VM-Based Shared Memory on Low-Latency, Remote-Memory-Access Networks*

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

Department of Computer Science
University of Rochester
Rochester, NY 14627–0226

† DEC Cambridge Research Lab
One Kendall Sq., Bldg. 700
Cambridge, MA 02139

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 false sharing. At the same time, it is severely constrained by limitations of the current Memory Channel hardware. In general, performance is better for TreadMarks.

*This work was supported in part by NSF grants CDA–9401142, CCR–9319445, CCR–9409120, and CCR–9510173; ARPA contract F19628–94–C–0057; an external research grant from Digital Equipment Corporation; and graduate fellowships from Microsoft Research (Galen Hunt) and CNPq–Brazil (Wagner Meira, Jr., Grant 200.862/93-6).

To appear in the Proceedings of the Twenth-Fourth Annual International Symposium on Computer Architecture, Denver, CO, June 1997.

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, 16]) 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 traditional 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 [17, 18]. 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.

In this paper we compare implementations of TreadMarks and a modified version of Cashmere on a 32-processor cluster (8 nodes, 4 processors each) of DEC AlphaServers, connected by DEC's Memory Channel [12] 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 aim is to compare a more fine-grain approach to shared memory to a state-of-the-art DSM that does

not rely on remote memory access. In future work we intend to study alternative fine-grain protocols in more detail; we believe the current version of Cashmere to be a good first cut.

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 for the Memory Channel. 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. The high remote memory channel interrupt overheads of Digital Unix significantly reduce performance for both TreadMarks and Cashmere. In general, TreadMarks is more sensitive to interrupt latencies since it uses request-response communication for both synchronization and data transfer. The Memory Channel implementation of Cashmere uses interrupts only for page (data) transfers. Cashmere takes advantage of remote memory access for program and meta-data resulting in fewer messages requiring a reply. In our experiments, Cashmere scales slightly better than TreadMarks for applications with false sharing since the use of home nodes reduces the number of request-response messages required to update the local copy of a page. In general, however, performance is better for TreadMarks due to reduced communication requirements. Overall, Cashmere spends less time in protocol code at the expense of extra communication, but is severely constrained by limitations of the current Memory Channel hardware.

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 [18], 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

Both TreadMarks and Cashmere 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. Mainly, TreadMarks does not take advantage of remote memory access for anything other than fast messaging, while Cashmere uses remote memory access for fine-grain communication of coherence information and shared data.

2.1 Cashmere

Cashmere maintains coherence information using a distributed directory data structure. Currently, directory entries are organized as a set of eight 4-byte words, one for each SMP node in the system. Each word contains presence bits for each of the four processors within the SMP (4 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), and an indication of whether a particular CPU within a node has exclusive read/write permission for the page (4 bits). The home node indications in separate words are redundant. Directory space overhead for the 8K pages supported by Digital Unix is about 3% and would be smaller if we did not have to replicate the directory on each of our eight nodes. Directory overhead could be reduced by packing the directory information into a single long word (64 bits). However, this would require locking of the directory entry. The current directory structure does not require the use of locks except to set the home node of a page.

The choice of home node 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 [23]. The home node is set only once during the lifetime of a program, and thus the use of locks does not impact performance.

In addition to the directory data structure, each processor also holds two globally accessible lists, the *write notice list* and the *no longer exclusive (NLE) list*. These globally accessible lists are protected by cluster-wide locks. The *write notice list* contains descriptors (write notices) for pages that are valid on the

processor and have been written by remote processors. A bit map is also associated with the *write notice list* in order to avoid duplicate write notices for the same page. The *NLE list* contains descriptors for pages to which the processor once held exclusive access, but which are now shared by multiple processors. A processor can avoid certain overhead for a page so long as it retains exclusive access (see below).

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 acquires exclusive access to the directory word representing its SMP node (using *load-linked/store-conditional (ll/sc)* within the node), and adds itself to the sharing set. It then checks to see if the home node has been assigned (valid home node bits). If the home node bits are not valid and home node ownership has already been asserted by another node, the faulting processor sets the home node bits in its node's directory entry. If home node ownership has not been asserted by any other node (checked by examining all other directory words), the faulting processor acquires the directory entry lock to assert home node ownership. It also scans all words in the directory entry to determine if another processor has previously acquired exclusive read/write permission on the page. If so, it appends a descriptor for the page to the *NLE list* of the former exclusive-sharer processor.

Locally, the faulting processor creates a mapping for the page. 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, a copy of the page is requested from the home node, and then written into the local copy (see below). A write page fault on an invalid page is treated the same as a read page fault. In addition, on each write page fault, the processor inserts the page number in a local list called the *dirty list*. The faulting operation is then restarted after providing the appropriate (read and write) permissions to the page.

At an acquire synchronization operation, the processor traverses its *write notice list* and removes itself from the sharing set of each page found therein. The removal operation requires that the processor obtain exclusive access (within the SMP node) for the corresponding word in the directory entry, and modify the bitmask of sharing processors.

At a release synchronization operation, the processor traverses its *dirty list* and informs other processors of the writes it has performed since the previous release operation. For each entry in the *dirty list* the processor scans the directory entry for that page. If other processors are found to be sharing the page, the releaser appends a notice to the *write notice list* of every sharing processor, if a write notice does not already exist for the page (checked by examining the bit mask). If no other processor is found sharing the page, the processor indicates that the page is in exclusive mode by appropriately modifying its directory word. If the page does not move to exclusive mode, the processor downgrades its permission to read-only so as to catch subsequent writes. After processing its *dirty list* the processor also traverse its *NLE* list. For each entry listed in the list it takes the same set of actions it took for entries in the *dirty list* with two differences. It sets a flag indicating that this page should never move to exclusive mode again and also avoids moving it to exclusive mode now, even if no other processors are found to be sharing the page.

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. Our protocol guarantees that a release operation cannot complete before all its writes have been applied at the unique home node. Future acquires are guaranteed to find data that is in their logical past 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 (write doubling).

This protocol differs in a number of ways from the Cashmere protocol employed in simulation-based studies [18]. The most important differences stem from differences in hardware platforms. The ideal simulated hardware (i.e. the one that yields the best performance for Cashmere-like protocols) assumes the ability to map remote pages and service cache misses to them in hardware as well as the ability to forward writes to home nodes using write-through. In the current implementation we were forced to use page copying and doubling of writes in software (see section 3).

We have also made modifications to the protocol itself. In particular we have removed the "weak" state from the implemented protocol. In simulation, any page with at least one writer resided in the weak state. Weak pages were optimistically assumed to be modified during every synchronization interval: all sharers would automatically invalidate the pages during acquire operations. The current protocol opts instead for the exclusive mode and for explicit write notices. Pages in exclusive mode experience only the initial write fault, the minimum of possible protocol overhead. At release operations, processors send write notices for all modified non-exclusive pages that do not already have such a notice pending. These two enhancements improve Cashmere's ability to efficiently handle private pages and producer-consumer sharing patterns.

2.2 TreadMarks

TreadMarks is a distributed shared memory system based on lazy release consistency (LRC) [16]. Lazy release consistency is a variant of release consistency [20]. 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. 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 run-length encoding of the changes, 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, and because all information is local and communication occurs only via messaging, 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 TreadMarks can be found in other papers [1].

3 Implementation Issues

3.1 Memory Channel

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 (a single virtual address mapping can only be either for transmit or for receive). 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 to transmit regions 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.

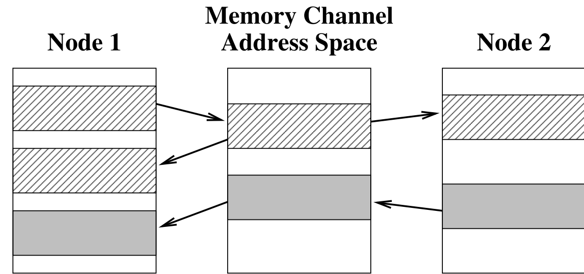

Figure 1: Memory Channel space. The lined region is mapped for both transmit and receive on node 1 and for receive on node 2. The gray region is mapped for receive on node 1 and for transmit on node 2.

```
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.

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 due to a limitation on the early implementation of the Memory Channel device driver. We expect that number to get closer to the 100MB/s supported by the hardware with the new driver release.

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.

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. The interrupt is filtered up through the kernel to the receiving process only when the receiving process subsequently enters the kernel, resulting in an average inter-node signal 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[1]. 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 programs that don't scale well it can actually be better than further parallelism). 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

Cashmere takes advantage of MC remote writes to collect shared memory writes, to update the page directory, and to implement synchronization. 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, write notice and NLE lists, 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

[1]We did not use ATOM since the current version can only insert instrumentation in the form of procedure calls incurring singificantly higher overhead

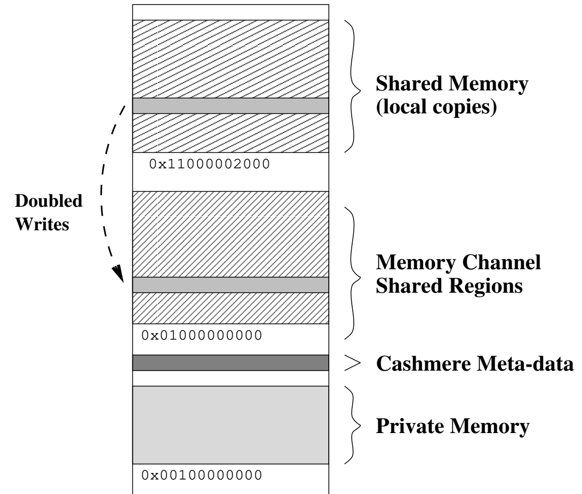

Figure 3: Cashmere process address space. Writes are duplicated from the private copy of a shared memory page to the corresponding page in the MC regions. MC regions are mapped for receive on home nodes and for transmit on all other nodes.

```
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.

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. Each MC region in our Cashmere implementation contains a *superpage*, the size of which is obtained by dividing the maximum shared memory size required by the application by the number of table entries. Superpages have no effect on coherence granularity: we still map and unmap individual pages. They do, however, constrain our "first touch" page placement policy: all pages of a given superpage must share the same home node.

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 ensures 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 eight 4-byte words.

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 *ll/sc*. 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. Most of that overhead is due to the crossing of thr kernel-user boundary. Application and protocol locks use the same implementation. Application barriers are tree-based. Upon arrival, each processor first waits for all of its children to arrive, then notifies its parent of the sub-tree's arrival, and finally waits for notification from the root of the barrier tree before continuing. All notifications are posted through explicit words in Memory Channel space.

3.4 TreadMarks

We have modified TreadMarks version 0.10.2 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—the polling instrumentation used for TreadMarks and Cashmere is identical) 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. No other processors are connected to the Memory Channel. Each AlphaServer runs in multi-user mode, but with the exception of normal Unix daemons no other processes were active during the tests. The underlying virtual memory page size is 8 Kbytes. A cache line is 64 bytes.

For Cashmere, we present results for three different mechanisms to handle remote requests—a dedicated "protocol processor" (`csm_pp`), `imc_kill` 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 sigio interrupts (tmk_udp_int), one that uses user-level messaging on MC with imc_kill interrupts (tmk_mc_int), and one that uses user-level messaging on MC with polling (tmk_mc_poll).

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 36% 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 16 μs for Cashmere, if locking is required, and 5 μs otherwise. In other words, 11 μs is spent acquiring and releasing the directory entry lock, but only when relocating the home node. 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. The red and black arrays are divided into roughly equal size bands of rows, with each band assigned to a different processor. Communication occurs across the boundaries between bands. Processors synchronize with barriers.

LU: a kernel from the SPLASH-2 [32] benchmark, which for a given matrix A finds its factorization $A = LU$, where L is a lower-triangular matrix and U is upper triangular. The matrix A is divided into square blocks for temporal and spatial locality. Each block is "owned" by a particular processor, which performs all computation on it.

Water: a molecular dynamics simulation from the SPLASH-1 [30] benchmark suite. The shared array of molecule structures is divided into equal contiguous chunks, with each chunk assigned to a different processor. The bulk of the interprocessor communication happens during a computation phase that computes intermolecular forces. Each processor accumulates its forces locally and then acquires per-processor locks to update the globally shared force vectors, resulting in a migratory sharing pattern.

TSP: a branch-and-bound solution to the traveling salesman problem. Locks are used to insert and delete unsolved tours in a priority queue. Updates to the shortest path are 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: a solver for a system of linear equations $AX = B$ using 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.

Ilink: a widely used genetic linkage analysis program from the FASTLINK 2.3P package that locates disease genes on chromosomes. We use the parallel algorithm described by Dwarkadas et al. [8]. The main shared data is a pool of sparse arrays of genotype probabilities. Updates to each array are parallelized. A master processor assigns individual array 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. 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 [30] 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 Barnes-Hut tree construction is performed sequentially, while all other phases are parallelized and dynamically load balanced. Synchronization consists of barriers between phases.

Em3d: a program to simulate electromagnetic wave propagation through 3D objects [7]. 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. For each phase in the computation, each processor updates the electromagnetic potential of its nodes based on the potential of neighboring nodes. While arbitrary graphs of dependencies between nodes can be constructed, the standard input assumes that nodes that belong to 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

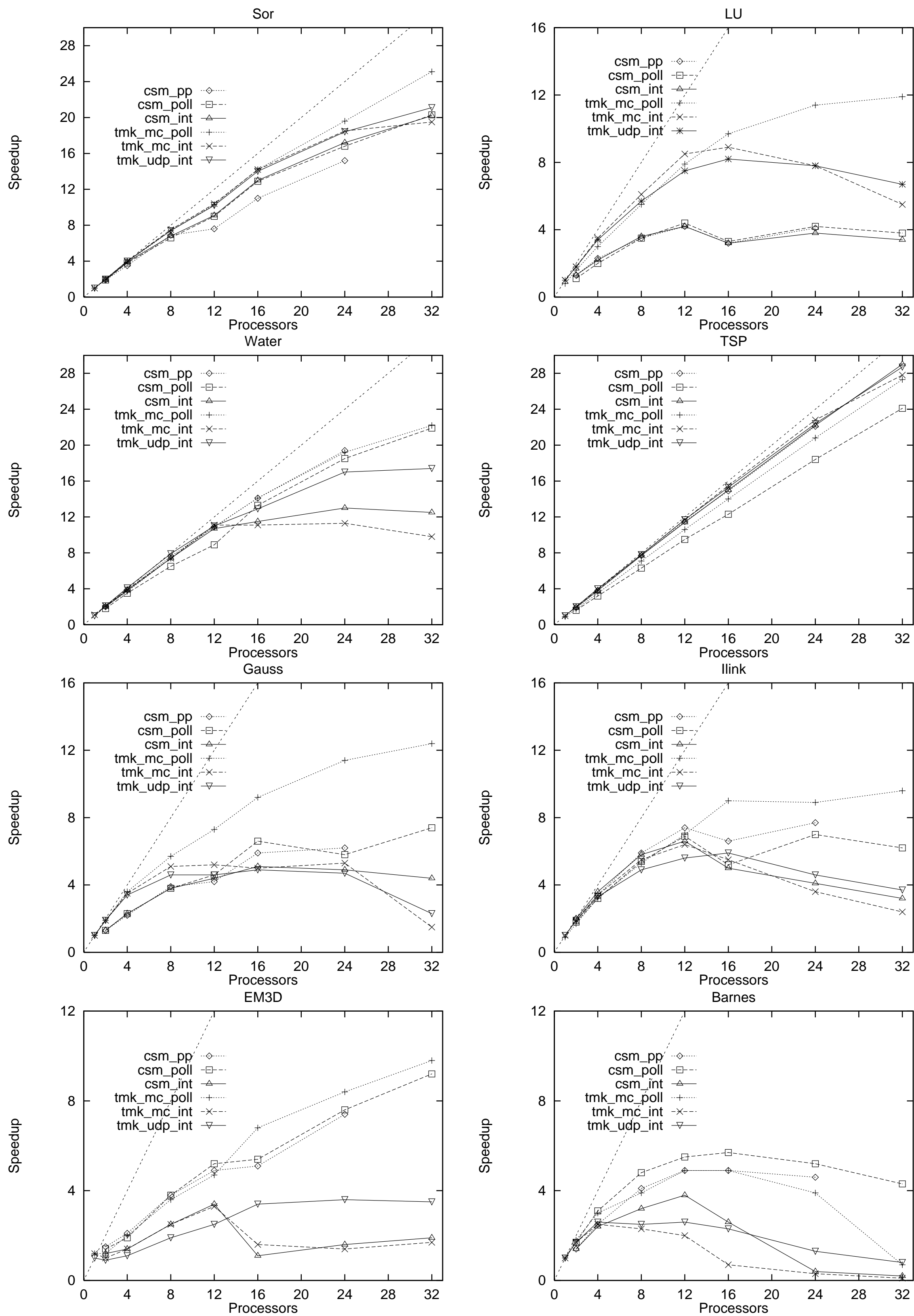

Figure 5: Speedups.

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.

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.

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: two processors in each of 8 nodes; 24: three processors in each of 8 nodes; and 32: trivial, but not applicable to csm_pp.

Table 3 presents detailed statistics on the communication incurred by each of the applications on the polling implementations of Cashmere and TreadMarks at 32 processors, except for Barnes, where the statistics presented are for 16 processors. Since the performance for Barnes drops significantly with more than 16 processors, we present the statistics at 16 processors in order to make a reasonable comparison. The statistics presented are the execution time, the number of lock and barrier synchronization operations, number of read and write faults, the number of page transfers for Cashmere, and the number of messages and data transferred for TreadMarks. Other than execution time, the statistics are aggregated over all 32 (16 in the case of Barnes) processors.

Figure 6 presents a break-down of the execution time processors for each application. The breakdown is normalized with respect to total execution time for Cashmere on 32 processors (16 for Barnes). The components shown represent time spent executing user code (`User`), the overhead of profiling for polling (`Polling`—Cashmere and TreadMarks) and write doubling (`Write doubling`—Cashmere only), time spent in protocol code (`Protocol`), and communication and wait time (`Comm & Wait`). Two of the components—`Comm & Wait` and `Protocol`—were measured directly on the maximum number of processors. The remaining three components—`User`, `Polling`, and `Write Doubling`—could not be measured directly and thus reflect extrapolation from the one-processor case. (On one processor we could turn polling and write doubling on and off and measure the change in execution time. On 32 [or 16] processors we assume that the ratios of user and

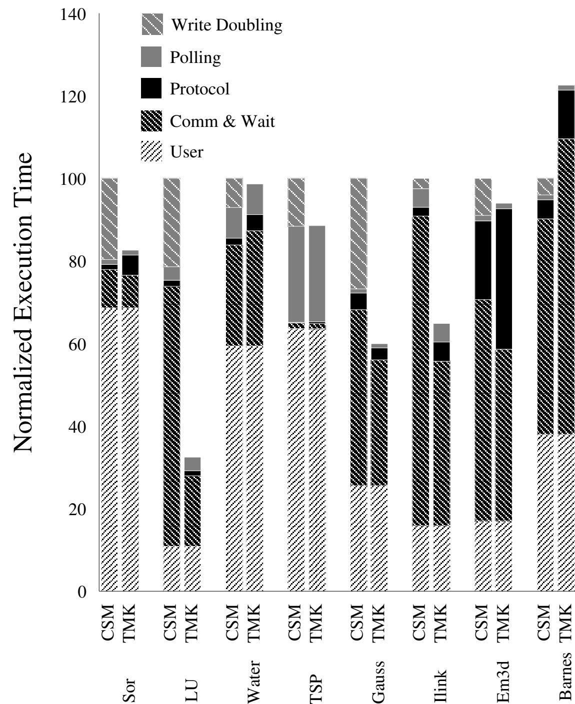

Figure 6: Breakdown of normalized execution time for the polling versions of Cashmere and TreadMarks (Barnes at 16 processors, the others at 32).

polling time remain the same as on a single processor, and use these times (calculated from TreadMarks) to determine the write doubling time for Cashmere).

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. Generalizing over varying numbers of processors (see figure 5), performance of Cashmere and TreadMarks is similar for TSP and Water. Cashmere outperforms TreadMarks on Barnes. TreadMarks outperforms Cashmere by significant amounts on LU and Gauss, and by smaller amounts on SOR, Em3d, and Ilink.

For Barnes, where Cashmere performs better than TreadMarks, the differences in performance stem primarily from Cashmere's ability to merge updates from multiple writers into a single home node, as well as to eliminate duplicate write notices for the same page: TreadMarks must generally merge diffs from multiple sources to update a page in Barnes (note the high message count in Table 3, and the larger fraction of communication

	Application	SOR	LU	Water	TSP	Gauss	Ilink	Em3d	Barnes
CSM	Exec. time (secs)	9.6	66.9	84.3	166.8	128.5	145.2	21.9	78.5
	Barriers	48	129	36	2	8	521	250	10
	Locks	0	0	3680	2638	133914	0	0	0
	Read faults	6242	27041	84271	20442	207035	228605	195268	127029
	Write faults	9116	7470	35573	9057	8181	45485	95875	98273
	Page transfers	9312	27046	84272	23805	207037	230010	195268	127032
TMK	Exec. time (secs)	7.8	21.7	83.1	147.8	77.0	94.2	20.6	96.3
	Barriers	48	129	36	2	8	521	250	10
	Locks	0	0	3680	2664	133914	0	0	0
	Read faults	2976	22828	84300	20479	220748	194462	194349	49577
	Write faults	2976	4302	33001	9133	8052	42562	95175	98279
	Messages	8990	83896	217590	57841	1008056	653402	406320	1451167
	Data (Kbytes)	27296	300400	816601	36021	1008820	511804	460774	592509

Table 3: Detailed statistics for the polling versions of Cashmere and TreadMarks (Barnes at 16 processors, the others at 32).

and wait time in Figure 6), and communicate a write notice for each process that writes a page. In addition, Figure 6 shows that TreadMarks spends a larger fraction of time executing protocol code (diffing and twinning). This overhead is avoided in Cashmere through the use of write-through to merge the changes by multiple writers.

For Ilink, where TreadMarks performs better than Cashmere, the differences in performance stem from the sparsity of Ilink's data structures. Only a small portion of each page is modified between synchronization operations. Hence, the diffs of TreadMarks result in less data communication than the page reads of Memory-Channel Cashmere, resulting in a much larger amount of time spent in communication for Cashmere (see Figure 6).

In Water the various competing effects more or less cancel out. TreadMarks spends more time in protocol code in diffing and twinning, while Cashmere spends more time communicating whole pages (the entire molecule data structure is not modified between any two synchronization points). It is difficult to draw conclusions from TSP, because of its non-determinism.

Write doubling to internal rows of an SOR or Em3d band is 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, but the doubled writes affect performance none-the-less. This effect can be seen in Figure 6, where write doubling comprises 19% of the total time in SOR and 8% of Em3d's execution time. Em3d spends significantly more time in protocol code for TreadMarks, performing diffing and twinning, while Cashmere spends more time in communication, once again because entire pages are transferred on a miss even though only part of the page was modified.

The most dramatic differences between Cashmere and TreadMarks occur in LU and Gauss, and can be traced to cache effects. Specifically, the increase in cache pressure caused by write doubling forces the working set of these applications out of the first-level cache of the 21064A. In LU, with a 32X32 block, the primary working set 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 fewer rows remain to be eliminated. For our problem size the working set starts by not fitting in the first level cache for Cashmere and 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 954. Similarly, LU for Cashmere on one processor takes 380 seconds; for TreadMarks it takes 255. 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. These effects show up in the write doubling sub-bars of Figure 6. Note that these sub-bars account for 21% of total execution time in LU, and 27% in Gauss. 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 the memory pressure effects of requiring space in the cache for diffs and twins.

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 for larger numbers of processors (the effect is a function of the number of remote operations). High interrupt latencies also result in a sudden reduction in performance for Em3d and Barnes (the two applications with the most active sharing) for both protocols when we move from 12 processors spread across 4 nodes to 16 processors spread across 8 nodes. At 16 processors in Barnes, the high cost of the *imc_kill* signal results in an 86% drop in performance for `tmk_mc_int` relative to `tmk_mc_poll`, with performance actually being worse than that for `tmk_udp_int`. At small

numbers of processors, `tmk_udp_int` does perform worse than `tmk_mc_int` as expected, because of the overhead of executing the `udp` protocol stack. In general, TreadMarks is more sensitive to interrupt latency than is Cashmere, due to the higher number of request-response communications (synchronization and data transfers). Cashmere requires request-response communication only for data transfer. For example, in Gauss at 32 processors, TreadMarks fields 231760 interrupts, while Cashmere fields only 186001 interrupts.

The tradeoff between polling and the use of a "protocol processor" in Cashmere 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 is consistent with the findings of Erlichson et al. [10]; it indicates that there is insufficient bandwidth on the link between each SMP and the hub, 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 [21]. Nitzberg and Lo [25] 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 [4, 19]. Lazy, multi-writer protocols were pioneered by Keleher et al. [16], 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 [26]. Recent work by the Alewife group at MIT [33] and the FLASH group at Stanford [10] has addressed the implementation of software coherence on a collection of hardware-coherent nodes.

Wisconsin's Blizzard system [29] 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 [28], 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 [14] is a multi-writer protocol designed for the Shrimp network interface [2]. Like Cashmere, AURC relies on remote memory access to write shared data updates to home nodes. Like TreadMarks, however, AURC uses distributed information in the form of timestamps and write notices to maintain sharing information. As a result, AURC has no need for directory, lock, or write notice/NLE list metadata: remote-mapped memory is used only for fast messages and doubled writes. The Shrimp network interface, like that of the Memory Channel, does not support remote reads, presumably because they are significantly more difficult than writes to implement, and less crucial for good performance. Where the Cashmere protocol was originally designed to read lines from remote memory on a cache miss, the AURC protocol was designed from the outset with the assumption that whole pages would be copied to local memory. Because the Shrimp interface connects to the memory bus of its 486-based nodes, AURC 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 [7] and Shrimp's Deliberate Update [2], the programmer must use special primitives to read and write remote data. In others, including Shared Regions [27], Cid [24], and CRL [15], 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 [34] 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 [6, 11, 31].

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 [9].

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 work-

station 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 [2], the HP Hamlyn interface [5] to Myrinet [3], and Dolphin's snooping interface [22] for the SCI cache coherence protocol [13].

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, and from the propagation of unnecessary write notices, especially on larger configurations. Cashmere moves much of the overhead of write collection off the critical path via write-through, 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).

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.

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). We are currently developing multi-level protocols that take advantage of hardware coherence within each AlphaServer node, and that combine the benefits of the TreadMarks and Cashmere protocols. These protocols will reduce the bandwidth requirements of write-through and eliminate write doubling overhead through the use of diffs and twins, while retaining the ability to merge modifications at a single node and to reduce coherence information by distributing it at a fine grain. Finally, we are continuing our research into the relationship between run-time coherence management and static compiler analysis [9].

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. *Computer*, 29(2):18–28, Feb. 1996.

[2] 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*, pp. 142–153, Apr. 1994.

[3] 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*, pp. 29–36, Feb. 1995.

[4] W. J. Bolosky, M. L. Scott, R. P. Fitzgerald, R. J. Fowler, and A. L. Cox. NUMA Policies and Their Relation to Memory Architecture. In *Proc. of the 4th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pp. 212–221, Apr. 1991.

[5] 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*, Oct. 1996.

[6] 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 *Proc. of the Twelfth ACM Symp. on Operating Systems Principles*, pp. 147–158, Dec. 1989.

[7] 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*, pp. 262–273, Nov. 1993.

[8] 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.

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

[10] 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*, Oct. 1996.

[11] M. J. Feeley, J. S. Chase, V. R. Narasayya, and H. M. Levy. Integrating Coherency and Recovery in Distributed Systems. In *Proc. of the 1st Symp. on Operating Systems Design and Implementation*, Nov. 1994.

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

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

[14] L. Iftode, C. Dubnicki, E. W. Felten, and K. Li. Improving Release-Consistent Shared Virtual Memory Using Automatic Update. In *Proc. of the 2nd Intl. Symp. on High Performance Computer Architecture*, Feb. 1996.

[15] K. L. Johnson, M. F. Kaashoek, and D. A. Wallach. CRL: High-Performance All-Software Distributed Shared Memory. In *Proc. of the Fifteenth ACM Symp. on Operating Systems Principles*, Dec. 1995.

[16] 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*, pp. 13–21, May 1992.

[17] 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, Nov. 1995.

[18] L. I. Kontothanassis and M. L. Scott. Using Memory-Mapped Network Interfaces to Improve the Performance of Distributed Shared Memory. In *Proc. of the 2nd Intl. Symp. on High Performance Computer Architecture*, Feb. 1996.

[19] 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, Nov. 1991.

[20] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor. In *Proc. of the 17th Intl. Symp. on Computer Architecture*, pp. 148–159, May 1990.

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

[22] O. Lysne, S. Gjessing, and K. Lochsen. Running the SCI Protocol over HIC Networks. In *2nd Intl. Workshop on SCI-based Low-cost/High-performance Computing (SCIzzL-2)*, Mar. 1995.

[23] 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 Ninth Intl. Parallel Processing Symp.*, Apr. 1995.

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

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

[26] K. Petersen and K. Li. Cache Coherence for Shared Memory Multiprocessors Based on Virtual Memory Support. In *Proc. of the 7th Intl. Parallel Processing Symp.*, Apr. 1993.

[27] H. S. Sandhu, B. Gamsa, and S. Zhou. The Shared Regions Approach to Software Cache Coherence on Multiprocessors. In *Proc. of the 4th ACM Symp. on Principles and Practice of Parallel Programming*, May 1993.

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

[29] 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 *Proc. of the 6th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pp. 297–306, Oct. 1994.

[30] 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.

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

[32] 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*, June 1995.

[33] D. Yeung, J. Kubiatowitcz, and A. Agarwal. MGS: A Multigrain Shared Memory System. In *Proc. of the 23rd Intl. Symp. on Computer Architecture*, May, 1996.

[34] M. J. Zekauskas, W. A. Sawdon, and B. N. Bershad. Software Write Detection for Distributed Shared Memory. In *Proc. of the 1st Symp. on Operating Systems Design and Implementation*, Nov. 1994.