Efficient Use of Memory-Mapped Network Interfaces for Shared Memory Computing

Nikolaos Hardavellas, Galen C. Hunt, Sotiris Ioannidis, Robert Stets,
Sandhya Dwarkadas, Leonidas Kontothanassis, and Michael L. Scott
Department of Computer Science
University of Rochester
e-mail: cashmere@cs.rochester.edu

Abstract

Memory-mapped network interfaces provide users with fast and cheap access to remote memory on clusters of workstations. Software distributed shared memory (DSM) protocols built on top of these networks can take advantage of fast messaging to improve performance. The low latencies and remote memory access capabilities of these networks suggest the need to re-evaluate the assumptions underlying the design of DSM protocols. This paper describes some of the approaches currently being used to support shared memory efficiently on such networks. We discuss other possible design options for DSM systems on a memory-mapped network interface and propose methods by which the interface can best be used to implement coherent shared memory in software.

1 Introduction

Software distributed shared memory (DSM) is an attractive design alternative for cheap shared memory computing on workstation networks. Traditional DSM systems rely on virtual memory hardware and simple message passing to implement shared memory. State-of-the-art DSM systems employ sophisticated protocol optimizations, such as relaxed consistency models, multiple writable copies of a page, and lazy processing of all coherence-related events. These optimizations recognize the very high (millisecond) latency of communication on workstation networks; their aim is to minimize the frequency of communication, even at the expense of additional computation.

Recent technological advances have led to the commercial availability of inexpensive workstation networks on which a processor can access the memory of a remote node safely from user space [3, 9, 5]. The latency of access is two to three orders of magnitude lower than that of traditional message passing. These networks suggest the need to re-evaluate the assumptions underlying the design of DSM protocols. This paper describes existing protocols that take advantage of the low-latency memory-mapped network interface, and discusses the trade-offs involved in efficiently exploiting its capabilities. We discuss the design options for DSM systems on a memory-mapped network interface and propose methods by which the interface can best be used to implement software coherent shared memory.

We restrict ourselves to a discussion of systems that support more-or-less "generic" shared-memory programs, such as might run on a machine with hardware coherence. The memory model presented to the user is release consistency [8], with explicit synchronization operations visible to the run-time system.

We begin with a summary of existing protocols and implementations. Cashmere [12] employs a directory-based multi-writer protocol that exploits the write-through capabilities of its network in order to merge updates by multiple processors. AURC [10] is an interval-based multi-writer protocol designed for the Shrimp network interface [3]. Shasta [21] uses a single-writer directory protocol with in-line protocol operations to support variable-size coherence blocks. Finally, TreadMarks [1] uses a multi-writer interval-based protocol to provide DSM on message-passing networks; on a memory-mapped network it uses the extra functionality only for fast messages.

These protocols represent only a small subset of choices in a very large design space. We enumerate alternatives for DSM systems on a memory-mapped network interface, and discuss methods to exploit that interface for high-performance software shared memory. Issues we consider include: 1) coherence granularity and miss detection mechanism; 2) metadata representation (directories v. intervals); 3) home node place-

This work is supported in part by the National Science Foundation under Grants CDA–9401142, and CCR–9319445.

ment/migration; 4) update collection mechanism; 5) use of remote-mapped address space; and 6) synchronous remote operations (interrupts, etc.)

The rest of the paper is organized as follows. Section 2 describes existing protocols implemented on memory-mapped network interfaces. Section 3 discusses performance tradeoffs in more detail, and presents a summary of results from a comparison of two of the protocols. Section 4 concludes with a summary of future directions.

2 Discussion of Existing Protocols

2.1 Background

Software distributed shared memory (DSM) combines the ease of programming of shared memory machines with the scalability of distributed memory (message passing) machines. DSM systems provide an avenue for affordable, easy-to-use supercomputing for computationally demanding applications in a variety of problem domains.

The original idea of using the virtual memory system on top of simple messaging to implement software coherence on networks dates from Kai Li's thesis work on Ivy [15]. A host of other systems were built following Li's early work; Nitzberg and Lo [18] provide a survey of early VM-based systems. Many of these systems often exhibited poor performance due to false sharing. The large granularity of the coherence blocks and the sequential consistency memory model used often resulted in page thrashing without real data sharing at the application level. Several groups employed similar techniques to migrate and replicate pages in early, cache-less shared-memory multiprocessors [4, 14].

Relaxed consistency models result in considerable improvements in DSM performance. Munin [6] was the first DSM system to adopt a release consistency model and to allow multiple processors to concurrently write the same coherence block. Processors kept track of what modifications they had made on a page by making a copy of the page before starting to write it (called *twinning*), and then comparing the page to its twin (called *diffing*). TreadMarks [1] uses a lazy implementation of release consistency [11], which further limits communication to only those processes that synchronize with one another.

Recent advances in network technology have narrowed the gap in communication performance between single-chassis systems and clusters of workstations. At the high end (in terms of cost), recent commercial offerings from Sequent and SGI construct large, cache-coherent systems from multiprocessor nodes on a high-speed network. Several other academic and commercial projects are developing special-purpose adaptors that extend cache coherence (at somewhat lower performance, but potentially lower cost) across a collection of SMP workstations on a commodity network; these include the Dolphin SCI adaptor [16], the Avalanche project [7] at the University of Utah, and the Typhoon project at the University of Wisconsin [20]. At still lower cost, memory-mapped network interfaces without cache coherence allow messages (typically triggered by ordinary loads and stores) to be sent from user space with microsecond latencies; examples here include the Princeton Shrimp [3], DEC Memory Channel [9], and HP Hamlyn [5] networks. Software DSM systems built on top of these very fast networks are an attractive cost-efficient alternative to full hardware coherence. In the rest of this section, we focus on four software DSM systems implemented on a memory mapped network interface.

2.2 Software DSMs on memory mapped network interfaces

TreadMarks [1] is a distributed shared memory system based on lazy release consistency (LRC) [11]. Lazy release consistency guarantees memory consistency only at synchronization points and permits multiple writers per coherence block. Time on each node is divided into intervals delineated by remote acquire synchronization operations. Each node maintains a timestamp consisting of a vector of such intervals: entry i on processor j indicates the most recent interval on processor i that logically precedes the current interval on processor j. When a processor takes a write page fault, it creates a write notice for the faulting page and appends the notice to the list of such notices associated with its current interval. During synchronization events the synchronizing processors exchange their vector timestamps and invalidate all pages that are described in write notices associated with intervals known to one processor but not known to the other. The write notices are appended to the data structures that describe the invalidated pages. In subsequent faults, the list of write notices associated with a page is perused and the changes made by the processors specified in the write notices are fetched. As in Munin, each processor keeps track of its own changes by using twins and diffs. There is one diff for every write notice in the system.

Our implementation of TreadMarks for the DEC Memory Channel makes use of the memory-mapped network interface for fast messaging and for a user-level implementation of polling, allowing processors to exchange asynchronous messages inexpensively. We do not currently use broadcast or remote memory access for either synchronization or protocol data structures, nor do we place shared memory in Memory Channel space.

Cashmere [12] is a software coherence system expressly designed for memory-mapped network interfaces. It was inspired by Petersen's work on coherence for small-scale, non-hardware-coherent multiprocessors [19]. Cashmere maintains coherence information using a distributed directory data structure. For each shared page in the system, a single directory entry indicates one of three possible page states: uncached, read-shared, or write-shared. At a release operation a processor consults the directory regarding pages it has written, and, if the page is not already in write-shared state, sends a write notice to all processors that have a copy of the page. At an acquire operation, a processor invalidates all write-shared pages in its sharing set. As in TreadMarks there may be multiple concurrent writers of a page. Rather than keeping diffs and twins, however, Cashmere arranges for every processor to write its changes through, as they occur, to a unique home copy of each page. When a processor needs a fresh copy of a page it can simply make a copy from the home; this copy is guaranteed to contain all changes made by all processors up to that point in time. Cashmere currently runs on the Memory Channel. Because the network adaptor does not snoop on the memory bus, Cashmere binaries must be modified to "double" every write to shared memory: one write goes to the local copy of the data; the other goes to I/O space, where it is caught by the adaptor and forwarded to the home node.

AURC [10] is a multi-writer protocol designed for the Princeton Shrimp [3]. Like TreadMarks, AURC uses distributed information in the form of timestamps and write notices to maintain sharing information. Like Cashmere, it relies on (remote) write-through to merge changes into a home copy of each page. Because the Shrimp interface connects to the memory bus of its 486-based nodes, it does not need to augment the executable like Cashmere does, thus avoiding a major source of overhead. Experimental results for AURC are currently based on simulation; implementation results await the completion of a large-scale Shrimp testbed.

Shasta [21], developed at DEC WRL, employs a single-writer relaxed consistency protocol with variable-size coherence blocks. Like TreadMarks, Shasta uses the Memory Channel only for fast messaging and for an inexpensive implementation of polling for remote requests. A major difference between Shasta and the above DSMs is the mechanism used to detect coherence misses. Rather than rely on VM, Shasta inserts consistency checks in-line when accessing shared memory. Aggressive compiler optimizations attempt to keep the cost of checks as low as possible. Because of its software miss detection, Shasta can maintain coherence at granularities smaller than a page, thus reducing false sharing effects seen by the previous three systems, and reducing the need for multiple writers. If different data ranges within an application display different access patterns, Shasta can use a different coherence granularity for each, thereby allowing the user to customize the protocol. Small coherence blocks have also been explored in the Blizzard system [22], but with a much less aggressive, sequentially-consistent protocol.

3 Performance Trade-Offs

Figure 1 presents results comparing two of the systems discussed above—TreadMarks and Cashmere. On the whole, the TreadMarks protocol with polling provides the best performance, though the Cashmere protocol comes close in several cases [13]. In general, the Cashmere protocol suffers from the overhead of write doubling, thrashing in the L1 cache, and the lack of bandwidth for write-throughs. The cache problems stem from the combination of a very small cache size (16KB on the 21064A processor used in our experiments) and cache pressure from write doubling. Both bandwidth and cache size will improve dramatically in future hardware from DEC. In Barnes, Cashmere already scales better than TreadMarks, by taking advantage of write-through to merge updates by multiple processors at a single location.

The DSM protocols discussed above and in section 2 represent only a small subset of choices in a very large design space. In this section we enumerate alternatives for DSM systems on a memory-mapped network interface, and discuss methods to exploit that interface for high-performance software shared memory.

3.1 Coherence Granularity and Miss Detection Mechanism

Software DSM systems require a mechanism for processes to detect when they are trying to access data that is not present in their sharing set, or for which

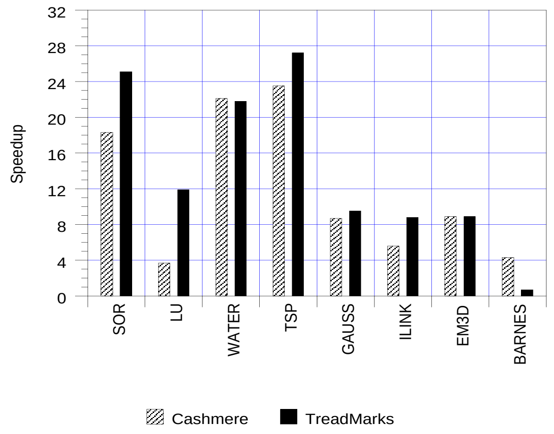

Figure 1: Cashmere and TreadMarks speedups on the Memory Channel, with 32 processors computing.

they do not have the appropriate access permissions. Two such mechanisms exist: hardware faults and in-line checks before shared loads and stores. Of the four systems outlined in Section 2, three—TreadMarks, Cashmere, and AURC—use page faults and therefore maintain coherence at the granularity of pages. In some systems it is possible to generate faults at a finer granularity using the ECC bits in memory [22].

Shasta checks a directory in software before shared loads and stores. If the check reveals an inconsistent state the program branches into a software handler that performs the necessary protocol actions. Inline checks are significantly cheaper than faults but have the disadvantage that they have to be performed on load and store instructions that would not cause a fault in a VM-based system. Aggressive compiler optimizations help to reduce the cost of inline checks; further reductions are an active area of research. Like ECC faults, in-line checks allow programs to use finer (and even variable) grain coherence blocks. Smaller blocks mitigate false sharing effects that might otherwise be observed with page-size coherence blocks. Multi-writer protocols also mitigate false sharing, but less effectively.

In general software DSM systems work best when applications employ larger blocks, thus minimizing the network startup latency and software overhead to transfer a given amount of data. The smaller blocks of Shasta would seem to be of the greatest benefit for applications in which hardware DSM enjoys the largest comparative advantage.

3.2 Metadata Representation

Cashmere and Shasta utilize directories in order to maintain sharing information on coherence blocks. Directories need to be updated when a processor misses on a coherence block, and also during synchronization events. Directories make more sense with a memory-mapped network than they do with a message-passing network, due to the low latency of messages. The fact that directories maintain global information (based on wall-clock time) means that processors may perform invalidations not strictly required by the happens-before relationship.

Distributed time-stamps allow processes to proceed without communication until they synchronize. A disadvantage of this approach is the accumulation of consistency information, requiring garbage collection. In addition, the lack of centralized information on a page may force processes to exchange write notices regarding pages that they are not actively sharing.

3.3 Home Node Placement/Migration

For protocols such as Cashmere, AURC, and Shasta, which maintain a "home" copy of each page, the choice of home node is important. Ideally, a page should be placed at the node that accesses it the most. At the very least, it should be placed at a node that accesses it some. Cashmere currently uses a "first touch after initialization" policy, resulting in reasonably good home node placement [17].

A fixed choice of home node may lead to poor performance for migratory data in Cashmere and AURC, because of high write-through traffic. Similarly, a home node for directory entries in Shasta implies a 3-way request for invalid data, which is directed through the home node to the current dirty copy. While data traffic is still 2-hop, care must be taken at compile-time to choose the correct granularity of directory access to avoid the *ping-pong* effect of false sharing. Unfortunately, in the presence of write-through there appears to be no way to change a home node at run-time without global synchronization. Remapping of pages is also fairly expensive.

Because modifications in TreadMarks are kept on the processor making the changes, and distributed only on demand, the issue of home node placement does not arise. Requests for updates to migratory data are automatically directed to the modifying processor. This localization of updates comes at the expense of additional computation for diffing and twinning. This overhead can be considerably reduced by combining single and multi-writer protocols [2].

3.4 Update Collection Mechanism

Cashmere and AURC avoid the computation overheads of diffing and twinning by writing data through to a home node. In the case of multiple concurrent writers, this has the advantage that subsequent requests for data by processors whose pages have been invalidated can be satisfied in their entirety by the home node. In contrast, TreadMarks must request diffs from all concurrent writers in order to update such a page. This results in a large number of messages in comparison to Cashmere or AURC, and a corresponding loss in performance. The amount of data traffic could, however, be lower, in the case of multiple writes to the same location since data is only transferred via an explicit request.

The Shasta protocol disallows multiple concurrent writers. Cache blocks that are actively write-shared will ping-pong between the sharing processors. Shasta reduces ping-ponging due to false sharing by using small coherence blocks. This of course increases the number of messages required to bring in the same amount of data. In order for small blocks to be profitable, the latency of messages must be low.

An attractive alternative would be to use home nodes to accumulate updates, but to collect them using twins and diffs, to minimize network traffic. We are adopting this approach in a version of Cashmere currently under development.

3.5 Use of Remote-Mapped Address Space

An important issue when designing DSM protocols for remote memory access networks is how best to exploit the ability to access remote memory directly. Choices include: a) make all shared memory accessible remotely, as in AURC and Cashmere, b) put all or some of the DSM metadata (directories, write notices, diffs) in shared space, but keep real data remotely inaccessible, and c) use the interface only for fast messaging and synchronization, as in Shasta and TreadMarks.

AURC and Cashmere make all shared memory accessible remotely. The advantage of this approach is that it allows in-place updates of home nodes, and direct transfers from the home to sharing nodes, without extra copies or remapping. The disadvantage (other than the bandwidth requirements of write-through, if any) is that the shared address space is limited to the amount of memory that can be remotely mapped. On the Memory Channel under Digital Unix, we are currently limited to about 100 MB.

TreadMarks and Shasta use the network interface only for fast messaging. The advantage of this approach is scalability, since the size of shared memory is not limited by the interface. However the latency of operations is higher, since data has to be transferred via the network into a message buffer and then copied to its final destination.

3.6 Synchronous Remote Operations

A side-effect of making all shared memory remotely-accessible is that there is no need to interrupt a remote processor in order to update its data. For networks such as Memory Channel and Shrimp, however, active cooperation of a remote processor is still required to *read* from a remote location—a reader must ask a remote processor to write the data back to it. Remote requests can be made using either interrupts or polling. The tradeoff between the two is similar to the one between fault-based and in-line access checks: polling incurs a modest constant overhead, while remote interrupts impose a large occasional overhead. At present polling outperforms remote interrupts for most of our applications on the Memory Channel under Digital Unix. Fast remote interrupts might tip the balance the other way.

Cashmere requires synchronous remote operations *only* to read remote data. Because all metadata and shared memory resides in Memory Channel space, (non-coherent) hardware reads would eliminate the need for either interrupts or polling. TreadMarks and Shasta place each processor in charge of maintaining its own data and metadata. As a result they use remote operations more extensively. AURC is somewhere in-between: it places shared data in remotely-accessible space, and updates it directly, but uses interrupts to trigger operations on locally-maintained metadata. A fourth alternative, which we are pursuing in a version of TreadMarks currently under development, is use remotely-accessible space only for metadata. This option permits very large data sets, while eliminating most of the interrupts associated with acquires and facilitating garbage collection of old intervals and diffs. A similar approach might also be attractive in Shasta.

4 Conclusion

In this paper, we have presented a survey of existing DSM protocols developed for remote memory-mapped network interfaces, and have discussed the trade-offs involved in their design and implementation. The trade-offs involved include coherence granularity (and method of detecting consistency violations), centralized directory-based coherence versus distributed vector timestamps, write-through of data to a central home node versus diffing and twinning versus single-writer mode, variable coherence granularity, and the use of interrupts in handling remote requests. We are in the process of systematically evaluating the design space for software DSM systems on top of memory-mapped network interfaces. Early experiments indicate that twinning and diffing tends to do better than write-through if the latter is not supported in hardware. However, write-through has advantages when used for the maintenance of protocol meta-data rather than application shared data. Based on our early findings, we are developing a VM-based protocol that will combine the advantages of write-through for protocol meta-data with the advantages of twinning and diffing for application data. In the near future, we expect to expand our evaluation to include coherence granularity issues and the choice of mechanism for detecting remote cache misses.

References

[1] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. TreadMarks: Shared Memory Computing on Networks of Workstations. *Computer*, 29(2):18–28, Feb. 1996.

[2] C. Amza, A. Cox, S. Dwarkadas, and W. Zwaenepoel. Software DSM Protocols that Adapt between Single Writer and Multiple Writer. In *Proc. of the 3rd Intl. Symp. on High Performance Computer Architecture*, Feb. 1997.

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

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

[5] G. Buzzard, D. Jacobson, M. Mackey, S. Marovich, and J. Wilkes. An Implementation of the Hamlyn Sender-Managed Interface Architecture. In *Proc. of the 2nd Symp. on Operating Systems Design and Implementation*, Oct. 1996.

[6] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and Performance of Munin. In *Proc. of the 13th ACM Symp. on Operating Systems Principles*, pp. 152–164, Oct. 1991.

[7] J. B. Carter, A. Davis, R. Kuramkote, and M. Swanson. The Avalanche Multiprocessor: An Overview. In *6th Workshop on Scalable Shared Memory Multiprocessors*, Boston, MA, Oct. 1996.

[8] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. L. Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *Proc. of the 17th Intl. Symp. on Computer Architecture*, pp. 15–26, May 1990.

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

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

[11] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy Release Consistency for Software Distributed Shared Memory. In *Proc. of the 19th Intl. Symp. on Computer Architecture*, pp. 13–21, May 1992.

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

[13] L. Kontothanassis, G. Hunt, R. Stets, N. Hardavellas, M. Cierniak, S. Parthasarathy, W. Meira, S. Dwarkadas, and M. L. Scott. VM-Based Shared Memory on Low-Latency, Remote-Memory-Access Networks. In *Proc. of the 24th Intl. Symp. on Computer Architecture*, June 1997.

[14] R. P. LaRowe Jr. and C. S. Ellis. Experimental Comparison of Memory Management Policies for NUMA Multiprocessors. *ACM Trans. on Computer Systems*, 9(4):319–363, Nov. 1991.

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

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

[17] M. Marchetti, L. Kontothanassis, R. Bianchini, and M. L. Scott. Using Simple Page Placement Policies to Reduce the Cost of Cache Fills in Coherent Shared-Memory Systems. In *Proc. of the 9th Intl. Parallel Processing Symp.*, Apr. 1995.

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

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

[20] S. K. Reinhardt, J. R. Larus, and D. A. Wood. Tempest and Typhoon: User-level Shared-Memory. In *Proc. of the 21st Intl. Symp. on Computer Architecture*, pp. 325–336, Apr. 1994.

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

[22] I. Schoinas, B. Falsafi, A. R. Lebeck, S. K. Reinhardt, J. R. Larus, and D. A. Wood. Fine-grain Access Control for Distributed Shared Memory. In *Proc. of the 6th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pp. 297–306, Oct. 1994.