Software Shared Memory Support on Clusters of Symmetric MultiProcessors Using Remote-Write Networks *

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-access networks have recently become commodity items. These networks can connect clusters of symmetric multiprocessors (SMPs) to form very cost-effective, large scale parallel systems. Software-based distributed shared memory (SDSM) is a natural choice for the underlying platform. However, to exploit the platform's full potential, sharing across SMPs must be managed without compromising the efficiency of sharing within an SMP. Cashmere-2L is a "two-level" SDSM protocol that delivers the platform's potential through novel software techniques that leverage, without compromising, the efficiency of the hardware coherence. The protocol implements a moderately lazy release consistency model with page directories, home-nodes, and multiple concurrent writers. By avoiding global meta-data locks and TLB shootdown, Cashmere-2L is able to maintain a high level of asynchrony.

The prototype Cashmere-2L system currently runs on an 8-node, 32-processor DEC AlphaServer cluster connected by a DEC Memory Channel remote-memory-acccess network. Across nine applications, the system obtains speedups from 5 to 26 on 32 processors. We quantify the performance of protocol optimizations and alternatives along two dimensions – SMP exploitation and remote-write network utilization. To determine the effectiveness of our two-level design, we compare its performance with a one-level protocol that does not leverage hardware coherence, and evaluate the benefits of our protocol's asynchrony. We also compare three alternative protocols that utilize the remote-write network to varying degrees: placing application data in (or out of) remotely accessible memory, and fine-grain versus coarse-grain updates of data. Our results show that our two-level design delivers an average of 25% performance improvement over the one-level protocol. Protocol asynchrony does not result in significant performance gain, at least in an environment with low-cost inter-processor interrupts. Remote-access to application data can reduce overhead, but the protocol that takes full advantage of the network produces an average improvement of only 3% across the nine applications.

1 Introduction

Conventional networks have long been used to connect workstations to form a large-scale parallel machine. Due to high communication latency, data sharing across the workstations, or *nodes*, has been too expensive and has limited overall performance. To address this bottleneck, recent work has investigated replacing the uniprocessor machines with symmetric multiprocessors (SMPs). Then ideally a large amount of sharing through fast hardware coherent shared memory. These new types of SMP-based parallel machines are typically called *SMP clusters.*

Much recent research has also focused on reducing network overhead, particularly by providing remote memory access support in the network. With this support, a machine on the network can access memory physically located on another machine connected to the network. This access can occur without intervention from any processor on the destination machine. Several projects, including Princeton's Shrimp, Hewlett-Packard's Hamlyn, Digital Equipment's Memory Channel and Intel's VIA,

*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).

have implemented remote memory access networks that are accessible through a user-level, rather than kernel-level, interface. The remote-access capabilities combine with the user-level interface to produce one-way latencies as low as 2.2μs (Memory Channel 2 [10]). The Memory Channel is also notable for establishing the commercial viability of these networks.

A very high-performance and cost-effective parallel cluster can be constructed from commodity SMPs and remote-access networks. Software-based distributed shared memory (SDSM) is an attractive programming paradigm for this new type of SMP cluster since it can fully exploit the uniqueness of the platform – SMP hardware coherence and the remote-access network. Transparent to the application, a "two-level" SDSM protocol can manage data sharing at two levels. The first level relies on hardware coherence to handle sharing occuring within an SMP node, while the second level uses the network to enforce consistency throughout the entire system.

Cashmere-2L [18] is a two-level SDSM protocol designed for SMP clusters connected by a remote-access network. The protocol leverages the available hardware coherence to both provide consistency within nodes and minimize the number of software protocol operations. The remote-access network is used to reduce the overhead of protocol meta-data maintenance.

Shared memory accesses are tracked through virtual memory (VM) faults, and the protocol uses invalidations, page directories and home nodes to manage the data and its sharers. As in Munin [5], multiple concurrent writers are supported by initially creating a pristine copy of a page, or a *twin*, and then subsequently comparing the working copy to the twin in order to uncover the modifications, or the *diffs*. As in HLRC [21], these diffs are then sent to the home node where they are merged into the master copy of the page. Cashmere-2L distinguishes this operation as an *outgoing diff*. In addition, Cashmere-2L uses twins to identify the remote modifications that need to be applied to the local working copy. This operation, called an *incoming diff*, replaces more synchronous operations used in other systems, e.g. TLB shootdown.

In previous work [18], we have shown that Cashmere-2L performs as much as 48% better than a comparable one-level protocol. The performance improvements are due to the exploitation of hardware coherence for sharing among nodes and coalescing of remote requests for data. Two-way diffing does not significantly affect application performance. This seems to contradict work in [9] that stated TLB shootdown is a major source of overhead. There are several reasons for this apparent contradiction. A shootdown operation is used to force other processors to downgrade their write permissions on a page. In Cashmere-2L, shootdown only occurs in the presence of false sharing among multiple writers within the same node. The shootdown will involve only the other writers on the node, at most three on our platform. For these reasons, Cashmere-2L is much less reliant on a shootdown mechanism than previous protocols. Finally, our shootdown mechanism is very inexpensive because of our polling-based messaging implementation. Only one application in our benchmark suite experienced frequent shootdowns, and for that application, two-way diffing and polling-based shootdown produced the same performance. However, when shootdown was switched to more expensive, interrupt-based messaging, the performance of the same application using shootdown degraded by 10%. On platforms with more expensive shootdown operations, two-way diffing may provide a significant performance advantage.

The remote-write network can be used to maintain application data, in addition to protocol meta-data. Diff operations and page updates can both be applied through remote-write. Our base Cashmere-2L protocol uses remote-write diffs. Unfortunately, due to limitations in the size of remotely-accessible memory, remote application of page updates is impractical. Our results show that in comparison to an alternative protocol that does not place application data in remotely-accessible memory, the remote-write diffs provide an average of only 3% improvement across the applications. Another alternative protocol replaces the twin and diff operations entirely with a write-through technique. This technique sends modifications to the home node as they occur. Our analysis shows that write-through can create excessive bus and network contention and generally results in poor performance.

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 their format in memory, 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 node using memory-mapped regions. Currently, the network only provides write-access capabilities.

Future generations will also include read-access. The network interfaces to a industry-standard PCI bus. A memory-mapped region can be mapped into a process' virtual address space for transmit, receive, or both, although a particular virtual address can only be mapped as transmit or receive. The virtual addresses for a transmit region map to addresses in I/O space, in particular addresses on the MC's PCI adapter. Receive regions are backed by physical RAM. Writes into transmit regions bypass all caches (although they are buffered in the Alpha's write buffer), 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. The upcoming second generation Memory Channel 2 network will drop one-way latency to 2.2 μs and offer per-link transfer bandwidths of 66 MB/s, on some platforms [10].

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 available hardware coherence to reduce demands on the inter-node level.

Consistency Model: Cashmere-2L implements a multiple-writer, *moderately* lazy release consistent protocol. This design decision is enabled by the requirement that applications adhere to the data-race-free programming model [1]. Simply stated, shared memory accesses must be protected by synchronization operations such that no data races exist. The synchronization operations must also be visible to the run-time system. Cashmere-2L provides routines for lock, barrier, and flag operations, with each constructed from simple acquire and release primitives. The primitives mark the beginning and end, respectively, of access to a piece of shared data. The Cashmere-2L consistency model implementation lies in between TreadMarks [3] and Munin [5]. 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.

Management of Shared Data: Accesses to shared data are detected through virtual memory (VM) faults, so making the coherence unit equal to the size of VM page. Each page has a single *home node* and an entry in a global *page directory*. The home node holds the master copy of the page, which is always updated with remote modifications in accordance with the consistency model. The page directory entry tracks the page's sharing set and other protocol meta-data. In order to compensate for the Memory Channel's lack of a remote-read capability, the page directory is replicated on each node and any modification is immediately broadcast.

Propagation of Shared Data: A process' initial access to a piece of shared data will trigger a page fault. After gaining control, the protocol will send a *page fetch* request to the home node. In reply, the home node will send back the latest copy of the page. If the process subsequently modifies the page, the protocol will again be invoked through a page fault. The protocol then creates a pristine copy of the page, called a *twin* [5]. Finally, the page id is inserted into a *dirty list* and the page's VM permissions are upgraded. At the next release operation, the protocol compares each page in the dirty list to its twin and sends all differences, denoting local modifications, to the home node [21]. After completing this *page flush* operation for each page, the protocol sends *write notices* to each node in the page's sharing set. Each write notice indicates a pending global modification. At each acquire operation, the process traverses its list of received write notices and invalidates the named pages. If pages are subsequently accessed, the latest copy can be obtained from the home node.

Two-Level Coherence Design: Processes inside a node share the same physical frame for a shared data page, so intra-node coherence is maintained entirely through hardware. The hardware coherence also allows two software protocol operations from different processors to be coalesced, as long as no intervening operation interferes. In a novel approach, each node uses a

private logical clock to track the temporal order of protocol operations. By keeping the clocks private to each node, the protocol is able to maintain a high level of asynchrony. The logical clock is incremented on protocol communication events, namely page fetches, page flushes, acquires, and releases.

By tracking temporal ordering, the protocol can determine if operations can be safely coalesced. As an example, a node's logical clock is used to timestamp both when a page was last updated and when the last write notice for a page was received. Before a page fetch request, the protocol compares the timestamps to determine if the page has been updated since the last write notice was received. If so, the page fetch is unnecessary; the request can be coalesced with the last request.

However, the two-level implementation does add extra overhead in one case. An incoming page fetch operation must take care not to overwrite any modifications from local concurrent writers. As described in section 2.6, a common solution has been to shoot down the write mappings on the other processes. Cashmere-2L however employs a novel *incoming diff* operation. The incoming data is simply compared to the twin, and any differences are written to both the working copy and the twin. Since applications are required to be data-race-free, these differences are exactly the modifications performed by the 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 during the next release. Incoming diffs fully avoid the explicit synchronization incurred in a shootdown operation.

Network Advantages: The home nodes place each page master directly in MC space. Remote nodes can then apply diffs through the network's remote-write mechanism and avoid interrupting the home node. Similarly, write notices can be delivered via remote-writes. Correct protocol operation requires that the diff be applied before any related write notices are processed, and also that all write notices associated with a lock are received at the destination before that lock is passed to another node. Rather than using an explicit acknowledgement scheme in these cases, Cashmere-2L relies on the MC's write-ordering property. By writing to a loopback region and waiting for the result, a processor can ensure that all packets on the network prior to the loopback write have been received at their destinations. Called a *MC flush*, this operation is performed after applying a diff and implicitly as part of an acquire.

The synchronization primitives have two-level implementations that also heavily exploit the remote-write and total ordering characteristics of the network. Within nodes, `load-linked/store-conditional` instructions provide synchronization, while MC remote-writes are used to perform synchronization throughout the cluster.

Protocol Optimizations: The fundamental protocol design outlined above is also improved by several optimizations. Cashmere-2L includes an *exclusive* page state for shared data pages currently only accessed by one node. These pages act as if they were private and incur no protocol overhead – no page faults, no twins, and no directory updates. When another sharer emerges, the exclusive holder is interrupted to flush its changes to the home node. The page then returns to the normal mode of operation.

In order to reduce contention on global structures, Cashmere-2L also uses lock-free structures for the page directory and write notice lists. These structures eliminate the need for lock protection at the expense of larger structures and slightly increased processing time.

2.3 Implementation Details

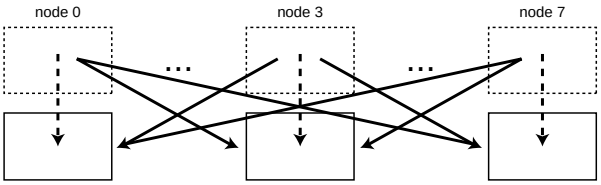
Memory and data structures of Cashmere-2L are grouped into four major classes, according to their functional use. These are:

Global Protocol Meta-Data: Global meta-data is mapped in MC space and maintained through remote-writes. The *global, or first-level, page directory* is replicated on each node, with each node mapping the same MC region for both receive and transmit in order to effect a broadcast (see Figure 1(a)). Writes are written to both the transmit and receive regions. Loopback mode is disabled. The *global write notice lists* require point-to-point messages, so each node is assigned a unique MC region, which it maps as receive and the other nodes map as transmit. (See Figure 1(b).) *Explicit message buffers* employ the same point-to-point scheme.

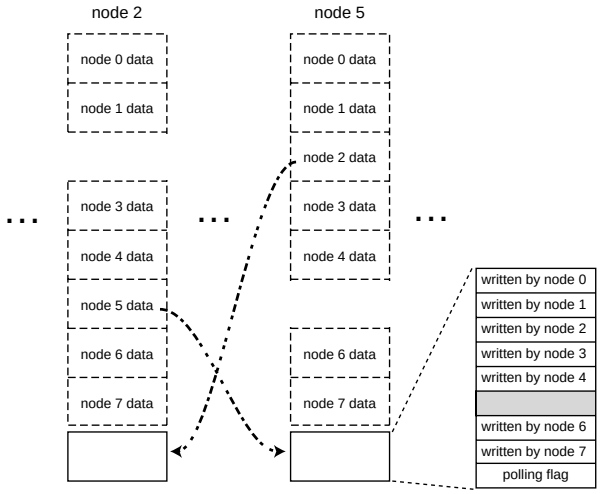
Node Protocol Meta-Data: Each node maintains protocol meta-data in standard shared memory. The meta-data includes *the second level directory, twins, per-processor dirty lists, a No-Longer-Exclusive list, a logical clock, and a Last-Release timestamp.*

Application Data: On the home node, application data is placed in MC space using a point-to-point scheme. (See Figure 1(c).) Non-home nodes hold the page in their shared memory, with each processor sharing the same virtual and physical addresses.

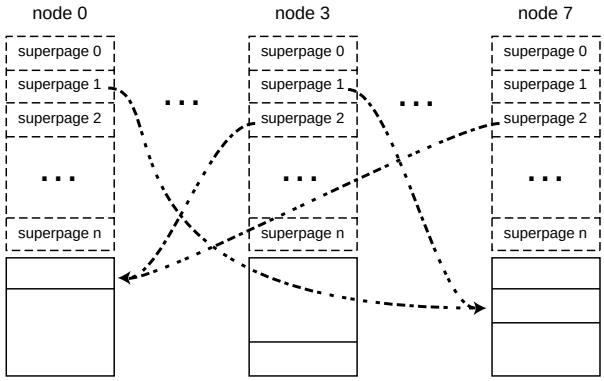
Synchronization objects: Barriers, locks, and flags are mapped in MC space in a broadcast scheme. Only locks use loopback mode, in order to allow processors to detect when their writes are globally performed. (See Figure 1(a).) Barriers and locks also have components in standard shared memory, in order to provide efficient synchronization within the node.



(a) The broadcast scheme used by the 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. Writes are performed in the receive regions either implicitly through loopback mode (locks) or explicitly though doubling—two writes to both the transmit and receive regions (directory, barriers, and flags). The former allows the writer to determine when a write has been globally performed.



(b) The point-to-point scheme for 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.



(c) 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.

Figure 1: Memory Channel Mappings.

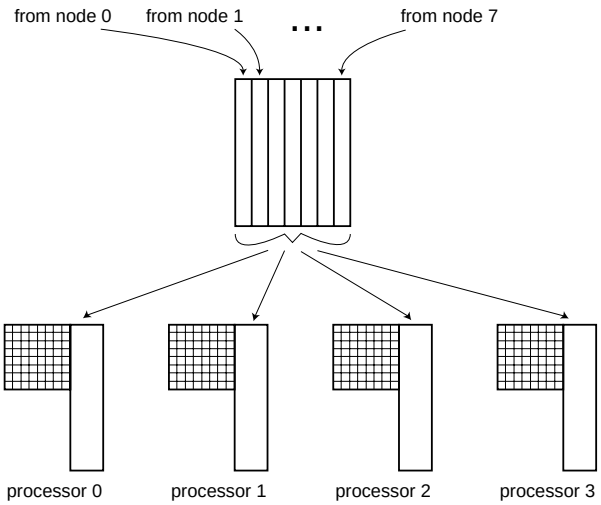


Figure 2: Write notices. A node's globally-accessible write notice list has one bin (a circular queue) for each other node in the system. On an acquire operation, a processor traverses all seven bins and distributes notices to the per-processor bins of processors with mappings for the page. Each per-processor bin consists of both a circular queue and a bitmap.

Protocol Meta-Data. The two-level page directory contains one entry for each page of shared data. In the global first-level directory, each entry consists of one word per node. Each word is written by a unique node and, for that node, indicates (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). The home node indications are redundant. This information could be compressed into one word, but then a global lock would be require to ensure atomic access (since 32-bits is the smallest unit that the 21064 can access atomically and since the MC does not support mutual exclusion in hardware.) For our system of eight nodes, however, the directory overhead for an 8K page is only 3%.

Within each node, each entry of the second-level directory indicates each local processor's sharing state (invalid, read-only, and read-write mappings). Each entry 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.

Write notice lists are also designed in a two-level manner. (See Figure 2.) The global level is a multi-bin (in our case, seven) structure where, again to avoid the need for mutually exclusive access, each bin is written by a unique remote node. Each bin contains a simple circular queue that holds the pending notices. When required by the protocol, a processor dequeues the pending notices from the circular queue of each incoming bin and then distributes the notices to the local write notice lists of each affected local processor. Each local write notice list consists of a simple queue and a bitmap. Each entry in the bitmap represents a shared page and is set if the page is described by a notice currently in the queue. Bitmap entries are set/reset by successful enqueue/dequeue operations, and used to avoid redundant notices in the lists.

The processing of the write notices, and several other protocol structures, requires mutual exclusion within a node. The protocol implements very efficient local locks with `ll/sc`. The `ll/sc` instruction is also used to atomically increment the logical clocks.

Each node contains two other important meta-data structures. Each processor maintains its own private *dirty list*, which is a list of shared pages that it has modified since its last release. Also, the node maintains a shared *No-Longer-Exclusive* list, which indicates what pages have recently left exclusive-mode.

Cashmere-2L only uses explicit inter-node requests to request a copy of the page from its home node and to break a page out of exclusive-mode. The request and reply buffers are structured very much like the global write notice lists. The buffers use a multi-bin structure to avoid the need for global locks.

Application Data. The home node places the master page in MC space, while other nodes uses standard shared memory. Initially home nodes are assigned in a round robin manner, and then are re-assigned dynamically after program initialization to the node that first touches a page. [14]. To relocate a master page, a processor must acquire a global lock and explicitly request a remapping from the initial home node. Because relocation occurs only once, the use of locks does not impact performance.

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

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 home-node-assignment code.

The Memory Channel subsystem of the Digital Unix kernel has a limitation on table space, and so each shared page can not be mapped as a unique MC region. For this reason, the protocol maps each MC region to a *superpage*, whose size is determined by dividing the application's maximum memory size by the number of MC regions. Coherence granularity is still a single page throughout the protocol. The only constraint is that home nodes are assigned to entire superpages.

Synchronization. Application locks (and protocol locks for home node selection) consist of a test-and-set flag on each node and an 8-entry array in Memory Channel space. To acquire a lock, a process first acquires the per-node test-and-set flag using `ll/sc`. Then the process 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. At 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.

Synchronization flags are mapped as MC regions following a broadcast scheme. A modification is simply written to both the receive and transmit regions, and is automatically broadcast through the system.

Polling. To compensate for the lack of remote-read capability in the first generation Memory Channel, our protocol uses explicit message passing. The request and reply buffers are implemented in MC space using a point-to-point scheme. To send a message, a processor writes the message data to the MC point-to-point link and use remote-write to assert a polling flag in the destination processor's memory.

All processors must check the polling flag 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 the polling instructions (see Figure 3) 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.

Kernel changes. We made two 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. Our philosophy has been to make only those kernel changes that are of clear benefit for many types of applications. We could improve performance significantly by moving protocol operations (e.g. the page fault handlers) 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. 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

Twins are used 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 [13], 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, our results in [18] indicate that twins and diffs perform better. More importantly, 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.

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. 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 [9, 20]. 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.

Treatment of Application Data. By placing data in MC address space, diff operations can be accomplished via remote-write operations. To avoid data set size limitations imposed by the 128MB of MC space, two protocol variants called Cashmere-2LOffMC and Cashmere-2LMigr have been developed. These variants place application data in regular node shared memory and then use explicit messages to perform diff operations. Cashmere-2LMigr also replaces the first-touch home node assignment policy with a migratory policy that re-assigns the master page location after a home node has completed its modifications. Since twins are not needed on the home node, twinning overhead can be reduced by migrating the home node to an active writer.

In order to support migration, the directory entry structure must be modified. Each entry still consists of one word per node, with each word containing two bits indicating the page status and one bit indicating if the node is performing a diff. In addition, an extra word is added to the entry and is shared by all nodes. This word contains five bits to indicate the home node, one bit to indicate exclusive mode, and one bit to signal that a migration is in progress. The per-node words can still be accessed in a lock-free manner, however modifications to the shared word must be synchronized through a global lock. Modifications to the shared word though are very infrequent and are aborted if another process is holding the global lock.

Before starting a diff, a process asserts the diff bit in its word of the directory entry. Then the process checks the shared word in the directory entry and waits if a migration is in progress. With no migration in progress, the diff can proceed. Upon completion, the process resets its node's diff bit.

A migration can occur only when the home node is not writing to the page. Inside the protocol write fault handler, a process checks if the home node is concurrently writing the page and if not, performs a conditional acquire on a per-page global migration lock. If another process is holding the lock, then a migration is already in progress and the lock attempt, along with the migration attempt, is aborted. On the other hand, if the acquire is successful, the process next verifies that the home node is still not modifying the page and that no diffs for the page are in progress. To ensure that its copy is up-to-date, the process searches the write notice list for the entries for the page and issues a page fetch request if such a write notice exists. Finally the process assumes ownership by updating the home node indication in the directory entry and then releases the migration lock.

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 [13]. 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. Like Cashmere-2L, both protocols are invalidation-based and use a global page directory and home nodes. The main difference, is that the protocols do not explicitly exploit intra-node hardware coherence.

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).

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 has two on-chip caches: a 16K instruction cache and 16K data cache. 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 represent 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 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

Dense Matrix Each of these applications operates on a dense matrix and have uniform computational requirement for each element.

CLU: a kernel from the SPLASH-2 [19] 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.

LU: Also from Splash-2, the implementation is identical to CLU except that blocks are not allocated contiguously, resulting in a very high degree of false sharing.

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.

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 band boundaries. Processors synchronize with barriers.

Sparse Data **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 [8]. 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

Program	Problem Size	Time (sec.)
CLU	2048x2048 (33Mbytes)	265.5
LU	2048x2048 (33Mbytes)	763.0
Gauss	2048x2048 (33Mbytes)	970.8
SOR	3072x4096 (50Mbytes)	162.2
Ilink	CLP (15Mbytes)	738.7
Volrend	Head (24MBytes)	11.8
Barnes	128K bodies (26Mbytes)	462.1
Em3d	64000 nodes (53Mbytes)	145.0
Water	4096 mols. (4Mbytes)	1566.4

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

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.

Work-Queue-Based Volrend: a modified SPLASH-2 application that renders a three-dimensional volume using a ray casting technique. The image plane is partitioned among processors in contiguous blocks of pixels. These blocks are further partitioned into small tiles, which serves as the unit of work. Load balancing is acheived by using distributed work queues with work stealing. To mitigate the effects of false sharing, the tile size has been increased and the task queue has been padded [11].

Irregular Access Patterns The communication patterns of these applications are determined at run-time and are often dynamic.

Barnes: an N-body simulation from the SPLASH-1 [17] 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 [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. 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.

Water: a molecular dynamics simulation from the SPLASH-1 [17] 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.

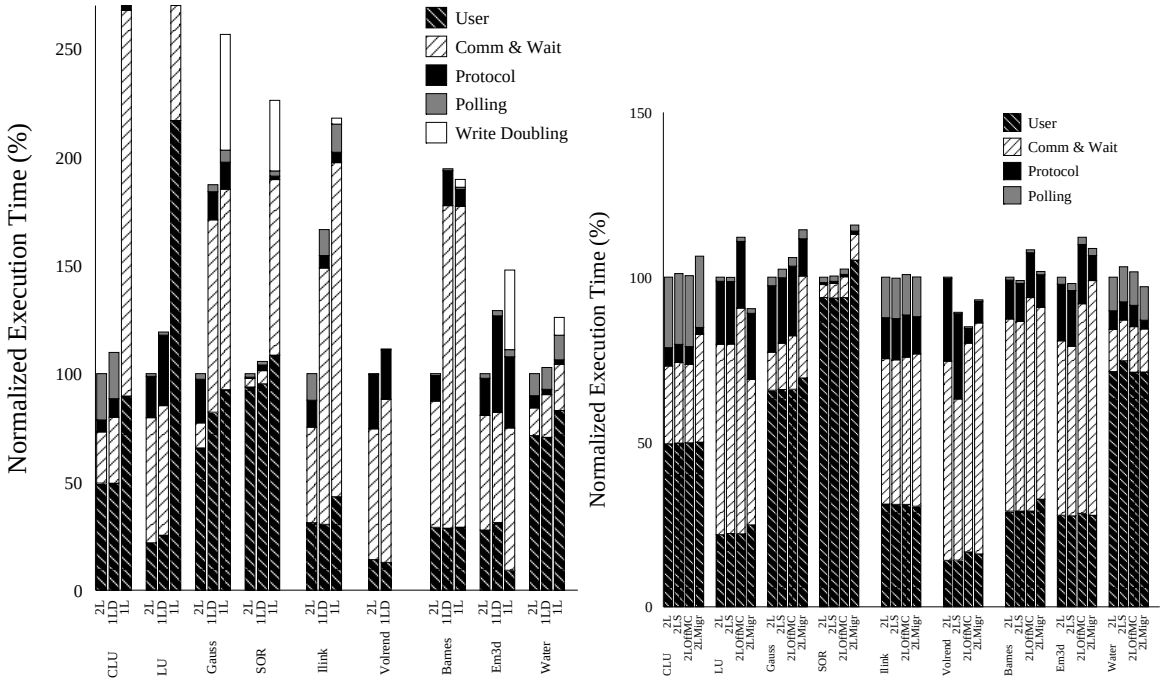
The data set sizes and uniprocessor execution times for each of the applications are presented in Table 2. The size of shared memory space is listed in parentheses. Execution times were measured by running each uninstrumented application sequentially without linking it to the protocol library.

3.3 Speedups Analysis

Figure 5 presents speedup bar charts for our applications on up to 32 processors. All calculations are with respect to the sequential times in Table 2. The configurations we use are specified as `P:C`, where `P` is the total number of processors per node and `C` is the number of processors per node. The total number of nodes then is P/C. For example 32:4 specifies 32-processors, 4 processors per node, and eight 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 amount of data transferred, and number of twins created. Statistics covering the number of incoming diff, flush-update, shootdown, and home node migration operations are also included under the appopriate protocol entries. All statistics, except for execution time, are aggregated over all 32 processors.

Figures 4(a) and 4(b) present breakdowns of execution time at 32 processors for each application on several protocol variants. The breakdowns are normalized with respect to total execution time for Cashmere-2L. The components shown represent



(a) Two-Level (2L), Two-Level-Shootdown (2LS), One-Level-Diff (1LD), and One-Level-Write-Doubling (1L) protocols.

(b) Two-Level (2L), Two-Level-Shootdown (2LS), Two-Level-Data-Off-Memory-Channel (2LOffMC), and Two-Level-Home-Migration (2LMigr) protocols.

Figure 4: Breakdown of percent normalized execution time 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).

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.

3.3.1 Comparison of 1LD to 1L

The two one-level protocols differ only in their mechanism for updating the home node. Cashmere-1LD uses diffs to forward modifications at synchronization release points, while Cashmere-1L updates the home node with modifications as they occur. The `1L` executable is instrumented with in-line code to double all writes to the home node, via a calculated address in MC space. This instrumentation increases computation, and applications with fine-grain access patterns can overwhelm the network's bandwidth with frequent communication. Primarily for these reasons, `1LD` improves execution time over `1L` by an average of 33% on eight of the applications, while the remaining application has similar performance on both protocols.

The dense matrix applications all have very fine-grain access patterns that favor `1LD`. CLU (66%), LU (76%), and Gauss (25%) show significant improvement on `1LD`. On `1L`, these applications suffer from lack of bandwidth. They also experience cache problems on the home node, where doubled-writes are sent to local memory rather than to uncacheable MC space. For example on CLU, the extra writes increase the working set size from 16K to 24K, which is larger the 21064 level 1 cache. On `1L`, SOR incurs a significant amount of unnecessary overhead due to the doubling of the processor's internal rows. Using diffs, `1LD` improves performance by 53%.

Application		SOR	CLU	LU	Water	Gauss	Ilink	Em3d	Barnes	Volrend
2L	Exec. Time (secs)	6.2	13.3	140.2	65.3	42.7	70.0	11.9	55.1	1.2
	Lock/Flag Acq. (K)	0	0	0	3.68	129.82	0	0	0	9.96
	Barriers	48	130	130	36	7	521	200	11	49
	Read Faults (K)	0.34	20.71	519.52	72.99	173.38	204.26	39.06	214.40	36.86
	Write Faults (K)	0.67	4.10	952.42	35.52	8.19	46.22	42.88	183.44	6.59
	Page Transfers (K)	0.34	8.30	134.60	31.90	42.14	51.31	32.91	65.98	14.14
	Dir. Updates (K)	2.02	16.50	779.97	111.59	67.76	144.64	154.31	261.35	27.46
	Write Notices (K)	0.34	0	185.88	77.47	23.27	105.41	32.60	260.39	10.49
	Data (Mbytes)	4.25	68.44	1293.09	263.94	345.48	424.98	275.91	564.54	118.45
	Twin Creations (K)	0.34	0	144.74	25.64	3.58	11.20	7.08	54.54	4.87
	Incoming Diffs	0	0	0	0	0	0	0	0	0.11
	Flush-Updates	0	0	0	0.17	0	0	0	0	0.67
2LS	Exec. Time (secs)	6.2	13.4	140.1	67.4	43.7	69.8	11.6	54.1	1.2
	Lock/Flag Acq. (K)	0	0	0	3.68	129.82	0	0	0	10.20
	Barriers	48	130	130	36	7	521	200	11	49
	Read Faults (K)	0.34	20.71	517.95	71.85	173.70	204.28	39.43	214.38	37.14
	Write Faults (K)	0.67	4.10	952.39	35.51	8.19	46.14	42.87	183.50	7.29
	Page Transfers (K)	0.34	8.31	134.25	30.61	41.87	51.31	33.27	65.98	14.04
	Dir. Updates (K)	2.02	16.50	777.65	109.40	70.88	144.55	154.95	261.42	28.44
	Write Notices (K)	0.34	0	186.21	80.00	23.13	105.55	32.95	259.66	10.80
	Data (Mbytes)	4.25	68.44	1293.31	267.64	345.58	425.00	276.03	664.63	116.97
	Twin Creations (K)	0.34	0	144.93	25.63	3.58	10.61	6.79	54.54	5.80
	Shootdowns	0	0	0	183	0	0	0	0	0.85
2LMIG	Exec. Time (secs)	7.2	14.2	126.8	63.5	48.9	70.2	13.3	55.2	1.1
	Lock/Flag Acq. (K)	0	0	0	3.68	129.82	0	0	0	18.84
	Barriers	48	130	130	36	7	521	200	11	49
	Read Faults (K)	0.67	20.70	584.57	80.68	169.82	211.84	47.41	203.68	38.51
	Write Faults (K)	0.67	4.16	938.88	35.50	13.28	46.13	44.40	183.64	4.44
	Page Transfers (K)	0.67	8.30	156.62	36.22	45.15	54.42	44.47	67.63	14.75
	Dir. Updates (K)	2.69	20.01	816.57	142.31	89.66	152.77	179.41	268.75	25.54
	Write Notices (K)	0.67	0	220.47	93.07	32.22	106.35	41.04	250.55	10.93
	Data (Mbytes)	5.64	68.37	1333.47	299.02	371.82	449.55	366.13	564.00	120.82
	Twin Creations (K)	0	0	105.79	0.86	0	8.32	0	46.63	0.56
	Incoming Diffs	0	0	0.01	0	0	0.03	0	0.69	0.76
	Flush-Updates	0	0	0	0.09	0	0	0	0	0.35
	Migrations	0	0	35.5	23.3	4.6	4.6	2.1	12.53	1.46
1LD	Exec. Time (secs)	6.6	15.4	167.3	69.7	80.2	116.6	15.4	106.6	1.7
	Lock/Flag Acq. (K)	0	0	0	3.68	129.82	0	0	0	12.01
	Barriers	48	130	130	36	7	521	200	11	49
	Read Faults (K)	2.98	22.83	1001.29	84.48	205.69	228.53	166.92	241.64	39.41
	Write Faults (K)	2.98	7.71	948.48	35.61	8.18	47.71	81.20	181.13	7.62
	Page Transfers (K)	2.98	22.83	1001.30	84.48	205.95	229.94	166.92	241.65	39.41
	Dir. Updates (K)	5.95	25.19	1937.72	163.17	312.44	459.19	328.63	483.33	49.77
	Write Notices (K)	2.98	2.36	936.42	78.69	106.74	229.26	161.71	241.67	10.27
	Data (Mbytes)	24.83	254.24	8243.34	696.42	1685.10	1894.22	1374.53	1983.61	324.92
	Twin Creations (K)	2.98		948.48	25.64	8.18	47.71	81.20	181.13	7.61

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

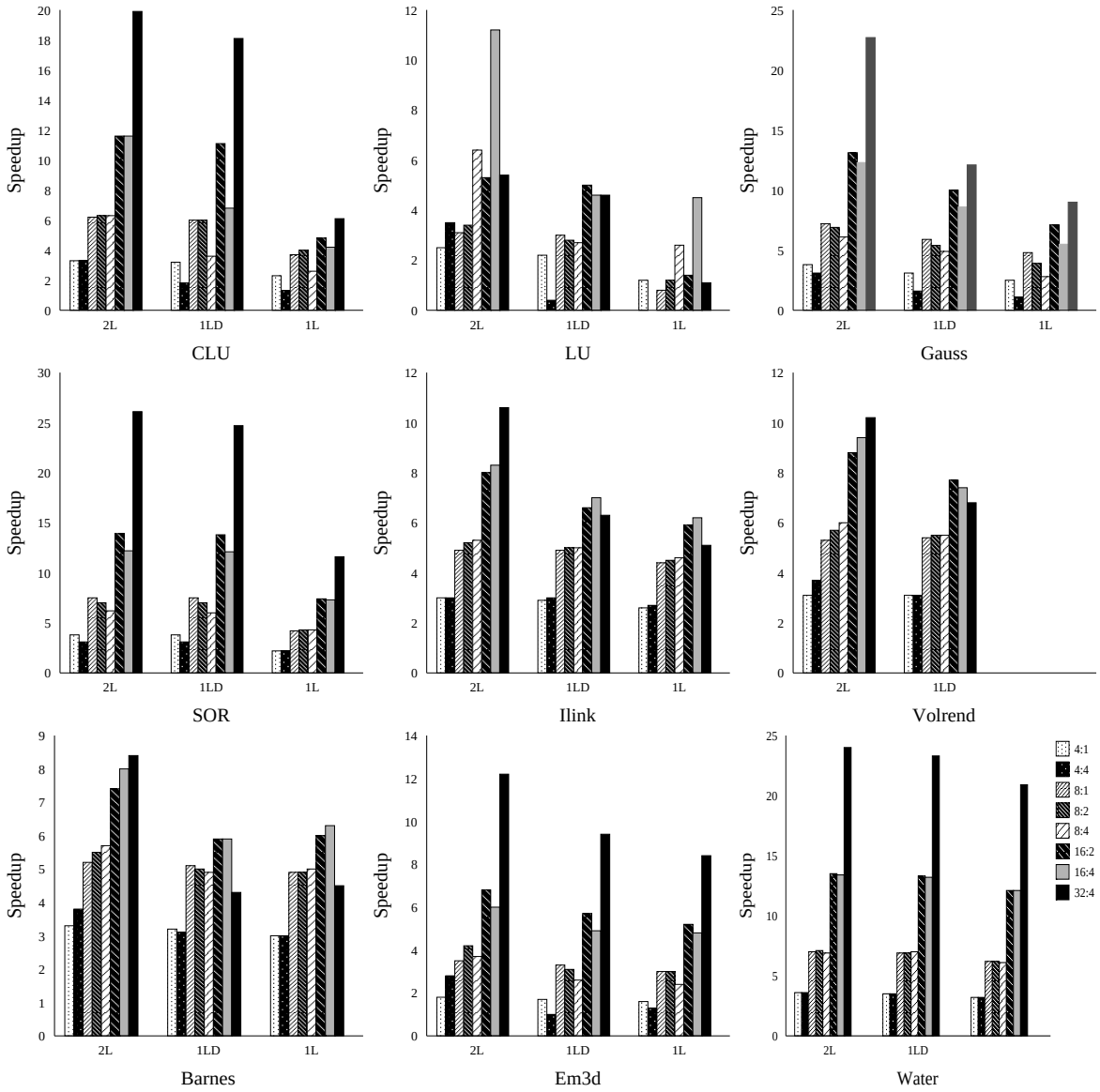


Figure 5: Speedups for Two-Level (2L), One-Level-Diffing (1LD), and One-Level-Write-Doubling (1L).

Ilink includes an all-to-one communication phase that places a significant strain on the available network bandwidth. However, the application also includes a one-to-all phase and works on a sparse matrix. Both of these characteristics do not require large bandwidth and so favor a write-doubling approach. Still, as Figure 4(a) shows, the communication and wait time is significantly longer in `1L`, and `1LD` outperforms `1L` by 19%.

On `1L`, Barnes spends less time in the protocol code since no twins or diffs are performed. As only small modifications per synchronization interval, the application does not incur bandwidth problems under `1L` and the protocol is able to outperform `1LD` by a thin margin of 3%. As another irregular application, Em3d can also benefit from avoiding twins in `1L`, however the application's communication-and-wait and write-doubling times are very high due to wasted write-doubling on each processor's interior data nodes. Em3d performs 11% better on `1LD`. Finally, Water also incurs lower protocol overhead in `1L`, but the write doubling overhead and associated cache pollution degrades performance. As on Em3d, `1LD` shows an 11% improvement on Water.

Volrend accesses shared data at a byte-level granularity, which is less than the granularity of write-doubling. As a result, Volrend can not be run on `1L` without major application restructuring.

3.3.2 Comparison of One- and Two-Level Protocols

In comparison to the best of the one-level protocols (1LD), the two-level protocols have superior performance for each application. SOR, CLU, and Water show small improvements of less than 10%, while LU and Em3d show decent improvements of approximately 20%. Volrend, Ilink, Gauss, and Barnes all show significant improvements between 30% and 48%. The key to the excellent performance is the two-level protocol's exploitation of the hardware coherence to minimize coherence overhead on the home node and cross-node software coherence activities.

Of the dense matrix applications, SOR (5%) and CLU (9%) both show small improvement. SOR has a high computation to communication ratio and entirely consists of nearest neighbor sharing. The two-level protocols improve on the already large speedup of 1LD by exploiting the hardware coherence to maintain the sharing within each node. The number/amount of page fetches, page transfers, and communicated data are reduced in 2L, as is the time spent in the protocol. (See Table 3 and Figure 4(a).) Similarly, 2L is able to reduce the amount of page transfers and data communicated in CLU. This application also benefits from the available hardware coherence and the coalescing of page requests.

The two-level protocols' use of hardware coherence is crucial in obtaining good performance in LU. The application's allocation and block assignment scheme produces an extremely high degree of false sharing at the page level. In the two-level protocols, the hardware coherence masks the effects of intra-node false sharing and reduces the amount of software protocol activity. There is an eight- to nine-fold reduction in number/amount of page transfers, data communicated, and twins created. As a result, 2L has much lower protocol overhead and communication and wait time. The 2L protocol outperforms 1LD at each configuration. There is however a large performance drop between 16:4 and 32:4 in 2L. Surprisingly, the drop provides proof of the effectiveness of the two-level design. At 16:4, the allocation and assignment scheme induces false sharing that is contained entirely within nodes and therefore is handled through the available hardware coherence. In the 32:4 configuration however, a significant amount of false sharing crosses node boundaries, resulting in heavy software coherence activity. At 16:4, 2L creates only two twins, but at 32:4, the number of twins jumps to 144K! The result at 16:4 shows how effectively the hardware coherence can be used to mask the presence of false sharing.

The fourth of the dense matrix applications, Gauss, improves dramatically on 2L. The application exhibits a single-producer/multiple-consumer sharing pattern. The two-level protocol is able to coalesce many page requests, resulting in four-fold reductions in number of page requests and amount of data transferred. This corresponds to a huge drop in communication and wait time and a 46% overall improvement. Like Gauss, Ilink also benefits from the ability to coalesce the page requests present in a single-producer/multiple-consumer sharing phase. The amount of data transferred is reduced by a factor of four and the performance improves 40% under 2L.

In Volrend, the work stealing introduces significant false sharing into the application, but again 2L is able to mitigate any negative performance effects by leveraging the hardware coherence to reduce twins and coalesce page requests. The false sharing does cause incoming diffs and flush-updates, which contributes to a slight increase in protocol overhead. However, the communication and wait time is decreased and 2L still improves the performance of Volrend by 33% over 1LD.

Of the irregular applications, Barnes shows the greatest improvement at 48%. The application benefits heavily from coalescing of requests to reduce the communication and wait time. The low computation-to-communication ratio allows for a substantial improvement in overall performance. After Barnes, Em3d shows the next largest improvement of the irregular applications. As much of the processing occurs on the home node, 2L is able to leverage available hardware coherence and boost performance by 23%. Finally, Water shows only a small 3% improvement under 2L. The amount of data communicated is halved, but the high computation-to-communication ratio limits improvements.

3.3.3 Effect of Clustering

With the total number of processors fixed, the level of *clustering* refers to the number of processes per node. As the clustering is increased, the amount of inter-node communication traffic should normally decrease, resulting in improved performance. However, this decrease in network traffic may be offset by increased contention for the node's shared memory bus or its single I/O connection.

The degradation resulting from bus contention is especially evident in SOR and Gauss. In both cases, the 4:1 configuration on 2L outperforms a hardware shared memory executable running on 4 processors. The performance difference is 20% in both cases. These two dense matrix applications have little computation per element, and the application working sets are both larger than the board-level cache. Significant contention for the memory bus results as the clustering is increased.

Neither CLU nor LU experience the same level of memory bus contention; however neither application benefits from increased clustering. The one-level protocol adds intra-node communication, and so increased clustering degrades performance significantly in both applications. For CLU, 2L eliminates the intra-node protocol communication and also the associated performance degradation. Performance does not improve with clustering due to the application's structure, which is already

designed to reduce communication. For LU, 2L is able to increasingly manage false sharing through hardware coherence as the clustering is increased.

Ilink and Volrend both show slight performance improvements as the clustering increases. Ilink benefits primarily from the coalescing of page requests in its single-producer, multiple-consumer phase. In Volrend, the two-level protocol leverages the available hardware coherence to reduce the penalty of false sharing in the task queue. This results in a drop in protocol overhead and better load balance. The one-level protocol does not leverage the hardware coherence to reduce protocol operations, and so bus contention increases and performance degrades.

Of the irregular applications, Barnes on 2L benefits the most from clustering. As clustering is increased at each step (from 1 to 2 to 4 processors per node), 2L is able to reduce the amount of data transferred by a factor of two. Water has a very high computation-to-communication ratio, so it is affected very little by increased clustering. Em3d is hurt by increased clustering at both eight and sixteen total processors. This result is unexpected since on 2L there are no signs of excess bus contention at four processors. At 4:4 however, the protocol does not make any twins. At eight and sixteen processors, the protocol does incur twinning and diffing overhead, which triggers bus contention problems. Contention is indicated by a 20% increase in user time and 40% increase in average twin and diff cost. With Em3d's nearest neighbor sharing patterns, the processors on the node boundaries incur the majority of the protocol overhead and are disproportionately affected by the bus contention. This leads to increased barrier synchronization times.

Note that our results compare the one and two-level protocols on the same clustered hardware. Both protocols share the same network bandwidth per processor. The one-level protocol may more naturally fit on non-clustered hardware, with higher network bandwidth per processor. For this reason, our results only appear to contradict other results that state bandwidth is a major factor in the performance of clustered systems [4, 6, 9]. Also, as the Memory Channel is a bus interconnect, our hardware platform favors protocols that reduce inter-node transactions.

3.3.4 Reducing Synchrony through Two-Way Diffing

In previous two-level protocols, TLB shootdown has been commonly used to synchronize processors during certain protocol operations. The extra synchrony has been reported as a major obstacle to high performance (e.g. SoftFLASH [9]). In Cashmere-2L, two-way diffing avoids added synchrony, but surprisingly shows no performance advantages (or disadvantages) over TLB shootdown. This result is explained by the protocol's multi-writer design and an efficient shootdown mechanism.

In SoftFLASH, a shootdown occurs when a page is "stolen" by a remote processor. The protocol must eliminate all local mappings. As SoftFLASH does not track the location of local mappings, the protocol must conservatively shootdown all processors that may have hosted an application process. In contrast, pages are never stolen in Cashmere-2L. Shootdown (or two-way diffing) is only necessary when more than one local processor is modifying a page at a release or a page fault. This can only occur in lock-based applications on pages that are falsely shared. In barrier-based applications, one processor can perform the page operations for the entire node. The last local processor to enter a barrier will perform the page flush for the entire node. Then the intra-node timestamps ensure that one page fetch after a barrier will suffice for the entire node. In addition, the Cashmere-2L second-level directories indicate the local processors that are modifying a page. Thus the shootdown can be focused directly on the affected processors.

Finally, due to the Cashmere-2L fast polling-based messaging layer, a shootdown of one processor requires only 72 μs. If intra-node interrupts are used instead of polling, the cost rises to 142 μs. Volrend is the only application with a significant number of shootdown operations. With polling-based messaging, 2LS matches the performance of 2L. If instead the shootdown mechanism is implemented with intra-node interrupts, the 2LS execution time increases by 10%. Our interrupts have already been modified to reduce their cost by an order of magnitude. Although two-way diffs do not affect results on our polling-based platform, the technique may yield significant benefits on platforms with more expensive shootdown operations.

3.3.5 The Impact of Storing Application Data in Remotely-Accessible Memory

The base Cashmere-2L protocol is able to perform some data maintenance, namely diff operations, through the MC's remote-write capabilities. The 2LOffMC and 2LMigr protocols are identical to 2L, except in their treatment of application data. Both protocol variants move data out of remotely-accessible memory. Instead of remote-write diff operations, both variants use explicit diff messages. The two variants differ in their home node assignment policy. As in 2L, 2LOffMC uses a first-touch policy. In 2LMigr, the home node assignment may be migrated as home nodes complete their modifications and new writers emerge. A migration operation requires 55μs. The cost of an explicit diff, depending on the size, ranges from 485–760 μs, which is roughly a 40% increase over a remote-write diff.

Due to the explicit diff messages, the performance of the nine applications on 2LOffMC averages 10% worse than on 2L. The migration policy in 2LMigr reduces the number of twins for all the applications, however the policy also generally increases the number of page fetches, as former home nodes eventually require page updates. Despite this tradeoff, 2LMigr

averages only 3% worse performance than 2L. In fact, 2LMigr actually *improves* performance over 2L for LU (10%), Volrend (10%), and Water (3%).

Of the dense matrix applications, SOR, Gauss, and CLU all lose performance when data is moved from MC space. All have a relatively small number of twin operations (see Table 3). The number of operations are reduced by the migration policy, but the gain is offset by an increase in the number of page fetches. The home node assignment policy has an especially subtle effect in SOR, where the two assignment policies give different home nodes to some boundary rows. The home node assignments produced by the first-touch policy result in less communication, and 2LMigr performs 16% worse than 2L. Gauss and Em3d similarly only experience a small decrease in twinning overhead, but a higher communication cost under 2LMigr leads to performance degradation of 14% and 12%, respectively. Like SOR, CLU suffers a 5% degradation on 2LMigr due to slight differences in home node assignments.

In LU, the migration policy cuts the number of twins from 144K in 2L to 100K in 2LMigr. Protocol overhead is unaffected, however (see Figure 4(b)). The cost of migrations adds to protocol overhead, but the large reduction in twinning overhead reduces the perturbation of application computation, leading to a drop in communication and wait time and a 10% overall improvement on 2LMigr.

Ilink surprisingly is unaffected by the diff mechanism or the home node assignment scheme. The application's sparse modifications produce small diffs that reduce the potential impact of the alternative approaches. As can be seen in Figure 4(b), the execution time breakdown is basically equivalent for the three variants.

Due to its task stealing, Volrend is very sensitive to any protocol changes that affect execution timing. The change to explicit diff messages in 2LOffMC degrades performance by 44%, compared to 2L performance. The extra overhead results in more load imbalance and increased task stealing, which is expensive due to the inherent false sharing. On the other hand, the migration policy in 2LMigr reduces the number of twins by a factor of four, thereby reducing protocol overhead and load imbalance. Overall, performance on 2LMigr is 10% better than on 2L.

Barnes is another application with inherent false sharing, which results in performance on 2LOffMC performance that is 8% worse than 2L. However, the migrating home nodes again mitigate the cost of the false sharing by reducing the number of twins. Performance on 2LMigr matches that on 2L. Water has some false sharing, but more importantly, its primary migratory sharing pattern is ideally suited to a migrating home node policy. However the application's small shared data size minimizes the potential improvements from protocol optimizations. Performance on 2LOffMC is 2% worse than on 2L, but 2LMigr boosts performance by 3% over 2L.

4 Related Work

In one of the first studies of layered hardware/software coherence protocols, Cox *et al.* simulated a system of 8-way, bus-based multiprocessors connected by an ATM network. Their coherence protocol was derived from TreadMarks [3]. Their findings stated that for clustering to provide significant benefits, reduction in inter-node messages and bandwidth requirements must be proportional to the degree of clustering. In simulating a similar system, Karlsson and Stenstrom [12] found that the limiting factor in performance was the latency rather than the bandwidth of the message-level interconnect.

The MGS system from MIT was the first implementation of a layered coherence protocol [20]. The system uses a multi-writer protocol, similar to Munin, to manage coherence across the nodes. A special optimization avoided unnecessary diffs in the case of a single writer. MGS performance results improved with large numbers of processors per node, however the implementation platform, Alewife [2], is a hardware DSM with a mesh interconnect. Commodity SMPs are bus-based and typically provide only a single network connection, so larger nodes are impractical.

SoftFLASH [9] is a two-level protocol that uses TLB faults to track shared data accesses. Despite being implemented at the kernel level, the protocol experiences a significant overhead from the required TLB shootdown operations. The overhead is exacerbated by the protocol's single-writer nature, which potentially results in frequent invalidations. Also, SoftFLASH must conservatively shoot down any processors that may have run an application process since TLB contents are unknown. The SoftFLASH results report that increased communication costs negate any benefits from clustering. This is primarily explained by the heavy shootdown overhead, especially in large nodes.

Shasta [16] implements an eager release consistent, variable-granularity, fine-grained, single-writer protocol for coherence across nodes. The variable granularity is enabled by using in-line checks instead of virtual memory or TLB faults. Each shared memory reference is validated against a state table that records the protocol state for each coherence object. Several intelligent techniques are used to minimize the overhead of the checks. The relative benefits of an instrumentation-based fine-grained and a coarse-grained VM-based approach are still to be determined.

The Princeton group has recently completed implementation of HLRCsmp [15], a two-level version of their HLRC protocol. Our two protocols share several similarities including the use of home nodes and two-way diffs. The main difference is

that rather than a centralized directory, HLRCsmp uses distributed meta-data consisting of vector timestamps that track the "happens-before" relationship. The effect of this difference on performance is an open question.

5 Conclusion

Cashmere-2L implements a two-level multiple-writer, moderately lazy release consistent protocol. By leveraging the available hardware coherence, the protocol can minimize sharing overhead within nodes and coalesce remote protocol operations. Protocol meta-data maintenance overhead is also reduced by exploiting the remote-write network. In addition, the protocol maintains a high level of asynchrony by employing a novel two-way diffing mechanism and lock-free global protocol structures.

Compared to a comparable one-level protocol, Cashmere-2L improves performance by an average of 25%, with the largest improvement being 48%. On the two-level design, the applications benefit from both reduced overhead on sharing within a node, and sharply reduced protocol communication. The use of two-way diffing, rather than TLB shootdown, has little effect on performance. Cashmere's multiple-writer design and use of separate page tables largely reduces the protocol's reliance on a shootdown mechanism. Shootdown is only needed in the event of active false sharing within a node and then only involves the active writers, a maximum of four on our platform. Only one application in our suite had these characteristics, and its performance for both mechanisms was very similar. Also, the shootdown mechanism on our platform is relatively inexpensive. On other platforms, incoming diffs may be more advantageous.

The Memory Channel's remote-write capabilities can be used to reduce the cost of application data maintenance operations, such as diffs. However, the gain in performance is on average only 3% compared to a protocol maintaining application data in private memory and using migrating home nodes to reduce the frequency of diffs.

The current Cashmere-2L design relies on the Memory Channel's broadcast, ordering, and reliable delivery capabilities. Work is in progress on a pure message-passing version of Cashmere-2L. By comparing these two Cashmere-2L versions, we can explore the actual impact of the MC's remote-write and total ordering and the relative benefits of centralized and distributed meta-data in a system with low-latency messages. In other future work, we will also continue to investigate the best use of network address space and also the necessary API functionality and performance.

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.-H. Lim, K. Mackenzie, and D. Yeung. The MIT Alewife Machine: Architecture and Performance. In *Proceedings of the Twenty-Second International Symposium on Computer Architecture*, 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] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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, Cambridge, MA, October 1996.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

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

[15] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. Methodological Considerations and Characterization of the SPLASH-2 Parallel Application Suite. In *Proceedings of the Twenty-Second International Symposium on Computer Architecture*, Santa Margherita Ligure, Italy, June 1995.

[16] 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.