Distributed Shared Memory for New Generation Networks *

Leonidas I. Kontothanassis and Michael L. Scott

Department of Computer Science
University of Rochester
Rochester, NY 14627-0226
{kthanasi,scott}@cs.rochester.edu

Technical Report 578

March 27, 1995

Abstract

Shared memory is widely believed to provide an easier programming model than message passing for expressing parallel algorithms. Distributed Shared Memory (DSM) systems provide the illusion of shared memory on top of standard message passing hardware at very low implementation cost, but provide acceptable performance on only a limited class of applications. In this paper we study the main sources of overhead found in software-coherent, distributed shared-memory systems and argue that recent revolutionary changes in network technology now allow us to design protocols that minimize such overheads and that approach the performance of full hardware coherence. Specifically, we claim that memory-mapped network interfaces that support a global physical address space can greatly improve the performance of DSM systems. To support this claim we study a variety of coherence protocols that can take advantage of the global physical address space and compare their performance with the best known protocol for pure message passing hardware. For the programs in our application suite, protocols taking advantage of the new hardware features improve performance by at least 50% and by as much as an order of magnitude.

*This work was supported in part by NSF Institutional Infrastructure grant no. CDA-8822724 and ONR research grant no. N00014-92-J-1801 (in conjunction with the DARPA Research in Information Science and Technology—High Performance Computing, Software Science and Technology program, ARPA Order no. 8930).

1 Introduction

Large-scale shared-memory multiprocessors can combine the computational power needed for some of the larger problems of science and engineering with the ease of programming required to make effective use of programmer time and effort. Unfortunately, large hardware-based shared-memory machines are substantially more difficult to build than small ones, mainly because of the difficulty of providing a scalable coherence mechanism in hardware. Distributed Shared Memory systems (DSM) provide programmers with the illusion of shared memory on top of message passing hardware while maintaining coherence in software. Unfortunately, the current state of the art in software coherence for networks and multicomputers provides acceptable performance for only a limited class of applications. To make software coherence efficient we need to overcome several fundamental problems with existing DSM emulations [14, 22, 34, 4]:

- DSM systems must interrupt the execution of remote processors every time data that is not present in local memory is required. Such data includes both application data structures and coherence protocol directory information. Synchronization variables pose a particularly serious problem: interrupt-processing overhead can make acquiring a a mutual exclusion on a DSM system several times as expensive as it is on a cache-coherent (CC-NUMA)) machine.
- Due to the high cost of messages, DSM systems try to minimize the number of messages sent. They tend to use centralized barriers (rather than more scalable alternatives) in order to collect and re-distribute coherence information with a minimal number of messages. They also tend to copy entire pages from one processor to another, not only to take advantage of VM support, but also to amortize message-passing overhead over as large a data transfer as possible. This of course works well only for programs whose sharing is coarse-grained. Finally, high message costs make it prohibitively expensive to use directories at home nodes to maintain caching information for pages. The best DSM systems therefore maintain their information in a distributed fashion using interval counters and vector timestamps, which increase protocol processing overhead.
- In order to maximize concurrency in the face of false sharing in page-size coherence blocks, the fastest DSM systems permit multiple copies of a page to be writable simultaneously. The resulting inconsistencies force these systems to compute diffs with older versions of a page in order to merge the changes [4, 14]. Copying and diffing pages is expensive not only in terms of time, but also in terms of storage overhead, cache pollution, and the need to garbage-collect old page copies, write notices, and records of diffs and intervals.

Revolutionary changes in network technology make it possible to address these problems, and to build DSM systems with performance approaching that of full hardware implementations. Several new network designs [3, 9, 33] provide cheap, user-level messages and permit direct access to remote workstation memory, effectively providing the protocol designer with a global physical address space, without the need to interrupt remote processors. These Network Non-Uniform Memory Access (Net-NUMA) systems can easily be built from commodity parts and can follow improvements in microprocessors and other hardware technology closely.

As part of the Cashmere[1] project we have developed a variety of protocols for Net-NUMAs with various levels of hardware support. The basic ideas behind all of the protocols are:

- Use ordinary loads and stores to maintain and access directory information for the coherence protocol, avoiding the need for vector timestamps and the creation and exchange of intervals.
- Use uncached remote references to implement fast synchronization operations both for the applications themselves and for the protocol operations that modify directory data structures.
- Allow multiple writers for concurrency, but avoid the need to keep old copies and compute diffs by using ordinary hardware write-through to a unique (and often remote) main-memory copy of each page.

[1] CaSHMERe stands for Coherence for SHared MEmory aRchitectures and is an ongoing effort to provide an efficient shared memory programming model on modest hardware. Other aspects of the project are discussed in section 5.

Using detailed execution-driven simulation, we have compared these protocols to the version of lazy release consistency used by TreadMarks/ParaNet [14], one of the best existing DSM systems for workstations on a LAN.[2] Assuming identical processor nodes, and networks of equal latency and bandwidth (but not identical interfaces), we see significant performance improvements for the programs in our application suite, ranging from as little as 50% to as much as an order of magnitude.

The rest of the paper is organized as follows. Section 2 discusses the protocols we have designed for Net-NUMAs, the different amount of hardware support required for each, and the intuition for algorithmic and architectural choices. Section 3 describes our experimental methodology and application suite. We present performance results in section 4 and compare our work to other approaches in section 5. We summarize our findings and conclude in section 6.

2 Protocol and Architectural Issues on Modern Networks

In this section we outline a variety of protocols for software cache coherence on potentially large-scale Net-NUMA systems. All of the protocols use virtual memory protection bits to enforce consistency at the granularity of pages. In order to minimize the performance impact of false sharing we allow multiple processors to write a page concurrently, and we use a variant of release consistency [18] to limit coherence operations to synchronization points. We assume that the hardware allows processors to read and write remote locations without interrupting remote processors, ideally by mapping virtual addresses to remote locations and issuing loads and stores, but possibly by issuing special instruction sequences. We exploit the resulting global address space for inexpensive directory maintenance and in-memory merging of dirty data via write-through.

Each shared page in the system has a *home node*, which maintains the master copy of the data, together with directory information. Pages are initially assigned to home nodes in round-robin order, but are moved by the operating system to the first processor to access the page after the program has completed its initialization phase. This simple placement policy has been shown to work very well [20]; it reduces the expected cost of a cache miss by guaranteeing that no page is assigned to a node whose processor does not use it. The directory information for a page includes a list of the current readers and writers, and an indication of the page's *state*, which may be one of the following:

Uncached – No processor is using the page. This is the initial state for all pages.

Shared – One or more processors are using the page read-only.

Dirty – A single processor is using the page read-write.

Weak – Two or more processors are using the page, and at least one is using it read-write.

The state of a page is a property of the system as a whole, not (as in most protocols) the viewpoint of a single processor. Borrowing terminology from PLATINUM [6], the distributed data structure consisting of this information stored at home nodes is called the *coherent map*.

In addition to its portion of the coherent map, each processor also holds a local *weak list* that indicates which of the pages for which there are local mappings are currently in the weak state. When a processor takes a page fault it locks the coherent map entry representing the page on which the fault was taken. It then changes the entry to reflect the new state of the page. If necessary (i.e. if the page has made the transition from shared or dirty to weak), the processor uses uncached remote references to update the weak lists of all processors that have mappings for the page. It then unlocks the entry in the coherent map. On an acquire operation, a processor must remove all mappings and purge from its cache all lines of all pages found in its local weak list. The process of updating another processor's weak list is referred to as posting a *write notice*.

Access costs to the coherent map can be minimized if lock operations are properly designed. If we employ a distributed queue-based lock [21], an uncached read of the coherent map entry can be initiated immediately

[2] One might argue that an object-oriented system such as Midway [34] or Log-Based DSM [8] could provide superior performance, but at the cost of a restricted programming model.

after starting the fetch-and-store operation that retrieves the lock's tail pointer. If the fetch-and-store returns nil (indicating that the lock was free), then the data will arrive right away. The write that releases the lock can subsequently be pipelined immediately after the write of the modified data, and the processor can continue execution. If the lock is held when first requested, then the original fetch-and-store will return the address of the previous processor in line. The queue-based lock algorithm will spin on a local flag, after writing that flag's address into a pointer in the predecessor's memory. When the predecessor finishes its update of the coherent map entry, it can write the data directly into the memory of the spinning processor, and can pipeline immediately afterwards a write that ends the spin. The end result of these optimizations is that the update of a coherent map entry requires little more than three end-to-end message latencies (two before the processor continues execution) in the case of no contention. When contention occurs, little more than *one* message latency is required to pass both the ownership of the lock and the data the lock protects from one processor to the next.

Additional optimizations allow us to postpone the processing of writes until a release synchronization point, and to avoid sending write notices for pages whose sharing behavior stays relatively constant during the execution of a program. A detailed description of the basic protocol design can be found in previous papers [17, 15].

The current paper examines several variants of the basic protocol. Some of the variants assume different levels of hardware support. Others simply represent alternative ways of using identical hardware. Altogether we consider four dimensions of the hardware and software design space that have the potential to significantly alter performance: a) copying pages v. filling cache lines remotely on demand, b) servicing cache misses in hardware v. software, c) writing through to home nodes in hardware v. via extra instructions, and d) merging nearly-contemporaneous writes in hardware v. sending them all through the network individually. The choices affect performance in several ways:

- Copying a page to local memory on an initial miss (detected by a page fault) can be advantageous if all or most of the data will be used before the page is invalidated again. If bandwidth is limited, however, or if the locality exhibited by the application is poor, then copying pages may result in unnecessary overhead.

- Servicing cache misses in hardware requires that the network interface snoop on the local memory bus in order to detect both coherence and capacity/conflict misses. We assume that mapping tables in the interface allow it to determine the remote location that corresponds to a given physical address on the local bus. If the interface does not snoop on ordinary traffic, then we can still service cache misses in software, provided that we have a "hook" to gain control when a miss occurs. We assume here that the hook, suggested by the Wind Tunnel group at the University of Wisconsin [26], is to map shared virtual addresses to local pages that have been set up to generate ECC errors when accessed. On an ECC "miss," the interrupt handler can retrieve the desired line, write it to local memory, and return to the application. If evicted, the data will be written back to local memory; ECC faults will not occur on subsequent capacity/conflict misses.

- A network interface that snoops on the memory bus can forward write-throughs to remote memory. If the writes are also recognized by local memory, then this forwarding constitutes automatic "doubling" of writes, which is perfect for systems that create local copies of pages on initial access faults, or that create them incrementally on cache fills, as described in the preceding bullet. Writes can also be forwarded in software by embedding extra instructions in the program, either via compiler action or by editing the object file [27, 34]. With software forwarding it is not necessary to write through to local memory; the single remote copy retains all of the written data, and capacity or conflict misses will update the local copy via write-back, so that re-loads will still be safe. The number of instructions required to forward a write in software varies from as little as three for a load/store interface with straightforward allocation of virtual addresses, to more than ten for a message-based interface. The extra instructions must be embedded at every potentially shared write in the program text, incurring both time and space overhead. Note that if the hardware writes through to remote memory *and* services cache misses from remote memory, then doubling of writes is not required: there is no local main-memory copy.

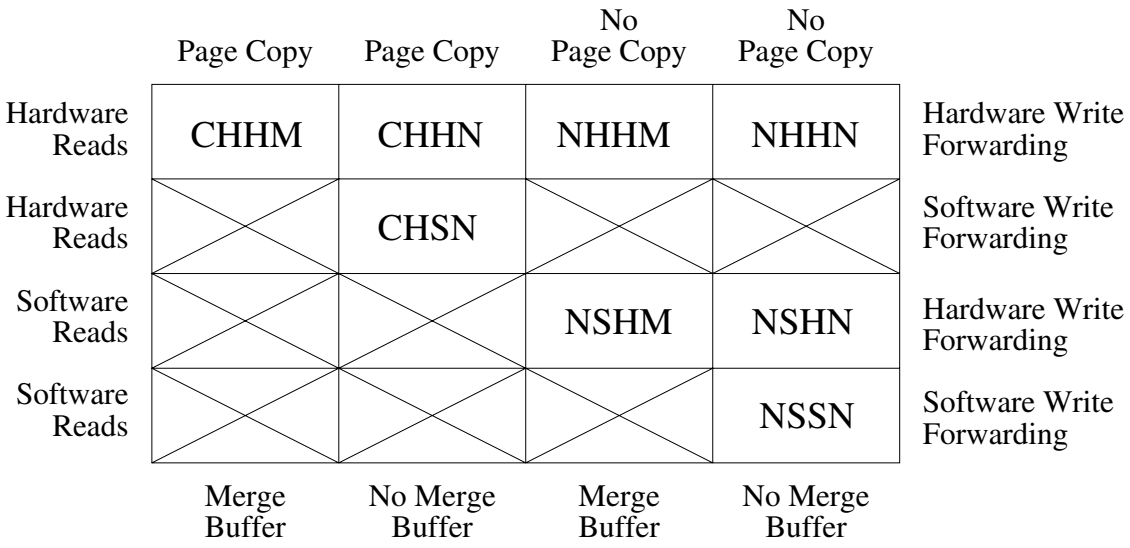

Figure 1: Cashmere protocol variants using different levels of hardware support.

- Merging write buffers [12] are a hardware mechanism that reduces the amount of traffic seen by the memory and network interconnect. Merge buffers retain the last few writes to the same cache line; they forward a line back to memory only when it is displaced by a line being added to the buffer. In order to successfully merge lines to main memory, per word dirty bits are required in the merge buffer. Apart from their hardware cost, merge buffers may hurt performance in some applications by prolonging synchronization release operations (which must flush the buffer through to memory).

Figure 1 depicts the four-dimensional design space. Crossed-out entries in the table imply that the particular protocol/hardware combination is either not feasible or does not make sense from an architectural point of view. For example, it is harder to implement loads from remote memory than it is to implement stores (loads stall the processor, raising issues of fault tolerance, network deadlock, and timing). It is therefore unlikely that a system would service remote cache fills in hardware, but require software intervention on writes. This rules out the last two boxes on the second row. In a similar vein, any protocol that copies whole pages to local memory will be able to read them (fill cache lines) in hardware, so the lower left quadrant of the chart makes no sense. Finally, merge buffers are an option only for hardware writes; the first and third boxes on the second and fourth rows make no sense.

In the remaining boxes, the first letter of the name indicates whether the protocol copies pages or not (C and N respectively). The second and third letters indicate whether caches misses (reads) and forwarded writes, respectively, are handled in hardware (H) or software (S). Finally the fourth letter indicates the existence or absence of a merge buffer (M and N respectively). The NHHM and CHHM boxes (hardware reads and writes, with a merge buffer) in some sense represent "ideal" Net-NUMA machines. The NHHN and CHHN boxes differ from them only in the absence of a merge buffer. All of the boxes other than NHHM and NHHN maintain both local and remote main-memory copies of active data. In the upper left quadrant of the chart (CHxx), a local copy of an entire page is created on an initial page fault (or a page fault following an invalidation). In the lower right quadrant (NSxx), the local copies are created incrementally on demand, in response to ECC faults. Two of the boxes—CHHN and NSHN—resemble systems that could be implemented on a machine like the Princeton Shrimp [3]. Shrimp will double writes in hardware, but cannot fill cache lines from remote locations. Two other boxes—CHSN and NSSN—resemble systems that could be implemented on top of the DEC Memory Channel [9] or HP Hamlyn [33] networks, neither of which will double writes in hardware or fill cache lines from remote locations. The Memory Channel and Hamlyn interfaces differ in that the former can perform (undoubled) stores to remote locations; it is therefore able to double writes in software with fewer instructions.

As noted at the beginning of this section, we assume that all systems are capable of reading remote locations, either with ordinary loads or via special instruction sequences. As of this writing, Hamlyn provides this capability for small-sized reads and DEC is considering it for future generations of the hardware [10]. To read a remote location on the Memory Channel, one must interrupt the remote processor, or require it to poll. Issuing remote interrupts to access directory information would clearly be prohibitively expensive, but if the individual nodes on the network are themselves small multiprocessors, one could consider dedicating one processor per node to polling for read requests.[3] Alternatively, one could consider a coherence protocol that copies pages, writes through to remote locations via software doubling, but does not employ directories. Researchers at the University of Colorado are currently considering such a protocol; rather than track the users of a page, they simply invalidate all local copies of shared data on every acquire operation [11].

3 Experimental Methodology

We use execution-driven simulation to simulate a network of 64 workstations connected with a dedicated, memory-mapped network interface. Our simulator consists of two parts: a front end, Mint [31], that simulates the execution of the processors, and a back end that simulates the memory system. The front end calls the back end on every data reference (instruction fetches are assumed to always be cache hits). The back end decides which processors block waiting for memory and which continue execution. Since the decision is made on-line, the back end affects the timing of the front end, so that the interleaving of instructions across processors depends on the behavior of the memory system and control flow within a processor can change as a result of the timing of memory references.

The front end is the same in all our experiments. It implements the MIPS II instruction set. Multiple modules in the back end allow us to explore the protocol design space and to evaluate systems with different amounts of hardware support and different architectural parameters. The back end modules are quite detailed, with finite-size caches, TLB behavior, full protocol emulation, network transfer costs including contention effects, and memory access costs including contention effects. Table 1 summarizes the default parameters used in our simulations.

Some of the transactions required by our coherence protocols require a collection of the operations shown in table 1 and therefore incur the aggregate cost of their constituents. For example a page fault on a read to an unmapped page consists of the following: a) a TLB fault service, b) a processor interrupt caused by the absence of read rights, c) a coherent map entry lock acquisition, and d) a coherent map entry modification followed by the lock release. Lock acquisition itself requires traversing the network and accessing the memory module where the lock is located. For a one μsec network latency and a 200Mhz processor the cost of accessing the lock (and the data it protects) is approximately 400 cycles (200 cycles each way). The total cost for the above transaction would then be $24 + 140 + 400 + 160 = 724$ cycles.

3.1 Workload

We report results for nine parallel programs. We have run each application on the largest input size that could be simulated in a reasonable amount of time and that provided good scalability for the 64-processor configuration we simulate. Five of the programs are best described as application kernels: `Gauss`, `sor`, `sor_l`, `mgrid`, and `fft`. The remaining are larger applications: `mp3d`, `water`, `em3d`, and `appbt`. The kernels are local creations.

`Gauss` performs Gaussian elimination without pivoting on a 448×448 matrix. `Sor` computes the steady state temperature of a metal sheet using a banded parallelization of red-black successive over-relaxation on a 640×640 grid. `Sor_l` is an alternative implementation of the `sor` program that uses locks instead of barriers

[3] In between polls, such a processor could be put to other uses as well, such as filling unused pages with "bad" ECC bits, in anticipation of using them for on-demand remote cache fills, or tracking remote references in the manner of a hardware stream buffer [23], in order to prefetch data being accessed at a regular stride. It is not clear whether dedicating a processor to this sort of work would be worthwhile; we consider it a topic for future research.

System Constant Name	Default Value
TLB size	128 entries
TLB fault service time	24 cycles
ECC interrupts	250 cycles
All other interrupts	140 cycles
Coherent map modification	160 cycles
Memory setup time	20 cycles
Memory bandwidth	2bytes/cycle
Page size	4K bytes
Total cache per processor	128K bytes
Cache line size	64 bytes
Network path width	16 bits (bidirectional)
Messaging software overhead	150 cycles
Switch latency	2 cycles
Wire latency	1 cycle
List processing	6 cycles/element
Page twinning	5 cycles/word
Diff application and creation	7 cycles/word
Software write doubling (overhead)	3 cycles/write

Table 1: Default values for system parameters

as its synchronization mechanism. `Mgrid` is a simplified shared-memory version of the multigrid kernel from the NAS Parallel Benchmarks [2]. It performs a more elaborate over-relaxation using multi-grid techniques to compute an approximate solution to the Poisson equation on the unit cube. We simulated 2 iterations, with 5 relaxation steps on each grid, and grid sizes of $64 \times 64 \times 32$. `Fft` computes a one-dimensional FFT on a 65536-element array of complex numbers, using the algorithm described by Akl [1].

`Mp3d` and `water` are part of the SPLASH suite [29]. `Mp3d` is a wind-tunnel airflow simulation. We simulated 40000 particles for 10 steps in our studies. `Water` is a molecular dynamics simulation computing inter- and intra-molecule forces for a set of water molecules. We used 256 molecules and 3 times steps. `Em3d` [7] simulates electromagnetic wave propagation through 3D objects. We simulate 65536 electric and magnetic nodes connected randomly, with a 5% probability that neighboring nodes reside in different processors. We simulate the interactions between nodes for 10 iterations. Finally `appbt` is from the NASA parallel benchmarks suite [2]. It computes an approximation to Navier-Stokes equations. It was translated to shared memory from the original message-based form by Doug Burger and Sanjay Mehta at the University of Wisconsin. We have modified some of the applications in order to improve their locality properties and to ensure that the large size of the coherence block does not create excessive amounts of false sharing. Most of our changes were mechanical and took no more than a few hours of programming effort.

Due to simulation constraints our input data sizes for all programs are smaller than what would be run on a real machine. We have also chosen smaller caches than are common on real machines, in order to capture the effect of capacity and conflict misses. Since we still observe reasonable scalability for most of our applications[4] we believe that the data set sizes do not compromise our results.

4 Results

Our principal goal is to determine the types of protocols that one should run on new generation networks. We start in section 4.1 by evaluating the tradeoffs between different software protocols on machines with an "ideal"

[4]Mp3d does not scale to 64 processor but we use it as a stress test to compare the performance of different coherence mechanisms.

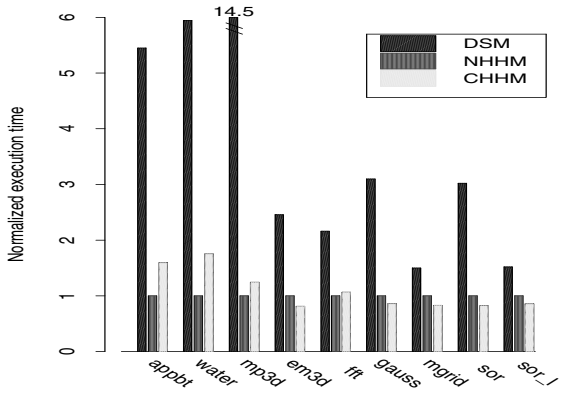

Figure 2: Normalized execution time for DSM and Cashmere protocols on 64 processors.

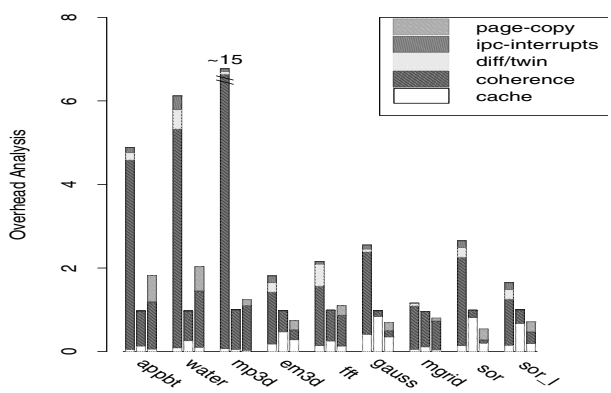

Figure 3: Overhead analysis for DSM and Cashmere protocols on 64 processors.

network interface. We then proceed in section 4.2 to study systems with simpler interfaces, quantifying the performance impact of hardware support for merging writes, doubling writes, and filling cache lines remotely. Finally in section 4.3 we look at the impact of network latency and bandwidth on the relative performance of protocols.

4.1 Protocol Alternatives

This section compares the two "ideal" Cashmere protocols—CHHM and NHHM—of section 2) to a state-of-the-art DSM system—TreadMarks/ParaNet [14]—designed for message-passing hardware. All simulations employ identical processors, write-through caches (except for the DSM system which does not need them), buses, and memories, and the same network bandwidth and latency. The Cashmere protocols further assume (and exploit) the ability to double write-throughs in hardware (with a write-merge buffer), and to fill cache lines remotely. Results appear in figure 2. Running time is normalized with respect to that of the no-page-copy (NHHM) protocol. For all programs in our application suite, the Cashmere protocols outperform the DSM system by at least 50% and in some cases almost by an order of magnitude. The choice between copying a page on a page fault or simply mapping the page remotely and allowing the cache to fetch lines on demand is less obvious. For some applications page copying proves inferior (often by a significant margin) while for others it improves performance by as much as 20%. We believe that given the ability to count cache misses (e.g. in special hardware or an ECC fault handler), an on-line hybrid protocol could choose the appropriate strategy for each page, and would probably out-perform both CHHM and NHHM; again this is a subject for future research.

The greatest performance benefits for Cashmere with respect to DSM are realized for the more complex applications: `appbt`, `mp3d` and `water`. This is to a certain extent expected behavior. Large applications often have more complex sharing patterns, and as a result the importance of the coherence protocol is magnified. In addition, these applications exhibit a high degree of synchronization, which increases the frequency with which coherence information must be exchanged. Fetching cache lines on demand, as in NHHM, is therefore better than copying whole pages, as in CHHM: page size transfers waste bandwidth, since the high frequency of synchronization forces pages to be invalidated before all of their contents have been used by the local processor. Page transfers also occupy memory and network resources for longer periods of time, increasing the level of contention.

`Sor` and `em3d` have very regular sharing patterns. Coherence information need only be exchanged between neighbors when reaching a barrier. Unfortunately, the natural implementation of barriers for a system such as TreadMarks exchanges information much more widely. To minimize barrier latency on modest numbers of processors with high message-passing overhead, TreadMarks employs a central barrier manager that gathers coherence information from, and disseminates it to, all participating processors. This centralized implementation limits scalability. It also forces each processor to examine the write notices of every other processor—not just its neighbors—leading to greater protocol processing overhead than is necessary based on the sharing patterns of the programs. The directory-based Cashmere protocols do not couple coherence and synchronization as tightly; they make a processor pay a coherence penalty only for the pages it cares about.

In order to assess the impact of barrier synchronization on the relative performance of the protocols we have produced a lock-based version of `sor` called `sor_l`. Instead of using a barrier, `sor_l` protects its boundary rows with locks, which processors must acquire before they can use the rows in their computation. As can be seen from the graphs the lock-based version shows a much smaller difference in performance across protocols. The additional time for DSM comes from computing diffs and twins for pages and servicing interprocessor interrupts. The source code differences between `sor` and `sor_l` are non-trivial: a single line of code (the barrier) in the former corresponds to 54 lines of lock acquisitions and releases in the latter. For `em3d`, in which sharing occurs between individual electric or magnetic nodes, associating locks with each node would be prohibitively expensive, and it is not obvious how to combine nodes together so that they can be covered by a single lock. Both `sor` (and its lock-based version) and `em3d` work better with page copying (CHHM) than with remote cache fills (NHHM). Rows are page aligned in `sor`, and nodes that belong to a processor are page aligned in `em3d`. Since both applications exhibit excellent spatial locality, they benefit from the low local cache miss penalty when pages are copied to local memory.

`Mgrid` also exhibits limited sharing and uses barrier synchronization. The three-dimensional structure of the grid however makes sharing more wide-spread than it is in `sor`. As a result the centralized-manager barrier approach of TreadMarks does not incur as much additional overhead and the Cashmere protocols reduce running time by less than half. In a similar vein, `fft` exhibits limited true sharing among different processors for every phase (the distance between paired elements decreases for each phase). The inefficiency of centralized barriers is slightly greater than in `fft`, but the Cashmere protocols cut program running time by only slightly more than half. Once again page copying proves valuable due to the good spatial locality of the applications.

The last application, `gauss`, is another lock-based program. Here running time under DSM is more than 3 times as long as with the Cashmere protocols. The main reason is that locks are used as flags to indicate when a row is available to serve as the pivot row. Without directories, lock acquisitions and releases in `gauss` must induce communication between processors. In Cashmere lock acquisitions are essentially free, since no write notices will be present for any shared page (no shared page is ever written by a remote processor in this application). Lock releases are slightly more expensive since data has to be flushed to main memory and directory information has to be updated, but flushing is overlapped with computation due to write-through, and directory update is cheap since no coherence actions need to be taken.

Figure 3 presents a breakdown of protocol overhead into its principal components: cache stall time, protocol processing overhead, time spent computing diffs, time spent processing interprocessor interrupts, and time spent copying pages. For DSM part of the cache stall time is encompassed in protocol processing time: since caches are kept consistent by applying diffs, there is no easy way to detect coherence related misses. Applying a diff will bring the new data into the cache immediately. As a consequence the measured cache-stall time in DSM is solely due to eviction and cold-start misses. Also, the protocol processing overhead includes any time processors spend stalled waiting for a synchronization variable. Due to the nature of the protocols it is hard to distinguish how much synchronization stall time is due to protocol processing and how much due to application-specific synchronization.

The relative heights of the bars may not agree between figures 2 and 3 because the former pertains to the critical path of the computation, while the latter provides totals over all processors for the duration of execution. Aggregate costs for the overhead components can be higher but critical path length can be shorter if some of the overhead work is done in parallel. As can be seen in figure 3 the coherence processing component

Application	Reduction in Running Time	
	8 processors	64 processors
appbt	22.6%	81.6%
water	31.3%	83.2%
mp3d	89.7%	93.2%
em3d	22.3%	66.9%
fft	11.4%	53.8%
gauss	46.8%	72.3%
mgrid	17.8%	44.8%
sor	11.7%	72.8%
sor_l	10.9%	43.7%

Table 2: Performance advantage of Cashmere protocols over traditional DSM on 8 and 64 processors.

is greatly reduced for the Cashmere protocols. The cost of servicing interprocessor interrupts and producing diffs and twins is also significant for DSM in many cases.

We have also run experiments with smaller numbers of processors in order to establish the impact of scalability on the relative performance of the DSM and Cashmere protocols. We discovered that for small numbers of processors the Cashmere protocols maintain only a modest performance advantage over DSM. For smaller systems, protocol overhead is significantly reduced; the Cashmere performance advantage stems mainly from the elimination of diff generation and interprocessor interrupt overhead. Table 2 shows the performance advantage (as percent reduction in running time) of the best Cashmere protocol over a Treadmarks-like DSM system on 8 and 64 processors for our application suite.

4.2 Relaxing Hardware Requirements

Current network implementations do not provide all the hardware support required for the best versions of our protocols. This is presumably due to a difference between the original goals of the network designers (fast user-level messages) and our goals (efficient software-coherent shared memory). In this section we consider the performance impact of the additional hardware on the relative performance of the DSM protocols. In particular we evaluate:

- The performance impact of merge buffers. Merge Buffers reduce the bandwidth requirements to the memory and network interconnect and can improve performance for applications that require large amounts of bandwidth.
- The performance impact of doubling writes in hardware. We compare hardware doubling to two types of software doubling, both of which rely on the ability to edit an executable and insert additional code after each write instruction. Assuming a memory-mapped interface (as on the DEC Memory Channel) that allows access to remote memory with simple write instructions, the minimal sequence for doubling a write requires something on the order of three additional instructions (with the exact number depending on instruction set details). If access to remote memory is done via fast messages (as in HP's Hamlyn) then the minimal sequence requires something on the order of twelve additional instructions. We have not attempted to capture any effects due to increased register pressure or re-scheduling of instructions in modified code.
- The performance impact of servicing cache misses in hardware. This option only affects the NHHM and NHHN protocols. In the absence of hardware support we can still service cache misses in software, provided that we have a "hook" to gain control when a miss occurs. As noted in section 2, one possibility is to map shared virtual addresses to local pages the have been set up to generate ECC errors when accessed. The ECC fault handler can then retrieve the desired line, write it to local memory, and return

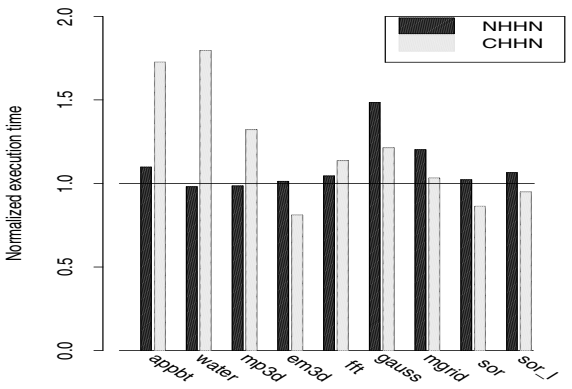

Figure 4: Normalized execution time for Cashmere protocols without a merge buffer (64 processors).

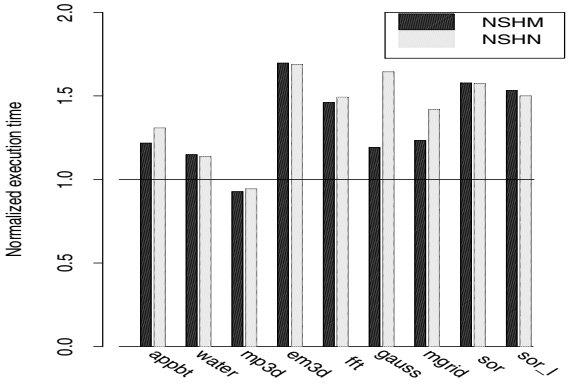

Figure 5: Normalized execution time for Cashmere protocols with software-mediated cache fills (64 processors).

to the application. This approach has the benefit that a cache miss does not stall the processor, thus preserving the autonomy of each workstation in the system. Should a remote node fail to produce an answer we can use a software timeout mechanism to gracefully end the computation and allow the workstation to proceed with other jobs. Furthermore the cost of the ECC fault has to be paid only for the initial cache miss. Subsequent cache misses due to capacity and conflict evictions can be serviced from local memory, to which evicted lines are written back.

Figure 4 shows the performance impact of merge buffers on both the NHHM and CHHM protocols. The Cashmere protocols shown in the figure are identical to those of figure 2, except for the absence of the merge buffer. Running time is once again normalized with respect to the NHHM protocol. For the architecture we have simulated we find that there is enough bandwidth to tolerate the increased traffic of plain write-through for all but two applications. `Gauss` and `mgrid` have writes to shared locations inside their inner loops. Those writes are frequent and have excellent spatial locality. The merge buffer is ideally suited to servicing such writes and reduces main memory traffic significantly.

Support for hardware reads does not seem to be as important. Six out of nine applications favor page transfers to start with. As expected the non-page-copying protocols behave even worse when cache misses to remote locations incur trap overhead. The one exception to this observation is `mp3d`. Closer inspection reveals that approximately two thirds of the application's misses are due to conflicts in the cache. The NSHM and NSHN protocols will service these misses from local memory while the all-hardware approach will have to pay the cost of a remote transfer. As a result the NSHM and NSHN protocols show a small performance improvement over their all-hardware counterparts. Normalized execution time for the protocols that do not assume hardware support for reads is shown in figure 5.

The final dimension of interest is the mechanism used to forward writes to a remote main-memory page. Figures 6 and 7 show the normalized running time of the Cashmere protocols when writes must be forwarded to the home node in software. For the sake of clarity we have separated the non-copying and page copying versions of the protocols across the two graphs. We consider two cases of software support for writes. A memory-mapped network interface requires an additional three instructions in order to forward the write to the home node, while a message-based interface increases this overhead to twelve instructions per shared write. For all applications, duplication of writes in software adds a substantial amount to program running time. For the message-passing interface, the overhead is high enough in some cases to eliminate the performance

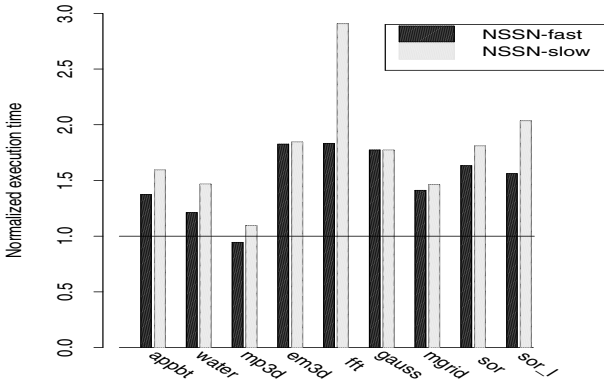

Figure 6: Normalized execution time for Cashmere protocols with software doubling of writes (64 processors; no page copy).

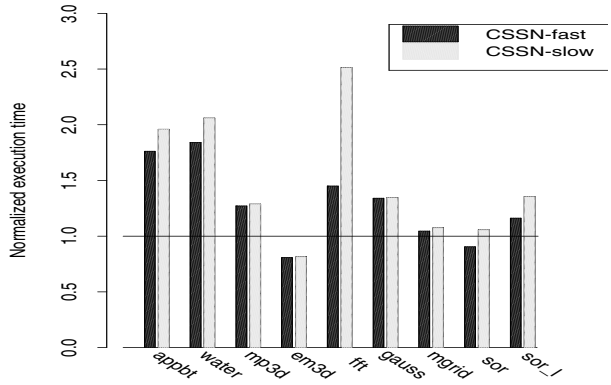

Figure 7: Normalized execution time for Cashmere protocols with software doubling of writes (64 processors; page copy).

advantage over DSM. Both types of network interfaces are available on the market. Our results indicate that the memory-mapped interface is significantly better suited to efficient shared memory emulations.

4.3 Impact of Latency and Bandwidth

Though not as important as the functional changes in network interfaces that allow us to access remote memory directly, ongoing improvements in network bandwidth and latency also play a significant role in the success of the Cashmere protocols. High bandwidth is important for write-through; low latency is important for cheap directory access. We expect bandwidth to continue to increase (bandwidths of 1G byte/sec are projected for the short- and medium-term future), but latency trends are less clear. It may be possible to achieve improvements in latency by further reducing the software overhead of messages, but most of the trends seem to be going in the opposite direction. As processors get faster, network latency (in processor cycles) will increase. In this section we evaluate the impact of changes in latency and bandwidth on the performance of the DSM and Cashmere protocols. For the sake of clarity in the graphs, we present results for only four applications: `appbt`, `water`, `gauss`, and `sor_l`. These applications represent important points in the application spectrum. Two are large programs, with complex sharing patterns and frequent synchronization, while the remaining two are smaller kernels, with simpler sharing patterns and lock-based synchronization. The kernels exhibit less of a performance advantage when run on the Cashmere protocols. The remaining applications exhibit qualitatively similar behavior.

Figures 8 and 9 show the performance impact of network latency on the relative performance of the DSM, NHHM, and CHHM protocols. The measures taken by TreadMarks to minimize the number of messages exchanged make the DSM protocol largely insensitive to changes in network latency. The Cashmere protocols are more affected by latency variations. The NHHM protocol in particular suffers severe performance degradation when forced to move cache lines at very high latency. For a 200 MHz processor a 10μsec network delay implies a minimum of 4,000 cycles to fetch a cache line. Such high cache miss penalties severely limit the performance of the system. The page copying protocol (CHHM) fares better, since it pays the high transfer penalty only on page fetches and satisfies cache misses from local memory. Both systems however must still pay for write-throughs and directory access, and suffer when latency is high.

We have also collected results (not shown) on the impact of latency for protocols that work with suboptimal hardware. We found that the impact of doubling writes and of mediating cache fills in software decreases as

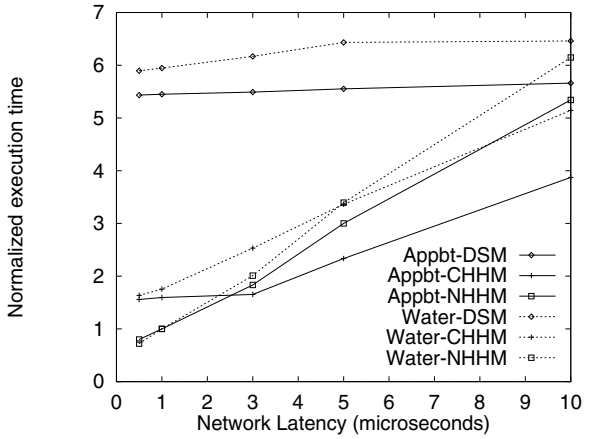

Figure 8: Normalized execution time for DSM and Cashmere protocols under different network latencies (64 processors).

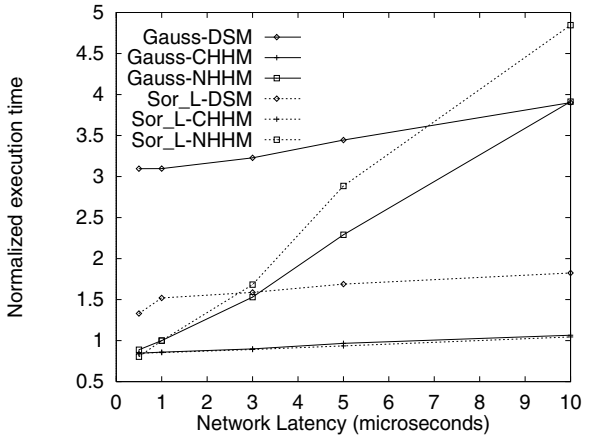

Figure 9: Normalized execution time for DSM and Cashmere protocols under different network latencies (64 processors).

latency increases, and that the performance gap between the ideal and sub-optimal protocols narrows. This makes sense: for high latencies the additional overhead of ECC faults for reads and the software overhead for doubling writes is only a small fraction of the total cost for the operation.

The impact of network bandwidth on the relative performance of the protocols is depicted in figures 10 and 11. Both the DSM and CHHM (to a lesser degree) protocols are sensitive to bandwidth variations and suffer significant performance degradation when bandwidth is limited. The reason for this sensitivity is the large size of some of the messages. Large messages exacerbate contention effects since they result in high occupancy times for network and memory resources. As bandwidth increases both protocols improve markedly. CHHM approaches or surpasses the performance of NHHM, while DSM tails off sooner; when enough bandwidth is available protocol processing overhead becomes the limiting factor. The NHHM protocol, which transfers cache lines only, is largely bandwidth-insensitive.

For the protocols that operate with sub-optimal hardware we have found that the absence of a write-merge buffer has a significant impact on performance at low bandwidth levels. Increasing the amount of bandwidth reduces the importance of merge buffers. Software-mediated cache fills and software doubling of writes cannot exploit the additional bandwidth; performing these operations in software increases their latency but leaves bandwidth requirements constant. As in the latency graphs we have omitted results for the remaining applications to avoid cluttering the figures. The applications shown here are representative of the whole application suite.

5 Related Work

Our work is closely related to that of Petersen and Li [24, 25]; we both use the notion of weak pages, and purge caches on acquire operations. The main difference is scalability: we distribute the coherent map and weak list, distinguish between "safe" and "unsafe" pages (those that are unlikely or likely, respectively, to be weak [17]), check the weak list only for unsafe pages mapped by the local processor, and multicast write notices for safe pages that turn out to be weak. Our work resembles Munin [4] and lazy release consistency [13] in its use of delayed write notices, but we take advantage of the globally-accessible physical address space for cache fills (in the Nxxx protocols) and for access to the coherent map and the local weak lists. We have also presented protocol variants [17] appropriate to a tightly-coupled NUMA multiprocessor such as the BBN TC2000 or the Cray T3D, with a globally-accessible physical memory but without hardware coherence. In such systems the lower latency of memory accesses allows the use of uncached remote references as an alternative to caching and provides us with the ability to tolerate small amounts of fine grain sharing.

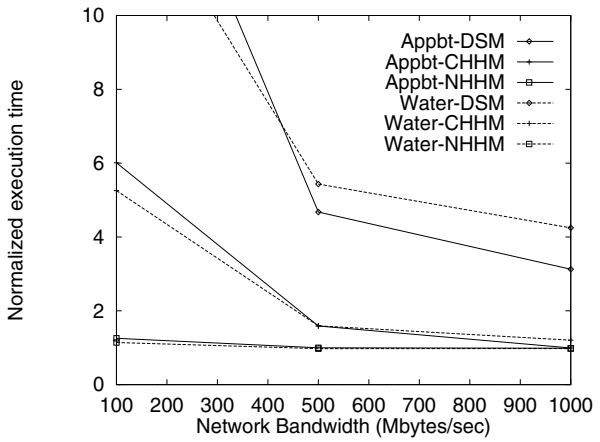

Figure 10: Normalized execution time for DSM and Cashmere protocols under different network bandwidths (64 processors).

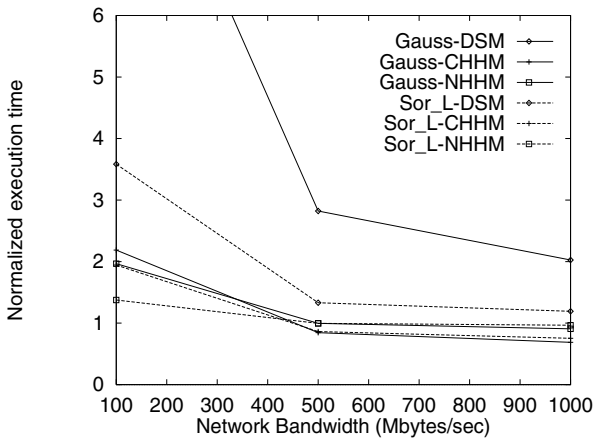

Figure 11: Normalized execution time for DSM and Cashmere protocols under different network bandwidths (64 processors).

On the hardware side our work bears a resemblance to the Stanford Dash project [19] in the use of a relaxed consistency model, and to the Georgia Tech Beehive project [28] in the use of relaxed consistency and per-word dirty bits for successful merging of inconsistent cache lines. Both these systems use their extra hardware to allow coherence messages to propagate in the background of computation (possibly at the expense of extra coherence traffic) in order to avoid a higher waiting penalty at synchronization operations.

Thekkath et al. have proposed using a network that provides remote memory access to better structure distributed and RPC-based systems [30]. The Blizzard system [27] at the University of Wisconsin uses ECC faults to provide fine-grain coherence with a sequential consistency memory model. In contrast we use standard address translation hardware to provide coherence support and use ECC faults to trigger data transfers (in the xxSx protocols) only. For the programs in our application suite, the coarse coherence granularity does not affect performance adversely [15]. Furthermore we have adopted a lazy protocol that allows multiple writers. In past work we have shown that very small coherence blocks for such protocols can be detrimental to performance [16].

The software-doubled writes and software-mediated reads of the xSxx and xxSx protocols are reminiscent of the `put` and `get` operations of active message systems [7, 32]. The use of object editing tools to insert coherence related code that maintains dirty bits or checks for write permission to a coherence block has also been proposed by the Blizzard [27] and Midway [34] systems. Our use is different in that we simply need to duplicate writes. As a consequence we need as little as three additional instructions for each shared memory write. Maintaining coherence related information increases this overhead to about eleven instructions per shared-memory write.

Coherence for distributed memory with per-processor caches can also be maintained entirely by a compiler [5]. Under this approach the compiler inserts the appropriate cache flush and invalidation instructions in the code, to enforce data consistency. The static nature of the approach, however, and the difficulty of determining access patterns for arbitrary programs, often dictates conservative decisions that result in higher miss rates and reduced performance.

6 Conclusions

In this paper we have shown that recent changes in network technology make it possible to achieve dramatic performance improvements in software-based shared memory emulations. We have described a set of software coherence protocols (known as Cashmere) that take advantage of Net-NUMA hardware to improve performance over traditional DSM systems by as much as an order of magnitude. The key hardware innovation is a

single physical address space: it allows remote data accesses (cache fills or page copies) to be serviced in hardware, permits protocol operations to access remote directory information with relatively little overhead, and eliminates the need to compute diffs in order to merge inconsistent writable copies of a page; ordinary write-through merges updates into a single main memory copy.

Recent commercial experience suggests that the complexity and cost of Net-NUMA hardware is much closer to that of message-based (e.g. ATM style) network interfaces than it is to that of hardware cache coherence. At the same time, previous work has shown the performance of Cashmere protocols to be within a few percent of the all-hardware approach [17]. Together, these findings raise the prospect of practical, shared-memory supercomputing on networks of commodity workstations.

We are currently pursuing the design of a protocol that will dynamically choose between cache line and page transfers based on the access patterns of the application. When cache misses are serviced in software, as in the ECC-fault approach, we have the necessary hooks to decide which policy to use on-line. Furthermore we expect that for fine-grain shared data structures it may be beneficial to disable caching and use uncached references instead of remote cache fills or page copying. Finally we are actively pursuing the design of annotations that a compiler can use to provide hints to the coherence system, allowing it to customize its actions to the sharing patterns of individual data structures.

Acknowledgements

Our thanks to Ricardo Bianchini and Alex Poulos for their help and support with this work. Thanks also to Pete Keleher for answering numerous question on the TreadMarks implementation, and to Rick Gillett for providing us with information on the Memory Channel.

References

[1] S. G. Akl. *The Design and Analysis of Parallel Algorithms.* Prentice Hall, Inc., Englewood Cliffs, NJ, 1989.

[2] D. Bailey, J. Barton, T. Lasinski, and H. Simon. The NAS Parallel Benchmarks. Report RNR-91-002, NASA Ames Research Center, January 1991.

[3] M. Blumrich, K. Li, R. Alpert, C. Dubnicki, E. Felten, and J. Sandberg. Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer. In *Proceedings of the Twenty-First International Symposium on Computer Architecture*, pages 142–153, Chicago, IL, April 1994.

[4] 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.

[5] H. Cheong and A. V. Veidenbaum. Compiler-Directed Cache Management in Multiprocessors. *Computer*, 23(6):39–47, June 1990.

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

[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] M. J. Feeley, J. S. Chase, V. R. Narasayya, and H. M. Levy. Log-Based Distributed Shared Memory. In *Proceedings of the First Symposium on Operating Systems Design and Implementation*, Monterey, CA, November 1994.

[9] R. Gillett. Memory Channel: An Optimized Cluster Interconnect. *Digital Technical Journal*, Fall 1995 (to appear). Digital Equipment Corporation, Maynard, MA.

[10] R. Gillett. Personal communication, Digital Equipment Corporation, Maynard, MA, March 1995.

[11] D. Grunwald. Personal communication, University of Colorado at Boulder, January 1995.

[12] N. Jouppi. Cache Write Policies and Performance. In *Proceedings of the Twentieth International Symposium on Computer Architecture*, San Diego, CA, May 1993.

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

[14] P. Keleher, A. L. Cox, S. Dwarkadas, and W. Zwaenepoel. ParaNet: Distributed Shared Memory on Standard Workstations and Operating Systems. In *Proceedings of the USENIX Winter '94 Technical Conference*, San Francisco, CA, January 1994.

[15] L. I. Kontothanassis and M. L. Scott. High Performance Software Coherence for Current and Future Architectures. *Journal of Parallel and Distributed Computing* (to appear).

[16] L. I. Kontothanassis, M. L. Scott, and R. Bianchini. Lazy Release Consistency for Hardware-Coherent Multiprocessors. TR 547, Computer Science Department, University of Rochester, December 1994.

[17] L. I. Kontothanassis and M. L. Scott. Software Cache Coherence for Large Scale Multiprocessors. In *Proceedings of the First International Symposium on High Performance Computer Architecture*, pages 286–295, Raleigh, NC, January 1995. Earlier version available as TR 513, Computer Science Department, University of Rochester, March 1994.

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

[19] D. Lenoski, J. Laudon, K. Gharachorloo, W. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. S. Lam. The Stanford Dash Multiprocessor. *Computer*, 25(3):63–79, March 1992.

[20] M. Marchetti, L. Kontothanassis, R. Bianchini, and M. L. Scott. Using Simple Page Placement Policies to Reduce the Cost of Cache Fills in Coherent Shared-Memory Systems. In *Proceedings of the Ninth International Parallel Processing Symposium*, Santa Barbata, CA, April 1995. Earlier version available as TR 535, Computer Science Department, University of Rochester, September 1994.

[21] J. M. Mellor-Crummey and M. L. Scott. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, February 1991.

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

[23] S. Palacharla and R. E. Kessler. Evaluating Stream Buffers as a Secondary Cache Replacement. In *Proceedings of the Twenty-First International Symposium on Computer Architecture*, Chicago, IL, April 1994.

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

[25] K. Petersen and K. Li. An Evaluation of Multiprocessor Cache Coherence Based on Virtual Memory Support. In *Proceedings of the Eighth International Parallel Processing Symposium*, pages 158–164, Cancun, Mexico, April 1994.

[26] S. K. Reinhardt, M. D. Hill, J. R. Larus, A. R. Lebeck, J. C. Lewis, and D. A. Wood. The Wisconsin Wind Tunnel: Virtual Prototyping of Parallel Computers. In *Proceedings of the 1993 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, Santa Clara, CA, May 1993.

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

[28] G. Shah and U. Ramachandran. Towards Exploiting the Architectural Features of Beehive. GIT-CC-91/51, College of Computing, Georgia Institute of Technology, November 1991.

[29] 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.

[30] C. A. Thekkath, H. M. Levy, and E. D. Lazowska. Separating Data and Control Transfer in Distributed Operating Systems. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 2–11, San Jose, CA, October 1994.

[31] J. E. Veenstra and R. J. Fowler. MINT: A Front End for Efficient Simulation of Shared-Memory Multiprocessors. In *Proceedings of the Second International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS '94)*, pages 201–207, Durham, NC, January – February 1994.

[32] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active Messages: a Mechanism for Integrated Communication and Computation. In *Proceedings of the Nineteenth International Symposium on Computer Architecture*, pages 256–266, Gold Coast, Australia, May 1992.

[33] J. Wilkes. Hamlyn — An Interface for Sender-Based Communications. Technical Report HPL-OSR-92-13, Hewllett Packard Laboratories, Palo Alto, CA, November 1992.

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