Cashmere-VLM:
Remote Memory Paging for Software Distributed Shared Memory *

Sandhya Dwarkadas,[‡] Nikolaos Hardavellas,[†] Leonidas Kontothanassis,[†]
Rishiyur Nikhil,[†] and Robert Stets[‡]

[†] Compaq Cambridge Research Lab
One Kendall Sq.. Bldg 700
Cambridge, MA, 02139
{kthanasi,nikhil}@crl.dec.com
nikos@eng.pko.dec.com

[‡] Department of Computer Science
University of Rochester
Rochester, NY, 14627
{sandhya,stets}@cs.rochester.edu

Abstract

Software distributed shared memory (DSM) systems have successfully provided the illusion of shared memory on distributed memory machines. However, most software DSM systems use the main memory of each machine as a level in a cache hierarchy, replicating copies of shared data in local memory. Since computer memories tend to be much larger than caches, DSM systems have largely ignored memory capacity issues, assuming there is always enough space in main memory in which to replicate data. Applications that access data that exceeds the capacity available in local memory will page to disk, resulting in reduced performance. We have developed a software DSM system based on Cashmere that takes advantage of system-wide memory resources in order to reduce or eliminate paging overhead. Experimental results on a 4-node, 16-processor AlphaServer system demonstrate the improvement in performance using the enhanced software DSM system for applications with large data sets.

1 Introduction

Shared memory is generally considered an attractive programming model that is readily available on symmetric shared-memory multiprocessors (SMPs). Software distributed shared memory (DSM) systems running on clusters of SMPs successfully extend the shared memory environment across nodes in the cluster for a large number of applications, providing an easy path to larger scale multiprocessing.

Most software DSM systems [2, 13, 15] use the main memory of each node in the cluster as a level in the cache hierarchy, and migrate and replicate data in that memory. Since computer memories tend to be much larger than caches, DSM systems have largely ignored memory capacity issues, assuming there is always enough space in main memory into which to replicate data and avoid remote access. Applications that access data that exceeds the capacity available in local memory will page to disk, reducing performance.

We have designed a new protocol, based on the Cashmere software DSM system [15], that attempts to maximize the use of **all** the memory available in a cluster. The key insight behind the protocol is that while a node may access a larger amount of memory than it has locally, the active working set is likely to be smaller than the amount of local memory. Paging from disk can be an order of magnitude slower than paging across a network. We have therefore augmented our protocol to take advantage of the inherent data replication in a software DSM system. When possible, disk paging is thus avoided, or replaced with network paging.

We have implemented the protocol on a cluster of AlphaServers connected by a Memory Channel network. Experimental results show that the new protocol outperforms the base protocol by as much as a factor of 1.58 for our test applications running with high memory pressure. In the absence of memory pressure, the protocol imposes minimal ($\sim$3% on average) overhead when compared to a protocol that does not attempt to evict coherence blocks. Eviction-capable DSM protocols add an important dimension to the scalability of clusters, making them more attractive as compute servers for very large applications.

2 Protocol Description

We begin this section with a description of our base protocol, and follow it by a description of the eviction enhancements.

2.1 Base Cashmere-2L Protocol

Our base protocol, Cashmere-2L (CSM) [15], is a page-based software DSM system that has been designed for SMP clusters connected via a remote-memory-write network. The system implements a multiple-writer, lazy release consistent

* This work was supported in part by NSF grants CDA–9401142, CCR–9702466, and CCR–9705594; and an external research grant from Compaq Computer Corporation.

protocol, and requires applications to adhere to the data-race-free, or properly-labeled, programming model [1]. The consistency model implementation lies in between those of TreadMarks [2] and Munin [3]. Invalidations in CSM are sent during a release and take effect at the time of the next acquire, regardless of whether they are causally related to the acquired lock.

Low-latency remote-write networks, such as Compaq's Memory Channel [9], allow processors in one node to modify the memory of another node safely from user space, with very low (microsecond) latency. CSM uses the broadcast capabilities of the Memory Channel network to maintain a replicated directory of sharing information for each page (i.e., each node maintains a complete copy of the directory). The directory is examined and updated during protocol actions. Initially, shared pages are mapped only on their associated home nodes. Page faults are used to trigger requests for an up-to-date copy of the page from the home node. Page faults triggered by write accesses are also used to keep track of data modified by each node. At the time of a write fault, the page is added to a per-processor *dirty list* (a list of all pages modified by a processor since the last release). If the home node is not actively writing the page, then the home node is also *migrated* to the current writer by simply modifying the directory to point to the new home node. As an optimization, we also move the page into *exclusive* mode if there are no other sharers, and avoid adding the page to the per-processor dirty list. If the page is not in exclusive mode and the faulting processor is not on the home node, a *twin* (or pristine copy of the page) is created. The *twin* is later used to determine local modifications.

At a release, each page in the dirty list is compared to its *twin*, and the differences are flushed to the home node. After flushing the differences, the releaser sends *write notices* (notifications of a page having been modified) to the sharers of each dirty page, as indicated by the page's directory entry. Finally, the releaser downgrades write permissions for the dirty pages and clears the list. At a subsequent acquire, a processor invalidates all pages for which write notices have been received, and which have not already been updated by another processor on the node.

The protocol exploits hardware coherence to maintain consistency within each SMP node. All processors in the node share the same physical frame for a shared data page. Hardware coherence then allows protocol transactions from different processors on the same node to be coalesced, resulting in reduced data communication, as well as reduced consistency overhead. One of the novelties in the base protocol (compared to other SMP-aware page-based schemes) is that TLB shootdown is avoided by using twins and diffs on both incoming and outgoing page-update operations. The use of diffs on incoming page updates allows a node to update regions of the page that were modified by a remote node without overwriting changes being made simultaneously by a local process. The correctness of this approach depends on the assumption that programs are data-race-free. A detailed description of the base protocol including the shootdown avoidance mechanism can be found in [15].

2.2 Eviction Extensions

Paging from disk can be an order of magnitude slower than paging across a network. Our eviction extensions attempt to replace disk paging with network paging. Also, our extensions take advantage of the inherent data replication in a software DSM system to avoid paging of any type whenever possible.

Our new protocol (termed Cashmere-VLM (CSM-VLM) which stands for `Cashmere - Very Large Memory`) is currently implemented completely at user-level. In order to exercise paging across the network rather than to disk, we need to determine several parameters: an estimate of memory usage and availability [1], the sharing status of each page of shared data, and the last reference timestamp on the page. Both the sharing status of a page and the last reference timestamp are important in order to minimize the overhead of paging.

Currently, the maximum available physical memory is determined at the start of the program. Each node tracks memory usage through a local counter representing the number of resident shared pages. The counter is incremented when the node references a non-resident page, and decremented when the node evicts a page.

This memory usage is examined at every page fault and at every entrance to a barrier or lock acquire. If the memory usage exceeds a certain percentage of the available physical memory, the protocol invokes the network paging routines to free physical memory and avoid possible paging to disk. The following section details the policy used by these routines to select a page for eviction.

2.2.1 Eviction Algorithm

Our protocol takes advantage of DSM knowledge in order to minimize the cost of evicting pages from a node. Each page on a node is placed in one of four categories, and then treated appropriately by the protocol:

Exclusive pages: There is only a single copy of this type of page, and the node on which it resides is the home, with read and write access. No replicas exist on other nodes. Exclusive pages are never selected as eviction victims. By definition, they are pages used only by the single node and thus eviction is unlikely to be of benefit unless the node is using a disproportionate amount of memory resources. Furthermore, these pages would have to be transitioned out of exclusive mode before they can be evicted. This transition can be an expensive operation, sometimes requiring synchronization among all processes within a node.

[1] Kernel hooks for more accurate estimation could greatly improve performance.

Home pages: These are pages for which the node is the home node, but for which other sharers may exist. To evict these pages, a new home node must be chosen through negotiation. Then the current contents of the page must be sent to the new home, in order to ensure consistency. The transfer to a new home is a complex operation since other asynchronous operations may be taking place in the system with respect to that page. We ensure that no processor within the SMP is accessing the page by acquiring an intra-node lock that ensures exclusive access to that page's meta-data within the node. Inter-node exclusive access is harder to achieve since inter-node locks are more expensive, and we want to make sure to leave the common case unaffected. We use the distributed shared directory space to ensure inter-node exclusive access. Certain bits in the directory are reserved to indicate the existence of in-progress operations (i.e., page fetches, or diff applications), while an additional bit is used to indicate that an eviction event is under way. When a process decides to evict a home page, it sets the eviction bit and waits for all page-fetch and diff-related bits to clear. To avoid deadlock and resolve races, we have made evictions the highest priority event. Page-fetch and diff operations will backoff in the presence of evictions unless they are already under way. Should a fetch/diff and eviction request arrive at the directory at the same time, the fetch/diff request will back off and wait for the eviction to complete. Livelock, however, is still possible, since a page may be evicted from node to node, forcing a fetch/diff request to back off all the time. To minimize the chance of livelock, newly evicted pages are placed at the tail of a LRU queue, making them unlikely candidates for eviction in their new home.

Read-write pages: Such pages have been replicated and subsequently modified in the node. Before eviction, the modifications must be flushed to the home node, and then other sharing nodes must be notified of the write event.

Read-only and Invalid pages: Such pages have been replicated in read-only mode in the node or have been invalidated by all processors on the node. Furthermore, an up-to-date copy of the page exists in the home node, and thus it can be dropped safely without requiring any communication.

Based on these categories, the protocol maintains three separate page lists: the *Exclusive*, *Replica*, and *Home* lists. The *Exclusive* and *Home* lists are self-explanatory. The *Replica* list includes the *read-write, read-only*, and *invalid* pages — pages for which a *replica* exists on another node. On each page fault, a page is timestamped with a local logical clock value and moved into the appropriate list. The lists are maintained in order according to the timestamp values, with the head of each list representing the least recently used page.

Also, during a page fault, the protocol examines the current memory usage level and will attempt a page eviction if necessary. The eviction policy considers both the page's sharing state and the access timestamp within the node. As mentioned above, Exclusive pages are never selected for eviction since they are only in use on one node. The eviction candidate is chosen from the head of either the *Replica* or the *Home* list. The protocol compares the timestamps of the pages at the head of the lists and selects the page with the smaller (i.e., older) access timestamp. In practice, a threshold is applied so that *Replica* pages are favored, since the eviction mechanism for replicas is less expensive than for home pages.

In addition to a full page transfer, a home page eviction requires some negotiation. Before assigning a new home node, the evicting node must first verify that a candidate node has enough free physical memory. This requires a round-trip message. If the candidate does not have enough free memory, then the evicting node must query another node. This communication can make the eviction mechanism rather expensive. Also, if a suitable home node candidate can not be found, the eviction will simply fail and the system will default to standard disk paging. On the other hand, pages from the replica list can be evicted with little (for *read-write* pages) or no (for *invalid or read-only* pages) communication. All the node has to do is write back any pending modifications and then drop the page.

Reclaiming physical memory is achieved through re-use of shared memory segments. Shared memory within each SMP is allocated using the `shmget` call and mapped into the application address space with the `shmat` call. Rather than destroying and recreating the segments (both relatively expensive operations under Tru64 Unix), we simply detach the segment from the application's address space (using `shmdt`) and then re-map the segment into a different portion of the application's address space, thereby re-using the same physical space for a different virtual address region. One final issue that must be addressed is the fact that before an eviction can complete, all node processes must drop the shared mapping. Unfortunately the only way to notify other node processes of the eviction event is through explicit messages that increase synchrony within a node.

3 Experimental Results

Our experimental environment consists of four Compaq AlphaServer 2100 4/233 computers. Each AlphaServer is equipped with four 21064A processors operating at 233 MHz and with 256MB of shared memory, as well as a Memory Channel network interface. The 21064A's primary data cache size is 16 Kbytes, and the secondary cache size is 1 Mbyte. A cache line is 64 bytes. Each AlphaServer runs Tru64 UNIX 4.0/878 with TruCluster v. 1.5 (Memory Channel) extensions. The systems execute in multi-user mode, but with the exception of normal Unix daemons, no other processes were active during the tests. In order to increase cache efficiency, application processes are pinned to a processor at startup. No other processors are connected to the Memory Channel.

3.1 Application Characteristics

We present results for 4 applications:

CLU: A kernel that performs a factorization of a given matrix. The matrix is divided in square blocks that are distributed among processors in a round-robin fashion. Blocks "owned" by a single processor are allocated contiguously in memory. The active working set consists of the blocks owned by the processor and the pivot block used to factor them. Over time, the size of the active working set is reduced as factored blocks no longer need to be accessed.

TCLOS: A kernel that computes the transitive closure for a directed graph. The graph is represented as a matrix with a unit value in position (i, j) representing an edge between vertices i and j. Rows of the matrix are distributed in blocked fashion. The working set for this application consists of the rows owned by each processor and the pivot row used to compute the partial transitive closure. The working set size stays constant during the execution of the programs.

GAUSS: A solver for a system of linear equations $AX = B$ using Gaussian Elimination and back-substitution. For load balance, the rows are distributed among processors cyclically. The active working set consists of the rows computed on by the processor and the pivot row used to eliminate them. Over time the working set size decreases as eliminated rows drop out of the computation.

QSORT: A sorting program that uses the parallel quicksort algorithm to sort a list of double-precision floating point numbers in place. Load-balancing is implemented using a work queue. The application implements limited affinity scheduling for work items in the queue. The active working set of the application is highly dependent on the work scheduling and can be quite random. However, the active working set size decreases over time as the large array is fragmented into smaller partially sorted pieces.

We have chosen these applications because they allow us to arbitrarily scale the input data-set sizes in order to examine the impact of memory pressure on the performance of our system, and because they are applications that access different regions of shared memory with time, helping demonstrate performance on a reasonable spectrum of access patterns.

3.2 Performance Comparison

In order to determine the performance effect of the modifications made to the protocol for the VLM enhancements, we compare the VLM protocol with the base Cashmere-2L protocol. For each application, we present results for two different data sets - one that fits in available physical memory and one that exceeds the available physical memory within a single node.

We have also conducted some microbenchmarks to allow us to quantify the cost of paging on our system. On average, the cost of paging a single page into the system from disk was measured to be 6 milliseconds. However, costs can vary, since paging occurs on blocks of pages rather than individual pages themselves. The cost of an `mprotect` varies from 18-29 μsecs depending on the relative positions of consecutive calls. Similarly, the cost of `shmat` and `shmdt` varies between 35 and 70 μsecs. The cost of an `shmget` increases dramatically with the number of shared segments, starting at 51 μsecs when there is one shared segment and going up to 2.5 msecs when there are 16384 segments. Since the use of many segments is required in order to effect user-level page eviction, this reinforces the need to eliminate `shmget` calls from the VLM protocol.

The cost of evicting a page can vary from 274 to 1130 μsecs. The lower cost corresponds to the average cost of evicting a read-only page. In this scenario no data needs to be transferred through the network and thus the cost is derived from shooting down other sharers on the node and from the system calls involved with unmapping a page. The higher cost is the average cost of evicting a home page and includes any wait time for the completion of outstanding transactions on that page, shooting down other sharers on the node, finding a new home node, and transferring the page data to the new home node. Since the transfer across the Memory Channel does not use direct memory access (DMA), the cost of eviction is visible to the process performing the eviction, if data actually does have to be moved across the network. The cost of retrieving a page across the network from the memory of a remote node is 530 μsecs.

3.3 Small Data Sets

Figure 1 shows the relative runtimes of the two systems, base Cashmere (`CSM`) and Cashmere-VLM (`CSM-VLM`), on the small data sets (data-set sizes are specified in the graph legends — each data element is a 64-bit quantity for all applications). Sequential execution times for `CLU`, `TCLOS`, `GAUSS`, and `QSORT` are 294.9, 828.4, 957.6, and 110.2 seconds, respectively. We see that adding eviction and remote-memory paging capability (`CSM-VLM`) does not affect the runtime of the applications appreciably, showing an average 3% degradation. The extra checks during page faults and at acquire synchronization points have a negligible impact on performance.

3.4 Large Data Sets

Figure 2 shows the runtime of our four applications on the larger data sets under the two protocols (again, data-set sizes are in the legends). We do not report sequential execution times for the large datasets since excessive results in poor sequential performance. When compared with the `CSM` protocol, `CSM-VLM` shows moderate to significant performance improvements on all four applications, ranging from a factor of 1.02 to 1.58. We have also collected statistics on the number of pages paged to (`Page-out`) and from (`Page-in`) disk, the number of pages evicted to remote memory across the network (`Page-net`), the number

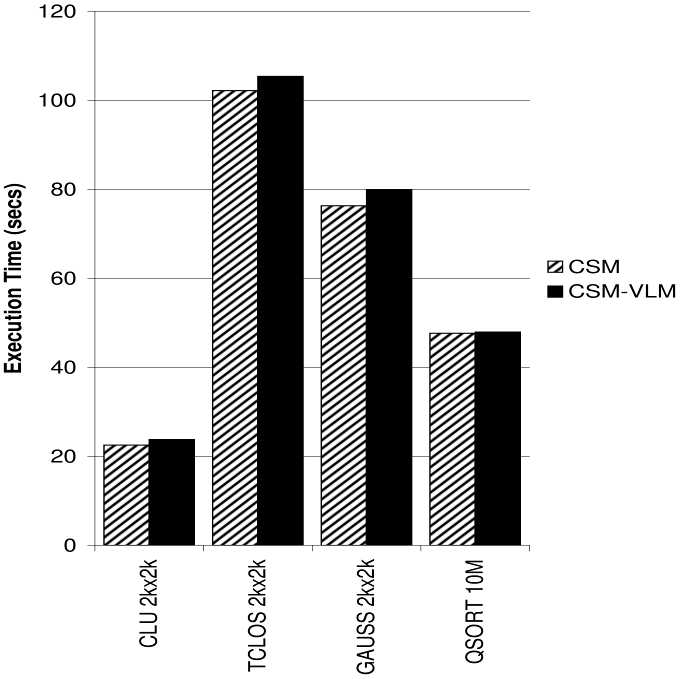

Figure 1. Execution times for small data sets at 16 processors.

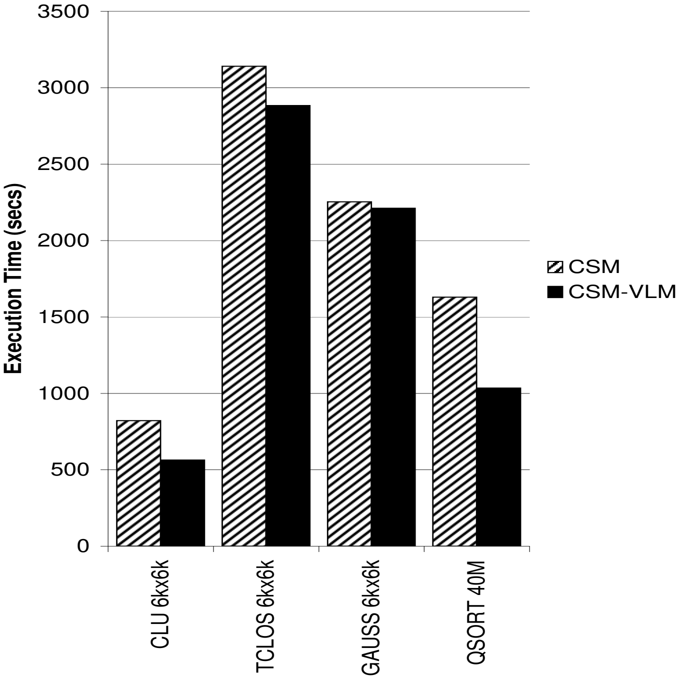

Figure 2. Execution times for large data sets at 16 processors.

of pages fetched from remote memory across the network (`Page-fetch`), the number of page faults encountered by each application (`Faults`), and the combined amount of data transferred through the network for both paging and coherence purposes (`Bytes`). The numbers in parentheses for both `Page-out` and `Page-in` indicate the maximum value at any node (the base numbers are an aggregate across all nodes). [2] These statistics are reported in Table 1.

Finally, for CSM-VLM, we have broken down the number of page evictions through the network into home evictions and other evictions and have collected statistics on the number of eviction attempts that failed due to the inability to find a new home node for the page. These statistics are reported in Table 2. Home evictions require sending the entire page to the new home, and therefore add to the total data traffic on the network. This is the statistic reported in Table 1 under `Page-net`, and represents the number of evictions that add a significant penalty in terms of time. (The Memory Channel uses programmed I/O and the communication cost is visible to the evicting process.)

Comparing CSM-VLM's `Page-fetch` and `Page-net` statistics with CSM's `Page-fetch` and `Page-in` statistics provides an indication of the reasons behind their relative performance. (As a rough approximation, `Page-ins` are assumed to be in the application's critical path, while `Page-outs` are performed in the background.) For three of the four applications, the `Page-fetch` numbers are fairly close. The only application with a significant increase in the number of pages fetched is QSORT, where the `Page-fetch` number is 4 times higher in CSM-VLM. However, disk I/O is replaced with cheaper network I/O in the form of `Page-fetches` and `Page-nets`, thereby resulting in improved performance for CSM-VLM.

CLU under CSM-VLM shows a runtime reduction of approximately 260 seconds (a performance improvement by a factor of 1.45) when compared with the CSM protocol. This performance improvement can be entirely attributed to the elimination of disk paging. The eviction policy we have chosen appears to do a reasonable job of choosing victim pages. This can be seen both in the modest increase in the number of faults and in the number of bytes transferred under CSM-VLM when compared with CSM. The number of pages fetched across the network also does not change. The

Application	Home Ev.	Other Ev.	Failed Ev.
CLU	20K	28K	2K
TCLOS	0	84K	2K
GAUSS	22K	65K	6K
QSORT	180K	65K	4K

Table 2. Characterization of network evictions in CSM-VLM.

[2] The operating system counts each page that is swapped out only once in the statistics, but if a number of processors simultaneously fault on a swapped-out page, the `Page-in` count will be incremented by the number of processors. This, along with the fact that clean pages do not add to the `Page-out` count, helps explain why some applications report more `Page-in` than `Page-out` counts.

Application	Protocol	Page-in	Page-out	Page-net	Page-fetch	Faults	Bytes
CLU	CSM	5.7K (1.6K)	14.0K (4.0K)	0	83.3K	205K	684M
	CSM-VLM	0.5K (0.2K)	0	20K	83.4K	205K	686M
TCLOS	CSM	7.1K (1.9K)	23.3K (6.0K)	0	110.6K	670K	913M
	CSM-VLM	0.9K (0.5K)	0	0	110.6K	627K	914M
GAUSS	CSM	20.2K (6.5K)	26.3K (7.3K)	0	121.6K	525K	999M
	CSM-VLM	0.5K (0.2K)	0K	22.2K	139.9K	554K	1151M
QSORT	CSM	181.9K (166.4K)	65.0K (49.2K)	0	81K	490.4K	2195M
	CSM-VLM	8.8K (3.6K)	13.6K (4.9K)	180K	331K	939.5K	3297M

Table 1. Paging and other statistics for regular Cashmere (CSM), and Cashmere-VLM (CSM-VLM). Numbers in parentheses for `Page-out` and `Page-in` are the highest values on any node.

modest increase in network traffic can be explained from the data in Table 2. As we can see, over two-thirds of the evicted pages did not require a home relocation and thus did not need to move any data through the network. Also, most of the evicted pages were not re-accessed.

TCLOS shows a smaller relative performance improvement than CLU. This is because CLU is more cache friendly due to its blocked nature and runs in significantly less time. As a result, the paging overhead is a more significant component of runtime and CLU benefits from CSM-VLM more than TCLOS does.

GAUSS shows a 2% runtime improvement over CSM when run under CSM-VLM. GAUSS' per-processor working set is not contiguous in memory. The lack of data contiguity confuses the operating system (OS) paging algorithm, and results in GAUSS paging four times as much data from disk as either TCLOS or CLU under base CSM. The existence of the back-substitution phase further complicates the application's access pattern as pages are accessed in reverse order from the first triangularization phase. Pages already evicted during the triangularization phase will be accessed again causing a higher number of `page-ins`. This latter effect renders our VLM eviction policy less effective as we evict pages that are later re-used. This can be seen by the increase in the number of pages fetched through the network when compared to the CSM protocol.

Quicksort is arguably the most interesting of the four applications since it has the highest amount of paging activity. Quicksort uses a work queue of sub-arrays that need to be sorted, which results in a non-deterministic access pattern by the processors. As can be seen, CSM-VLM reduces the runtime by over 36% when compared to CSM. The performance improvement over CSM can be entirely attributed to the significant reduction in paging activity without a corresponding increase in network traffic. However, due to the random access pattern of the application and limitations of our user-level network eviction policy, some disk paging remains even for the VLM protocol.

In our experiments, we also found that an efficient OS implementation of shared and virtual memory mappings is essential for the effective performance of CSM-VLM, and more generally, software DSMs. On an earlier, less optimized version of the OS, QSORT executed 2.2 times slower on CSM-VLM than on base CSM. This slowdown was due to the OS's poor management of large numbers of shared segments, which resulted in significant cost increases for `mprotect`, `shmat`, and `shmdt` calls. In particular, the cost of `mprotect` calls in moving from one segment to 16384 segments increased from 18μsecs to 1.7 msecs, while the cost of `shmat` and `shmdt` calls increased from 33 μsecs to 1.1 msecs. A recent upgrade to our OS solved these problems by greatly improving the management of shared segments. With the new, efficient OS implementation, CSM-VLM now outperforms base CSM by a factor of 1.58 on QSORT.

4 Related Work

Several previous studies have examined methods for using remote memory in a workstation cluster. The Global Memory System (GMS [6]) is one such system that provides network-wide page replacement decisions on a network of workstations. The system uses idle or "older" pages in the network to house replaced pages from highly active nodes. It does not, however, take advantage of application-specific information in order to make global page replacement decisions, nor does it try to optimize shared page placement.

Other implementations of network-based paging systems include those of Comer and Griffioen [4], and Felten and Zahorjan [7]. Comer and Griffioen use dedicated remote memory servers, while Felten and Zahorjan extend the idea to use memory on idle client machines, with a centralized registry process. Schilit and Duchamp [14] use network-based paging for mobile computing with limited resources. Franklin *et al* [8] describe a DBMS system using centralized remote memory paging at the server as a backing store. Markatos [12] extends the use of remote memory to commit transaction writes. These implementations generally target different environments (mobile computing, file systems, DBMSes), and do not attempt to take advantage of application-specific information to reduce the cost of paging in parallel applications.

Li and and Petersen [10] have implemented a system where they add memory modules on the I/O bus (VME bus) of a computer system. This memory can be used both as backing store (another level in the memory hierarchy), and as slower main memory accessed via simple load and store operations. Although this approach allows the use of memory as backing store, it does not attempt to take advantage of redundant resources on a cluster. Rather, it requires that dedicated resources be added to each individual workstation.

Perhaps the most closely related work is [11], in which Li describes a strategy to reduce page swapping in software DSM systems. The eviction policy described is similar to the one we use, with three key differences, however. First, we describe a working implementation using *multiprocessor* nodes rather than uniprocessor nodes. Additional issues such as the handling of shared segments arise in this case. Second, our implementation is for a network of workstations, rather than a multicomputer. Lastly, our implementation is for a release consistent rather than a sequentially consistent system, resulting in differences in the categorization of page types, and in the cost of network paging.

Finally, the JIAJIA DSM system [5] also attempts to extend a DSM protocol with evictions. However, it requires the user to decide on an appropriate assignment of homes to pages and disallows eviction of such home pages even if they are unused by the home node.

5 Conclusion and Future Work

We have implemented and evaluated a software DSM system that uses all of the memory in the cluster in order to reduce paging to disk. Leveraging application information can often eliminate expensive paging operations by taking advantage of copies of data that have been replicated throughout the cluster due to application data accesses. Our results show a performance improvement of up to a factor of 1.58 for our test applications run using large data sets.

Further work is needed to explore alternative eviction policies that can leverage application knowledge to a greater extent. One possibility would be to provide application upcalls from the runtime library and ask the application (or compiler) to indicate which data is no longer needed. Another issue that needs to be solved is the extension of the eviction mechanisms to account for private data as well as shared data. Even though private data is not replicated and thus the opportunity to take advantage of existing replicas does not exist, it might still be preferable to page through a fast high-bandwidth network if memory on a remote node is free. Finally, either an in-kernel implementation or the provision of kernel hooks to provide the necessary low-level information should improve performance considerably.

References

[1] S. V. Adve and K. Gharachorloo. Shared Memory Consistency Models: A Tutorial. In *Computer*, pages 66–76, December 1996.

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

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

[4] D. Comer and J. Griffioen. A New Design for Distributed Systems : The Remote Memory Model. In *Proceedings of the Summer 1990 USENIX Conference*, pages 127–135. June 1990.

[5] M. R. Eskicioglu, T. A. Marsland, W. Hu, and W. Shi. Evaluation of the JIAJIA Software DSM System on High Performance Computer Architectures. In *Proceedings of the '99 Hawaii International Conference on System Sciences Conference*. January 1999.

[6] M. J. Feeley, W. E. Morgan, F. P. Pighin, A. R. Karlin, H. M. Levy, and C. A. Thekkath. Implementing Global Memory Management in a Workstation Cluster. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, Copper Mountain, CO, December 1995.

[7] E. W. Felten and J. Zahorjan. Issues in the Implementation of a Remote Memory Paging System. Submitted for publication, November 1991.

[8] M. J. Franklin, M. J. Carey, and M. Livny. Global Memory Management in Client-Server DBMS Architectures. In *Proceedings of the 18th VLDB Conference*. August 1992.

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

[10] K. Li and K. Petersen. Evaluation of Memory System Extensions. In *Proceedings of the Eighteenth International Symposium on Computer Architecture*, pages 84–93, Toronto, Canada, May 1991.

[11] K. Li. *Scalability Issues of Shared Virtual Memory for Multicomputers*, Scalable Shared Memory Multiprocessors, pages 263–280. Kluwer Academic Publishers, 1991.

[12] E. P. Markatos. Using Remote Memory to Avoid Disk Thrashing: A Simulation Study. In *International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS'96)*, San Jose, CA, February 1996.

[13] D. J. Scales, K. Gharachorloo, and C. A. Thekkath. Shasta: A Low Overhead, Software-Only Approach for Supporting Fine-Grain Shared Memory. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 174–185, Cambridge, MA, October 1996.

[14] B. N. Schilit and D. Duchamp. Adaptive Remote Paging. Technical Report CUCS-004091, Department of Computer Science, Columbia University, February 1991.

[15] R. Stets, S. Dwarkadas, N. Hardavellas, G. Hunt, L. Kontothanassis, S. Parthasarathy, and M. Scott. Cashmere-2L: Software Coherent Shared Memory on a Clustered Remote-Write Network. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, St. Malo, France, October 1997.