Cashmere-2L: Software Coherent Shared Memory on a Clustered Remote-Write Network*

Robert Stets, Sandhya Dwarkadas, Nikolaos Hardavellas,
Galen Hunt, Leonidas Kontothanassis,† Srinivasan Parthasarathy,
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

cashmere@cs.rochester.edu

Abstract

Low-latency remote-write networks, such as DEC's Memory Channel, provide the possibility of transparent, inexpensive, large-scale shared-memory parallel computing on clusters of shared memory multiprocessors (SMPs). The challenge is to take advantage of hardware shared memory for sharing within an SMP, and to ensure that software overhead is incurred only *when actively sharing data across SMPs in the cluster. In this paper, we describe a "two-level" software coherent shared memory system—Cashmere-2L—that meets this challenge. Cashmere-2L uses hardware to share memory within a node, while exploiting the Memory Channel's remote-write capabilities to implement "moderately lazy" release consistency with multiple concurrent writers, directories, home nodes, and page-size coherence blocks across nodes. Cashmere-2L employs a novel coherence protocol that allows a high level of asynchrony by eliminating global directory locks and the need for TLB shootdown. Remote interrupts are minimized by exploiting the remote-write capabilities of the Memory Channel network.*

Cashmere-2L currently runs on an 8-node, 32-processor DEC AlphaServer system. Speedups range from 8 to 31 on 32 processors for our benchmark suite, depending on the application's characteristics. We quantify the importance of our protocol optimizations by comparing performance to that of several alternative protocols that do not share memory in hardware within an SMP, and require more synchronization. In comparison to a one-level protocol that does not share memory in hardware within an SMP, Cashmere-2L improves performance by up to 46%.

*This work was supported in part by NSF grants CDA–9401142, CCR–9319445, CCR–9409120, CCR–9702466, CCR–9705594, and CCR–9510173; ARPA contract F19628–94–C–0057; an external research grant from Digital Equipment Corporation; and a graduate fellowship from Microsoft Research (Galen Hunt).

To appear in the Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles, Saint-Malo, France, October 1997.

1 Introduction

The shared-memory programming model provides ease-of-use for parallel applications. Unfortunately, while small-scale hardware cache-coherent "symmetric" multiprocessors (SMPs) are now widely available in the market, larger hardware-coherent machines are typically very expensive. Software techniques based on virtual memory have been used to support a shared memory programming model on a network of commodity workstations [3, 6, 12, 14, 17]. In general, however, the high latencies of traditional networks have resulted in poor performance relative to hardware shared memory for applications requiring frequent communication.

Recent technological advances are changing the equation. Low-latency remote-write networks, such as DEC's Memory Channel [11], provide the possibility of transparent and inexpensive shared memory. These networks allow processors in one node to modify the memory of another node safely from user space, with very low (microsecond) latency. Given economies of scale, a "clustered" system of small-scale SMPs on a low-latency network is becoming a highly attractive platform for large, shared-memory parallel programs, particularly in organizations that already own the hardware. SMP nodes reduce the fraction of coherence operations that must be handled in software. A low-latency network reduces the time that the program must wait for those operations to complete.

While software shared memory has been an active area of research for many years, it is only recently that protocols for clustered systems have begun to be developed [7, 10, 13, 22]. The challenge for such a system is to take advantage of hardware shared memory for sharing within an SMP, and to ensure that software overhead is incurred *only* when actively sharing data across SMPs in the cluster. This challenge is non-trivial: the straightforward "two-level" approach (arrange for each SMP node of a clustered system to play the role of a single processor in a non-clustered system) suffers from a serious problem: it requires the processors within a node to synchronize very frequently, e.g. every time one of them exchanges coherence information with another node.

Our Cashmere-2L system is designed to capitalize on both intra-node cache coherence and low-latency inter-node messages. All processors on a node share the same physical frame for a shared data page. We employ a "moderately lazy" VM-based implementation of release consistency, with multiple concurrent writers, directories, home nodes, and page-size coherence blocks. Updates by multiple writers are propagated to the home node using *diffs* [6]. Cashmere-2L exploits the capabilities of a low-latency remote-write network to apply these *outgoing* diffs without remote assistance, and to implement low-cost directories, notification queues, and application locks and barriers.

Cashmere-2L solves the problem of excess synchronization due to protocol operations within a node with a novel technique called *two-way diffing*: it uses *twins* (pristine page copies) and *diffs* (comparisons of pristine and dirty copies) not only to identify local changes that must be propagated to the home node (*outgoing* diffs), but also to identify remote changes that must be applied to local memory (*incoming* diffs). The coherence protocol is highly asynchronous: it has no global directory locks, no need for intra-node TLB shootdown[1] or related operations, and only limited need for remote interrupts of any kind: namely, to fetch pages on a miss and to initiate sharing of pages that previously appeared to be private.

We have implemented Cashmere-2L on an 8-node, 32-processor DEC AlphaServer cluster connected by a Memory Channel network. Speedups for our benchmark suite range from 8 to 31 on 32 processors. We have found that exploiting hardware coherence and memory sharing within SMP nodes can improve performance by as much as 46% at 32 processors, in comparison to a protocol designed to view each processor as a separate node. The two-level protocol is able to exploit intra-node locality to coalesce page fetch requests and reduce inter-node protocol overhead. Also, our results show that the elimination of global locks in the protocol provides performance increases up to 7%.

The use of *two-way diffing* does not provide significant performance advantages over the more common, TLB-shootdown implementation of the two-level protocol. There are three reasons for this contradiction of previous findings by other researchers [10] that have characterized TLB shootdown as a significant source of overhead for software DSM on clustered architectures. First, the nature of our protocol is such that TLB shootdown is only rarely required (i.e. in the presence of false sharing with active multiple writers, and then only among processors that have writable copies of the page). Second, even when shootdown is required it is only necessary at worst across the four processors of an SMP. Finally, shootdown is relatively inexpensive because of our polling-based messaging implementation. However, there is no performance disadvantage to using the two-way diffing mechanism, and it is likely to provide performance gains in clusters of larger SMPs, or in systems using more expensive interrupt mechanisms.

The remainder of this paper is organized as follows. We describe the Cashmere-2L protocol in Section 2, together with alternative protocols used for performance comparisons. In Section 3 we describe our experimental setting and present performance results. In the final two sections, we discuss related work and summarize our conclusions.

2 Protocol Description

We begin this section with a description of our base hardware platform, which consists of a cluster of eight 4-processor AlphaServer nodes connected by the Memory Channel. We then provide an overview of the Cashmere-2L protocol, followed by a description of the protocol data structures and memory classes, implementation details, principal operations, and alternative protocols.

2.1 Memory Channel Characteristics

Digital Equipment's Memory Channel (MC) is a low-latency remote-write network that provides applications with access to memory on a remote cluster using memory-mapped regions. Only writes to remote memory are possible—reads are not supported on the current hardware. The smallest granularity for a write is 32 bits (32 bits is the smallest grain at which the current-generation Alpha can read or write atomically). The adapter for the MC network sits on the PCI bus. A memory-mapped region can be mapped into a process' address space for transmit, receive, or both (a particular virtual address mapping can only be for transmit or 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 bypass all caches (although they are buffered in the Alpha's write buffer), and are collected by the source MC adapter, forwarded to destination MC adapters through a hub, and transferred via DMA to receive regions (physical memory) with the same global identifier. 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* through the hub has been enabled for the region. In our protocols we use loop-back only for synchronization primitives.

MC has page-level connection granularity, which is 8 Kbytes for our Alpha cluster. The current hardware supports 64K connections for a total of a 128 Mbyte MC address space. 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 29 MB/s with the limiting factor being the 32-bit AlphaServer 2100 PCI bus. MC peak aggregate bandwidth, calculated with loop-back disabled, is about 60 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. Synchronization operations must be implemented with reads and writes; there is no special hardware support. As described in Section 2.3, our implementation of global locks requires 11 μs. These locks are used almost exclusively at the application level. Within the coherence protocol, they are used only for the initial selection of home nodes.

2.2 Overview

Cashmere-2L is a *two-level* coherence protocol that extends shared memory across a group of SMP nodes connected (in our prototype) via a MC network. The protocol is designed to exploit the special features available in this type of platform. The inter-node level of the protocol significantly benefits from the low-latency, remote-write, in-order network, while the intra-node level fully leverages the available hardware coherence. The protocol is also designed to exploit the intra-node hardware coherence to reduce the demands on the inter-node level.

"Moderately" Lazy Release Consistency Implementation: Cashmere-2L implements a multiple-writer, release consistent protocol. This design decision is enabled by the requirement that

[1] When a process reduces access permissions on a page in a shared address space, it must generally interrupt the execution of any processes executing in that address space on other processors, in order to force them to flush their TLBs, and to update the page table atomically [5, 16, 20]. This interrupt-based inter-processor coordination is known as TLB shootdown.

applications adhere to the data-race-free programming model [1]. Simply stated, shared memory accesses must be protected by locks and barriers that are explicitly visible to the run-time system. The consistency model implementation lies in between TreadMarks [3] and Munin [6]. Invalidations in Munin take effect at the time of a release. Invalidations in TreadMarks take effect at the time of a causally related acquire (consistency information is communicated *only* among synchronizing processes at the time of an acquire). Invalidations in Cashmere-2L take effect at the time of the next acquire, regardless of whether it is causally related or not.

Synchronization Primitives: The synchronization primitives provided include locks, barriers, and flags. Their implementation is two-level, requiring a local `load-linked/store-conditional` (*ll/sc*) acquire, followed by reads and writes to MC space to implement a distributed lock. Consistency actions dictated by the protocol are performed in software on completion of an acquire, or prior to a release.

Page-Size Consistency Unit: The run-time library provides synchronization and fault handling routines, which serve as entry points into the protocol. Standard virtual memory (VM) faults combine with the synchronization routines to implement consistency throughout the system. Since VM faults are used to track shared accesses, the coherence granularity is naturally a VM page.

Home-Based Protocol: Shared memory pages are mapped into MC space. Each page in shared memory has a single distinguished *home node,* which contains the master copy of the page. Processors on the home node work directly on the master copy. Processors on remote nodes update the master copy according to the definition of release consistency. Modifications from remote nodes are transparently merged at the home node. The protocol is directory-based, with each shared page having an associated entry in the distributed *page directory*. This entry holds information describing the current page state. Page directory modifications are broadcast using the MC's remote-write capabilities.

Directory-Based Coherence: Cashmere-2L uses the broadcast capabilities of the MC network to maintain a distributed (replicated) directory of sharing information for each page on a "per-node" basis. A broadcast of directory modifications is performed due to the lack of remote-read capability on the Memory Channel. Not using broadcast would imply having to explicitly request data from the home of a directory entry every time it is read. The directory is examined and updated during protocol actions.

Initially, shared pages are mapped only on their associated home nodes. Page faults are used to trigger requests for an up-to-date copy of the page from the home node. Page faults triggered by write accesses are also used to keep track of data modified by each node. At the time of a write fault, the page is added to a per-processor *dirty list* (a list of all pages modified by a processor since the last release). A *twin* or a pristine copy of the page is also created. As in Munin, the twin is later used to determine local modifications.

At a release, each page in the dirty list is compared to its twin, and the differences are flushed to the home node. This operation, called a *page flush*, also applies to pages that are no longer in exclusive-mode (see Section 2.4). *Write notices* (notification of a page having been modified) for all pages in the processor's dirty list are then sent to each node in the sharing set, as indicated by the page's directory entry. Both of these operations take advantage of low-latency remote writes to communicate the data without interrupting remote process operation. The releaser then downgrades write permissions for the pages that are flushed and clears the dirty list. At a subsequent acquire, a processor invalidates all pages for which write notices have been received, and that have not already been updated by another processor on the node.

Exclusive Mode: The protocol also includes special optimizations to avoid overhead in the absence of active sharing. A node may hold a page in *exclusive* mode when no other node is concurrently accessing the page (information maintained in the global directory). The exclusive holder can act as though the page were private, eliminating all coherence overhead, including page faults, directory, and home node updates. When another node joins the sharing set, the exclusive holder is interrupted in order to update the copy at the home node. Subsequent modifications to the page will then be flushed to the home node at the next release.

Hardware Coherence Exploitation: The protocol exploits hardware coherence to maintain consistency within each node. All processors in the node share the same physical frame for a shared data page. Hardware coherence then allows protocol transactions from different processors on the same node to be coalesced, resulting in reduced data communication, as well as reduced consistency overhead. Of course, two transactions can only be coalesced if there is no intervening inter-node protocol transaction that invalidates the effects of the hardware coherence. Through novel optimizations, the protocol tracks the temporal ordering of protocol transactions and ensures that processes within a node can execute with a high level of asynchrony.

Hardware-Software Coherence Interaction: Temporal ordering of protocol operations is maintained through intra-node timestamps. These timestamps hold the value of a logical clock, which is incremented on protocol events, namely, page faults, page flushes, acquires, and releases—operations that result in communication with other nodes. This logical clock is used to timestamp a page at the time of its last update, as well as to indicate when a write notice for the page was last processed. With this information, for example, page fetch requests can safely be eliminated if the page's last update timestamp is greater than the page's last write notice timestamp.

One example of extra protocol overhead due to the two-level implementation is during a page update operation. Incoming page data cannot simply be copied into the destination page frame since the new data may overwrite the modifications performed by other concurrent writers on the node. As described in Section 2.6, a common solution to this problem has been to *shootdown* the other processors' mappings. To avoid the explicit synchronization involved in this approach, Cashmere-2L employs a novel *incoming diff* page update operation. The processor simply compares the incoming data to the existing twin (if any) and then writes the differences to the working page as well as the twin. Since applications are required to be data-race-free, these differences are exactly the modifications made on remote nodes, and will not overlap with those on the local node. Updating the twin ensures that only local modifications are flushed back to the home node at the time of the next release. This approach altogether avoids explicit synchronization within the node.

2.3 Implementation Details

Memory and data structures of Cashmere-2L are grouped into five major classes, according to their use of Memory Channel mappings. These are:

Global directory – replicated on each node, with MC receive and transmit regions (see Figure 1). No loop-back; writes are doubled manually to local copy.

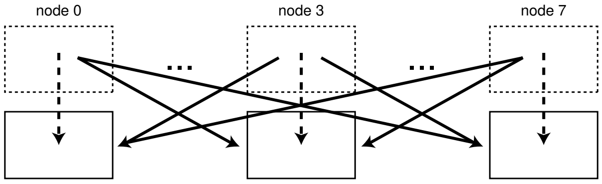

Figure 1: Memory channel mappings for global directory and synchronization objects. Transmit regions (shown with dashed lines) are mapped to I/O (PCI) space. Receive regions (shown with solid lines) are mapped to local physical memory. Directory entries are "doubled" in software—written to both the transmit and receive regions of the writer. Synchronization objects employ loop-back: writes to the local transmit region are written back by the Memory Channel to the local receive region, as shown by the dashed arrow. Loop-back allows the writer to determine that a write has been globally performed.

Synchronization objects – replicated on each node, with MC receive and transmit regions (see Figure 1). Uses loop-back to wait for writes to be globally performed (see the following section); no doubling of writes required.

Other global meta-data – receive region on each node, mapped for transmit by every other node (see Figure 2). Includes: *global write notice lists*, *page read buffers*, and *buffers for explicit requests.*

Home node page copies – receive regions on each node, mapped for transmit by every other node (see Figure 3).

Private to node– no MC mappings. Includes: *code* (replicated, read-only, to all nodes; shared by processors within a node); *stacks*, *non-shared static data*, and *per-processor dirty lists* (private to each processor); *local copies of shared pages* (shared, with various permissions, by all local processors); *page twins*, *second-level directories*, *current logical time*, and *last release time* (shared by all local processors); and *per-processor write notice lists* and *no-longer-exclusive (NLE) lists* (one per processor, writable by all local processors).

Each page is represented in the global directory by eight 32-bit words, each of which is written by only one node in the system, and indicates that node's view of the page (more words would be required if there were more than eight SMP nodes). The word for a given page on a given node contains (1) the page's loosest permissions on any processor on that node (invalid, read-only, read-write—2 bits), (2) the id of any processor accessing the page in *exclusive* mode (6 bits), and (3) the id of the home processor (and consequently the id of the home node) (6 bits). Home node indications in separate words are redundant.

Global directory information could be compressed to save space, but a global lock would then be required to ensure atomic access (32 bits is the smallest grain at which the Alpha can read or write atomically, and the Memory Channel does not support the Alpha's *ll/sc* instructions). By expanding the directory entry to one word per node, global locks are avoided and synchronization within the protocol library is dramatically reduced. Moreover, even at four bytes per node on each of eight nodes, the directory overhead for an 8K page is only 3%.

Within each node, the second-level directory maintains information on which processors have invalid, read-only, and read-write

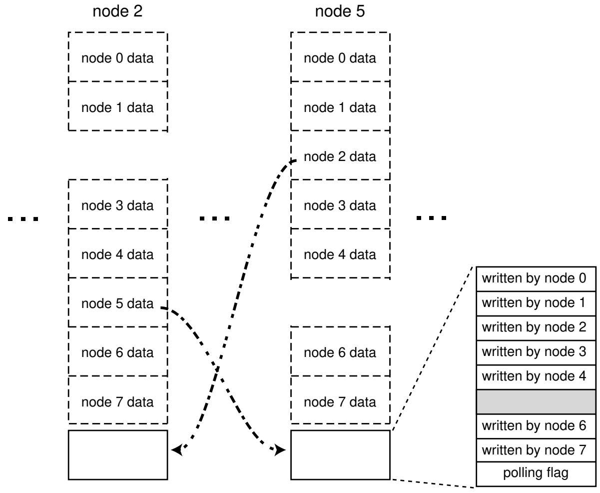

Figure 2: Memory channel mappings for coherence meta-data other than the global directory. Each node has a receive region that is written by other nodes and that it (alone) reads. With one exception (the remote request polling flag) every word of the receive region is written by a unique remote node, avoiding the need for global locks.

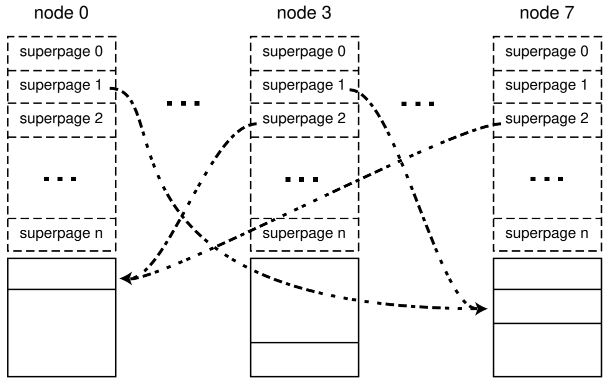

Figure 3: Memory channel mappings for home node copies of pages. Pages are grouped together into "superpages" to reduce the total number of mappings. Each superpage is assigned a home node near the beginning of execution. Every node that creates a local copy of a page in a given superpage creates a transmit mapping to the home node copy. The mapping is used whenever a local processor needs (at a release operation) to update data at the home node.

mappings. It also includes three timestamps for the page: the completion time of the last home-node *flush* operation, the completion time of the last local *update* operation, and the time the most recent *write notice* was received.

Timestamps are simply integers. Each node maintains a notion of the current time and an indication of the time of the most recent release by any local processor. The current time is incremented every time the protocol begins an acquire or release operation and applies local changes to the home node, or vice versa. The timestamps are updated atomically with *ll/sc*.

Write notice lists are implemented using a multi-bin, two-level structure, as shown in Figure 4, to avoid the need for mutually exclusive access. Each node has a seven-bin *global write notice list*

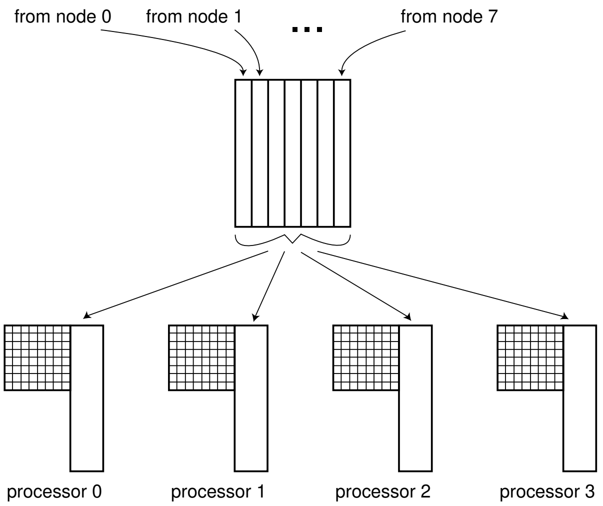

Figure 4: Write notices. A node's globally-accessible write notice list has one bin (a circular queue) for each other node in the system. On an acquire operation, a processor traverses all seven bins and distributes notices to the per-processor bins of processors with mappings for the page. Each per-processor bin consists of both a circular queue and a bitmap.

and a separate, second-level per-processor write notice list. Each bin of the global list is written by a unique remote node (again, more bins would be required on a larger system). Whenever a processor inspects the global list it distributes the notices found therein to the second-level lists of all affected processors.

Several intra-node data structures, including the per-processor write notice lists, are protected by local locks, implemented with *ll/sc*. To minimize the overhead of redundant write notices, per-processor second-level write-notice lists consist of both a bit map and a queue. When placing a notice in a list, a processor acquires a local lock, sets the appropriate bit in the bit map, adds a notice to the queue if the bit was previously clear, and releases the lock. If the write notice is already present, no action is necessary. When processing its own list, a processor acquires the lock, flushes the queue, clears the bitmap, and releases the lock.

In addition to data structures shared among the processors within a node, each individual processor maintains its own *dirty list*—the list of shared pages it has written since its last release.

Explicit inter-node requests are used for only two purposes in Cashmere-2L: to request a copy of a page from its home node, and to break a page out of exclusive-mode. Like global write notice lists, request buffers and reply buffers (for page transfers) employ a multi-bin structure to avoid the need for global locks.

Synchronization. Application locks (and protocol locks for home node selection) are represented by an 8-entry array in Memory Channel space, and by a test-and-set flag on each node. Like global directory entries, lock arrays are replicated on every node, with updates performed via broadcast. Unlike directory entries, however, locks are configured for loop-back (see Figure 1). 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 the kernel-user boundary.

Application barriers employ a two-level implementation. During each barrier, the processors inside a node synchronize through shared memory and upon full arrival the last arriving processor uses the Memory Channel to communicate the node's arrival to the rest of the nodes in the system, using an array similar to that employed for locks. Each processor within the node, as it arrives, performs page flushes for those (non-exclusive) pages for which it is the last arriving local writer. Waiting until all local processors arrive before initiating any flushes would result in unnecessary serialization. Initiating a flush of a page for which there are local writers that have not yet arrived would result in unnecessary network traffic: later-arriving writers would have to flush again.

Home node selection. Home nodes are initially assigned in a round robin manner, and then are re-assigned dynamically after program initialization to the processor that first touches a page. [15]. To relocate a page a processor must acquire a global lock and explicitly request a remapping from the initial home node. Because we only relocate once, the use of locks does not impact performance. Ordinary page operations (fetches, updates, flushes) need not acquire the lock, because they must always follow their processor's initial access in time, and the initial access will trigger the lock-acquiring home-node-selection code.

Explicit requests. 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.

We have experimented with both interrupt- and polling-based approaches to handling explicit requests. Polling provides better performance in almost every case (TSP is the only exception in our application suite —see [14] for more details on a comparison of polling versus interrupts on our platform).

Polling requires that processors check for messages frequently, and branch to a handler if one has arrived. 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 instructions at the start of all labeled basic blocks that are internal to a function and are backward referenced—i.e. at the tops of all loops. The polling instruction sequence appears in Figure 5.

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

Figure 5: Polling. Polling code is inserted at all interior, backward-referenced labels. The address of the polling flag is preserved in register $13 throughout execution.

Superpages. Due to limitations on the size of tables in the Memory Channel subsystem of the Digital Unix kernel, we were unable to place every shared page in a separate MC region for applications with large data sets. We therefore arrange for each MC region to contain 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.

Kernel changes. We made several minor changes to the Digital Unix kernel to accommodate our experiments. Specifically, we allow a user program to both specify the virtual address at which an MC region should be mapped and also to control the VM access permissions of an MC receive region. To address the very high cost of interprocessor interrupts, we also arranged to effect a user-to-kernel transition immediately upon interrupt delivery, rather than waiting for the next kernel entry (e.g. `hardclock`). This change reduced the average latency of intra-node interrupts from 980 to 80 μs and the cost of inter-node interrupts from 980 to 445 μs. However, for all but one application, the improvement was insufficient to allow interrupts to outperform polling as a means of delivering explicit requests. We report only polling-based results in Section 3. Our philosophy with respect to the kernel has been to make only those changes that are of clear benefit for many types of applications. We could improve performance significantly by moving protocol operations (the page fault handlers in particular) into the kernel, but at a significant loss in modularity, and of portability to other sites or other variants of Unix.

2.4 Principal Consistency Operations

2.4.1 Page Faults

In response to a page fault, a processor first modifies the page's second-level directory entry on the local node to reflect the new access permissions. If no other local processor has the same permissions, the global directory entry is modified as well. If no local copy for the page exists, or if the local copy's update timestamp precedes its write notice timestamp or the processor's acquire timestamp (whichever is earlier), then the processor fetches a new copy from the home node.

In addition to the above, write faults also require that the processor check to see if any other node is currently sharing the page. If there are any other sharers, the processor adds the page to its (private) per-processor dirty list and possibly creates a twin of the page (see Section 2.5); otherwise it shifts the page into *exclusive* mode (see the following section). Finally, for either a read or a write fault, the processor performs an `mprotect` call to establish appropriate permissions, and returns from the page fault handler.

Exclusive Mode: If the global directory reveals that a desired page is currently held exclusively, then the faulting processor must send an explicit request to some processor on the holder node. Upon receipt of such a request, the holder flushes the entire page to the home node, removes the page from exclusive-mode, and then returns the latest copy of the page to the requestor. All subsequent page fetches will be satisfied from the home node, unless the page re-enters exclusive-mode.

If any other processors on the node have write mappings for the page, the responding processor creates a twin and places notices in the *no-longer-exclusive* list of each such processor, where they will be found at subsequent release operations (see below). After returning the copy of the page, the responding processor downgrades its page permissions in order to catch any future writes. A page in exclusive-mode incurs no coherence protocol overhead. It has no twin; it never appears in a dirty list; it generates no write notices or flushes at releases.

2.4.2 Acquires

Consistency actions at a lock acquire operation (or the departure phase of a barrier) begin by traversing the bins of the node's global write notice list and distributing the notices therein to the affected local processors (i.e. those with mappings). Distributing the notices to the local lists of the affected processors eliminates redundant work and book-keeping in traversals of the global write notice bins. As the write notices are distributed, the most recent "write-notice" timestamp for the page is updated with the current node timestamp.

After distributing write notices, an acquiring processor processes each write notice in its per-processor list (note that this list may hold entries previously enqueued by other processors, as well as those just moved from the global list). If the update timestamp precedes the write notice timestamp for the page associated with the write notice, the page is invalidated using an `mprotect`. Otherwise, no action is taken.

2.4.3 Releases

During a release operation, a processor must flush all dirty, non-exclusive pages to the home node. (If a processor is on the home node, then the flush can be skipped.) These pages are found in the processor's (private) dirty list and in its no-longer-exclusive list, which is written by other local processors. In addition, the releasing processor must send write notices to all other nodes, excluding the home node, that have copies of the dirty page. Since modifications are flushed directly to the home node, the home node does not require explicit notification.

If the release operations of two different processors on the same node overlap in time, it is sufficient for only one of them to flush the changes to a given page. As it considers each dirty page in turn, a releasing processor compares the page's flush timestamp (the time at which the most recent flush began) with the node's last release time (the time at which the protocol operations for the most recent release began). It skips the flush and the sending of write notices if the latter precedes the former (though it must wait for any active flush to complete before returning to the application). After processing each dirty page, the protocol downgrades the page permissions so that future modifications will be trapped.

2.5 Twin Maintenance

Cashmere-2L uses twins to identify page modifications. The twin always contains the node's latest view of the home node's master copy. This view includes all the local modifications that have been made globally available and all the global modifications that the node has witnessed. Twins are unnecessary on the home node.

A twin must be maintained whenever at least one local processor, not on the home node, has write permissions for the page, and the page is not in exclusive-mode. The twins are used to isolate local and global modifications, i.e. to perform "outgoing" and "incoming" diffs. The latter operation is used in place of TLB-shootdown to ensure data consistency, while avoiding inter-processor synchronization.

Special care must also be exercised to ensure that flush operations do not introduce possible inconsistencies with the twin. Since multiple concurrent writers inside a node are allowed, the twin must be updated not only during page-fault-triggered updates to the local copy, but also during flushes to the home node at release time. A *flush-update* operation writes all local modifications to both the home node and the twin. Subsequent release operations within the node will then realize that the local modifications have already been flushed, and will avoid overwriting more recent changes to the home by other nodes.

2.5.1 Prior Work

In earlier work on a one-level protocol [14], we used write-through to the home node to propagate local changes on the fly. On the current Memory Channel, which has only modest cross-sectional bandwidth, the results in Section 3 indicate that twins and diffs perform better. More important, for the purposes of our two-level protocol, twins allow us to identify remote updates to a page and eliminate TLB shootdown, something that write-through does not. We briefly considered a protocol that would use write-through to collect local changes, and that would compare home and local copies to identify remote changes. Such a protocol would eliminate the need for twins, but would require that we use explicit intra-node messages to synchronize with other processors on the node in order to ensure that any writes in transit are flushed to the home node and to stall other local writers during a page update.

2.6 Alternative Protocols

Shootdown In order to quantify the value of two-way diffing, we have also implemented a shootdown protocol called Cashmere-2LS. As described earlier, the shootdown protocol avoids races between a processor incurring a page fault and concurrent local writers by shooting down all other write mappings on the node, flushing outstanding changes, and then creating a new twin only on a subsequent write page fault. It behaves in a similar fashion at releases: to avoid races with concurrent writers, it shoots down all write mappings, flushes outstanding changes, and throws away the twin. In all other respects, the shootdown protocol is the same as Cashmere-2L.

It should be noted that this shootdown protocol is significantly less "synchronous" than single-writer alternatives [10, 22]. It allows writable copies of a page to exist on multiple nodes concurrently. Single-writer protocols must shoot down write permission on all processors of all other nodes when one processor takes a write fault.

One-level protocols We also present results for two one-level protocols. The first of these (Cashmere-1L) is described in more detail in a previous paper [14]. In addition to treating each processor as a separate node, it "doubles" its writes to shared memory on the fly using extra in-line instructions. Each write is sent both to the local copy of the page and to the home node copy.

Our second one-level protocol (Cashmere-1LD) abandons write-through in favor of twins and (outgoing) diffs. Both protocols share many of the characteristics of Cashmere-2L. While simpler, they do not explicitly exploit intra-node hardware coherence. Each directory entry is again organized as eight 32-bit words, one for each node in the cluster. Each word contains presence flags for the processors within the node (4 bits), the id of the home node (5 bits), an indication of 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 any processor within a node has exclusive read/write permission for the page (4 bits). Write notice lists are on a per processor rather than per node basis and are protected by cluster-wide locks.

Read faults always fetch a page from the home node, and set the presence bit on the word corresponding to the faulting processor's node. Write faults fetch the page if necessary, create a twin (in the case of Cashmere-1LD), and insert a descriptor of the page in a local dirty list. If the page is found in exclusive-mode, then the holder is contacted and asked to flush the page to the home node before the fetch operation can proceed.

At a release operation, a processor traverses its dirty list. For each dirty page it sends write notices to other sharing processors which do not already hold the necessary write notices. In the case of Cashmere-1LD, the home node is also updated based on a comparison of the local copy and the twin. If no sharers are found, the page moves to exclusive mode: it will not participate in coherence transactions until it leaves this mode. At an acquire operation, the processor traverses its write notice list and removes itself from the sharing set of each page found therein.

One-level protocols with home-node optimization We also have prepared a special version of both one-level protocols that takes greater advantage of our prototype platform, in particular the SMP nodes. By allowing home-node accesses to occur directly on the page's master copy (the MC receive region), the home-node is able to avoid twin operations and page invalidations. Also since each home-node processor now shares the same physical page frame, the node will benefit from the underlying hardware coherence. Since pages on remote nodes do not benefit from hardware coherence, this is essentially an intermediate implementation that resides somewhere between a strict one-level and a full two-level protocol. The results are presented only to show that a full two-level protocol is necessary to obtain robust performance.

3 Experimental Results

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. The 21064A's primary data cache size is 16 Kbytes, and the secondary cache size is 1 Mbyte. A cache line is 64 bytes. Each AlphaServer runs Digital UNIX 4.0D with TruCluster v. 1.5 (Memory Channel) extensions. The systems execute in multi-user mode, but with the exception of normal Unix daemons no other processes were active during the tests. In order to increase cache efficiency, application processes are pinned to a processor at startup. No other processors are connected to the Memory Channel. Execution times were calculated based on the best of three runs.

3.1 Basic Operation Costs

Memory protection operations on the AlphaServers cost about 55 μs. Page faults on already-resident pages cost 72 μs. The overhead for

Operation	Time (μs)	
	2L/2LS	1LD/1L
Lock Acquire	19	11
Barrier	58 (321)	41 (364)
Page Transfer (Local)	-	467
Page Transfer (Remote)	824	777

Table 1: Costs of basic operations for the two-level protocols (2L/2LS) and the one-level protocols (1LD/1L).

Program	Problem Size	Time (sec.)
SOR	3072x4096 (50Mbytes)	195.0
LU	2046x2046 (33Mbytes)	254.8
Water	4096 mols. (4Mbytes)	1847.6
TSP	17 cities (1Mbyte)	4029.0
Gauss	2046x2046 (33Mbytes)	953.7
Ilink	CLP (15Mbytes)	899.0
Em3d	60106 nodes (49Mbytes)	161.4
Barnes	128K bodies (26Mbytes)	469.4

Table 2: Data set sizes and sequential execution time of applications.

polling ranges between 0% and 36% compared to a single processor execution, 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 is 199 μs. The cost of outgoing diff creation varies according to the diff size. If the home node is local (only applicable to the one-level protocols), the cost ranges from 340 to 561 μs. If the home node is remote, the diff is written to uncacheable I/O space and the cost ranges from only 290 to 363 μs per page. An incoming diff operation applies changes to both the twin and the page and therefore incurs additional cost above the outgoing diff. Incoming diff operations range from 533 to 541 μs, again 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. All times are for interactions between two processors. The barrier times in parentheses are for a 32 processor barrier. The lock acquire time increases slightly in the two-level protocols because of the two-level implementation, while barrier time decreases.

3.2 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 [21] 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 processor, which performs all computation on it.

Water: a molecular dynamics simulation from the SPLASH-1 [19] 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 occurs during a phase that updates intermolecular forces using locks, 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. For load balance, the rows are distributed among processors cyclically, with each row computed on by a single processor. 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 in [9]. The main shared data is a pool of sparse arrays of genotype probabilities. For load balance, non-zero elements are assigned to processors in a round-robin fashion. The computation is master-slave, with one-to-all and all-to-one data communication. 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 [19] suite, using the hierarchical Barnes-Hut Method. The major shared data structures are two arrays, one representing the bodies and the other representing the cells, a collection of bodies in close physical proximity. 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 [8]. 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. 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. Barriers are used for synchronization.

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 uninstrumented application sequentially without linking it to the protocol library.

3.3 Comparative Speedups

Figure 7 presents speedup bar charts for our applications on up to 32 processors. The unshaded extensions to the one-level bars indicate the results of the one-level protocols with home-node optimization. All calculations are with respect to the sequential times in Table 2. The configurations we use are as follows: 4:1—four processors with 1 process in each of 4 nodes; 4:4—four processors with 4 processes in 1 node; 8:1—eight processors with 1 process in each of 8 nodes; 8:2—eight processors with 2 processes in each of 4 nodes; 8:4—eight processors with 4 processes in each of 2 nodes; 16:2—sixteen processors with 2 processes in each of 8 nodes; 16:4—sixteen processors with 4 processes in each of 4 nodes; 24:3—

twenty four processors with 3 processes in each of 8 nodes; and 32:4—thirty two processors with 4 processes in each of 8 nodes.

Table 3 presents detailed statistics on the communication incurred by each of the applications under each of the four protocols at 32 processors. The statistics presented are the execution time, the number of lock and barrier synchronization operations, the number of read and write faults, the number of page transfers, the number of directory updates, the number of write notices, the number of transitions in to and out of exclusive-mode, and the amount of data transferred. The Cashmere-2L and Cashmere-2LS entries also include statistics concerning twin maintenance. The number of twin creations is listed for both protocols. In the presence of multiple writers, Cashmere-2L maintains the twins via incoming diffs and flush-update operations, while Cashmere-2LS employs shootdown operations. Statistics covering these operations are also included in the table. All statistics, except for execution time, are aggregated over all 32 processors.

Figure 6 presents a breakdown of execution time at 32 processors for each application. The breakdown is normalized with respect to total execution time for Cashmere-2L. The components shown represent time spent executing user code (`User`), time spent in protocol code (`Protocol`), the overhead of polling in loops (`Polling`), communication and wait time (`Comm & Wait`), and the overhead of "doubling" writes for on-the-fly write-through to home node copies (`Write Doubling`; incurred by `1L` only). In addition to the execution time of user code, `User` time also includes cache misses and time needed to enter protocol code, i.e. kernel overhead on traps and function call overhead from a successful message poll. Two of the components—`Protocol` and `Comm. & Wait`—were measured directly on the maximum number of processors. `Polling` overhead could not be measured directly: it is extrapolated from the single-processor case (we assume that the ratios of user and polling time remain the same as on a single processor). `Write doubling` overhead in the case of `1L` is also extrapolated by using the minimum `User` and `Polling` times obtained for the other protocols.

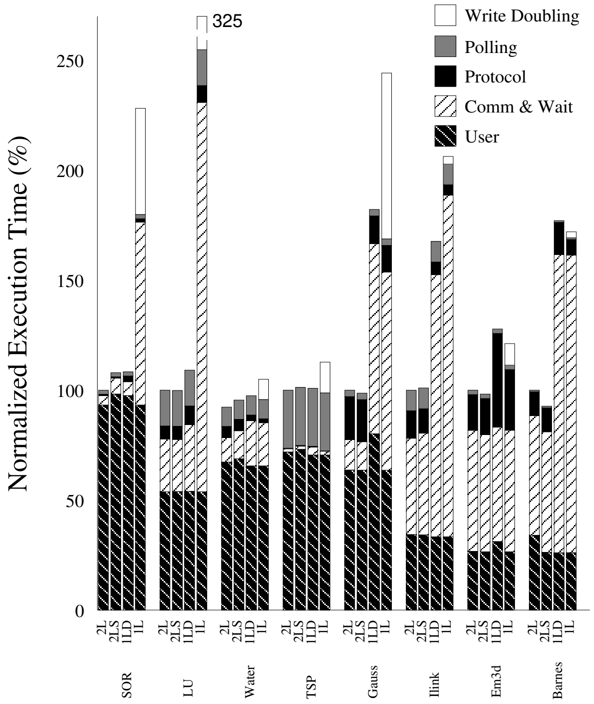

Figure 6: Breakdown of percent normalized execution time for the Two-Level (2L), Two-Level-Shootdown (2LS), One-Level-Diffing (1LD), and One-Level-Write-Doubling (1L) protocols at 32 processors. The components shown represent time spent executing user code (`User`), time spent in protocol code (`Protocol`), the overhead of polling in loops (`Polling`), communication and wait time (`Comm & Wait`), and the overhead of "doubling" writes for on-the-fly write-through to home node copies (`Write Doubling`; incurred by `1L` only).

3.3.1 Comparison of 1LD to 1L

The main difference between the two one-level protocols is the mechanism used to merge changes into home-node copies of pages. `1L` writes changes through to the home node on the fly, by using an extra compilation pass to "double" writes in-line. `1LD` uses twins and (outgoing) diffs.

Write doubling incurs additional computational cost for each write to shared data. On the home node, write doubling also has a large cache penalty in the case of Gauss and LU, since the working set size is increased beyond the size of the first-level cache. (On the home node, writes to the local copy and to the home node copy are both to physical memory. On a remote node, the write to the home node copy goes to I/O space which bypasses all caches.) See [14] for more details.

In addition, for all applications, write-through on the Memory Channel often incurs the full latency and overhead of Memory Channel access for each word written (in addition to contention). Although writes are coalesced in the processor's write buffer before being sent out onto the Memory Channel, the benefits of coalescing writes to reduce overhead on the Memory Channel is not fully exploited. In the case of `1LD`, which uses *diffing*, writes are contiguous and have a greater chance of being coalesced. Hence, `1LD` shows better performance for most applications.

We present the results of `1L` mainly as a base point for comparison to our earlier paper describing the one-level protocol with write-through [14]. (Note that these latest results have been executed on a new version of the Memory Channel which offers slightly higher bandwidth.) We will use `1LD` (without the home-node optimization) as the basis for comparison of the two-level protocols in the rest of this section since it has the best over-all performance of the true one-level protocols, and is similar to the two-level protocols described in this paper.

3.3.2 Comparison of One- and Two-Level Protocols

Compared to the base `1LD` protocol, the two-level protocols show slight improvement for LU, SOR, TSP, and Water (1–9%), good improvement for Em3d (22%), and significant improvement for Ilink, Gauss, and Barnes (40–46%). With the home-node optimization, both the one-level protocols improve dramatically for Em3d, but the two-level protocols still hold their performance edge for the applications overall.

SOR benefits mainly from the ability of `2L` and `2LS` to use hardware coherence within nodes to exploit application locality. This locality results in a major reduction in the number/amount of page faults, page transfers, write notices, directory updates, and communicated data (see Table 3); and a reduction in time spent in the protocol code. Given the high computation-to-communication ratio

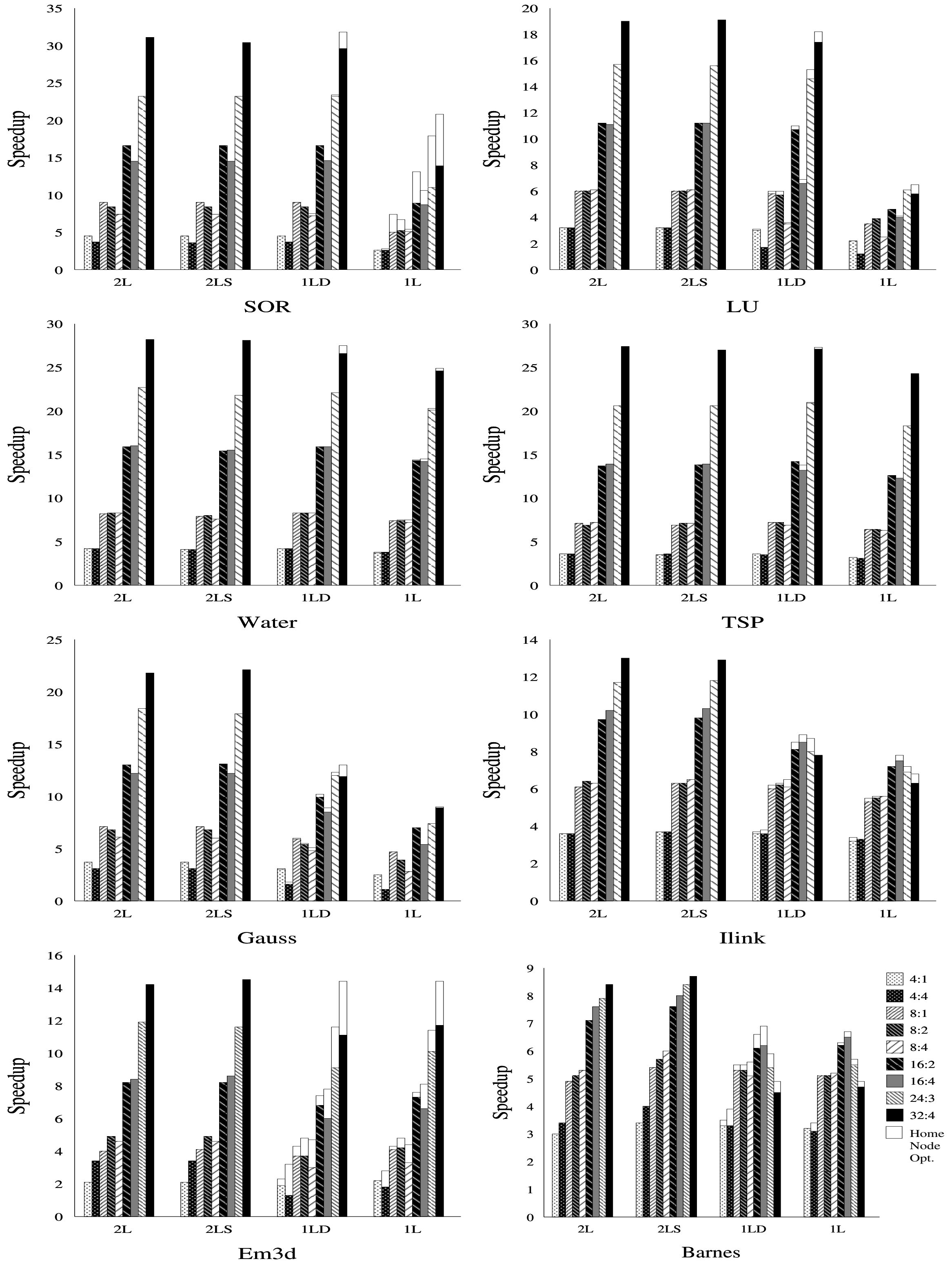

Figure 7: Speedups for Two-Level (2L), Two-Level-Shootdown (2LS), One-Level-Diffing (1LD), and One-Level-Write-Doubling (1L).

Application		SOR	LU	Water	TSP	Gauss	Ilink	Em3d	Barnes
2L	Exec. time (secs)	6.3	13.4	65.8	147.1	44.0	69.5	14.2	60.2
	Lock/Flag Acquires (K)	0	0	3.68	2.61	129.82	0	0	0
	Barriers	48	130	36	2	7	521	250	11
	Read Faults (K)	0.34	22.14	72.55	14.32	174.51	204.28	44.49	214.45
	Write Faults (K)	0.67	4.81	35.54	9.08	8.18	46.16	50.97	183.05
	Page Transfers (K)	0.34	8.76	30.35	12.41	42.51	51.31	38.43	68.61
	Directory Updates (K)	2.02	18.74	111.91	23.01	70.12	144.66	181.69	261.11
	Write Notices (K)	0.34	0.37	75.88	53.25	23.03	105.34	38.12	269.76
	Excl. Mode Transitions (K)	0	4.42	2.03	0.03	4.14	4.13	2.92	3.49
	Data (Mbytes)	4.25	116.56	277.83	103.23	385.31	479.90	345.92	616.75
	Twin Creations (K)	0.34	0.18	25.56	7.91	3.59	10.56	8.84	54.80
	Incoming Diffs	0	0	1	0	23	0	0	0
	Flush-Updates	0	0	163	0	0	0	0	0
2LS	Exec. time (secs)	6.4	13.4	68.1	148.9	43.3	70.2	13.9	55.7
	Lock/Flag Acquires (K)	0	0	3.68	2.61	129.82	0	0	0
	Barriers	48	130	36	2	7	521	250	11
	Read Faults (K)	0.34	22.14	73.10	14.35	173.04	199.19	44.77	214.42
	Write Faults (K)	0.67	4.81	35.53	9.06	8.18	46.43	50.85	183.50
	Page Transfers (K)	0.34	8.76	31.49	12.46	42.01	50.06	38.70	66.88
	Directory Updates (K)	2.02	18.73	109.32	22.96	70.70	142.40	181.71	261.78
	Write Notices (K)	0.34	0.37	81.83	53.13	23.08	103.44	38.39	261.39
	Excl. Mode Transitions (K)	0	4.42	2.04	0.03	3.12	4.24	2.55	3.58
	Data (Mbytes)	4.25	116.88	287.32	103.99	383.10	481.03	346.86	616.40
	Twin Creations (K)	0.34	0.98	25.61	7.87	3.59	12.54	8.84	54.79
	Shootdowns	0	0	161	3	0	0	0	0
1LD	Exec. time (secs)	6.6	14.7	69.5	148.4	80.1	116.6	18.2	103.43
	Lock/Flag Acquires (K)	0	0	3.68	2.57	129.82	0	0	0
	Barriers	48	130	36	2	7	521	250	11
	Read Faults (K)	2.98	22.83	84.48	19.81	205.95	228.53	195.87	241.65
	Write Faults (K)	2.98	7.70	35.61	8.78	8.18	47.71	95.88	181.13
	Page Transfers (K)	2.98	22.83	84.48	23.18	205.95	229.94	195.88	241.66
	Directory Updates (K)	5.95	25.19	163.17	45.59	312.94	459.19	386.84	483.33
	Write Notices (K)	2.98	2.36	78.69	22.40	106.99	229.26	190.97	241.67
	Data (Mbytes)	24.83	254.69	696.42	190.58	1779.95	1894.22	1612.47	1983.61
1L	Exec. time (secs)	14.0	43.8	75.0	166.0	107.5	143.59	17.3	103.4
	Lock/Flag Acquires (K)	0	0	3.68	2.65	129.82	0	0	0
	Barriers	48	130	36	2	7	521	250	11
	Read faults (K)	2.98	22.83	84.17	20.62	207.11	228.53	195.88	240.00
	Write faults (K)	2.98	7.71	35.61	9.15	8.18	47.71	95.88	181.13
	Page transfers (K)	2.98	22.83	84.18	24.01	207.11	229.94	195.87	240.99
	Directory Updates (K)	5.95	25.19	162.56	47.20	315.26	459.19	386.84	480.03

Table 3: Detailed statistics for the Two-Level (2L), Two-Level-Shootdown (2LS), One-Level-Diffing (1LD), and One-Level-Write-Doubling (1L) protocols at 32 processors.

for this application, however, these benefits do not translate into significant performance improvements. Em3d also benefits mainly from exploiting locality of access and avoiding intra-node messaging. Since the computation-to-communication ratio is much lower than for SOR, we see more gain in performance. The reduction in the number of page faults, page transfers, and total data transferred is similar to that in SOR, as demonstrated in Table 3. Due to the high frequency of page accesses by processors within the page's home node, each of these applications can benefit greatly from available hardware coherence within the home node. Hence, the one-level protocols with the home-node optimization are able to obtain speedups comparable to the full two-level protocols.

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

Gauss shows significant benefits from coalescing remote requests, since the access pattern for the shared data is essentially single producer/multiple consumer (ideally implemented with broadcast). There is a four-fold reduction in the amount of data transferred, but little reduction in the number of read and write page faults. These gains come at the expense of a slightly larger protocol overhead due to increased maintenance costs of protocol meta-data (Figure 6).

However, there is a 45% improvement in execution time, a result of the reduction in communication and wait time. Ilink's communication pattern is all-to-one in one phase, and one-to-all in the other (a master-slave style computation). Hence, its behavior is similar to that of Gauss, with a 40% performance improvement in performance. Barnes shows a 46% improvement in performance. The benefits in this case come from the ability to coalesce page fetch requests, which significantly reduces the amount of data transferred. Since the computation-to-communication ratio is low for this application, this reduction has a marked effect on performance.

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

3.3.3 Effect of Clustering

The performance of SOR and Gauss decreases for all protocols when the number of processors per node (degree of clustering) is increased, while keeping the number of processors fixed (see Figure 7). Both applications are matrix-based, with only a small amount of computation per element of the matrix. Their data set sizes do not fit in the second level caches and hence a large amount of traffic is generated between the caches and main memory due to capacity misses. Increasing the number of processors per node increases the traffic on the shared node bus, thereby reducing performance. These results are corroborated by the fact that the performance of the 4:1 configuration is better than using pure hardware shared memory within the 4 processors of an SMP. Running the application on an SMP without linking it to any protocol code takes 51.8 seconds, a 21% drop in performance from the 4:1 1LD protocol case. Cox *et al.* [7] report similar results in their comparison of hardware and software shared memory systems.

LU exhibits negative clustering effects only for the one-level protocols. The 4:4, 8:4, and 16:4 configurations experience a significant performance drop, which is not present in either of the two-level protocols. This can be attributed to LU's bursty nature of page accesses. As a pivot row is factored, the enclosed pages are only accessed by their owner and are therefore held in exclusive-mode. After factoring is complete, a number of processors immediately access the pages in the row, thus generating explicit requests for each page to leave exclusive-mode. As the number of processors per node increases, an increasing number of the requests are sent to the same node, thereby creating a communication bottleneck. The two-level protocols alleviate this bottleneck by exploiting the available hardware coherence.

Em3d and Barnes show a performance improvement when the number of processors per node is increased in the two-level protocols. The one level protocols do not show similar performance gains. These applications have low computation-to-communication ratios, allowing the reduction in inter-node traffic due to the use of intra-node sharing in the two-level protocols to yield benefits even at 8 and 16 processor totals. At 4 processors, the two-level protocols also benefit from sharing memory in hardware thus avoiding software overheads and extra traffic on the bus. These results only *appear* to contradict those in [4, 7, 10], which report that bandwidth plays a major role in the performance of clustered systems. Our results compare one and two-level protocols on the *same* clustered hardware, as opposed to two-level protocols on *clustered* hardware versus one-level protocols on *non-clustered* hardware (with consequently higher network bandwidth per processor). In addition our hardware platform has a serial global interconnect (MC is a bus) that favors protocols that reduce the number of inter-node transactions.

The performance of Water, TSP, and Ilink is unaffected by the number of processors per node at 8 and 16 processor totals, regardless of the protocol.

The reduction in inter-node communication due to the use of hardware coherence in the two-level protocols improves performance significantly at 24 and 32 processors for Gauss, Ilink, Em3d, and Barnes. SOR, LU, TSP, and Water show only slight overall benefits from exploiting hardware coherence, due to their high computation-to-communication ratios.

Overall the extra synchronization and data structures in the two-level protocols have little effect on performance. This observation is supported by the similar performance of the 8:1 configuration for all of the (diff-based) protocols, for all of the applications studied.

3.3.4 TLB Shootdown Versus Two-Way Diffing

The use of two-way diffing in place of TLB shootdown has little effect on the performance of the two-level protocol for most of our applications. This result stands in sharp contrast to results reported by others (e.g. SoftFLASH [10]), and is explained by our use of a multi-writer protocol and our implementation of the shootdown mechanism.

Shootdown happens in SoftFLASH mainly when a page is "stolen" by a remote processor, and all local mappings must be eliminated. In Cashmere-2L, pages are never stolen. Shootdown (or two-way diffing) is only necessary when there is more than one local processor writing a page at a release or at a page fault. This situation occurs only in the presence of false sharing, in lock-based applications. It does not occur with barriers, because only the last local processor to arrive at the barrier flushes changes to the home node, and because the timestamps on page updates allow processors to realize that the first of them to experience a page fault after the barrier is retrieving a copy that will suffice for all. The Cashmere-2L second-level directories indicate the node processors that have read-only and read-write mappings for a page, so a shootdown operation can be efficiently limited to the set of processors concurrently holding read-write mappings for the page. In contrast, SoftFLASH must conservatively shootdown all node processors that have at any time hosted an application process.

Additionally, the Cashmere-2L polling-based messaging layer enables an extremely efficient shootdown mechanism. With polling, the cost of shooting down one processor is only 72 μs. If intra-node interrupts are used in place of message polling, then the cost to shootdown one processor rises to 142 μs. In the case of Water, a lock-based application with false sharing, `2LS` matches the performance of `2L` when the shootdown mechanism is implemented with polling-based messaging. If instead the shootdown mechanism is implemented with intra-node interrupts, the `2LS` execution time increases by 6%. Note that our kernel modifications have already decreased the latency of intra-node interrupts by almost an order of magnitude, so even our interrupt-based shootdown mechanism is highly optimized. Other environments will most likely experience a more significant performance degradation when using an interrupt-based shootdown mechanism.

3.3.5 Impact of Lock-Free Structures

To quantify the impact of the lock-free protocol structures, the `2L` protocol was modified to use global locks to control access to the global directory entries and the remote write notice lists. By using global locks, the directory entries can safely be represented in one 32-bit word, and the write notices can be stored as a single list on each node. In addition to saving space, processing time is slightly reduced since both structures are compressed into a smaller area. Of course, protocol asynchrony is decreased.

Our benchmark suite contains a wide range of applications in terms of the number of directory accesses and write notices produced. (See Table 3.) Barnes has by far the most directory accesses and write notices, and, not surprisingly, the application benefits from the lock-free structures. Execution time improves by 5% when lock-free structures are used instead of structures protected by global locks. Em3d, Ilink, and Water have a moderate amount of directory accesses and write notices. Both Em3d (5%) and Ilink (7%) also show improvements by using lock-free structures. Water, however, has similar performance for the lock-free and lock-based implementations of protocol data structures. The remaining applications have relatively few accesses to these structures and show no significant differences between the lock-free or lock-based approaches.

4 Related Work

Cox *et al.* were among the first to study layered hardware/software coherence protocols [7]. They simulate a system of 8-way, bus-based multiprocessors connected by an ATM network, using a protocol derived from TreadMarks [3], and show that for clustering to provide significant benefits, reduction in inter-node messages and bandwidth requirements must be proportional to the degree of clustering. Karlsson and Stenstrom [13] examined a similar system in simulation and found that the limiting factor in performance was the latency rather than the bandwidth of the message-level interconnect. Bilas *et al.* [4] present a simulation study of the automatic update release consistent (AURC) protocol on SMP nodes. They find that the write-through traffic of AURC, coupled with the fact that processors within an SMP have to share the same path to the top-level network, can result in TreadMarks-style lazy release consistency performing better than AURC with clustering.

Yeung *et al.* [22], in their MGS system, were the first to implement a layered coherence protocol. They use a Munin-like multi-writer protocol to maintain coherence across nodes. The protocol also includes an optimization that avoids unnecessary diff computation in the case of a single writer. The MGS results indicate that performance improves with larger numbers of processors per node in the cluster. MGS is implemented on the Alewife [2] hardware shared memory multiprocessor with a mesh interconnect. In contrast, commercial multiprocessors currently provide a single connection (usually through the I/O bus) that all processors within a node have to share, thus limiting the benefits of larger nodes.

SoftFLASH [10] is a kernel-level implementation of a two-level coherence protocol, on a cluster of SMPs. The protocol is based on the hardware coherent FLASH protocol. Shared data is tracked via TLB faults, and thus TLB shootdown with costly inter-processor interrupts is required to avoid consistency problems. This cost is further compounded by the use of shared page tables. At shootdown operations, all processors that may have run an application process must conservatively be interrupted in order to invalidate their TLB entry. The use of a single-writer protocol also results in potentially more frequent invalidations (and shootdowns). The SoftFLASH results indicate that any beneficial effects of clustering are offset by increased communication costs. This can be explained largely by the heavy shootdown overhead, especially for larger sized nodes.

Finally, the Shasta system [18], which runs on hardware similar to our system, implements an eager release consistent, variable-granularity, fine-grained, single-writer protocol for coherence across nodes. Unlike Cashmere-2L, Shasta does not use virtual memory to generate access faults. Instead it uses in-line checks on every shared memory reference to manage the state table entry for each coherence object. Novel techniques are used to minimize the overhead of the in-line checks. Moving to a clustered two-level protocol increases the complexity of the in-line checks and requires internal locks in some protocol operations, which are more frequent than in the VM-based systems. The relative benefits of variable-granularity single-writer coherence with in-line checks is still to be determined.

5 Conclusion

We have presented a two-level software coherent shared memory system—Cashmere-2L—that exploits hardware cache coherence within SMP nodes and takes advantage of a low-latency remote-write network. The protocol implements "moderately lazy" release consistency, with multiple concurrent writers, directories, home nodes, and page-size coherence blocks. Software is invoked only when there is sharing across SMPs. A novel coherence protocol based on *two-way diffing* is used to reduce intra-node synchronization overhead and avoid TLB shootdown. Global protocol locks are eliminated to avoid inter-node protocol synchronization overhead.

Results indicate that Cashmere-2L outperforms its one-level counterpart (which treats each processor as a separate node), with performance gains sometimes as high as 46%. Cashmere-2L is able to exploit locality of access within a node, and to coalesce multiple off-node requests for data. The extra complexity of the two-level protocol does not add significant protocol overhead. Elimination of global protocol locks results in up to a 7% improvement over a lock-based protocol.

Cashmere-2L does not exhibit any significant performance advantage over its shootdown-based counterpart. Careful investigation reveals that shootdown is unnecessary in most situations, due to the use of a multi-writer protocol (which eliminates page "stealing") and to the generally small amounts of false sharing in our applications (which makes it uncommon that there are multiple local writers at the time of a release or page fault). Shootdown is also relatively inexpensive on our system, since we use polling to detect shootdown requests. As a result, the added asynchrony of two-way diffing provides only a small performance advantage, which is at least partially offset by its somewhat higher computation cost. Performance gains could be higher with larger degrees of clustering, or high interrupt costs.

An important open question is the extent to which global directories (as opposed to TreadMarks-style tracking of the "happens-before" relationship) help or hurt performance. Another issue is the relative benefits of VM-based multiple-writer protocols versus variable-granularity single-writer protocols, such as Shasta, in this environment. We are currently exploring these alternatives in the realm of two-level protocols.

References

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

[2] A. Agarwal, R. Bianchini, D. Chaiken, K. Johnson, D. Kranz, J. Kubiatowicz, B. Lim, K. Mackenzie, and D. Yeung. The MIT Alewife Machine: Architecture and Performance. In *Proceedings of the Twenty-Second International Symposium on Computer Architecture*, pages 2–13, Santa Margherita Ligure, Italy, June 1995.

[3] 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, February 1996.

[4] A. Bilas, L. Iftode, D. Martin, and J. P. Singh. Shared Virtual Memory Across SMP Nodes Using Automatic Update: Protocols and Performance. Technical Report TR-517-96, Department of Computer Science, Princeton University, October 1996.

[5] D. L. Black, R. F. Rashid, D. B. Golub, C. R. Hill, and R. V. Baron. Translation Lookaside Buffer Consistency: A Software Approach. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 113–122, Boston, MA, April 1989.

[6] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and Performance of Munin. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, pages 152–164, Pacific Grove, CA, October 1991.

[7] A. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, and W. Zwaenepoel. Software Versus Hardware Shared-Memory Implementation: a Case Study. In *Proceedings of the Twenty-First International Symposium on Computer Architecture*, pages 106–117, Chicago, IL, April 1994.

[8] 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.

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

[10] A. Erlichson, N. Nuckolls, G. Chesson, and J. Hennessy. SoftFLASH: Analyzing the Performance of Clustered Distributed Virtual Shared Memory. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 210–220, Boston, MA, October 1996.

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

[12] 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*, pages 14–25, San Jose, CA, February 1996.

[13] M. Karlsson and P. Stenstrom. Performance Evaluation of a Cluster-Based Multiprocessor Built from ATM Switches and Bus-Based Multiprocessor Servers. In *Proceedings of the Second International Symposium on High Performance Computer Architecture*, pages 4–13, San Jose, CA, February 1996.

[14] L. Kontothanassis, G. Hunt, R. Stets, N. Hardavellas, M. Cierniak, S. Parthasarathy, W. Meira, S. Dwarkadas, and M. L. Scott. VM-Based Shared Memory on Low-Latency, Remote-Memory-Access Networks. In *Proceedings of the Twenty-Fourth International Symposium on Computer Architecture*, pages 157–169, Denver, CO, June 1997.

[15] 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.

[16] B. Rosenburg. Low-Synchronization Translation Lookaside Buffer Consistency in Large-Scale Shared-Memory Multiprocessors. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, pages 137–146, Litchfield Park, AZ, December 1989.

[17] 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*, pages 174–185, Boston, MA, October 1996.

[18] D. J. Scales, K. Gharachorloo, and A. Aggarwal. Fine-Grain Software Distributed Shared Memory on SMP Clusters. WRL Research Report 97/3, DEC Western Research Laboratory, February 1997.

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

[20] P. J. Teller. Translation-Lookaside Buffer Consistency. *Computer*, 23(6):26–36, June 1990.

[21] 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*, pages 24–36, Santa Margherita Ligure, Italy, June 1995.

[22] D. Yeung, J. Kubiatowitcz, and A. Agarwal. MGS: A Multigrain Shared Memory System. In *Proceedings of the Twenty-Third International Symposium on Computer Architecture*, pages 44–55, Philadelphia, PA, May, 1996.