HPCA '96

Using Memory-Mapped Network Interfaces to Improve the Performance of Distributed Shared Memory *

Leonidas I. Kontothanassis and Michael L. Scott

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

In 2nd International Conference on High Performance Computer Architecture

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 for only a limited class of applications. We argue that the principal sources of overhead in DSM systems can be dramatically reduced with modest amounts of hardware support (substantially less than is required for hardware cache coherence). Specifically, we present and evaluate a family of protocols designed to exploit hardware support for a global, but non-coherent, physical address space. We consider systems both with and without remote cache fills, fine-grain access faults, "doubled" writes to local and remote memory, and merging write buffers. We also consider varying levels of latency and bandwidth. We evaluate our protocols using execution driven simulation, comparing them to each other and to a state-of-the-art protocol for traditional message-based networks. For the programs in our application suite, protocols taking advantage of the global address space improve performance by a minimum of 50% and sometimes by as much as an order of magnitude.

1 Introduction

Distributed Shared Memory systems (DSM) provide programmers with the illusion of shared memory on top of message passing hardware while maintaining coherence in software [4, 14, 22, 35]. Unfortunately, the current state of the art in software coherence for networks and multicomputers provides acceptable performance for only a limited class of applications. We argue that this limitation is principally the result of inadequate hardware support. To make software coherence efficient we need to overcome several fundamental problems:

- 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 and coherence protocol data structures. Synchronization variables pose a particularly serious problem: interrupt-processing overhead can make acquiring a mutual exclusion lock 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 now make it possible to address these problems, and to build software coherent systems with performance approaching that of full hardware implementations. Several new workstation networks, including the Princeton Shrimp [3], DEC Memory Channel [8], and HP Hamlyn [33], allow a processor to access the memory of other nodes (subject to VM protection) without trapping into the kernel or interrupting a re-

*This work was supported in part by NSF Infrastructure grants nos. CDA–88-22724 and CDA–94-01142, and ONR research grant no. N00014-92-J-1801 (in conjunction with the ARPA Research in Information Science and Technology—High Performance Computing, Software Science and Technology program, ARPA Order no. 8930).

mote processor. These Non-Cache-Coherent Non-Uniform Memory Access (NCC-NUMA) systems effectively provide the protocol designer with a global physical address space. In comparison to large-scale cache-coherent multiprocessors, NCC-NUMA systems are relatively easy to build, and can follow improvements in microprocessors and other hardware technology closely. As part of the CASHMERe[1] project we have shown that a lazy software protocol on a tightly-coupled NCC-NUMA system can rival the performance of traditional hardware coherence [15, 17]. In this paper we examine the other end of the architectural spectrum, quantifying the performance advantage of software coherence on NCC-NUMA hardware with respect to more traditional DSM systems for message-passing hardware.

Using detailed execution-driven simulation, we have compared the performance of NCC-NUMA systems running our protocols to the version of lazy release consistency used by TreadMarks [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.

We are currently in the process of building a 32-processor Cashmere prototype in collaboration with Digital Equipment Corporation. The implementation is based on eight 4-processor DEC 4/233 mulitprocessors and a Memory Channel network. The prototype will provide a benchmark against which to compare our current results, and will allow us to address implementation details that are difficult to examine in the context of simulation. We expect the prototype to confirm that Cashmere protocols on NCC-NUMA hardware lie at or near the knee of the price-performance curve for shared-memory parallel systems.

The rest of the paper is organized as follows. Section 2 discusses network and processor features as well as protocol options that affect the performance of software coherence on NCC-NUMA hardware. 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 Architectural and Protocol Issues on Modern Networks

In this section we describe a variety of software coherence protocols. For simplicity's sake, we assume both here and in the simulations that each node contains a single processor (the protocols can easily be extended to accommodate multiprocessor nodes). We start in section 2.1 by describing TreadMarks and Lazy Release Consistency, the protocol of choice for message passing multicomputers. We then proceed in section 2.2 to describe our protocol for NCC-NUMA machines. Finally in section 2.3 we discuss several variants of NCC-NUMA hardware and how they affect the design of our protocol.

2.1 TreadMarks and Lazy Release Consistency

Lazy release consistency [13, 14] is a variant of release consistency [18]. It guarantees memory consistency only at synchronization points and permits multiple writers per coherence block. Lazy release consistency divides time on each node into intervals. Each node maintains a vector of such intervals, with entry i on processor j representing the most recent interval on processor i that logically precedes the current interval on processor j. Every time a node executes a synchronization operation it starts a new interval and increments its own entry in its local vector of timestamps. The protocol associates writes to shared pages with the interval in which they occur. When a processor takes a write protection fault, it creates a write notice for the faulting page and appends the notice to a list of notices associated with its current interval.

When a processor acquires a lock, it sends a copy of its current vector timestamp to the previous lock owner. The previous lock owner compares the received timestamp with its own, and responds with a list of all intervals (and the write notices associated with them) that the new owner has not seen. The acquiring processor sets its vector timestamp to be the pairwise maximum of its old vector and the vector of the previous lock owner. It also incorporates in its local data structures all intervals (and the associated write notices) that the previous lock owner knew about, but which the acquiring processor has not yet seen. Finally it invalidates all pages for which it receives a write notice, since the receipt of a write notice indicates that there is a write to that page in the processor's logical past and the processor needs to get an up-to-date copy of the page.

Pages are brought up-to-date by merging the writes of remote processors in software. Before a processor writes a page, it makes a copy of the page, also called a *twin*. When a processor is asked to produce the set of changes it has made to a page, it computes the set by comparing the current version of the page to its twin. The result is a list of modified addresses with the new contents, also called a *diff*. There is one *diff* for every write notice in the system. On an access fault to an invalid page, the faulting processor asks for all the diffs it needs to bring the page up-to-date, based on its list of write notices. It then applies the diffs to its copy of the page in the causal order defined by the timestamps of the write notices.

Barrier synchronization is dealt with somewhat differently. Upon arrival at a barrier, all processors send their vector timestamps (and intervals and write notices), to a barrier manager based on what values they think the manager's vector timestamp contains. The manager merges all timestamps, intervals, and write notices into its local data structures, and then sends to each processor its new updated timestamp along with the intervals and write notices that that processor has not seen.

Because it must guarantee the correctness of arbitrary future references, the TreadMarks protocol must send notices of all logically previous writes to synchronizing processors even if the processors have no copy of the page to which the write notice refers. If a processor is not going to acquire a copy of the page in the future (something the protocol cannot of course predict), then sending and processing these

[1] CASHMERe stands for Coherence Algorithms 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.

[2] One might argue that an object-oriented system such as Midway [35] or Log-Based DSM [7] could provide superior performance, but at the cost of a restricted programming model.

notices may constitute a significant amount of unnecessary work, especially during barrier synchronization, when all processors need to be made aware of all other processors' writes.

2.2 Basic Cashmere Protocol

Like most DSM systems, Cashmere uses virtual memory protection bits to enforce consistency at the granularity of pages. Like TreadMarks, it minimizes the performance impact of false sharing by allowing multiple processors to write a page concurrently, and uses a variant of lazy release consistency to limit coherence operations to synchronization points. The basic Cashmere data structures are a distributed directory that maintains page state information, and per-processor lists that contain notices for pages that must be invalidated at lock acquisition points.

Each shared page in the system has a *home node*, which maintains the master copy of the data, together with the 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 quite 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 *global 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 coherence protocols) the viewpoint of a single processor or node.

Processors execute coherence protocol operations in response to four events: read-protection page faults, write-protection page faults, synchronization release operations, and synchronization acquire operations. On protection faults (both read and write), the faulting processor locates the directory entry of the faulting page. It then acquires the lock for that entry, fetches the entry into its local cache, modifies it, writes it back, and releases the lock. Modifying the entry involves computing the new state of the page, and sending write notices to sharing processors if the page made the transition to the weak state. The cost of fetching a directory entry can can be minimized if lock operations are properly designed.

If we employ a distributed queue-based lock [21], a fetch of the directory entry can be initiated immediately 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 directory 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 directory 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.

To post a write notice, a processor locks the write-notice list of the remote node, appends a notice to the end of the list, and releases the lock. The write notice simply names the page to be invalidated. Our current implementation uses a circular array to represent the list.

The last thing the page-fault handler does is to set up a mapping to the remote page; data will be transferred to the local cache in response to cache misses. Alternatively, the handler can choose to copy the remote page to local memory and map the local copy. This is appropriate for applications and systems with high spatial locality and relatively high remote memory latencies. We examine the tradeoff between page copying and remote cache fills in section 4.

On a synchronization *acquire* operation, the processor examines its local write notice list and self-invalidates all listed pages. To self-invalidate a page, the processor (a) acquires the lock for the page's directory entry; (b) removes mappings for the page from its local TLB/page table; (c) purges all lines belonging to the page from its cache; (d) removes itself from the directory's list of processors sharing the page; (e) changes the state of the page to uncached if the sharing list is now empty; and (f) releases the directory entry lock.

The protocol uses a write-through policy to ensure that the master copy of shared pages—located at the home node—, has the most recent version of the data. Synchronization *release* operations stall the processor until all write-through operations have completed. This guarantees that any processor that invalidates its mapping to a modified page as a result of a subsequent acquire operation will see the releasing processor's changes when it goes back to main memory for the data.

We have applied two optimizations to this basic protocol. The first optimization postpones sending write notices until the processor reaches a synchronization release operation. The processor treats all page faults as described above for read faults, but creates both read and write mappings for the page on which it faulted. When it reaches a release operation, the processor examines the dirty bits for all pages it shares. If it finds the dirty bit set it then proceeds to take the same actions it would have taken on a write page fault. It also sets a "processed bit" to ensure that it does not reprocess the same pages again on a subsequent release. The "processed bit" is reset when a page is invalidated.

The second optimization tries to minimize the cost of sending write notices to processors. It takes advantage of the fact that page behavior tends to be relatively constant over the execution of a program, or at least a large portion of it. When the protocol notices that it has to send a large number of write notices on behalf of a particular page, it

marks the page as *unsafe*. (The unsafe state is orthogonal to the four other page states.) When an unsafe page becomes weak due to a page fault, the faulting processor does not send write notices to the sharing set. Instead it only changes the page state to weak in the directory. An acquiring processor, in addition to processing its local write notice list, must now check the directory entry for all its unsafe pages, and invalidate the weak ones. A processor knows which of its pages are unsafe because it maintains a local list of them (this list is never modified remotely). Correctness is guaranteed by the fact that the transition to the unsafe state is a side-effect of the transition to the weak state. All processors in the sharing set at the point of the transition will receive write notices. Processors joining the sharing set after the transition will notice that the page is unsafe and will know to check the directory entry on their next acquire operation.

Experiments (not reported here) reveal that delayed write notices significantly improve performance. The distinction between safe and unsafe pages has a comparatively minor effect.

2.3 Network Interface Variants

Current NCC-NUMA systems provide different amounts of hardware support for the type of protocol outlined in the previous section. In this section we consider three dimensions in the hardware design space together with an initial, software dimension that affects the performance of the others: (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.
- If a process decides to not copy a page on an initial miss, the mechanism used to detect cache misses to the remote memory can have a significant impact on performance. Servicing cache misses to remote memory 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 protocols 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, 35]. 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.
- 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).

Not all combinations of the dimensions discussed above make sense. 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. In a similar vein, a protocol that copies pages to local memory will not need software intervention to fetch data into the cache from local memory. Finally, merge buffers only make sense on a system with hardware writes.

We use four-letter acronyms to name the protocols, based on the four dimensions. The first letter indicates whether the protocol copies pages or not (C or 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 protocol and network interface design space is then as follows:

NHHM: This system assumes hardware support for cache misses, hardware support for write forwarding, and a merge buffers. It transfers data a cache line at a time.

CHHM: This system assumes hardware support for write forwarding. Cache misses are to local memory, and

thus are in hardware by definition. It copies data in page-sized chunks.

NHHN: Identical to NHHM but without a merge buffer.

CHHN: Identical to CHHM but without a merge buffer.

CHSN: This system is similar to CHHM but in that it forwards writes to remote pages in software and does not have a merge buffer.

NSHM: This system is similar to NHHM. However the mechanism used to detect a cache miss is ECC faults as described earlier in the section.

NSHN: Identical to NSHM but without a merge buffer.

NSSN: This system transfers cache lines in software, forwards writes to remote memory in software, and does not have a merge buffer. It employs the simplest NCC-NUMA network interface.

The CHHM and NSHM systems could be implemented on a machine similar to the Princeton Shrimp [3]. Shrimp will double writes in hardware, but cannot fill cache lines from remote locations. CHSN and NSSN resemble systems that could be implemented on top of the DEC Memory Channel [8] 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 access remote locations with regular store instructions as opposed to special instruction sequences; it is therefore able to double writes in software with less overhead.

In our work 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 and the Memory Channel provide this capability, but Shrimp does not. None of the systems will fill cache lines from remote memory in response to a cache miss but DEC is considering this feature for future generations of the hardware [9]. Shrimp is the only system that doubles writes in hardware: both the Memory Channel and Hamlyn require software intervention to achieve the same effect.

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 for the bus and network interface, and memory access costs including contention for the bus and memory module. Table 1 summarizes the default parameters used in our simulations. Contention is captured by splitting each transaction into pieces at the points of potential contention, and running each piece separately through the simulator's discrete event queue. Local requests receive priority over remote requeusts for access to the bus.

System Constant Name	Default Value
TLB size	128 entries
TLB fill service time	100 cycles
ECC interrupts	400 cycles
Page faults	400 cycles
Directory 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 (bi-dir)
Messaging software overhead	150 cycles
Switch latency	2 cycles
Wire latency	1 cycle
Page twinning	5 cycles/word
Diff application and creation	7 cycles/word
List processing overhead	7 cycles/write notice
SW write doubling (overhead)	3 cycles/write

Table 1: Default values for system parameters

ECC interrupt time is based on figures for the Wisconsin Wind Tunnel, scaled to account for comparatively slower memory access times on a higher clock rate processor [34]. The figure in the table does not include the time to fix the bad ECC. That time is overlapped with the fetch of the data from remote memory. Similarly, page twinning and diffing overhead counts only the instruction overhead. Any memory stall time due to cache misses while twinning or diffing is taken into account separately in the simulator. Most of the transactions required by the Cashmere 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 fill service, (b) a page fault caused by the absence of read rights, (c) a directory entry lock acquisition, and (d) a directory 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 450 cycles (200 cycles each way plus 50 cycles for the memory to produce the directory entry). The total cost for the above transaction would then be $100 + 400 + 450 + 160 = 1110$ cycles, plus any additional time due to bus or network contention.

3.1 Workload

We report results for nine parallel programs. 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 640 $\times$ 640 matrix. `Sor` computes the steady state temperature of a metal sheet using a banded parallelization of red-black successive over-relaxation on a 1024 $\times$ 1024 grid. `Sor_l` is an alternative implementation of the `sor` program that uses locks instead of barriers 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 131072-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 60000 particles for 10 steps in our studies. `Water` is a molecular dynamics simulation computing inter- and intramolecule forces for a set of water molecules. We used 256 molecules and 3 times steps. `Em3d` [6] 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 the changes were mechanical and took no more than a few hours of programming effort.

We ran each application on the largest input size that could be simulated in a reasonable amount of time and that provided good scalability for a 64-processor system These sizes are smaller than would occur on a real machine, but we have chosen similarly smaller caches in order to capture the effect of capacity and conflict misses. Since we still observe reasonable scalability for most of our applications,[3] we believe that the data set sizes do not compromise our results.

4 Results

In this section we evaluate the performance improvement that can be achieved over more traditional DSM systems, with the addition of a memory-mapped network interface. We start by evaluating the tradeoffs between different software protocols on machines with an "ideal" NCC-NUMA 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 performance impact of network latency and bandwidth.

[3] Mp3d does not scale to 64 processors; we use it as a stress test to compare the performance stability of the various systems and protocols.

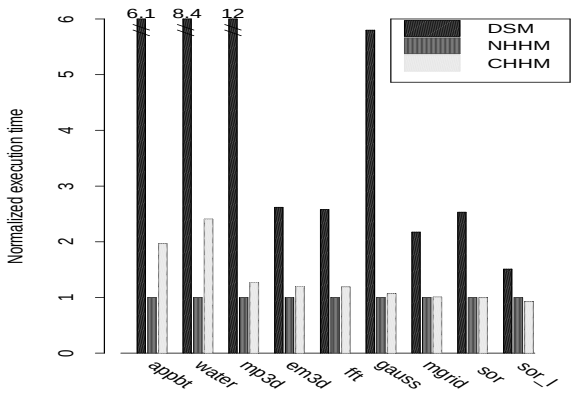

Figure 1: Execution time of DSM and Cashmere systems, normalized with respect to NHHM.

4.1 Protocol Alternatives

This section compares two versions of the Cashmere protocol running on an "ideal" NCC-NUMA system to a state-of-the-art DSM protocol—TreadMarks—designed for message-passing hardware. The first version copies a page to local memory on a page fault (CHHM), while the second simply maps it and fetches data in response to cache misses (NHHM). All simulations employ identical processors, write-through caches (except for the DSM protocol, 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), to fill cache lines remotely, and to perform uncached references directly to remote memory. Results appear in figure 1. 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 by as much as 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 is resolved in favor of line transfers. Only `sor` favors page copying and even in that case, the performance advantage is only 9%.

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 reduced coherence overhead realized by Cashmere 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 significantly 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

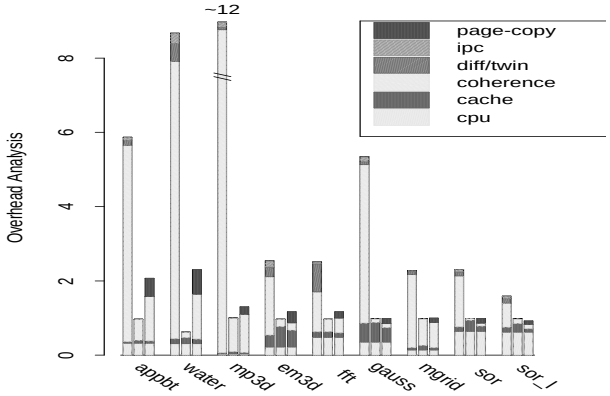

Figure 2: Overhead analysis for 64-processor DSM and Cashmere systems.

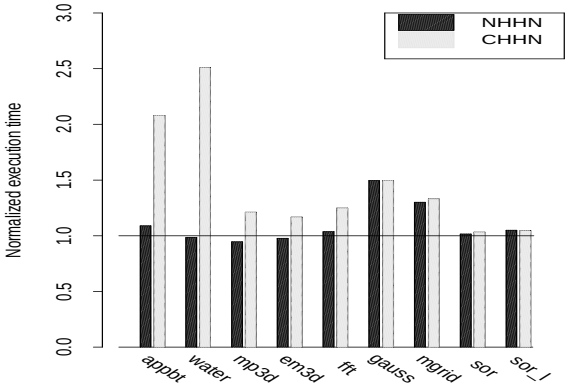

Figure 3: Execution time of Cashmere systems without a merge buffer, normalized with respect to NHHM.

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 longer messages and greater list processing overhead than is necessary based on the sharing pattern of the program. 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 wrote 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. 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

Application	Running Time ratio	
	8 processors	64 processors
appbt	1.51	6.16
water	1.94	8.43
mp3d	8.17	12.01
em3d	1.43	2.61
fft	1.15	2.58
gauss	2.24	5.80
mgrid	1.39	2.17
sor	1.13	2.53
sor_l	1.14	1.51

Table 2: Running time ratio of traditional DSM to NHHM system on 8 and 64 processors.

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. `Sor`, and its lock-based version work better with page copying (CHHM) than with remote cache fills (NHHM). Rows are page aligned in `sor`, and electromagnetic 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.

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 almost 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 2 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 the DSM system part of the cache stall time is encompassed in protocol processing time: since caches

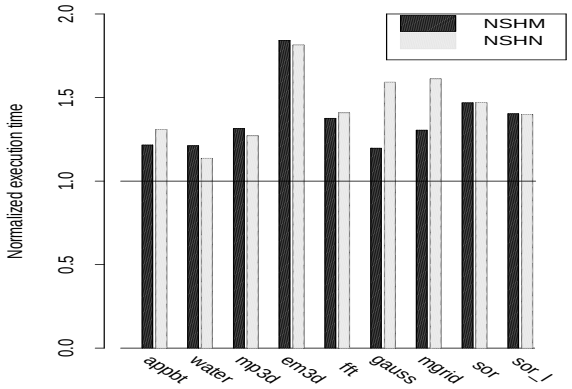

Figure 4: Execution time for Cashmere systems with software-mediated cache fills, normalized with respect to NHHM.

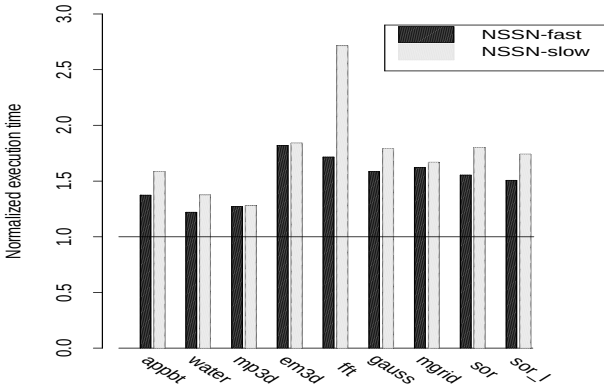

Figure 5: Execution time for Cashmere systems with software doubling of writes and line transfers, normalized with respect to NHHM.

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 capacity, conflict, and cold-start misses. Also, the protocol processing overhead includes any time that 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. As can be seen from the figure, the coherence processing component 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 systems 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 management and interprocessor interrupt overhead. Table 2 shows the run time ratio of a Treadmarks-like DSM and the NHHM Cashmere protocol on 8 and 64 processors for our application suite. The 64-processor numbers are those shown in graphs. They are repeated in the table for comparison purposes.

4.2 Relaxing the Hardware Requirements

Current network implementations do not provide all the hardware support required to realize an ideal NCC-NUMA system. This is presumably due to a difference between the original goals of the network designers (fast user-level messages) and the goals of a system such as Cashmere (efficient software-coherent shared memory). In this section we consider the performance impact of the extra hardware. In particular we consider:

Merge buffers. These reduce the bandwidth required of memory and the network, and can improve performance for applications that require large amounts of bandwidth.

Write doubling. 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.

Remote cache fills. In the absence of hardware support we can still service cache misses in software, provided that we have a "hook" such as ECC faults to gain control when a miss occurs. The fault handler can then retrieve the desired line, write it to local memory, and return 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 3 shows the performance impact of merge buffers for protocols that either copy or do not copy pages (systems NHHN and CHHN). The Cashmere systems shown in the figure are identical to those of figure 1, except for the absence of the merge buffer. Running time is once again normalized with respect to the NHHM system. 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.

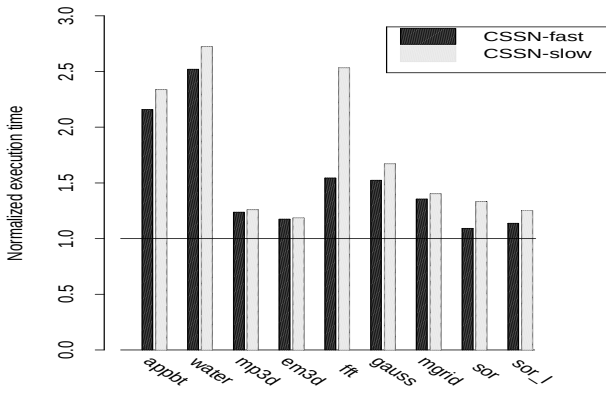

Figure 6: Execution time for Cashmere systems with software doubling of writes and page copying, normalized with respect to NHHM.

Support for hardware reads is more important. When cache misses to remote locations incur trap overhead the NSHM and NSHN protocols suffer a significant performance hit, with performance loss ranging from 14 to 84%. As a matter of fact performance loss is high enough, that in the absence of hardware support for remote cache fill, copying the page to local memory on a page fault becomes the better choice. Normalized execution time for the systems that do not assume hardware support for reads is shown in figure 4.

The final dimension of interest is the mechanism used to forward writes to a remote main-memory page. Figures 5 and 6 show the normalized running time on the Cashmere systems 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 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 systems. High bandwidth is important for write-through; low latency is important for cheap directory access. We expect bandwidth to continue to increase (bandwidth for next year's version of the Memory Channel is expected to be over 1Gbyte/sec [9]), 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 systems. 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 over more traditional DSM when run on the Cashmere protocols. The remaining applications exhibit qualitatively similar behavior.

Figure 7 shows the performance impact of network latency on the relative performance of the DSM, NHHM, and CHHM systems. We kept bandwidth at 400Mbytes/sec and varied message latency between 0.5 and 10 μsec. Running times in the graph are normalized with respect to the running time of the NHHM protocol with 1 μsec latency. The measures taken by TreadMarks to minimize the number of messages exchanged make its performance largely insensitive to changes in network latency. The Cashmere systems are more affected by latency variations. The NHHM system in particular suffers severe performance degradation when forced to move cache lines at very high latency. For a 200 MHz processor, a 10 μsec 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 system (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. Currently an end-to-end 4-byte write takes about 3.5 μsec on the Memory Channel; this number will decrease by almost a factor of three in the next generation release.

We have also collected results (not shown) on the impact of latency for systems with suboptimal hardware. We found that the impact of doubling writes and of mediating cache fills in software decreases as 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 systems is depicted in figure 8. For these experiments we kept latency at 1 μsec and varied bandwidth between 100Mbytes/sec and 1Gbyte/sec. Running times in the graph are normalized with respect to the running time of the NHHM protocol with 400Mbytes/sec bandwidth. Both the DSM and (to a lesser degree) CHHM systems 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, bus, and memory resources. As bandwidth increases, performance for both systems improves markedly. CHHM approaches or surpasses the performance of NHHM, while DSM tails off sooner; when enough bandwidth is available

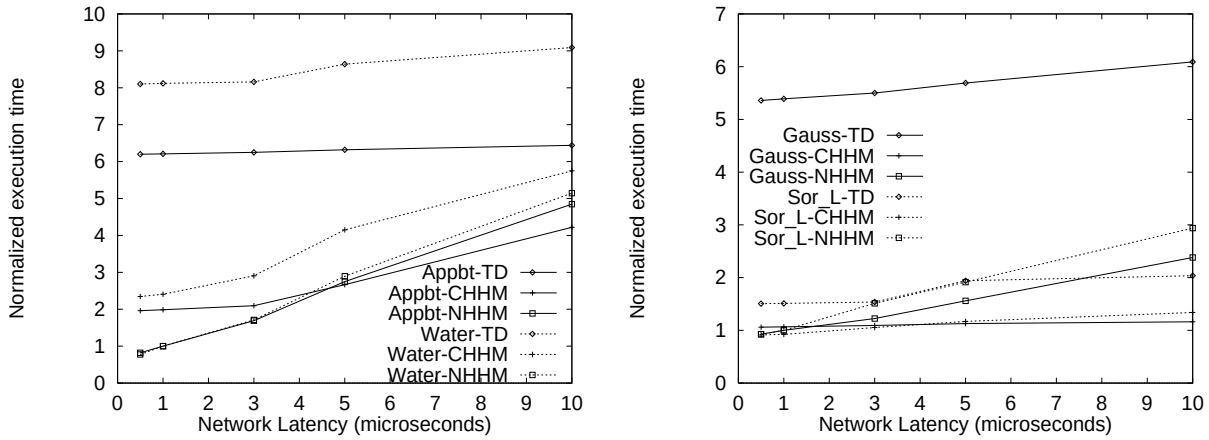

Figure 7: Execution time for DSM and Cashmere under different network latencies, normalized with respect to NHHM.

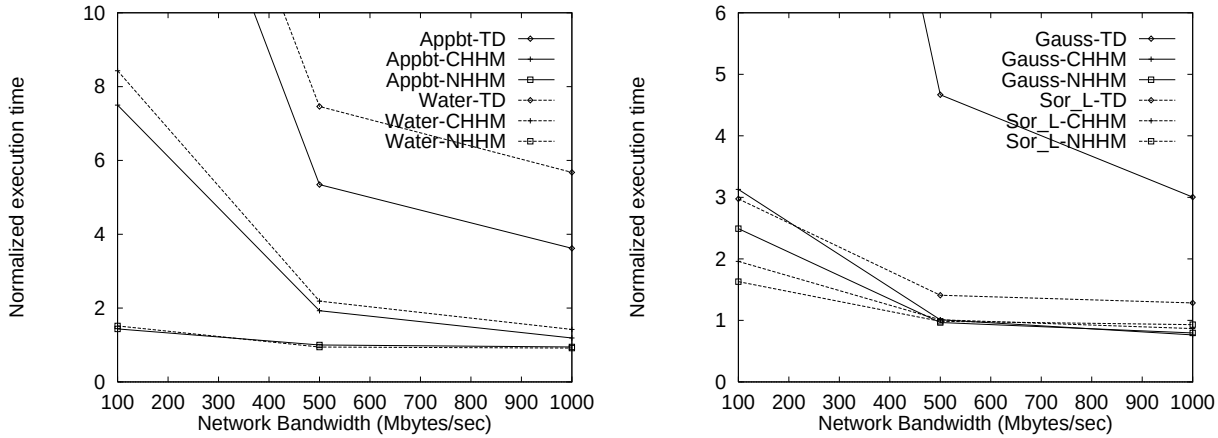

Figure 8: Execution time for DSM and Cashmere under different network bandwidths, normalized with respect to NHHM.

protocol processing overhead becomes the limiting factor. The NHHM system, which transfers cache lines only, is largely bandwidth-insensitive.

For systems 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.

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 directory and weak list, distinguish between "safe" and "unsafe" pages, check the weak list only for unsafe pages mapped by the local processor (i.e., for those that appear in the local write notice list), and multicast write notices only 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 systems) and for access to the directory and the local write notice lists. We have also presented protocol variants [17] suitable for a 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.

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. Recent work on the Shrimp project at Princeton [11] has addressed the design of coherence protocols based on write-through to remote locations. The Shrimp protocol resembles what we call CHHN, with the added twist that when only two processors have mappings to a page they write through to each others' copies, rather than to a single master copy. Researchers at the University of Colorado also propose the use of NCC-NUMA hardware for efficient

shared memory emulations [10]. In their approach however, the coherence protocol is simplified to invalidate all shared pages on lock acquisition 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 NSxx systems) 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 [6, 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 [35] systems. Our use is different in that we simply need to duplicate writes. As a consequence we need as few 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 NCC-NUMA hardware to improve performance over traditional DSM systems by as much as an order of magnitude. The key hardware innovation is a memory-mapped network interface, with which processors can access remote locations without trapping into the operating system or interrupting other processors. This type of interface 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 NCC-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 [15, 17]. Together, these findings raise the prospect of practical, shared-memory supercomputing on networks of commodity workstations.

We are currently building a 32-processor NCC-NUMA prototype based on DEC's Memory Channel and 4-processor AlphaServer 2100 multiprocessors. We are continuing research on protocol issues, including the design of a protocol variant that chooses dynamically between cache line and page transfers based on the access patterns of the application, and another that automatically invalidates "unsafe" pages without consulting the corresponding directory entry. We are also 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. Finally we are looking into whether it makes sense to dedicate one processor from a multiprocessor node to protocol processing tasks. Such a processor could be used to fill unused pages with "bad" ECC bits, in anticipation of using them for on-demand remote cache fills; track remote references in the manner of a hardware stream buffer [23] in order to prefetch data being accessed at a regular stride; or process write faults on behalf of the other processors, allowing them to overlap computation with coherence management.

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, to Rick Gillett for providing us with information on the DEC Memory Channel, and to David Wood Steve Reinhardt, and Mark Hill for answering questions on the Wisconsin Wind Tunnel. The comments of Kai Li and the anonymous referees led to significant improvements in this paper.

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, Jan. 1991.

[3] M. Blumrich, K. Li, R. Alpert, C. Dubnicki, E. Felten, and J. Sandberg. Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer. *21st Intl. Symp. on Computer Architecture*, pp. 142–153, Chicago, IL, Apr. 1994.

[4] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and Performance of Munin. *13th ACM Symp. on Operating Systems Principles*, pp. 152–164, Pacific Grove, CA, Oct. 1991.

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

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

[7] M. J. Feeley, J. S. Chase, V. R. Narasayya, and H. M. Levy. Log-Based Distributed Shared Memory. *1st*

Symp. on Operating Systems Design and Implementation, Monterey, CA, Nov. 1994.

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

[9] R. Gillett. Personal communication, Digital Equipment Corp., Maynard, MA, Mar. 1995.

[10] D. Grunwald. Personal communication, Univ. of Colorado at Boulder, Jan. 1995.

[11] L. Iftode, C. Dubnicki, E. Felten, and K. Li. Improving Release-Consistent Shared Virtual Memory using Automatic Update. *2nd Intl. Symp. on High Performance Computer Architecture*, San Jose, CA, Feb. 1996.

[12] N. Jouppi. Cache Write Policies and Performance. *20th Intl. Symp. 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. *19th Intl. Symp. on Computer Architecture*, pp. 13–21, Gold Coast, Australia, May 1992.

[14] P. Keleher, S. Dwarkadas, A. L. Cox, and W. Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. *USENIX Winter '94 Technical Conf.*, pp. 115–131, San Francisco, CA, Jan. 1994.

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

[16] L. I. Kontothanassis, M. L. Scott, and R. Bianchini. Lazy Release Consistency for Hardware-Coherent Multiprocessors. *Supercomputing '95*, San Diego, CA, Dec. 1995.

[17] L. I. Kontothanassis and M. L. Scott. Software Cache Coherence for Large Scale Multiprocessors. *1st Intl. Symp. on High Performance Computer Architecture*, pp. 286–295, Raleigh, NC, Jan. 1995.

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

[19] D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. S. Lam. The Stanford Dash Multiprocessor. *Computer*, 25(3):63–79, Mar. 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. *9th Intl. Parallel Processing Symp.*, Santa Barbara, CA, Apr. 1995.

[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, Feb. 1991.

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

[23] S. Palacharla and R. E. Kessler. Evaluating Stream Buffers as a Secondary Cache Replacement. *21st Intl. Symp. on Computer Architecture*, Chicago, IL, Apr. 1994.

[24] K. Petersen and K. Li. Cache Coherence for Shared Memory Multiprocessors Based on Virtual Memory Support. *7th Intl. Parallel Processing Symp.*, Newport Beach, CA, Apr. 1993.

[25] K. Petersen and K. Li. An Evaluation of Multiprocessor Cache Coherence Based on Virtual Memory Support. *8th Intl. Parallel Processing Symp.*, pp. 158–164, Cancun, Mexico, Apr. 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. *1993 ACM SIGMETRICS Intl. Conf. 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. *6th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pp. 297–306, San Jose, CA, Oct. 1994.

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

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

[30] C. A. Thekkath, H. M. Levy, and E. D. Lazowska. Separating Data and Control Transfer in Distributed Operating Systems. *6th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pp. 2–11, San Jose, CA, Oct. 1994.

[31] J. E. Veenstra and R. J. Fowler. MINT: A Front End for Efficient Simulation of Shared-Memory Multiprocessors. *2nd Intl. Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS '94)*, pp. 201–207, Durham, NC, Jan. – Feb. 1994.

[32] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active Messages: A Mechanism for Integrated Communication and Computation. *19th Intl. Symp. on Computer Architecture*, pp. 256–266, Gold Coast, Australia, May 1992.

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

[34] D. Wood. Personal communication, Univ. of Wisconsin — Madison, Nov. 1995.

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