kinds of communication. Overall, we find that only three of our ten benchmark applications can obtain significant performance advantages (more than 12%) from a protocol that takes full advantage of the Memory Channel's special features in comparison with an alternative protocol based entirely on point to point messages. The message-only protocol has simpler hardware requirements, and allows the size of shared memory to grow beyond the addressing limits of the network interface.[1] It also enables us to implement variants of Cashmere that employ home node migration. These variants improve performance by as much as 67%, more than offsetting the advantage of using special Memory Channel features. These results suggest that for systems of modest size (up to 8 nodes), low latency is much more important for SDSM performance than are remote writes, broadcast, or total ordering. However, broadcasting using remote writes, if it can be scaled to larger numbers of nodes, can be beneficial for applications with widely shared data. Results on an emulated 32-node system suggest that the availability of inexpensive broadcast can improve the performance of these applications by as much as 51%.

The next section discusses the Memory Channel and its special features, along with the Cashmere protocol. Section 3 evaluates the impact of the Memory Channel features and the home node migration and broadcast optimizations. Section 4 covers related work, and Section 5 outlines our conclusions.

[1] Most current commodity remote access networks have a limited remotely-accessible memory space. Methods to eliminate this restriction are a focus of ongoing research [7, 30].

2. Protocol Variants and Implementation

Cashmere was designed for SMP clusters connected by a high performance system area network, such as Compaq's Memory Channel [15]. Earlier work on Cashmere [12, 28] and other systems [12, 14, 22, 23, 24] has quantified the benefits of SMP nodes to SDSM performance. In this paper, we examine the performance impact of the special network features.

We begin by providing an overview of the Memory Channel network and its programming interface. Following this overview is a description of the Cashmere protocol and of its network communication in particular. A discussion of the design decisions related to SMP nodes can be found in earlier work [28].

2.1. 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* or *receive regions* in the Memory Channel's address space. 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.

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 is 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 given channel. A loopback message goes out through the hub 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 network hub. Broadcast and total ordering, together with loopback, are useful in implementing cluster-wide synchronization, to be described in the following section.

2.2. Protocol Overview

Cashmere is an *SMP-aware* protocol. The protocol allows all data sharing within an SMP to occur through the hardware coherence mechanism in the SMP. Software coherence overhead is incurred only when sharing spans nodes.

Cashmere uses the virtual memory (VM) subsystem to track data accesses. The coherence unit is an 8KB VM page. Cashmere implements “moderately lazy” release consistency [17]. Invalidation messages are sent at *release* operations, but need not be processed until a subsequent *acquire* operation. Cashmere requires all applications to follow a *data-race-free* [1] 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.

In Cashmere, each page of shared memory has a single, distinguished *home node* and an entry in a global *page directory*. The home node maintains a master copy of the page. The directory entry identifies the home node of the page and the members of its *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 the sharing set information in the 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 will also create a pristine copy of the page (called a *twin*) and add 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.

At a release operation, the protocol examines each page in the dirty list and compares the page to its twin in order to identify the 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 performing 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 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 for which it has a mapping, and which have not subsequently been updated by another processor.

2.3. Protocol Variants

In order to isolate the effects of Memory Channel features on shared data propagation, protocol metadata maintenance, and synchronization, we evaluate seven variants of the Cashmere protocol, summarized in Table 1. For each of the areas of protocol communication, the protocols either leverage the full Memory Channel capabilities (*i.e.* remote write access, total ordering, and inexpensive broadcast) or instead send explicit messages between nodes. We assume a reliable network (as is common in current SANs). Since we wish to establish ordering, however, most explicit messages require an acknowledgement.

Message Polling: All of our protocol variants rely in some part on efficient explicit messages. To minimize delivery overhead [18], 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. CSM-DMS: Data, Metadata, and Synchronization using Memory Channel.

The base protocol, denoted CSM-DMS, is the Cashmere-2L protocol described in our study on the effects of SMP clusters [28]. This protocol exploits the Memory Channel for all SDSM communication: to propagate shared data, to maintain metadata, and for synchronization.

Data: All shared data is mapped into the Memory Channel address space. Each page is assigned a home node, which is chosen to be 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 is 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 128MBytes of Memory Channel address space, this significantly limits the maximum dataset size. (For eight nodes, the maximum dataset would be only about 16MBytes.)

Modifications are written back to the home node in the form of diffs.[2] With home node copies kept in Memory Channel space these diffs can be applied with remote writes, avoiding the need for processor intervention at the home. Address space limits still constrain dataset size, but the limit is reasonably high.

To avoid race conditions, Cashmere must be sure all diffs are completed before exiting a release operation. To

[2] An earlier Cashmere study [18] investigated using write-through to propagate data modifications. Diffs were found to use bandwidth more efficiently than write-through, and to provide better performance.

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 (32 Nodes)	MC/ADB	MC	MC	No
CSM-ADB (8 Nodes)	Explicit/ADB	MC	MC	Yes

Table 1. These protocol variants have been chosen to isolate the performance impact of special network features on the areas of SDSM 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.

avoid the need for explicit acknowledgements, 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.

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. 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 acknowledgements.

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 asserts and broadcasts its node 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.

2.3.2. 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. CSM-MS, however, cannot use remote-write diffs. Instead, diffs are sent as explicit messages, which require processing assistance from the home node and explicit acknowledgements to establish ordering. In CSM-MS, metadata and synchronization still leverage all Memory Channel features.

2.3.3. 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. The home node is the only node guaranteed to have an up-to-date directory entry.

Directory updates (or reads) 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 acknowledgement. In fact, an explicit directory message is needed only when a page is invalidated.

2.3.4. CSM-None: No Use of Special Memory Channel Features.

The fourth protocol, CSM-None, uses explicit messages (and acknowledgements) 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 message polling mechanism, described above, should be considered independent of remote write; similarly efficient polling can be implemented on other networks [10, 30].

2.3.5. CSM-MS-Mg and CSM-None-Mg: Home Node Migration.

All of the above protocol variants use first-touch home node assignment [20]. Home assignment is extremely important because processors on the home node write directly to the master copy 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.

Migrating home nodes cannot be used when data is remotely accessible. The migration would force a re-map of Memory Channel space that can only be accomplished through a global synchronization. The synchronization would be necessary to ensure that no diffs or other remote memory accesses occur while the migration is proceeding. 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. When a processor incurs a write fault, these protocols check the local copy of the directory to see if the home is actively writing the page. If not, a migration request is sent to the home. The request is granted if received when the home is not writing the page. The home changes the directory entry to point to the new home. Since the new home node has touched the page, the transfer of data occurs as part of the corresponding page update operation. The marginal cost of changing the home node identity is therefore very low.

CSM-None-Mg uses a local copy of page directory information to see whether the home node is writing the page. If this copy is out of date, useless migration requests can occur. We do not present CSM-S-Mg because its performance does not differ significantly from that of CSM-S.

2.3.6. CSM-ADB: Adaptive Shared Data Broadcast. The protocol variants described in the previous sections all use invalidate-based coherence: data is 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 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 multiple-consumer access patterns that are repetitive, as well as those that are not. 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 scaling, we also report CSM-ADB results using 32 processors on a one-level protocol (one that does not leverage hardware shared memory for sharing within the node) described in earlier work [18].

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

Table 2. Basic 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 discuss the results of our investigation of the impact of Memory Channel features and the home node migration and broadcast optimizations.

3.1. Platform and Basic Operation Costs

Our experimental environment 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 II system area network, a PCI-based network with a peak point-to-point bandwidth of 75 MBytes/sec and a one-way, cache-to-cache latency for a 64-bit remote-write operation of 3.3 μs.

Each AlphaServer node runs Digital 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 processors are connected to the Memory Channel. Execution times represent the lowest values of three runs.

In practice, the 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, and a twin operation requires 68 μs.

As described earlier, Memory Channel features can be used to significantly reduce the cost of diffs, directory updates, write notice propagation, and synchronization. Table 2 shows the costs for diff operations, lock acquires, and barriers, both when leveraging (*MC Features*) and not

Program	Problem Size	Time (s)
Barnes	128K bodies (26MBytes)	120.4
CLU	2048x2048 (33MBytes)	75.4
LU	2500x2500 (50MBytes)	143.8
EM3D	64000 nodes (52MBytes)	30.6
Gauss	2048x2048 (33MBytes)	234.8
Ilink	CLP (15MBytes)	212.7
SOR	3072x4096 (50MBytes)	36.2
TSP	17 cities (1MByte)	1342.49
Water-Nsquared	9261 mols. (6MBytes)	332.6
Water-Spatial	9261 mols. (16MBytes)	20.2

Table 3. Data set sizes and sequential execution time of applications.

leveraging (*Explicit Messages*) special Memory Channel features. The cost of diff operations varies according to the size of the diff. Directory updates, write notices, and flag synchronization all use the Memory Channel's remote-write and total ordering features. Directory updates and flag synchronization also rely on the inexpensive broadcast support. Without these features, these operations are accomplished via explicit messages. 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 will depend greatly on the write notice count and destinations. Write notices sent to different destinations can be overlapped, thus reducing the operation's overall latency. Flags are inherently broadcast operations, 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

Our applications consist primarily of well-known benchmarks from the Splash [25, 31] and TreadMarks [2] suites. Due to space limitations, we refer the reader to earlier descriptions [12]. The applications are Barnes, an N-body simulation from the TreadMarks [2] distribution (based on the same application in the SPLASH-1 [25] suite); CLU and LU, lower and upper triangular matrix factorization kernels with and without contiguous allocation of a single processor's data, respectively,[3] from the SPLASH-2 [31] suite; EM3D, a program to simulate electromagnetic wave propagation through 3D objects [9]; 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 [11] package that 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 where the data is partitioned spatially.

The data set sizes and uniprocessor execution times for these applications are presented in Table 3. The size of shared memory is listed in parentheses. Execution times were measured by running each uninstrumented application sequentially without linking it to the protocol library.

3.3. Performance

Throughout this section, we will refer to Figure 1 and Table 4. Figure 1 shows a breakdown of execution time, normalized to that of the CSM-DMS protocol, for the first six protocol variants. 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 number of page transfers, diff operations, home node migrations, and migration attempts (listed in parentheses).

3.3.1. The Impact of Memory Channel Features. This subsection begins by discussing the impact of Memory Channel support, in particular, remote-write capabilities, inexpensive broadcast, and total-ordering properties, on the three types of protocol communication: shared data propagation, protocol metadata maintenance, and synchronization. All protocols described in this subsection use a first-touch initial home node assignment.[4]

Five of our ten applications show a measurable performance advantage running on CSM-DMS (fully leveraging Memory Channel features) as opposed to CSM-None (using explicit messages). Barnes runs 80% faster on CSM-DMS than it does on CSM-None, while 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.

Barnes exhibits a high degree of sharing and incurs a large Wait time on all protocol variants (see Figure 1). CSM-DMS runs roughly 40% faster than CSM-MS and 80% faster than CSM-S and CSM-None. This performance

[3] 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 have modified LU to assign a column of tiles to processors within the same SMP, thereby reducing false sharing across node boundaries.

[4] In the case of multiple sharers per page, the timing differences between 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.

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. As is true with most of our applications, the use of Memory Channel features to optimize synchronization has little impact on overall performance. Synchronization time is dominated by software coherence protocol overhead, and in general limits the performance of applications with active fine-grain synchronization on SDSM.

At the given matrix size, LU incurs a large amount of protocol communication due to write-write false sharing at row boundaries. In this application, 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 diffing overhead. The Message time in CSM-DMS is much lower than in the other protocols. In CSM-MS, CSM-S, and CSM-None, some of the increased Message time is hidden by existing Wait time.

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 1 shows this effect, as Message time increases slightly from CSM-DMS to CSM-MS (18% and 24%, respectively), but Wait time increases dramatically (41% and 65% for CSM-DMS and CSM-MS, respectively). This application provides an excellent example of the sensitivity of synchronization Wait time to any protocol perturbation.

Water-Nsquared obtains its best performance again on CSM-DMS. As can be seen in Figure 1, CSM-MS, CSM-S, and CSM-None all have much higher Protocol times than CSM-DMS. Detailed instrumentation shows that the higher Protocol time is spent in write fault handlers, contending for a set of per-page locks shared by the write fault and diff message handlers. The average time spent acquiring these locks shows a four-fold increase from CSM-DMS to CSM-MS. CSM-DMS does not experience this contention since it relies on remote writes and total ordering to deliver diffs without a message handler. 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.

Like EM3D, Water-Spatial is sensitive to data propagation costs. The higher cost of data propagation 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 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. Of the remaining applications, Gauss, SOR, and TSP are not noticeably dependent on the use of Memory Channel features.

3.3.2. 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 twin/diffs and invalidations, 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; four (LU, Water-Spatial, Barnes, and Water-Nsquared) benefit.

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.[5]

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 show large reductions in 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 lower by 12% and 27% compared to CSM-MS and CSM-None, bringing performance to within 30% of CSM-DMS for CSM-None-Mg. Water-Nsquared shows an 8% and 12% improvement 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.

[5] As described earlier, migration cannot be used when data is placed in the Memory Channel address space (for example, in CSM-DMS), because of the high cost of remapping.

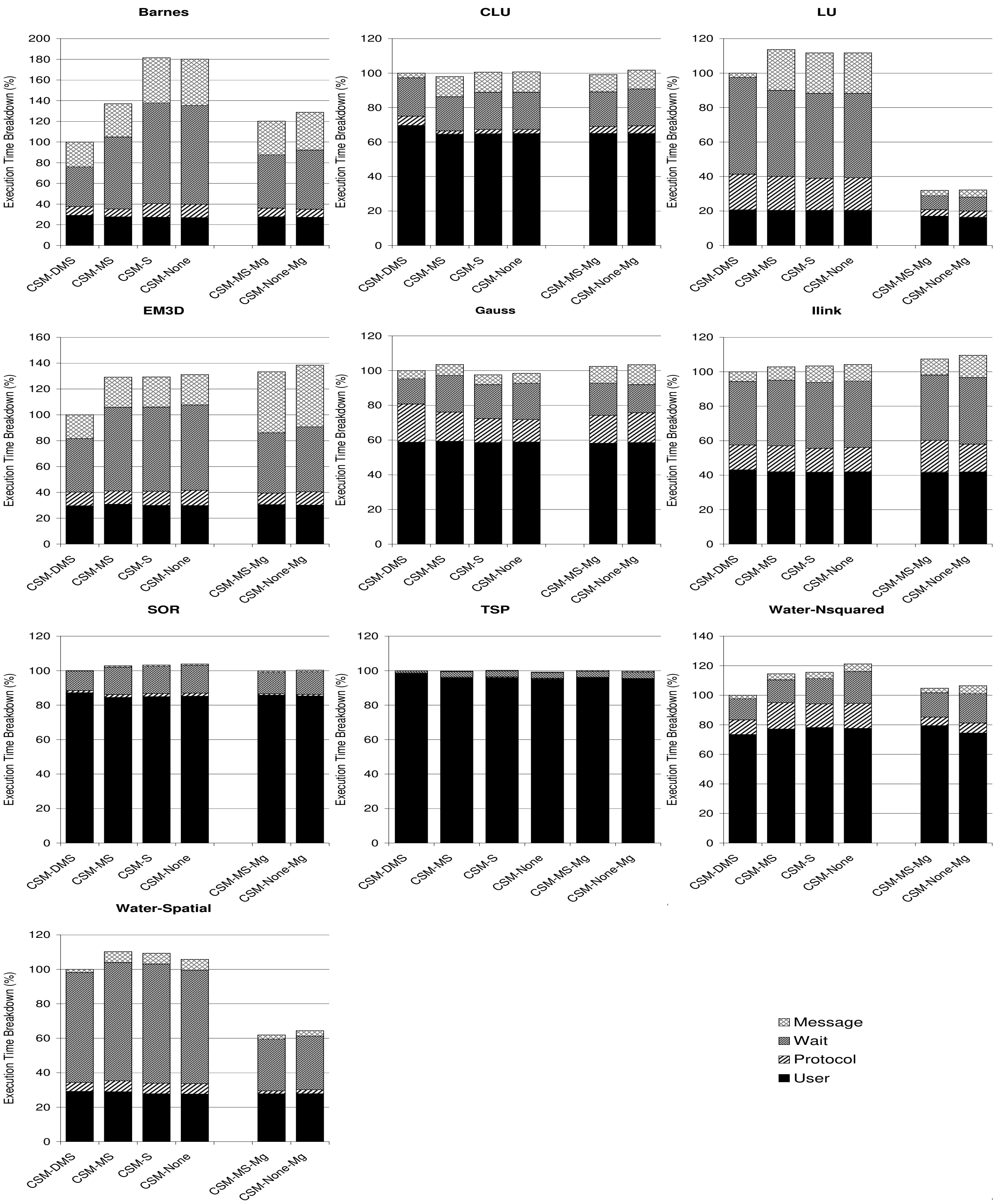

Figure 1. Normalized execution time breakdown for the applications and protocol variants on 32 processors. The suffix on the protocol name indicates the kinds of communication using special Memory Channel features (D: shared Data propagation, M: protocol Meta-data maintenance, S: Synchronization, None: No use of Memory Channel features). Mg indicates a migrating home node policy.

Application		CSM-DMS	CSM-MS	CSM-S	CSM-None	CSM-MS-Mg	CSM-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)	0	0	0	0	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)	0	0	0	0	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)	0	0	0	0	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)	0	0	0	0	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)	0	0	0	0	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)	0	0	0	0	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	0	0	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)	0	0	0	0	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)	0	0	0	0	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	0	0	0	0.3 (0.3)	0.3 (0.3)

Table 4. Application speedups and statistics at 32 processors.

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. 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. For EM3D, CSM-MS-Mg and CSM-None-Mg perform 3% and 6% worse than CSM-MS and CSM-None, respectively. Similarly, for Ilink, CSM-MS-Mg and CSM-None-Mg both perform 5% worse than their first-touch counterparts. Also, CSM-None-Mg suffers from a large number of unsuccessful migration requests (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.

Overall, the migration-based protocol variants deliver very good performance, while avoiding the need to map shared data into the limited amount of remotely addressable 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 to 67%).

3.3.3. Selective Broadcast for Widely Shared Data. Selective use of broadcast for data that is 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). However, at 8 nodes (see Figure 2), the performance improvement across all applications is a maximum of 13%.

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 [6]. Performance improvements at 32 nodes jumps to 18, 49, and 51% for LU, Ilink and Gauss, respectively . The large gains for LU, Ilink, and Gauss come from a reduction in the message and wait time. In Gauss, the protocol is able to detect and optimize the communication of each pivot row to the mul-

[6]Emulation differs from a real 32-node system in that the (four) processors within a node share the same network interface and messages among processors within a node are exchanged through shared memory. We also 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.

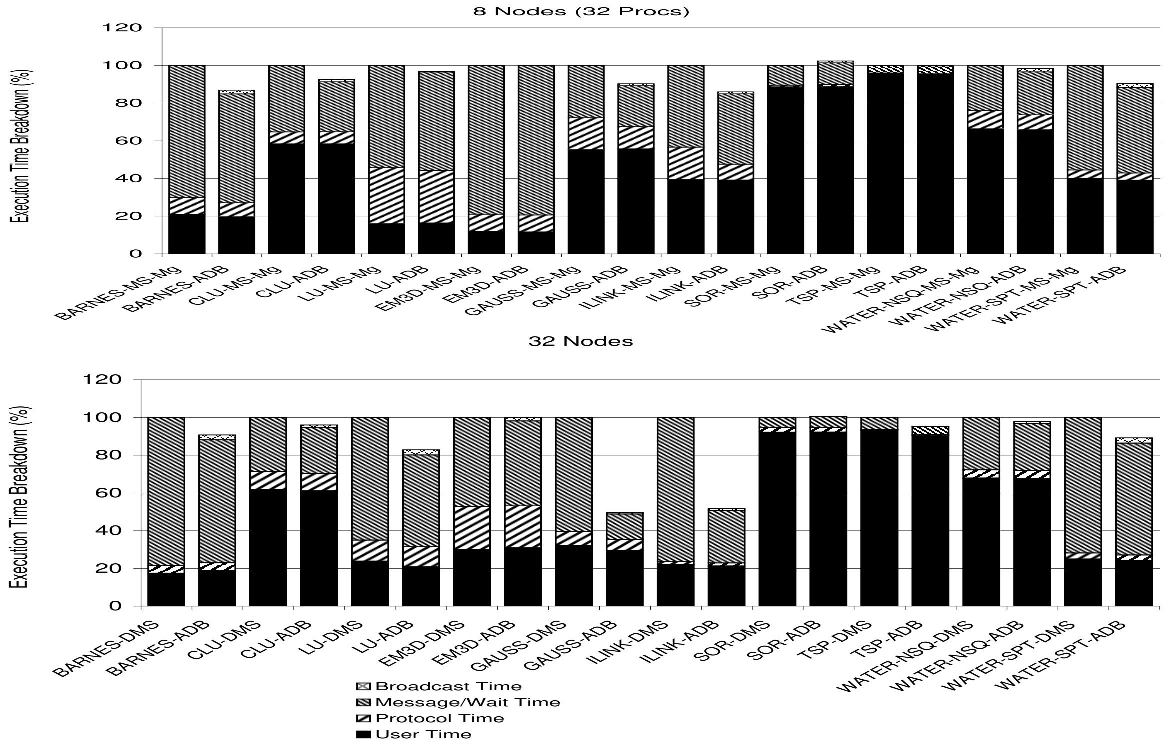

Figure 2. Normalized execution time breakdown for the applications using adaptive broadcast of data (CSM-ADB) in comparison to CSM-MS-Mg at 8 and CSM-DMS at 32 nodes.

tiple consumers: 172K out of a total of 182K page updates are satisfied by the broadcast buffers, while 10K pages are actually placed in the buffers at 32 nodes. In the case of Ilink, 191K out of a total of 205K page updates are satisfied by the broadcast buffers, while only 12K pages are placed in the broadcast buffers at 32 nodes. The number of consumers in LU is not as large: 400K out of a total of 1.19M page updates are satisfied by the broadcast buffers, while 106K pages are placed in the broadcast buffers. All other applications, with the exception of SOR, also benefit from the use of CSM-ADB by smaller amounts.

4. Related Work

Bilas *et al.* [4] use their GeNIMA SDSM to examine the impact of special network features on SDSM performance. Their network has remote write, remote read, and specialized lock support, but no broadcast or total ordering. GeNIMA disseminates write notices through broadcast and so could benefit from the appropriate support. In base Cashmere, the lock implementation uses remote writes, broadcast, and total ordering to obtain the same benefits as GeNIMA's specialized lock support.

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 SDSM performance. However, their base protocol 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 [18]. Our evaluation assumes that messages can be detected through a much more efficient polling mechanism, as is found with other SANs [10, 13], and so each of our protocols benefits from the same low messaging latency. We also extend 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 is remotely accessible, while the other optimization, adaptive data broadcast, relies on a very efficient mapping of remotely accessible memory.

Speight and Bennett [26] evaluate the use of multicast and multithreading in the context of SDSM 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, we rely 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 a current page migration policy found in some CC-NUMA multiprocessors [19, 29]. 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. The page transfer has most likely already occurred on a processor's previous (read) access to the page. Since the migration mechanism is so lightweight, Cashmere can afford to be very aggressive.

Amza *et al.* [3] describe adaptive extensions to the TreadMarks [2] 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. 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) [21] also describes 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.

5. Conclusions

In this paper we have studied the effect of special network features, specifically, remote writes, inexpensive broadcast, and total packet ordering, on the state-of-the-art Cashmere SDSM.

We have found that these network features do indeed provide a performance benefit. Two applications improve by 18% and 23% when all features are exploited. A third improves by 44%, but this improvement leads to a speedup of only 7.6 on 32 processors. The remaining seven applications improve by less than 12%. The network features have little impact on synchronization overhead: the actual cost of a lock, barrier, or flag is typically dwarfed by that of the attendant software coherence protocol operations. The features are somewhat more useful for protocol metadata maintenance. They are primarily useful, however, for data propagation. The direct application of diffs reduces synchronization wait time and the cost of communication due to false sharing, and minimizes the extent to which protocol operations perturb application timing.

On the other hand, we found that home node migration, made possible by moving shared data out of the network address space, is very effective at reducing the number of twin/diff operations and associated protocol overhead. In fact, the benefits of migration sometimes outweigh those of using special network features for shared data propagation. Moreover, by allowing shared data to reside in private memory, we eliminate the need for page pinning and allow the size of shared memory to exceed the addressing limits of the network interface. Our work on out-of-core data sets [12] depends on this more scalable use of the address space.

Overall, our results suggest that for systems of modest size, low latency is much more important for SDSM performance than are remote writes, broadcast, or total ordering. On larger networks, however, we found that an adaptive protocol capable of identifying widely-shared data can potentially make effective use of broadcast with remote writes.

In the future, we would like to examine the impact of other basic network issues on SDSM performance. These issues include DMA versus programmed I/O interfaces, messaging latency, and bandwidth. We are also interested in incorporating predictive migration mechanisms [8, 21, 27] that would identify migratory pages and then trigger migration at the time of an initial read fault.

Acknowledgement: The authors would like to thank Ricardo Bianchini and Alan L. Cox for many helpful discussions concerning this paper.

References

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

[2] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. TreadMarks: Shared Memory Computing on Networks of Workstations. *Computer*, 29(2):18–28, Feb. 1996.

[3] C. Amza, A. Cox, S. Dwarkadas, and W. Zwaenepoel. 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.

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

[5] M. Blumrich, K. Li, R. Alpert, C. Dubnicki, E. Felten, and J. Sandberg. Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer. In *Proc. of the*

21st Intl. Symp. on Computer Architecture, pages 142–153, Chicago, IL, Apr. 1994.

[6] G. Buzzard, D. Jacobson, M. Mackey, S. Marovich, and J. Wilkes. 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.

[7] Y. Chen, A. Bilas, S. N. Damianakis, C. Dubnicki, and K. Li. UTLB: A Mechanism for Address Translation on Network Interfaces. In *Proc. of the 8th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 193–203, San Jose, CA, Oct. 1998.

[8] A. L. Cox and R. J. Fowler. Adaptive Cache Coherency for Detecting Migratory Shared Data. In *Proc. of the 20th Intl. Symp. on Computer Architecture*, San Diego, CA, May 1993.

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

[10] D. Dunning, G. Regnier, G. McAlpine, D. Cameron, B. Shubert, F. Berry, A. M. Merritt, E. Gronke, and C. Dodd. The Virtual Interface Architecture. *IEEE Micro*, 18(2):66–76, Mar. 1998.

[11] S. Dwarkadas, A. A. Schäffer, R. W. Cottingham Jr., A. L. Cox, P. Keleher, and W. Zwaenepoel. Parallelization of General Linkage Analysis Problems. *Human Heredity*, 44:127–141, 1994.

[12] S. Dwarkadas, K. Gharachorloo, L. Kontothanassis, D. J. Scales, M. L. Scott, and R. Stets. Comparative Evaluation of Fine- and Coarse-Grain Approaches for Software Distributed Shared Memory. In *Proc. of the 5th Intl. Symp. on High Performance Computer Architecture*, Orlando, FL, Jan. 1999.

[13] T. v. Eicken, A. Basu, V. Buch, and W. Vogels. 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.

[14] A. Erlichson, N. Nuckolls, G. Chesson, and J. Hennessy. 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*, pages 210–220, Cambridge, MA, Oct. 1996.

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

[16] R. W. Horst and D. Garcia. ServerNet SAN I/O Architecture. In *Proc. of Hot Interconnects V Symposium*, Palo Alto, CA, Aug. 1997.

[17] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy Release Consistency for Software Distributed Shared Memory. In *Proc. of the 19th Intl. Symp. on Computer Architecture*, pages 13–21, Gold Coast, Australia, May 1992.

[18] L. Kontothanassis, G. Hunt, R. Stets, N. Hardavellas, M. Cierniak, S. Parthasarathy, W. Meira, S. Dwarkadas, and M. L. Scott. VM-Based Shared Memory on Low-Latency, Remote-Memory-Access Networks. In *Proc. of the 24th Intl. Symp. on Computer Architecture*, pages 157–169, Denver, CO, June 1997.

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

[20] M. Marchetti, L. Kontothanassis, R. Bianchini, and M. L. Scott. 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.

[21] L. R. Monnerat and R. Bianchini. 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.

[22] R. Samanta, A. Bilas, L. Iftode, and J. P. Singh. Home-based SVM Protocols for SMP clusters: Design and Performance. In *Proc. of the 4th Intl. Symp. on High Performance Computer Architecture*, pages 113–124, Las Vegas, NV, Feb. 1998.

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

[24] D. J. Scales, K. Gharachorloo, and A. Aggarwal. 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.

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

[26] E. Speight and J. K. Bennett. Using Multicast and Multithreading to Reduce Communication in Software DSM Systems. In *Proc. of the 4th Intl. Symp. on High Performance Computer Architecture*, pages 312–322, Las Vegas, NV, Feb. 1998.

[27] P. Stenström, M. Brorsson, and L. Sandberg. An Adaptive Cache Coherence Protocol Optimized for Migratory Sharing. In *Proc. of the 20th Intl. Symp. on Computer Architecture*, San Diego, CA, May 1993.

[28] R. Stets, S. Dwarkadas, N. Hardavellas, G. Hunt, L. Kontothanassis, S. Parthasarathy, and M. Scott. Cashmere-2L: Software Coherent Shared Memory on a Clustered Remote-Write Network. In *Proc. of the 16th ACM Symp. on Operating Systems Principles*, St. Malo, France, Oct. 1997.

[29] B. Verghese, S. Devine, A. Gupta, and M. Rosenblum. 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.

[30] M. Welsh, A. Basu, and T. V. Eicken. Incorporating Memory Management into User-Level Network Interfaces. Technical Report TR97-1620, Cornell University, Aug. 1997.

[31] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. 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.