VM-Based Shared Memory on Low-Latency, Remote-Memory-Access Networks[1]

Leonidas Kontothanassis[2], Galen Hunt, Robert Stets, Nikolaos Hardavellas,
Michał Cierniak, Srinivasan Parthasarathy, Wagner Meira, Jr.,
Sandhya Dwarkadas, and Michael Scott

Department of Computer Science   [2] DEC Cambridge Research Lab
University of Rochester   One Kendall Sq., Bldg. 700
Rochester, NY 14627–0226   Cambridge, MA 02139

Technical Report # 643

`cashmere@cs.rochester.edu`
November 1996

Abstract

Recent technological advances have produced network interfaces that provide users with very low-latency access to the memory of remote machines. We examine the impact of such networks on the implementation and performance of software DSM. Specifically, we compare two DSM systems—Cashmere and TreadMarks—on a 32-processor DEC Alpha cluster connected by a Memory Channel network.

Both Cashmere and TreadMarks use virtual memory to maintain coherence on pages, and both use lazy, multi-writer release consistency. The systems differ dramatically, however, in the mechanisms used to track sharing information and to collect and merge concurrent updates to a page, with the result that Cashmere communicates much more frequently, and at a much finer grain.

Our principal conclusion is that low-latency networks make DSM based on fine-grain communication competitive with more coarse-grain approaches, but that further hardware improvements will be needed before such systems can provide consistently superior performance. In our experiments, Cashmere scales slightly better than TreadMarks for applications with fine-grain interleaving of accesses to shared data. At the same time, it is severely constrained by limitations of the current Memory Channel hardware. On average, performance is better for Tread-Marks. Because most of its messages require remote processing, TreadMarks depends critically on the latency of interrupts: performance improves greatly when processors poll for messages.

Keywords: Distributed Shared Memory, Memory-Mapped Network Interface, Cashmere, TreadMarks.

1 Introduction

Distributed shared memory (DSM) is an attractive design alternative for large-scale shared memory multiprocessing. Traditional DSM systems rely on virtual memory hardware and simple message passing to implement shared memory. State-of-the-art DSM systems (e.g. TreadMarks [1, 19]) employ sophisticated protocol optimizations, such as relaxed consistency models, multiple writable copies of a page, and lazy processing of all coherence-related events. These optimizations recognize the very high (millisecond) latency of communication on workstation networks; their aim is to minimize the frequency of communication, even at the expense of additional computation.

Recent technological advances, however, have led to the commercial availability of inexpensive workstation networks on which a processor can access the memory of a remote node safely from user space, at a latency two to three orders of magnitude lower than that of tra-ditional message passing. These networks suggest the need to re-evaluate the assumptions underlying the design of DSM protocols, and specifically to consider protocols that communicate at a much finer grain. The Cashmere system employs this sort of protocol. It uses directories to keep track of sharing information, and merges concurrent writes to the same coherence block via write-through to a unique (and possibly remote) main-memory copy of each page. Simulation studies indicate that on an "ideal" remote-write network Cashmere will significantly outperform other DSM approaches, and will in fact approach the performance of full hardware cache coherence [20, 21].

In this paper we compare implementations of Cashmere and TreadMarks on a 32-processor cluster (8 nodes, 4 processors each) of DEC AlphaServers, connected by DEC's Memory Channel [15] network. Memory Channel allows a user-level application to write to the memory of remote nodes. The remote-write capability can be used for (non-coherent) shared memory, for broadcast/multicast, and for very fast user-level messages. Remote reads are not directly supported. Where the original Cashmere protocol used remote reads to access directory information, we broadcast directory updates on the Memory Channel. Where the original Cashmere protocol would read the contents of a page from the home node, we ask a processor at the home node to write the data to us. In TreadMarks, we use the Memory Channel simply to provide a very fast messaging system.

Our performance results compare six specific protocol implementations: three each for TreadMarks and Cashmere. For both systems, one implementation uses interrupts to request information from remote processors, while another requires processors to poll for remote requests at the top of every loop. The third TreadMarks implementation uses DEC's standard, kernel-level implementation of UDP. The third Cashmere implementation dedicates one processor per node to handling remote requests. This approach is similar to polling, but without the additional overhead and the unpredictability in response time: it is meant to emulate a hypothetical Memory Channel in which remote reads are supported in hardware. The emulation is conservative in the sense that it moves data across the local bus twice (through the processor registers), while true remote reads would cross the bus only once.

In general, both Cashmere and TreadMarks provide good performance for many of the applications in our test suite. Low interrupt overhead is crucial to the performance and scalability of TreadMarks, resulting in greatly improved results with polling. The difference between interrupts and polling has a smaller effect on performance in Cashmere: fewer interrupts are required than in Tread-Marks, since Cashmere takes advantage of remote memory access for program and meta-data resulting in fewer messages requiring a reply. The Cashmere protocol scales

better than TreadMarks for applications with relatively fine-grain synchronization, or with false sharing within a page. However, for applications with low communication requirements, performance is hurt due to the cost of "doubling" writes to the home node as well as the need to check directory entries for data that is not actively shared.

Three principal factors appear to contribute to making the differences between TreadMarks and Cashmere on the Memory Channel smaller than one would expect on an "ideal" remote-memory-access network. First, the current Memory Channel has relatively modest cross-sectional bandwidth, which limits the performance of write-through. Second, it lacks remote reads, forcing Cashmere to copy pages to local memory (rather than fetching them incrementally in response to cache misses), and to engage the active assistance of a remote processor in order to make the copy. With equal numbers of compute processors, Cashmere usually performs best when an additional processor per node is dedicated to servicing remote requests, implying that remote-read hardware would improve performance further. Third, our processors (the 21064A) have very small first-level caches. Our write-doubling mechanism increases the first-level working set for certain applications beyond the 16K available, dramatically reducing performance. The larger caches of the 21264 should largely eliminate this problem.

We are optimistic about the future of Cashmere-like systems as network interfaces continue to evolve. Based on previous simulations [21], it is in fact somewhat surprising that Cashmere performs as well as it does on the current generation of hardware. The second-generation Memory Channel, due on the market very soon, will have something like half the latency, and an order of magnitude more bandwidth. Finer-grain DSM systems are in a position to make excellent use of this sort of hardware as it becomes available.

The remainder of this paper is organized as follows. We present the coherence protocols we are evaluating in section 2. In section 3 we discuss our implementations of the protocols, together with the mechanisms we employed (write doubling, page copying, directory broadcast, and remote read emulation) to overcome limitations of the hardware. Section 4 presents experimental results. The final two sections discuss related work and summarize our conclusions.

2 Protocols

In this section, we discuss the TreadMarks and Cashmere coherence protocols. Both systems maintain coherence at page granularity, employ a relaxed consistency model, and allow multiple concurrent writers to the same coherence block. There are significant differences, however, in the way sharing information for coherence blocks is maintained, and in the way writes to the same coherence block by multiple processors are merged.

2.1 Cashmere

Cashmere maintains coherence information using a distributed directory data structure. For each shared page in the system, a single directory entry indicates the page state. For a 32-processor system, the entry fits in a 64-bit longword that encodes the page state (2 bits), the id of the *home node* for the page (5 bits), whether the home node is still the original default or has been set as the result of a "first touch" heuristic (1 bit), the number of processors writing the page (5 bits), the id of the last processor to take a write fault on the page (5 bits), a bitmask of the processors that have a copy of the page (32 bits), the number of processors that have a copy of the page (5 bits), a lock bit used for obtaining exclusive access to a directory entry, and an initialization bit indicating whether the directory entry has been initialized or not. Directory space overhead for the 8K pages supported by DEC Unix is only about 1%, and would be smaller if we did not have to replicate the directory on each of our eight nodes.

A page can be in one of three states: **Uncached** – No processor has a mapping to the page. This is the initial state for all pages. **Shared** – One or more processors have mappings to the page, but none of them has made changes that need to be globally visible. **Weak** – Two or more processors have mappings to the page and at least one has made changes that need to be globally visible, because of a subsequent release. Notice that we have omitted the **dirty** state present in some previous Cashmere designs. We have chosen to leave a page in the shared state when there is a single processor reading and writing it; we check at release synchronization points to see if other processors have joined the sharing set and need to be told of changes.

In addition to the directory data structure, each processor also holds a globally accessible *weak list* that indicates which of the pages with local mappings are currently in the weak state. Protocol operations happen in response to four types of events: **read** page faults, **write** page faults, **acquire** synchronization operations, and **release** synchronization operations.

When a processor takes a read page fault it locks the directory entry representing the page, adds itself to the sharing set, increments the number of processors that have a copy of the page, and changes the page state to shared if it was uncached. It also inserts the page number in the local *weak list* if the page was found in the weak state. The directory operation completes by unlocking the entry. Locally, the processor creates a mapping for the page and re-starts the faulting operation. Ideally, the mapping would allow the processor to load cache lines

from the home node on demand. On the Memory Channel, which does not support remote reads, we must copy the page to local memory (see below). A write page fault is treated the same as a read page fault, except that the processor also inserts the page number in a local list called the *write list*.

At an acquire synchronization operation the processor traverses its weak list and removes itself from the sharing set of each page found therein. The removal operation requires that the processor obtain the lock for the corresponding directory entry, modify the bitmask of sharing processors, decrement the count of processors that have a copy of the page, and return the page to the uncached state if this was the last processor with a mapping.

At a release synchronization operation the processor traverses the write list and informs other processors of the writes it has performed since the previous release operation. For each entry in the write list the processor acquires the lock for the corresponding directory entry. If other processors are found to be sharing the page, the releaser changes the page state to weak and appends a notice to the weak list of every sharing processor. No state change or notices are needed if the page is already in the weak state. We also apply an optimization for single-writer sharing. If a processor is the only writer of a page it never needs to invalidate itself, since it has the most up-to-date version of the data, and thus it refrains from placing a notice in its own weak list. Furthermore it does not move the page to the weak state since that would prevent other (future) writers from sending write notices that the first writer needs to see. The count of processors writing a page and the id of the last processor to write the page, found in the directory entry, allow us to identify this particular sharing pattern.

Ideally, a releasing processor would inspect the dirty bits of TLB/page table entries to determine which pages have been written since the previous release. Pages could then be mapped writable at read page faults, avoiding the need to take an additional fault (to place the page in the write list) if a write came after the read. Similarly, a single writer would clear its dirty bit and then leave its copy of the page writable at a release operation. Unfortunately, we do not currently have access to the dirty bits on our AlphaServers. We therefore map a page read-only on a read page fault, and wait for a subsequent write fault (if any) to add the page to the write list. At a release, a single writer leaves the page writable (to avoid the overhead of repeated faults), but must then leave the page in its write list and recheck for readers at subsequent synchronization points. As a result, a processor that writes a page once and then keeps it for a very long time may keep the page in the weak state all that time, forcing readers of the page to perform unnecessary invalidations at every subsequent acquire. Similarly, a logically shared page that is in fact used only by one processor (e.g. data internal to a band

in SOR) will remain in the shared state forever, forcing the processor that uses the page to inspect the directory at every release, just to make sure no readers have arrived and necessitated a transition to the weak state. We expect to overcome these limitations shortly by using dirty bits.

The final issue to be addressed is the mechanism that allows a processor to obtain the data written by other processors. For each page there is a unique home node to which processors send changes on the fly. Because acquires and releases serialize on access to a page's directory entry, a processor that needs data from a page is guaranteed that all writes in its logical past are reflected in the copy at the home node. On a network with remote reads there would be only one copy of each page in main memory—namely the copy at the home node. Every page mapping would refer to this page; cache fills would be satisfied from this page; and the collection of changes would happen via the standard cache write-through or write-back. On the Memory Channel, we must create a local copy of a page in response to a page fault. Normal write-back then updates this local copy. To update the copy at the home node, we insert additional code into the program executable at every shared memory write.

The choice of home node for a page can have a significant impact on performance. The home node itself can access the page directly, while the remaining processors have to use the slower Memory Channel interface. We assign home nodes at run-time, based on which processor first touches a page after the program has completed any initialization phase [27].

More detailed information on the Cashmere protocol (and its network-interface-specific variants) can be found in other papers [21, 20].

2.2 TreadMarks

TreadMarks is a distributed shared memory system based on lazy release consistency (LRC) [19]. Lazy release consistency is a variant of release consistency [24]. It guarantees memory consistency only at synchronization points and permits multiple writers per coherence block. Lazy release consistency divides time on each node into intervals delineated by remote synchronization operations. Each interval is represented by a vector of timestamps, with entry i on processor j representing the most recent interval on processor i that logically precedes the current interval on processor j. The protocol associates writes to shared pages with the interval in which they occur. When a processor takes a write page fault, it creates a write notice for the faulting page and appends the notice to a list of notices associated with its current interval.

When a processor acquires a lock, it sends a copy of its current vector timestamp to the previous lock owner. The previous lock owner compares the received timestamp

with its own, and responds with a list of all intervals (and the write notices associated with them) that are in the new owner's past, but that the new owner has not seen. The acquiring processor sets its vector timestamp to be the pairwise maximum of its old vector and the vector of the previous lock owner. It also incorporates in its local data structures all intervals (and the associated write notices) sent by the previous owner. Finally, it invalidates (unmaps) all pages for which it received a write notice. The write notice signifies that there is a write to the page in the processor's logical past and that the processor needs to bring its copy of the page up to date.

To support multiple writers to a page, each processor saves a pristine copy of a page called a *twin* before it writes the page. When asked for changes, the processor compares its current copy of the page to the page's twin. The result of the comparison is a list of modified addresses with the new contents, called a *diff*. There is one diff for every write notice in the system. When a processor takes a read page fault (or a write page fault on a completely unmapped page), it peruses its list of write notices and makes requests for all unseen modifications. It then merges the changes into its local copy in software, in the causal order defined by the timestamps of the write notices.

Barrier synchronization is dealt with somewhat differently. Upon arrival at a barrier, all processors send their vector timestamps (and intervals and write notices), to a barrier manager, using a conservative guess as to the contents of the manager's vector timestamp. The manager merges all timestamps, intervals, and write notices into its local data structures, and then sends to each processor its new updated timestamp along with the intervals and write notices that the processor has not seen.

The TreadMarks protocol avoids communication at the time of a release, and limits communication to the processes participating in a synchronization operation. However, because it must guarantee the correctness of arbitrary future references, the TreadMarks protocol must send notices of all logically previous writes to synchronizing processors even if the processors have no copy of the page to which the write notice refers. If a processor is not going to acquire a copy of the page in the future (something the protocol cannot of course predict), then sending and processing these notices may constitute a significant amount of unnecessary work, especially during barrier synchronization, when all processors need to be made aware of all other processors' writes. Further information on Tread-Marks can be found in other papers [1].

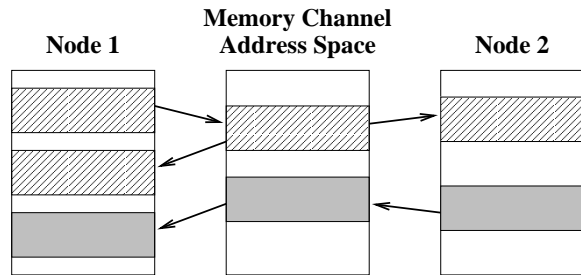# 3 Implementation Issues

3.1 Memory Channel

Figure 1: Memory Channel space. The lined region is mapped for both transmit and receive on node 1 and for receive on node 2. The gray region is mapped for receive on node 1 and for transmit on node 2.

Digital Equipment's Memory Channel (MC) network provides applications with a global address space using memory mapped regions. A region can be mapped into a process's address space for transmit, receive, or both. Virtual addresses for transmit regions map into physical addresses located in I/O space, and, in particular, on the MC's PCI adapter. Virtual addresses for receive regions map into physical RAM. Writes into transmit regions are collected by the source MC adapter, forwarded to destination MC adapters through a hub, and transferred via DMA to receive regions with the same global identifier (see Figure 1). Regions within a node can be shared across processors and processes. Writes originating on a given node will be sent to receive regions on that same node only if *loop-back* has been enabled for the region. In our implementation of Cashmere, we use loop-back only for synchronization primitives. TreadMarks does not use it at all.

Unicast and multicast process-to-process writes have a latency of 5.2 μs on our system (latency drops below 5μs for other AlphaServer models). Our MC configuration can sustain per-link transfer bandwidths of 30 MB/s with the limiting factor being the 32-bit AlphaServer 2100 PCI bus. MC peak aggregate bandwidth is also about 32 MB/s.

Memory Channel guarantees write ordering and local cache coherence. Two writes issued to the same transmit region (even on different nodes) will appear in the same order in every receive region. When a write appears in a receive region it invalidates any locally cached copies of its line.

3.2 Remote-Read Mechanisms

Although the first-generation Memory Channel supports remote writes, it does not support remote reads. To read remote data, a message passing protocol must be used to send a request to a remote node. The remote node responds by writing the requested data into a region that is mapped for receive on the originating node.

4

```
label:
    ldq    $7, 0($13)          ; Check poll flag.
    beq    $7, nomsg           ; If message,
    jsr    $26, handler        ;   call handler.
    ldgp   $29, 0($26)
nomsg:
```

Figure 2: Polling. Polling code is inserted at all interior, backward-referenced labels. The address of the polling flag is preserved in register $13 throughout execution.

Three different mechanisms can be used to process transfer requests. The most obvious alternative uses a MC provided inter-node interrupt to force a processor on the remote node into a handler. The second alternative is for processes to poll for messages on a periodic basis (e.g. at the tops of loops). The third alternative is to dedicate one processor on every node to servicing remote requests, and have that processor poll on a continual basis.

Memory Channel allows a processor to trigger an inter-node interrupt by means of a remote write for which the recipient has created a special receive region mapping. This capability is exported to applications as an imc_kill function which is called with a remote host name, process identifier and UNIX signal number. Because the interrupt must be filtered up through the kernel to the receiving process, inter-node signals have a cost of almost 1 millisecond.

Polling requires instrumentation that checks the message receive region frequently, and branches to a handler if a message has arrived. Applications can be instrumented either by hand or automatically. We instrument the protocol libraries by hand and use an extra compilation pass between the compiler and assembler to instrument applications. The instrumentation pass parses the compiler-generated assembly file and inserts polling instrumentation at the start of all labeled basic blocks that are internal to a function and are backward referenced—i.e. at tops of all loops. The polling instruction sequence appears in Figure 2.

Dedicating a processor to polling on each node is the least intrusive mechanism: it requires neither application changes nor expensive interrupt handlers. Of course, a dedicated processor is unavailable for regular computation. In general, we would not expect this to be a productive way to use an Alpha processor (though in a few cases it actually leads to the best program run-times). Our intent is rather to simulate the behavior of a hypothetical (and somewhat smaller) Memory Channel system that supports remote reads in hardware.
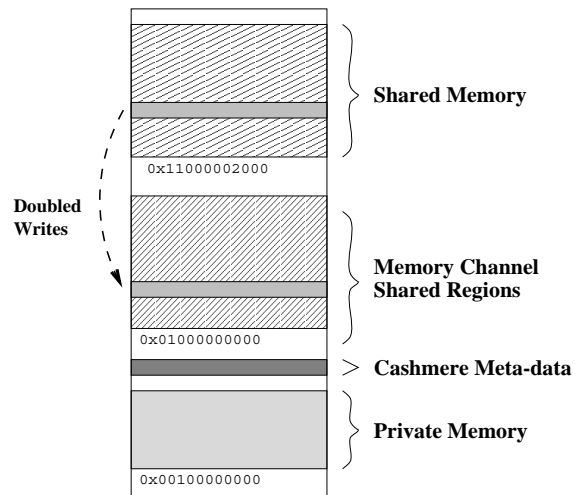
3.3   Cashmere

Figure 3: Cashmere process address space. Writes are duplicated from the private copy of a shared memory page to the corresponding page in the MC regions. MC regions are mapped for receive on home nodes and for transmit on all other nodes.

Cashmere takes advantage of MC remote writes to collect shared memory writes and to update the page directory. The Cashmere virtual address space consists of four areas (see Figure 3). The first area consists of process private memory. The second contains Cashmere meta-data, including the page directory, synchronization memory, and protocol message-passing regions (for page fetch requests). The other two areas contain Cashmere shared memory. The higher of these two contains the processor's local copies of shared pages and the lower contains MC regions used to maintain memory consistency between nodes.

The page directory and synchronization regions are mapped twice on each node: once for receive and once for transmit. Remote meta-data updates are implemented by writing once to the receive region in local memory and again to the write region for transmit. MC loop-back is not used because it requires twice the PCI bandwidth: once for transmit and once for receive, and because it does not guarantee processor consistency.

For every actively-shared page of memory, one page is allocated in the process's local copy area and another in the MC area. If a process is located on the home node for a shared page, the MC page is mapped for receive. If a process is not located on the home node, then the MC page is mapped for transmit. In the current version of Digital Unix, the number of separate Memory Channel regions is limited by fixed-size kernel tables. As a temporary expedient, each MC region in our Cashmere implementation contains 32 contiguous pages, called a superpage. Superpages have no effect on coherence granularity: we still

```
srl    $3, 40, $7    ; Create page offset
sll    $7, 13, $7    ;   if shared write.
subq   $3, $7, $7
zap    $7, 0x20, $7  ; Subtract MC offset.
stq    $1, 0($3)     ; Original write.
stq    $1, 0($7)     ; Doubled write.
```

Figure 4: Write Doubling, in this case for an address in register 3 with 0 displacement.

map and unmap individual pages. They do, however, constrain our "first touch" page placement policy: all 32 pages of a given superpage must share the same home node. We plan to modify the kernel to allow every page to lie in a separate region.

3.3.1 Write Doubling

All writes to shared memory consist of an original write to the private copy and a duplicate write to the MC area. The duplicate write and associated address arithmetic are inserted into applications by parsing and updating assembly code, in a manner analagous to that used to insert polling. As in polling, we rely on the GNU C compiler's ability to reserve a register for use in the instrumented code. Single-processor tests confirm that the additional register pressure caused by the reservation has a minimal impact on the performance of our applications. A compiler-supported Cashmere implementation would be able to use one of the registers ($7) for other purposes as well.

Figure 4 contains the assembly sequence for a doubled write. The sequence relies on careful alignment of the Cashmere shared-memory regions. The local copy of a shared page and its corresponding Memory Channel region differ in address by 0x0100 0000 2000. The high bit in the offset places the addresses far enough apart that all of shared memory can be contiguous. The low bit in the offset increases the odds that the two addresses will map to different locations in the first-level cache on the home node. To obtain the address to which to double a write, we use the value of the 40th bit to mask out the 13th bit. We also clear the 40th bit. For a truly private reference, which should not be doubled, these operations have no effect, because the 40th bit is already 0. In this case, the private address is written twice. By placing the address arithmetic before the original write, so that the writes are consecutive instructions, we make it likely that spurious doubles of private writes will combine in the processor's write buffer. As an obvious optimization, we refrain from doubling writes that use the stack pointer or global pointer as a base.

3.3.2 Page Directory and Locks

Page directory access time is crucial to the overall performance of Cashmere. Directory entries must be globally consistent and inexpensive to access. As mentioned previously, the page directory is mapped into both receive and transmit regions on each node. Each entry consists of a single 64-bit value and a lock.

Both application and protocol locks are represented by an 8-entry array in Memory Channel space, and by a test-and-set flag on each node. To acquire a lock, a process first acquires the per-node flag using load-linked/store-conditional. It then sets the array entry for its node, waits for the write to appear via loop-back, and reads the whole array. If its entry is the only one set, then the process has acquired the lock. Otherwise it clears its entry, backs off, and tries again. In the absence of contention, acquiring and releasing a lock takes about 11 μs. Digital Unix provides a system-call interface for Memory Channel locks, but while its internal implementation is essentially the same as ours, its latency is more than 280 μs.

3.4 TreadMarks

We have modified TreadMarks version 0.10.1 to use the MC architecture for fast user-level messages. Only the messaging layer was changed: all other aspects of the implementation are standard. In particular, we do not use broadcast or remote memory access for either synchronization or protocol data structures, nor do we place shared memory in Memory Channel space.

We present results for three versions of TreadMarks. The first uses DEC's kernel-level implementation of UDP for the Memory Channel, with regular `sig_io` interrupts. The second uses user-level message buffers, and sends `imc_kill` interrupts (see Section 3.2) to signal message arrival. The third is built entirely in user space, with polling (see Section 3.2) to determine message arrival. Interrupts (and in particular signals) for the first two implementations are expensive in Digital Unix, and are a principal factor limiting scalability. While polling in our implementation makes use of the Memory Channel hardware, it could also be implemented (at somewhat higher cost) on a more conventional network. In our experiments it allows us to separate the intrinsic behavior of the TreadMarks protocol from the very high cost of signals on this particular system.

The UDP version of TreadMarks creates a pair of UDP sockets between each pair of participating processors: a request socket on which processor A sends requests to processor B and receives replies, and a reply socket on which processor A receives requests from processor B and sends replies. The MC version replaces the sockets with a pair of message buffers. Two sense-reversing flags (variables

allocated in Memory Channel space) provide flow control between the sender and the receiver: the sender uses one flag to indicate when data is ready to be consumed; the receiver uses the other to indicate when data has been consumed. Reply messages do not require interrupts or polling: the requesting processor always spins while waiting for a reply. Because both DEC's MC UDP and our user-level buffers provide reliable delivery, we have disabled the usual timeout mechanism used to detect lost messages in TreadMarks. To avoid deadlock due to buffer flow-control, we have made the request handler re-entrant: whenever it spins while waiting for a free buffer or for an expected reply it also polls for, and queues, additional incoming requests.

In both TreadMarks and Cashmere each application-level process is bound to a separate processor of the AlphaServer. The binding improves performance by taking advantage of memory locality, and reduces the time to deliver a page fault. To minimize latency and demand for Memory Channel bandwidth, we allocate message buffers in ordinary shared memory, rather than Memory Channel space, whenever the communicating processes are on the same AlphaServer. This is the only place in either TreadMarks or Cashmere that our current implementations take advantage of the hardware coherence available within each node.

4   Performance Evaluation

Our experimental environment consists of eight DEC AlphaServer 2100 4/233 computers. Each AlphaServer is equipped with four 21064A processors operating at 233 MHz and with 256MB of shared memory, as well as a Memory Channel network interface.

For Cashmere, we present results for three different mechanisms to handle remote requests—a dedicated "protocol processor" (csm_pp), interrupts (csm_int), and polling (csm_poll). The protocol processor option approximates the ability to read remote memory in hardware. For TreadMarks, we present results for three different versions—one that uses DEC's kernel-level MC UDP protocol stack with interrupts (tmk_udp_int), one that uses user-level messaging on MC with interrupts (tmk_mc_int), and one that uses user-level messaging on MC with polling (tmk_mc_poll). Both TreadMarks and Cashmere treat every processor as a separate protocol node; neither takes advantage of hardware shared memory for user data sharing within a node.

4.1   Basic Operation Costs

Memory protection operations on the AlphaServers cost about 62 μs. Page faults cost 89 μs. It takes 69 μs to deliver a signal locally, while remote delivery costs the sender 584 μs and incurs an end-to-end latency of about 1 ms. The overhead for polling ranges between 0% and 15% compared to a single processor execution, depending on the application.

The overhead for write doubling ranges between 0% and 39% compared to a single processor execution for Cashmere, depending on the application. Directory entry modification takes 17 μs for Cashmere. Most of that time, 11 μs, is spent acquiring and releasing the directory entry lock. The cost of a twinning operation on an 8K page in TreadMarks is 362 μs. The cost of diff creation ranges from 289 to 534 μs per page, depending on the size of the diff.

Table 1 provides a summary of the minimum cost of page transfers and of user-level synchronization operations for the different implementations of Cashmere and TreadMarks. All times are for interactions between two processors. The barrier times in parentheses are for a 16 processor barrier.

4.2   Application Characteristics

We present results for 8 applications:

SOR: a Red-Black Successive Over-Relaxation program for solving partial differential equations. In our parallel version, the program divides the red and black arrays into roughly equal size bands of rows, assigning each band to a different processor. Communication occurs across the boundaries between bands. Processors synchronize with barriers.

LU: a kernel from the SPLASH-2 [38] benchmark, which for a given matrix A finds its factorization $A = LU$, where L is a lower-triangular matrix and U is upper triangular. This program promotes temporal and spatial locality by dividing the matrix A into square blocks. Each block is "owned" by a particular processor, which performs all computation on it. The "first touch" location policy in Cashmere ensures that a processor serves as home node for the blocks it owns.

Water: a molecular dynamics simulation from the SPLASH-1 [35] benchmark suite. The main shared data structure is a one-dimensional array of records, each of which represents a water molecule. The parallel algorithm statically divides the array into equal contiguous chunks, assigning each chunk to a different processor. The bulk of the interprocessor communication happens during a computation phase that computes intermolecular forces. Each processor updates the forces between each of its molecules and each of the $n/2$ molecules that follow it in the array in wrap-around fashion. The processor accumulates its

Operation	Time (μs)					
	csm_pp	csm_int	csm_poll	tmk_udp_int	tmk_mc_int	tmk_mc_poll
Lock Acquire	11	11	11	1362	976	79
Barrier	90 (173)	88 (208)	86 (205)	987 (8006)	568 (5432)	75 (1213)
Page Transfer	736	1960	742	2932	1962	784

Table 1: Cost of Basic Operations

forces locally and then acquires per-processor locks to update the globally shared force vectors.

TSP: a branch-and-bound solution to the traveling salesman problem. The major data structures are a pool of partially evaluated tours, a priority queue containing pointers to tours in the pool, a stack of pointers to unused tour elements in the pool, and the current shortest path. All the major data structures are shared. Locks are used to insert and delete unsolved tours in the priority queue. Updates to the shortest path are also protected by a separate lock. The algorithm is non-deterministic in the sense that the earlier some processor stumbles upon the shortest path, the more quickly other parts of the search space can be pruned.

Gauss: Gauss solves a system of linear equations $AX = B$ through Gaussian Elimination and back-substitution. The Gaussian elimination phase makes A upper triangular. Each row of the matrix is the responsibility of a single processor. For load balance, the rows are distributed among processors cyclically. A synchronization flag for each row indicates when it is available to other rows for use as a pivot. The algorithm performs a column operation only if the pivot for a given row is zero. An additional vector, SWAP, stores the column swaps in the order that they occur. A barrier separates the above phase from the back-substitution step, which serves to diagonalize A. The division of work and synchronization pattern is similar to the first step. If there were any column operations performed in the first step these operations (which are stored in the vector SWAP) have to be carried out in reverse order on the vector X to calculate the final result.

ILINK: ILINK [9, 23] is a widely used genetic linkage analysis program that locates disease genes on chromosomes. The input consists of several family trees. The main data structure is a pool of sparse *genarray*s. A genarray contains an entry for the probability of each genotype for an individual. As the program traverses the family trees and visits each nuclear family, the pool of genarrays is reinitialized for each person in the nuclear family. The computation either updates a parent's genarray conditioned on the spouse and all children, or updates one child conditioned on both parents and all the other siblings. We use the parallel algorithm described by Dwarkadas et al. [12]. Updates to each individual's genarray are parallelized. A master processor assigns individual genarray elements to processors in a round robin fashion in order to improve load balance. After each processor has updated its elements, the master processor sums the contributions. The bank of genarrays is shared among the processors, and barriers are used for synchronization. Scalability is limited by an inherent serial component and inherent load imbalance.

Barnes: an N-body simulation from the SPLASH-1 [35] suite, using the hierarchical Barnes-Hut Method. Each leaf of the program's tree represents a body, and each internal node a "cell": a collection of bodies in close physical proximity. The major shared data structures are two arrays, one representing the bodies and the other representing the cells. The program has four major phases in each time step, to (1) construct the Barnes-Hut tree, (2) partition the bodies among the processors, (3) compute the forces on a processor's bodies, and (4) update the positions and the velocities of the bodies. Phase (1) executes on only the master process. Parallel computation in the other phases is dynamically balanced using the cost-zone method, with most of the computation time spent in phase (3). Synchronization consists of barriers between phases.

Em3d: a program to simulate electromagnetic wave propagation through 3D objects [11]. The major data structure is an array that contains the set of magnetic and electric nodes. These are equally distributed among the processors in the system. Dependencies between nodes are static; most of them are among nodes that belong to the same processor. For each phase in the computation, each processor updates the electromagnetic potential of its nodes based on the potential of neighboring nodes. Only magnetic nodes are taken into account when updating the electric nodes, and only electric nodes are looked at when updating the magnetic nodes. While arbitrary graphs of dependencies between nodes can be constructed, the standard input assumes that nodes that belong to

Program	Problem Size	Time (sec.)
SOR	3072x4096 (50Mbytes)	194.96
LU	2046x2046 (33Mbytes)	254.77
Water	4096 mols. (4Mbytes)	1847.56
TSP	17 cities (1Mbyte)	4028.95
Gauss	2046x2046 (33Mbytes)	953.71
ILINK	CLP (15Mbytes)	898.97
Em3d	60106 nodes (49Mbytes)	161.43
Barnes	128K bodies (26Mbytes)	469.43

Table 2: Data Set Sizes and Sequential Execution Time of Applications

a processor have dependencies only on nodes that belong to that processor or neighboring processors. Processors use barriers to synchronize between computational phases.

Table 2 presents the data set sizes and uniprocessor execution times for each of the eight applications, with the size of shared memory space used in parentheses. The execution times were measured by running each application sequentially without linking it to either TreadMarks or Cashmere.

4.3 Comparative Speedups

Figure 5 presents speedups for our applications on up to 32 processors. All calculations are with respect to the sequential times in Table 2. The configurations we use are as follows: 1 processor: trivial; 2: separate nodes; 4: one processor in each of 4 nodes; 8: two processors in each of 4 nodes; 12: three processors in each of 4 nodes; 16: for csm_pp, two processors in each of 8 nodes, otherwise 4 processors in each of 4 nodes; 24: three processors in each of 8 nodes; and 32: trivial, but not applicable to csm_pp.

Table 3 presents detailed statistics on communication and delays incurred by each of the applications on the various protocol implementations, all at 16 processors. The interrupt numbers in parentheses represent the number of interrupts that actually result in signal handler dispatch. Additional events are detected via polling at obvious places within the protocol.

TSP displays nearly linear speedup for all our protocols. Speedups are also reasonable in SOR and Water. Em3d, LU, Ilink, Barnes, and Gauss have lower overall speedup. Performance of Cashmere and TreadMarks is similar for TSP, Water, and Em3d. Cashmere outperforms TreadMarks on Barnes. TreadMarks outperforms Cashmere by significant amounts on LU and smaller amounts on SOR, Gauss, and Ilink.

For Barnes, the differences in performance stem primarily from Cashmere's ability to merge updates from mul-

tiple writers into a single home node: TreadMarks must generally merge diffs from multiple sources to update a page in Barnes (note the high message count in Table 3). The overhead of merging diffs is also a factor in Ilink. It is balanced in part by the overhead of write doubling in Cashmere. In the presence of false sharing (an issue in Barnes and Water) it is also balanced by extra invalidations and page faults, which TreadMarks avoids through more careful tracking of the happens-before relationship. Finally, in applications such as Ilink, which modify only a small portion of a page between synchronization operations, the diffs of TreadMarks can be smaller than the page reads of Memory-Channel Cashmere.

In Water, TSP, and Em3d, the various competing effects more or less cancel out. It is difficult to draw conclusions from TSP, because of its non-determinism.

Write doubling to internal rows of an SOR band is entirely wasted in Cashmere, since no other processor ever inspects those elements. The impact is reduced to some extent by the "first write" placement policy, which ensures that the doubled writes are local.

Cashmere suffers in SOR, LU, and Gauss from the use of a single-writer optimization due to the lack of access to dirty bits in our current implementation. The single-writer optimization avoids the use of a dirty state and reduces page faults at the expense of checking directory entries for sharing at each release. In SOR, a processor must inspect directory entries for interior rows on every release to make sure there are no readers that need to receive write notices. In LU and Gauss, the single-writer optimization causes a writing processor to keep a copy of its blocks or rows even when they are no longer being written. At each release, the writer sends write notices to all the reading processors, causing them to perform invalidations. If a reader needs a page again, the fault it takes will be wasted, because the data has not really been re-written. If all readers are done with the page, the writer still wastes time inspecting directory entries. With access to dirty bits, a non-re-written page would leave the writer's write list, and would not be inspected again.

The most dramatic differences between Cashmere and TreadMarks occur in LU, and can be traced to cache effects. Specifically, the increase in cache pressure caused by write doubling forces the working set of this applications out of the first-level cache of the 21064A. With a 32X32 block the primary working set of this application is 16 Kbytes which fits entirely in the first level cache of the 21064A. Write doubling increases the working set to 24K for Cashmere forcing the application to work out of the second level cache and thus significantly hurting performance. Gauss exhibits similar behavior. In Gauss the primary working set decreases over time as less rows remain to be eliminated. For our problem size the working set starts by not fitting in the first level cache for neither
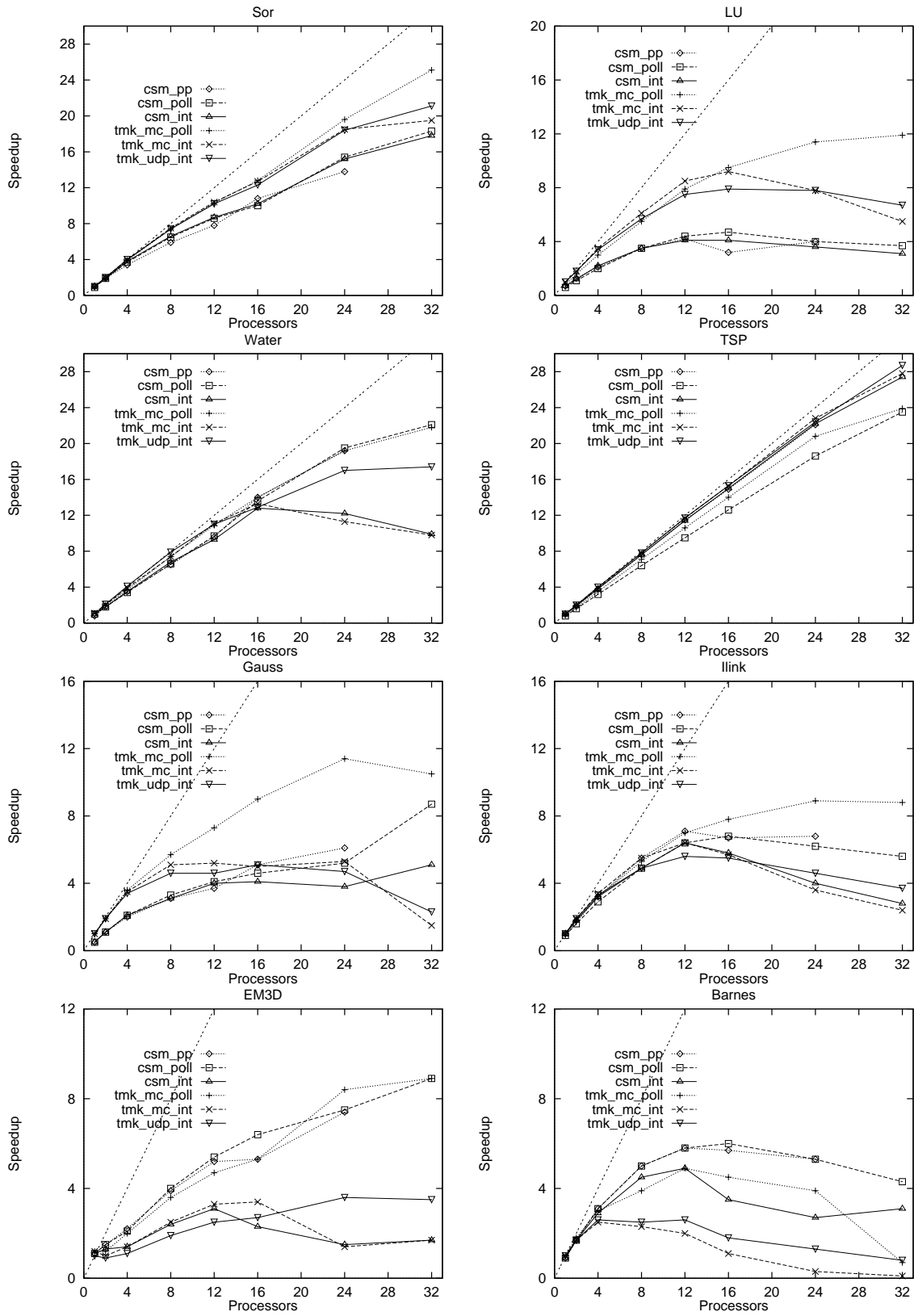
Figure 5: Speedups

10

	Application	Sor	LU	Water	TSP	Gauss	Ilink	Em3d	Barnes
CSM	Barriers	832	2128	592	48	128	8352	4016	336
	Locks	0	0	1200	2560	0	0	0	0
	Read faults	4610	18426	47131	21907	115284	119713	188426	258053
	Write faults	6140	4095	19460	8739	8181	23304	4440	191769
	Interrupts	2429	7361	28156	18992	56261	47843	50853	57234
	Directory updates ($X10^3$)	319	560	239	54	1296	329	555	725
	Page transfers	7680	18431	47136	25349	115286	121948	188426	258076
Tmk	Barriers	832	2128	608	48	160	8384	4016	336
	Locks	0	0	1165	2552	68316	0	0	0
	Read faults	4414	18165	42733	16329	106388	93232	187293	129694
	Write faults	7199	4430	18721	8772	7924	27006	91659	195652
	Interrupts ($X10^3$)	8(0.8)	25(3)	50(10)	25(2.2)	275(150)	138(21)	192(12)	1740(55)
	Messages	16277	49437	99193	47419	490735	286969	384927	3479621
	Data	59789	176036	317395	16953	737592	226769	452854	1411434

Table 3: Detailed Statistics at 16 Processors for the Interrupt Versions of Cashmere and TreadMarks

Cashmere nor Treadmarks. As it reduces in size it starts fitting in the cache first for Treadmarks and at a later point for Cashmere. The importance of the effect can be seen in the single-processor numbers. When compiled for Cashmere (with write doubling and protocol operations), Gauss takes about 1750 seconds on one processor. When compiled for TreadMarks, it takes 950. Similarly, LU for Cashmere on one processor takes 380 seconds; for TreadMarks it takes 250. In both cases, modifying the write-doubling code in the Cashmere version so that it "doubles" all writes to a single dummy address reduces the run time to only slightly more than TreadMarks. On a machine with a larger first-level cache (e.g. the 21264), we would not expect to see the same magnitude of effect on performance.

In addition to the primary working set (data accessed in the inner loop) Gauss has a secondary working set which affects performance. The secondary working set for our input size is 32Mbytes/P where P is the number of processors. At 32 processors the data fits in the processor's second level cache resulting in a jump in performance for the Cashmere protocols. Treadmarks does not experience the same performance jump due to memory pressure effects. When running on 32 processors the memory requirements on each node increase beyond what is available in our SMPs. Swapping effects at that point limit performance. Em3d also requires more memory than is available and experiences a performance drop-off when run on 16 and 32 processors.

With the possible exception of TSP, whose results are non-deterministic, polling for messages is uniformly better than fielding signals in both TreadMarks and Cashmere. The difference can be dramatic especially for TreadMarks, with performance differences on 16 processors as high as 49% in Em3d and 60% in Barnes (the two applications with the most amount of active sharing). Polling is also uniformly better than fielding interrupts as a means of emulating remote reads in Cashmere. The differences are again largest in Em3d and Barnes, though generally smaller than in TreadMarks, because the number of page requests—the only events that require interrupts—in Cashmere is always significantly smaller than the number of messages in TreadMarks (Table 3). The tradeoff between polling and the use of a "protocol processor" is less clear from our results, and there seems to be no clear incentive for a protocol processor, if the sole use of the protocol processor is to service page requests. Polling imposes overhead in every loop, but often results in page requests being serviced by a processor that has the desired data in its cache. The protocol processor must generally pull the requested data across the local bus before pushing it into the Memory Channel. Hardware support for reads could be expected to outperform either emulation: it would not impose loop overhead, and would use DMA to ensure a single bus traversal.

Additional bandwidth should also help Cashmere since it has higher bandwidth requirements than Treadmarks. While bandwidth effects are hard to quantify, we have observed that applications perform significantly better when bandwidth pressure is reduced. An 8-processor run of Gauss, for example, is 40% faster with 2 processors on each of 4 nodes than it is with 4 processors on each of 2 nodes. This result indicates that there is insufficient bandwidth on the link between each SMP and MC, something that should be remedied in the next generation of the network.

5 Related Work

Distributed shared memory for workstation clusters is an active area of research: many systems have been built, and more have been designed. For purposes of discussion we group them into systems that support more-or-less "generic" shared-memory programs, such as might run on a machine with hardware coherence, and those that require a special programming notation or style.

5.1 "Generic" DSM

The original idea of using virtual memory to implement coherence on networks dates from Kai Li's thesis work [25]. Nitzberg and Lo [29] 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 [5, 10, 22]. Lazy, multi-writer protocols were pioneered by Keleher et al. [19], 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 [30]. Recent work by the Alewife group at MIT has addressed the implementation of software coherence on a collection of hardware-coherent nodes [39].

Wisconsin's Blizzard system [34] maintains coherence for cache-line-size blocks, either in software or by using ECC. It runs on the Thinking Machines CM-5 and provides a sequentially-consistent programming model. The more recent Shasta system [33], developed at DEC WRL, extends the software-based Blizzard approach with a relaxed consistency model and variable-size coherence blocks. Like Cashmere, Shasta runs on the Memory Channel, with polling for remote requests. Rather than rely on VM, however, it inserts consistency checks in-line when accessing shared memory. Aggressive compiler optimizations attempt to keep the cost of checks as low as possible.

AURC [17] is a multi-writer protocol designed for the Shrimp network interface [3]. Like TreadMarks, AURC uses distributed information in the form of timestamps and write notices to maintain sharing information. Like Cashmere, it relies on remote memory access to write shared data updates to home nodes. Because the Shrimp interface connects to the memory bus of its 486-based nodes, it is able to double writes in hardware, avoiding a major source of overhead in Cashmere. Experimental results for AURC are currently based on simulation; implementation results await the completion of a large-scale Shrimp testbed.

5.2 Special Programming Models

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 [11] and Shrimp's Deliberate Update [3], the programmer must use special primitives to read and write remote data. In others, including Shared Regions [31], Cid [28], and CRL [18], remote data is 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 [40] 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 [8, 14, 36], or support inter-process communication via sharing of special concurrent objects [7, 32].

Because they provide the coherence system with information not available in more general-purpose systems, special programming models have the potential to provide superior performance. It is not yet clear to what extent the extra effort required of programmers will be considered an acceptable burden. In some cases, it may be possible for an optimizing compiler to obtain the performance of the special programming model without the special syntax [13].

5.3 Fast User-Level Messages

The Memory Channel is not unique in its support for user-level messages, though it is the first commercially-available workstation network with such an interface. Large shared memory multiprocessors have provided low-latency interprocessor communication for many years, originally on cache-less machines and more recently with cache coherence. We believe that a system such as Cashmere would work well on a non-cache-coherent machine like the Cray T3E. Fast user-level messages were supported without shared memory on the CM-5, though the protection mechanism was relatively static.

Among workstation networks, user-level IPC can also be found in the Princeton Shrimp [3], the HP Hamlyn interface [6] to Myrinet [4], and Dolphin's snooping interface [26] for the SCI cache coherence protocol [16].

6 Conclusion and Future Work

We have presented results for two different DSM protocols—Cashmere and TreadMarks—on a remote-memory-access network, namely DEC's Memory Channel. TreadMarks uses the Memory Channel only for fast messaging, while Cashmere uses it for directory maintenance and for fine-grained updates to shared data. Our work is among the first comparative studies of fine and coarse-grained DSM to be based on working implementations of "aggressively lazy" protocols.

Our principal conclusion is that low-latency networks make fine-grain DSM competitive with more coarse-grain approaches, but that further hardware improvements will be needed before such systems can provide consistently superior performance. TreadMarks requires less frequent communication and lower total communication bandwidth, but suffers from the computational overhead of twinning and diffing, which occur on the program's critical path, from the need to interrupt remote processors (or otherwise get their attention), and from the propagation of unnecessary write notices, especially on larger configurations. Cashmere moves much of the overhead of write collection off of the critical path via write-through, and eliminates the need for synchronous remote operations (other than emulated page reads), but at the expense of much more frequent communication, higher communication bandwidth, and (in the absence of snooping hardware) significant time and cache-footprint penalties for software write doubling. Unlike TreadMarks, Cashmere sends write notices only to processors that are actually using the modified data, thereby improving scalability (though the use of global time can sometimes lead to invalidations not required by the happens-before relationship).

Polling for remote requests is a major win in Tread-Marks, due to the frequency of request-reply communication. Polling is less important in Cashmere, which uses it only to get around the lack of a remote-read mechanism: polling works better than remote interrupts, and is generally comparable in performance to emulating remote reads with a dedicated processor.

The fact that Cashmere is able to approach (and sometimes exceed) the performance of TreadMarks on the current Memory Channel suggests that DSM systems based on fine-grain communication are likely to out-perform more coarse-grain alternatives on future remote-memory-access networks. Many of the most severe constraints on our current Cashmere implementation result from hardware limitations that are likely to disappear in future systems. Specifically, we expect the next generation of the Memory Channel to cut latency by more than half, to increase aggregate bandwidth by more than an order of magnitude, and to significantly reduce the cost of synchronization. These changes will improve the performance of TreadMarks, but should help Cashmere more. Eventually, we also expect some sort of remote read mechanism, though probably not remote cache fills. It seems unlikely that inexpensive networks will provide write doubling in hardware, since this requires snooping on the memory bus. The impact of software write doubling on cache footprint, however, should be greatly reduced on newer Alpha processors with larger first and second level caches. In addition, access to the TLB/page table dirty bit (currently in the works) will allow us to make protocol changes that should largely eliminate unnecessary invalidations.

We are currently pursuing several improvements to the Cashmere implementation. In addition to using dirty bits, we plan to experiment with hierarchical locking for directories [37] and with alternatives to write-doubling based on twins and diffs or on software dirty bits [33]. To reduce page fault handling overhead (both for TreadMarks and for Cashmere), we plan to modify the kernel to eliminate the need for `mprotect` system calls. When a page is present but not accessible, the kernel will make the page writable before delivering the `sig_segv` signal, and will allow the return call from the handler to specify new permissions for the page. This capability should be useful not only for DSM, but for any application that uses virtual memory protection to catch accesses to particular pages [2]. At a more aggressive level, we plan to evaluate the performance impact of embedding the page-fault handling software directly in the kernel.

We are also continuing our evaluation of protocol alternatives. Cashmere and TreadMarks differ in two fundamental dimensions: the mechanism used to manage coherence information (directories v. distributed intervals and timestamps) and the mechanism used to collect data updates (write-through v. twins and diffs). The Princeton AURC protocol [17] combines the TreadMarks approach to coherence management with the Cashmere approach to write collection. We plan to implement AURC on the Memory Channel, and to compare it to Cashmere, to TreadMarks, and to the fourth alternative, which would combine directories with twins and diffs (or with software dirty bits). In addition, we are developing multi-level protocols that take advantage of hardware coherence within each AlphaServer node, and that also allow a remote-memory-access cluster to function as a single node of a larger, message-based shared-memory system. Finally, we are continuing our research into the relationship between run-time coherence management and static compiler analysis [13].

References

[1] 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. In *Computer*, to appear.

[2] A. W. Appel and K. Li. Virtual Memory Primitives for User Programs. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 96–107, Santa Clara, CA, April 1991.

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

of the Twenty-First International Symposium on Computer Architecture, pages 142–153, Chicago, IL, April 1994.

[4] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. E. Seitz, J. N. Seizovic, and W.-K. Su. Myrinet: A Gigabit-per-Second Local Area Network. In *IEEE Micro*, pages 29–36, February 1995.

[5] W. J. Bolosky, R. P. Fitzgerald, and M. L. Scott. Simple But Effective Techniques for NUMA Memory Management. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, pages 19–31, Litchfield Park, AZ, December 1989.

[6] G. Buzzard, D. Jacobson, M. Mackey, S. Marovich, and J. Wilkes. An Implementation of the Hamlyn Sender-Managed Interface Architecture. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation*, Seattle, WA, October 1996.

[7] N. Carriero and D. Gelernter. Linda in Context. *Communications of the ACM*, 32(4):444–458, April 1989. Relevant correspondence appears in Volume 32, Number 10.

[8] J. S. Chase, F. G. Amador, E. D. Lazowska, H. M. Levy, and R. J. Littlefield. The Amber System: Parallel Programming on a Network of Multiprocessors. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, pages 147–158, Litchfield Park, AZ, December 1989.

[9] R. W. Cottingham Jr., R. M. Idury, and A. A. Schaffer. Faster Sequential Genetic Linkage Computations. *American Journal of Human Genetics*, 53:252–263, 1993.

[10] A. L. Cox and R. J. Fowler. The Implementation of a Coherent Memory Abstraction on a NUMA Multiprocessor: Experiences with PLATINUM. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, pages 32–44, Litchfield Park, AZ, December 1989.

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

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

[13] S. Dwarkadas, A. L. Cox, and W. Zwaenepoel. An Integrated Compile-Time/Run-Time Software Distributed Shared Memory System. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, Boston, MA, October 1996.

[14] M. J. Feeley, J. S. Chase, V. R. Narasayya, and H. M. Levy. Integrating Coherency and Recovery in Distributed Systems. In *Proceedings of the First Symposium on Operating Systems Design and Implementation*, Monterey, CA, November 1994.

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

[16] D. B. Gustavson. The Scalable Coherent Interface and Related Standards Projects. *IEEE Micro*, 12(2):10–22, February 1992.

[17] L. Iftode, C. Dubnicki, E. W. Felten, and K. Li. Improving Release-Consistent Shared Virtual Memory Using Automatic Update. In *Proceedings of the Second International Symposium on High Performance Computer Architecture*, San Jose, CA, February 1996.

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

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

[20] L. I. Kontothanassis and M. L. Scott. High Performance Software Coherence for Current and Future Architectures. *Journal of Parallel and Distributed Computing*, 29(2):179–195, November 1995.

[21] L. I. Kontothanassis and M. L. Scott. Using Memory-Mapped Network Interfaces to Improve the Performance of Distributed Shared Memory. In *Proceedings of the Second International Symposium on High Performance Computer Architecture*, San Jose, CA, February 1996.

[22] R. P. LaRowe Jr. and C. S. Ellis. Experimental Comparison of Memory Management Policies for NUMA Multiprocessors. *ACM Transactions on Computer Systems*, 9(4):319–363, November 1991.

[23] G. M. Lathrop, J. M. Lalouel, C. Julier, and J. Ott. Strategies for Multilocus Linkage Analysis in Humans. *Proceedings of the National Academy of Science, USA*, 81:3443–3446, June 1984.

[24] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor. In *Proceedings of the Seventeenth International Symposium on Computer Architecture*, pages 148–159, Seattle, WA, May 1990.

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

[26] O. Lysne, S. Gjessing, and K. Lochsen. Running the SCI Protocol over HIC Networks. In *Second International Workshop on SCI-based Low-cost/High-performance Computing (SCIzzL-2)*, Santa Clara, CA, March 1995.

[27] 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 *Proceedings of the Ninth International Parallel Processing Symposium*, Santa Barbara, CA, April 1995.

[28] R. S. Nikhil. Cid: A Parallel, "Shared-memory" C for Distributed-Memory Machines. In *Proceedings of the Seventh Annual Workshop on Languages and Compilers for Parallel Computing*, August 1994.

[29] B. Nitzberg and V. Lo. Distributed Shared Memory: A Survey of Issues and Algorithms. *Computer*, 24(8):52–60, August 1991.

[30] K. Petersen and K. Li. Cache Coherence for Shared Memory Multiprocessors Based on Virtual Memory Support. In *Proceedings of the Seventh International Parallel Processing Symposium*, Newport Beach, CA, April 1993.

[31] H. S. Sandhu, B. Gamsa, and S. Zhou. The Shared Regions Approach to Software Cache Coherence on Multiprocessors. In *Proceedings of the Fourth ACM Symposium on Principles and Practice of Parallel Programming*, San Diego, CA, May 1993.

[32] D. J. Scales and M. S. Lam. The Design and Evaluation of a Shared Object System for Distributed Memory Machines. In *Proceedings of the First Symposium on Operating Systems Design and Implementation*, Monterey, CA, November 1994.

[33] D. J. Scales, K. Gharachorloo, and C. A. Thekkath. Shasta: A Low Overhead, Software-Only Approach for Supporting Fine-Grain Shared Memory. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, Boston, MA, October 1996.

[34] I. Schoinas, B. Falsafi, A. R. Lebeck, S. K. Reinhardt, J. R. Larus, and D. A. Wood. Fine-grain Access Control for Distributed Shared Memory. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 297–306, San Jose, CA, October 1994.

[35] 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, March 1992.

[36] A. S. Tanenbaum, M. F. Kaashoek, and H. E. Bal. Parallel Programming Using Shared Objects and Broadcasting. *Computer*, 25(8):10–19, August 1992.

[37] R. C. Unrau, O. Krieger, B. Gamsa, and M. Stumm. Experiences with Locking in a NUMA Multiprocessor Operating System Kernel. In *Proceedings of the First Symposium on Operating Systems Design and Implementation*, Monterey, CA, November 1994.

[38] 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 *Proceedings of the Twenty-Second International Symposium on Computer Architecture*, Santa Margherita Ligure, Italy, June 1995.

[39] D. Yeung, J. Kubiatowitcz, and A. Agarwal. MGS: A Multigrain Shared Memory System. In *Proceedings of the Twenty-Third International Symposium on Computer Architecture*, Philadelphia, PA, May, 1996.

[40] M. J. Zekauskas, W. A. Sawdon, and B. N. Bershad. Software Write Detection for Distributed Shared Memory. In *Proceedings of the First Symposium on Operating Systems Design and Implementation*, Monterey, CA, November 1994.