Making Distributed Shared Memory Simple, Yet Efficient

Mark Swanson
Leigh Stoller
John Carter
Computer Systems Laboratory
University of Utah

Abstract

Recent research on distributed shared memory (DSM) has focussed on improving performance by reducing the *communication overhead* of DSM. Features added include lazy release consistency-based coherence protocols and new interfaces that give programmers the ability to hand tune communication. These features have increased DSM performance at the expense of requiring increasingly complex DSM systems or increasingly cumbersome programming. They have also increased the *computation overhead* of DSM, which has partially offset the communication-related performance gains. We chose to implement a simple DSM system, Quarks, with an eye towards hiding most computation overhead while using a very low latency transport layer to reduce the effect of *communication overhead*. The resulting performance is comparable to that of far more complex DSM systems, such as Treadmarks and Cashmere.

1 Introduction

Distributed shared memory (DSM) is a software abstraction of shared memory on a distributed memory multiprocessor or workstation cluster. The first DSM system, Ivy [18], was fairly simple: data was transparently managed in page-grained chunks, normal data and synchronization were treated identically, and all data was kept sequentially consistent [16]. Although Ivy performed well on some applications, it performed poorly for programs with significant degrees of sharing, which limited its applicability. Since this time, the driving goal of DSM research has been to improve performance by any means necessary [1, 2, 4, 7, 9, 11, 13, 14, 21].

The performance of DSM systems is dominated by two issues: (i) the *communication overhead* required to maintain consistency and (ii) the *computation overhead* required to service remote requests and compute the information needed to maintain consistency. Over the years, there has been extensive research aimed at dramatically reducing the *communication overhead* required to keep DSM-managed data consistent. These efforts have included:

- exploiting object encapsulation to give programmers the ability to precisely specify the coherence unit [1, 2, 7, 9],
- allowing programmers to explicitly lock data to specific nodes and thereby control at a fine grain when data is communicated [9, 11],
- supporting multiple programmer-selected coherence protocols [4],
- supporting multiple concurrent writers to eliminate false sharing [4, 14], and
- using various forms of relaxed consistency models, thereby increasing the restrictions imposed on the programmer but increasing the flexibility of the underlying DSM system in when it must communicate [2, 4, 14].

These innovations improved DSM performance to varying degrees, sometimes to the point where software DSM systems can rival the performance of hardware DSM systems for moderately-grained applications [8]. However, these efforts to reduce *communication overhead* often increase the system complexity and *computation overhead*. Modern DSM systems have added other overheads as well. Many modern systems require programmers to write their programs in very constrained ways [1, 2, 7, 13], in effect making the DSM programmer perform much of the work required of message passing programmers in terms of specifying what data should be shipped when. Other DSM systems require complex runtime systems or large amounts of DRAM for storing DSM state [14, 20].

To attack communication overhead, we implemented a very low overhead communication subsystem using a Direct Deposit Protocol[23] developed as part of the Avalanche multiprocessor project. In this paper we explore the extent to which DSM performance can be improved by attacking the other source of DSM overhead: the *computation overhead*. The availability of a low latency, high bandwidth communication layer allowed us to discard the traditional assumption that the frequency and volume of communication is the overriding factor determining performance, to be avoided at the cost of additional complexity in the basic protocols. Thus, rather than make heroic efforts to reduce the number of DSM operations performed, we decided to keep the core DSM system simple but implement it in a way that hides most of the computation overhead. We did so by eliminating many high overhead system calls and overlapping computation and communication whenever possible. The result is the Quarks DSM system [15].

DSM computation overhead comes in many forms in different systems, including:

- page fault and signal handling,
- system call overheads to protect and unprotect memory,
- thread context switch overheads,
- the time spent handling incoming coherence requests instead of useful user computation,
- the time spent calculating what pieces of shared data were modified (so-called *diff* computation),
- copying data to/from communication buffers, and
- time spent blocked on synchronous I/O.

Many DSM systems have attacked individual components of this overhead. For example, CRL [13], Midway [2], and Shasta [20] eliminate memory protection and page fault handling overheads by detecting memory references via annotated load and store operations. Unfortunately, these efforts turned out to increase system overhead [20] or add cumbersome programming requirements [2, 13].

Among the optimizations we applied to Quarks to reduce the impact of DSM computation overhead are the following:

- We employed a release-consistent write-update protocol, similar to that employed by Munin [4], which allowed *diff* creation to be performed in parallel with communication.
- Local *diffs* are applied to the clean *twins* as they are computed rather than copying a new twin on the next write to a page.
- Incoming messages may be handled asynchronously, when the local node is idle or at a convenient blocking point, rather than invoking a high overhead signal.
- Outgoing messages are sent using non-blocking I/O operations so that we can overlap *diff* computation with the transmission of *diffs*.
- We avoided using multithreading within the Quarks runtime system to support DSM operations, because we found that the impact of context switching exceeded the benefits provided.
- Twin creation and memory protection-related system calls are performed while other pages are being requested (i.e., we use split-phase RPC to do useful work while loading a remote page).

- *Diffs* are computed and sent before acquiring locks when the local node node is not the current owner (i.e., we overlap synchronization and DSM overhead).

The rest of this paper describes the detailed implementation of Quarks and the DDP communication layer (Section 2), discusses Quarks' performance (Section 3), compares our work with related systems (Section 4), and draws conclusions (Section 5).

2 Design of Quarks

A software DSM system must provide a small number of basic mechanisms: (i) shared memory creation and naming, (ii) shared memory consistency maintenance, and (iii) synchronization. Quarks' basic abstraction is that of *shared regions*, which are variable-length, page-aligned byte ranges. Quarks supports two consistency protocols: a release-consistent write update protocol and a sequentially-consistent write invalidate protocol. For synchronization, Quarks supports both locks and barriers, where locks are distributed while barriers are centralized. Many of Quarks' core mechanisms are derived from its predecessor, Munin[4].

Conventional wisdom holds that minimizing communication overhead is the key factor required to make DSM systems efficient. This has led to a wide variety of optimizations that reduce the number of messages required to maintain consistency, as described in Section 1. Recently, DSM systems have emerged that exploit highly specialized communication hardware to attack the problem of communication overhead head-on [17, 21]. In these systems, low message latency and high bandwidth communication translates directly into improved performance. Along these lines, when designing Quarks we exploited a very low latency messaging layer called Direct Deposit (DD) that was developed as part of the Avalanche multiprocessor project. The details of our implementation of DD can be found in Section 2.1. Although the results presented in Section 3 assume a DD communication layer with limited hardware support, we expect that the use of other modern lightweight protocols, such as Fast Messages[19], and commodity high-speed interconnects, such as Myrinet[3], would produce similar results.

The availability of a low latency, high bandwidth communication layer led us to focus on building simple, easy to implement, computationally cheap, and portable mechanisms wherever possible, even if they required additional messages. The only times messaging determined the design was when it affected synchronization between processes. In Section 2.2 we discuss Quarks' design, followed by a description of computation-reducing optimizations that explored, both successful (Section 2.3) and unsucessful (Section 2.4).

2.1 Direct Deposit Based Communication

The Quarks implementation described herein uses the Direct Deposit Protocol[23] as its transport layer. DD is an instance of a sender-based protocol[24] (SBP). The key concept of SBPs is a connection based mechanism that enables the sender to manage a reserved receive buffer within the receiving process' address space that is obtained when the connection is established. The sender directs placement of messages within that buffer via an offset carried within the message header. Coupled with an appropriate network interface, an SBP can achieve 0-copy message reception directly to the receiver's virtual address space.

DD uses a system call-based interface for sends to provide safety and flexibility at overhead and latency costs that are modest. The semantics of DD allow for *asynchronous* sends, i.e., the send call can simply request transmission of the message and return immediately. Given a network interface with DMA capanbility, the transmission actually occurs in parallel with continued computation within the user process. Quarks makes heavy use of this.

DD supports a completely user mode message reception. On message arrival, a *notification object*, or note, is written into a circular queue specified by the connection. This queue can be in kernel or user memory, and can be shared by several connections within a single process, at the discretion of the user. A user level receive consists simply of checking the valid flag of the next note. Polling for incoming messages is thus an extremely inexpensive operation.

When a note's valid bit is set, the user extracts the message address, size, and connection identifier from the note. As the message buffers can be in user virtual memory, this yields a completely user-mode receive capability. A system call variant of receive is provided to allow a process to sleep and wait for the arrival of a message notification on a specified queue.

DD is sufficiently integrated into the operating system that one can optionally request that a process be signalled when an incoming message is available. Quarks makes use of this to provide asynchronous handling of incoming requests, an alternative we chose to provide to explicit polling within the application code. With DD as its transport, Quarks has a very inexpensive polling capability. The choice of polling versus asynchronous, signal-driven message handling is left to the user; the best choice is application-dependent.

2.2 Consistency Management

In Quarks, consistency protocols are assigned on a per-region basis. Currently two protocols are supported: sequentially consistent write invalidate and release consistent write update. As in most DSM systems, the Unix `mprotect()` system call is used to modify the virtual memory protection of pages to determine when accesses to invalid (not locally mapped) pages or writes to read-only pages are made. The implementation of Quarks' two consistency protocols are detailed in this section.

Sequentially Consistent Write Invalidate

Quarks' write invalidate protocol is the same basic single-writer protocol that originated in Ivy[18]. Any number of nodes can have read-only copies of a given page. When any node attempts to modify such a shared region, the attempted store instruction will trigger a page fault, which is caught by the Quarks runtime system. In response, the runtime system will acquire an exclusive copy of the region, invalidate all other shared copies on other nodes, remap the region to read-write mode, and re-execute the failed store instruction. Reads from regions that have never been loaded or have been invalidated also cause page faults. In response to these faults, the Quarks runtime system locates a copy of the region, loads it, maps it read-only, and (if necessary) remaps any exclusive (read-write) copies that exist on other nodes before re-executing the failed load.

Release Consistent Write Update

Quarks' write update protocol was derived from Munin, and supports multiple concurrent writers to different portions of a shared region. It exploits the release consistency memory model [12], which (informally) allows processors to independently modify regions of shared data between synchronization points. When a process performs a *release* operation, it is required to ensure that any changes to shared data that have occurred on this node are flushed to remote nodes before they can perform the corresponding *acquire*[1]. To avoid such complexities as vector timestamps, indefinite buffering of changes, and distributed garbage collection[14], Quarks uses an eager implementation of release consistency, where changes to shared data are flushed to remote nodes immediately when a release operation is performed.

This design requires that Quarks track the specific bytes within each shared page that each writer modifies between flushes - writes to exclusive pages are not tracked. It does so using a *delayed update queue* and a simple *diffing* mechanism, as illustrated in Figures 1 and 2. Whenever a process writes to a read-only page, the page is added to a *delayed update queue* (DUQ), its access is changed to writable, and a twin (copy) of the page is made. When a process reaches a *release* point, all modifications to shared pages are made visible to other processes by the `purge_DUQ` routine. This is done by iterating over the pages contained on the DUQ, computing diffs and sending updates to other processes that share that page. Diffs are computed by comparing the current contents of the page to the contents of the twin made before the page was first modified. The resulting update mes-

[1] For Quarks, lock acquisition and release are acquire and release operations for the sake of release consistency, while barrier waits are effectively a release-acquire pair.

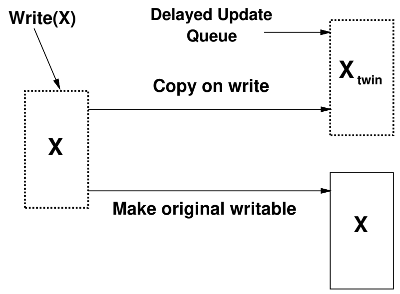

Figure 1: Tracking Changes Using the DUQ

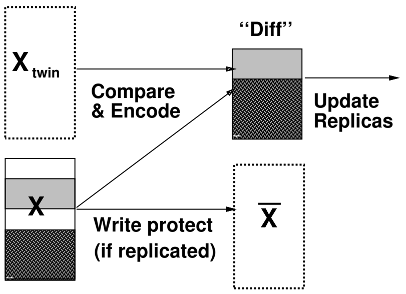

Figure 2: Sending Out *Diffs*

sage is sent to each member of the page's *copyset*, the set of other nodes that a node believes have a copy of a given page. On any given node, copysets may not be complete. However, the owner process always has a complete copyset and is always part of the copyset. The header of an update message contains the sender's idea of the current copyset. To ensure that updates are propagated to all copyholders, the owner does three things when it receives an update:

1. It forwards the update to nodes that are in the actual copyset, but which are not contained in the sender's idea of the copyset.

2. It returns the correct copyset to the sender in its acknowledgement.

3. It returns a count to the sender of the number of nodes to which it has forwarded the update.

As each node must know when its updates have been observed, it keeps a count of the number sent plus the number forwarded by owners on its behalf. The `purge_DUQ` routine does not return until it has received a number of acknowledgements equal to that count.

2.3 Useful Optimizations

Hiding Page Acquisition Stalls

Whenever a process accesses shared memory that is not valid, it cannot proceed until the page is supplied by the owner. However, the time spent waiting for the page need not be wasted. In Quarks, a number of operations can be - and are -performed after the page request is sent:

- The protection of the target page is changed to writable on the assumption that the page that is returned will need to written into it.

- If the failing access was a write, then a twin is allocated and made writable and the page is added to the delayed update queue.

When the page arrives, it is copied into the target page. It is also copied into the twin if the failing access was a write. If the access was a read, the target page is then downgraded to read-only.

On the owner side, the node requesting the page is added to the page's copyset. The owner then sends two reply messages to the requester: a header message containing the current copyset of the page and a second containing the data. Two messages are sent to avoid copying the page data into a buffer with the header. With the DD transport, it is much cheaper to send an additional message than to copy the page, both in raw latency seen by the requestor and in the overhead experienced by the owner. Since the only copying required is done by the network interface's DMA engine, the owner spends only a minimal amount of time servicing a page request. The impact of servicing remote page requests is a major factor in system performance, because it often interrupts useful work, so keeping it small can dramatically improve performance.

Hiding Diff Creation

Calculating diffs is one of the most time consuming activities in Quarks. Fortunately it is one activity that can frequently be overlapped with communication. `Purge_DUQ` will send out as many updates as it can, overlapping their calculation with their actual transmission. As each page in the DUQ is processed, it is removed from the DUQ and its protection is changed to read-only, to facilitate capturing future changes to the page. In practice, the protection changes are all performed together, interleaved with the reception of acknowlegdements, at the end of `purge_DUQ` for two reasons:

1. Other nodes may concurrently send updates for the same pages; it would then be necessary to change the page protection to write and back to read-only again for any such updates.

2. Doing so allows the process to get its updates out to other processes faster, so they can acknowledge them that much sooner.

No-Copy Twin Creation

We found that twin creation, which involves a 4-kilobyte `bcopy()`, adds significant overhead. This same observation is what drove Midway to detect writes explicitly rather than via diffs[2]. We reduced the impact of twin creation in the common case as follows. Originally, whenever a *clean* page is dirtied, a twin is saved to facilitate diff creation at the next release point. When a page's diffs are propagated, its twin become stale. As an optimization, we updated the twins to match the base page as part of the diff creation process. The benefits of this optimization are two-fold: (i) updating the twin occurs while both the real page and the twin are in the cache as a result of the diff calculation and thus is essentially free, and (ii) only those bytes that have actually changed need be written to the twin, rather than the entire 4-kilobytes.

Two factors might make this optimization less useful: (i) the page may not be modified again, so no twin would have been required, and (ii) updates for the page from other processes must be applied to both the page and the twin to keep them in sync. Depending on the number and size of these other updates, the simple copying approach might have been cheaper. Our experience showed that, while eager updating of twins sometimes lengthens the update propagation time, it has overall positive effects on runtime.

Reducing Synchronization Stalls

When a `lock_acquire()` occurs, the lock may either already reside in the locking process, or it may reside in another process. In the latter case, the DUQ is processed before the lock is acquired, even though release consistency only requires that changes to shared data be flushed before a lock is *released.* This optimization results in two update phases, one prior to acquiring the lock and one prior to releasing the lock, which might seem counterproductive at first glance. The logic behind the optimization is as follows. If many processes are contending for a lock, processes will spend significant time blocked prior to successfully acquiring the lock. In this case, it is better to perform most of the updates before blocking on the lock than it is to wait until the lock is released, because the flushing process would not be able to make useful progress anyway. Waiting to flush all data until the release point can serialize the update activity, which increases the time processes spend holding exclusive locks, since the DUQ must be processed prior to releasing the lock. This effect reduces performance. In the case where the lock is not heavily contended-for, our approach has little or no negative impact.

Sending Uninitialized Data

We optimize the case of accessing unitialized data. If a page has not been initialized by the owner when a page request is received, the owner simply sets a special flag in the response message to indicate a zero page and no data message is seent. In this case, ownership is passed to the node performing the page request.

2.4 Failed Optimizations

Two possible optimizations were attempted and subsequently rejected: (i) early propagation of updates and (ii) self-invalidation of unused pages. This should not be viewed as a general result showing them to be of no use, but rather a caution that they do not represent low-hanging fruit on the optimization tree.

Early Propagation of Updates

One problem with release consistent protocols is the potentially large number of update messages that each process must send at a release point. This can result in increased synchronization times or network congestion as many or all nodes simultaneously flood the network with update messages. We attempted to minimize these effects by setting a target length for the DUQ. If adding a new page to the DUQ would cause it to exceed this target length, the oldest entry would be removed and processed first.

We experienced two problems with this attempted optimization. First, for certain programs, the modification lifetimes of all the shared pages being modified overlapped, i.e., all of the modified pages had to be writable at the same time. A fix was added to allow such programs to avoid the livelock situatuion that occurred. The fix effectively turned the optimization off (dynamically) once the behavior was recognized.

The second problem was that the optimization slowed programs down by interleaving the propagation of updates with the real work of the program. Update propagation overhead that might have been at least partially hidden with otherwise idle time near a synchronization point was no longer hidden. In general, any DSM operation that interferes with useful user computation directly impacts performance and should be avoided.

Self-invalidation of Unused Pages

In self-invalidation [4], a process decides that certain shared pages are no longer of interest, and it unilaterally drops them from its working set. The advantage in doing this is that updates for those pages will no longer need to be applied by the process, saving it time. Other processes, when they find out about the diminished copyset, can save time by not having to send an update to the self-invalidated process, nor wait for an acknowledgement from it.

The drawbacks to this optimization are the cost of its implementation and the possibility of decreasing performance by self-invalidating pages that are subsequently referenced. The mechanism for selecting a page for self-invalidation is to remove read access to it and wait to see (by the occurrence of a read fault) if it is referenced. Together, the removal of read access and the handling of a signal, including unprotecting the page, represent a heavy cost for pages that are still part of the active working set. There is also a significant cost to any false-positives identified and then self-invalidated. The subsequent access, which defines the case as a false positive, causes a page reload since the process no longer has a valid copy of the data.

We used self-invalidation to support a competitive update scheme[4, 10]. However, we were unable to achieve any performance improvement using this method over experiments using a variety of threshhold values.

3 Experimental Evaluation

In this section, we evaluate the performance of software DSM system we have developed. First, the experimental setup is described. Next, Quark's is characterized in terms of basic operation costs. Finally, results for benchmark programs are presented.

3.1 Experimental Setup

The architectural support for DD is still under construction by the Avalanche project, so development and evaluation of Quarks was performed using that project's simulation tools.

The simulation results reported here were obtained from execution-driven simulations using the Paint simulator[22]. Paint simulates the HP PA RISC 1.1 architecture and includes an instruction set interpreter and detailed cycle-level models of a first level cache, system bus, and memory system similar to those found in HP J-class systems. In addition, it contains a detailed cycle-level model of the network interface and a simple Myrinet simulator that models contention only at node inputs. For these particular experiments, a simple processor is modeled at 120 MHz, with non-blocking loads and stores. Most instructions (except floating point) take one cycle. The cache is direct mapped, virtually indexed, 256 KB with 32 byte lines. It provides single-cycle loads and stores, a one-cycle load-use penalty, and data streaming. The Runway bus is also modeled at 120 MHz. The memory system contains 4 banks and a 17 entry write buffer. The Myrinet is modeled at 160 MHz, giving nearly 160 MB/second bandwidth; fall through time for the single switch is 27 processor cycles (approximately 200 ns.), propagation delays are 1 cycle on each side of the switch. In the experiments, cache misses to main memory had an average latency of 30 cycles. The simulation environment includes a small BSD-based kernel, which provides device drivers and provides system calls to set up connections, perform message send operations, support virtual memory (e.g., `mprotect()`, and handle signals.

Figure 3: Cost of basic OS operations.

Operation	Times	
	cycles	μsecs
Signal Delivery	1997	16.6
Signal Return	1474	12.3
Page Protect	941	7.8
Page Zero	1329	11.1
Page Copy	2576	21.5

Operation	Times	
	cycles	μsecs
Poll/Receive	70	0.6
Msg Transmit	160	1.33
1-way latency/64B	508	4.23
1-way latency/4KB	4160	34.66

Figure 4: Overheads and Latencies of Basic Messaging Operations.

3.2 Results

3.2.1 Basic Operations

To understand whole program performance, it is first necessary to know the cost of the basic operations underlying that performance. Figure 3 reports those costs for Quarks. Each time is the average obtained over several hundred iterations, so cold miss cache effects should be minimal. Page copy is an optimized version that assumes proper alignment of the data, which is the case in Quarks. The times for signal delivery and return from a signal handler are simply that: entry and exit costs. Any work performed to actually deal with the signal entails additional overhead.

It is interesting to compare those times with basic messaging overheads and latencies, given in Figure 4. The first two rows of Figure 4 represent the directly observable overhead. The next two rows present latency; a process with other work to do can overlap most of this time with useful work. The contribution of signal handling and data moving operations is easily equivalent to message latency. The effects of these same operations on *overhead* is an order of magnitude greater than that of messaging. This supports our claim that it can be appropriate to trade more messages for reduced complexity and reduced overhead.

Figure 5 reports times for a variety of basic Quarks costs. Once again, each time is the average obtained over several hundred iterations, so cold miss cache effects should be minimal. Barrier time is for 2 and 16 nodes; the latter is in parantheses. Getpage times are for non-zero pages; "R" means the page is initially in read mode, "W" means it is in write mode and a twin is made immediately. The getpage times with "Z" are for all-zero (not yet initialized) pages. Note that the bulk of the time in getpage is spent in local, non-message related activities: signal entry/exit, copying the page to its final destination, changing the protection of the page. Also note the seemingly anomalous larger cost of getting a read-only page. This results

Operation	Times	
	cycles	μsecs
Lock Acquire	1328	11.1
Barrier	2007(13726)	16.7(114.4)
Getpage (R)	16317	136.0
Getpage (W)	12775	106.5
Getpage (Z/R)	7152	59.6
Getpage (Z/W)	8003	66.7

Figure 5: Cost of Basic Quarks Operations.

Program	Problem Size	Times (ms)	
		Total	Compute
Radix	1024K keys	3368	1261
FFT	256K points	5737	3564
LU	1024x1024	42672	35313
Water	1331 mols	34621	34411
SOR	4096x4096	10129	10125
Gauss	768x768	20233	19579

Figure 6: Data set sizes and sequential execution tims.

from two factors: first, there is less work (relative to the "W" case) to overlap with the RPC to get the page from the owner. This results in the receive operation actually blocking in the kernel. Second, read only pages suffer a final setting of the page protection to read-only, which is not required for a writable page and which cannot be hidden.

3.2.2 Whole Program Results

In this section we report results using six programs. Three (FFT, LU,and water) are from the Splash2 suite[25]. SOR and gauss are from a study by Chandra, et al.[6]. Radix is from the Splash1 suite. Figure 6 gives the problem size for each application, and the uniprocessor version runtime.

Figure 7 displays the speedups in the parallel runtime phases for the 6 benchmarks. Note that the scale of the x axis (node count) is not linear. The to-

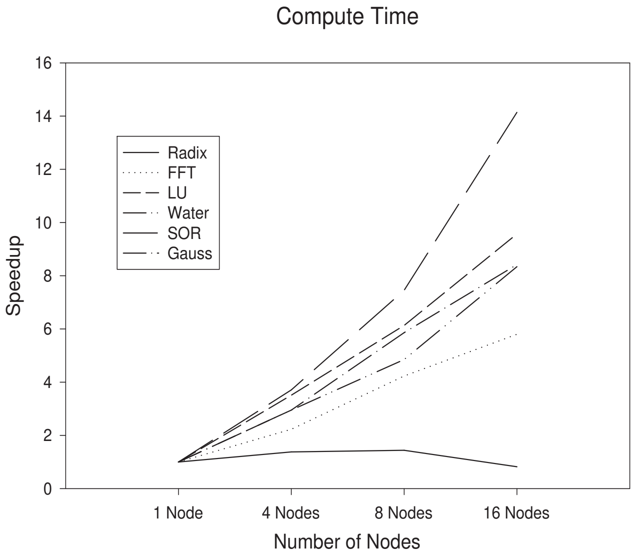

Figure 7: Speedup curves from parallel phase.

tal times include all initialization of data structures which is generally performed by the master process, and thus represents a serial computation.

In considering performance, we note first off that radix performs dismally for all configurations, due to its pathological sharing patterns. All processes write to all shared pages, resulting in all-to-all update propagation of a large number of pages at each synchronization. We include it only for completeness and as an indication of robustness under near-worst case sharing behavior.

On the other end of the spectrum is SOR, which only uses shared data for distributed computation of a global norm (basically a reduction operation). As a result, the near-linear speedup even out to 16 nodes is not surprising.

Lu also does quite well, with a speedup of 9.5 at 16 nodes. The sharing pattern in lu is such that each portion of the array is only written once, and then only by one process. More importantly, the writes occur before there are other sharers of the row, so almost no updates need be propagated. Lu does represent a good test of getpage performance, since processes continually read additional pages throughout the computation.

Gauss displays a speedup of 8 at 16 nodes. It has two primary uses of write-shared data. First, it performs a distributed reduction operation to select the next pivot row. Second, it uses shared memory to pass the selected row between the owner and the other processes. This approach is used rather than having the other processes access the row directly so that rows do not become widely shared, which would induce significant amounts of useless update traffic. Such sharing is useless since, other than the step at which a row is the pivot, only its owner needs to access it, i.e., sharing is one time only.

Fft performance, like that of lu, is primarily determined by getpage performance, for similar reasons. The speedups tail off quickly, which we attribute primarily to the small size of the problem that we were able to simulate.

Water achieves a speedup of about 8 at 16 nodes. It is by far the most challenging of the programs (other than radix), as measured by the amount of actively shared data. The run shown performs 16 synchronizations, sending out an average of 6000+ updates per node (in a 16 node run).

In summary, performance of Quarks is very good for one program (SOR) and reasonable for a number of others that vary in the emphasis of their sharing behavior on getpage (producer-consumer) or active write sharing. We expect that speedup would scale better with problem sizes that are currently beyond our ability to simulate. It is interesting to note that aggressive, complex optimization of updates would do little to help the getpage-dependent applications that are sensitive to reaction time on the part of the page's owner. A fine-grained implicit polling mechanism backed by a low cost polling primitive such as Direct Deposit's would be much more useful.

Figure 8: Parallel phase compute times for 4, 8, and 16 node runs. Times are in milliseconds.

Program	Nodes			
	1	4	8	16
Radix	1261	917	873	1539
FFT	3564	1466	756	614
LU	35313	10056	5760	3690
Water	34411	11656	7095	4128
SOR	10125	2732	1350	716
Gauss	19579	6637	3210	1964

4 Related Work

This section compares our work with a number of existing software DSM systems, focusing on the mechanisms used by these other systems to reduce the amount of communication necessary to provide shared memory. We limit our discussion to those systems that are most related to the work presented in this paper.

Ivy was the first software DSM system [18]. It uses a single-writer, write-invalidate protocol for all data, with virtual memory pages as the units of consistency. The large size of the consistency unit and the single-writer protocol makes Ivy prone to large amounts of communication due to false sharing.

Lazy release consistency [14], as used in TreadMarks [8], is an algorithm for implementing release consistency different from the one presented in this paper. Instead of updating every cached copy of a data item whenever the modifying thread performs a release operation, only the cached copies on the processor that next acquires the released lock are updated. Lazy release consistency reduces the number of messages required to maintain consistency, but the implementation is more expensive in terms of computation and memory overhead [14].

Midway [2] proposes a DSM system with *entry consistency*, a memory consistency model weaker than release consistency. The goal of Midway is to minimize communication costs by aggressively exploiting the relationship between shared variables and the synchronization objects that protect them. *Entry consistency* only guarantees the consistency of a data item when the lock associated with it is acquired. To exploit the power of entry consistency, the programmer must associate each individual unit of shared data with a single lock. For some programs, making this association is easy. However, for programs that use nested data structures or arrays, it is not clear if making a one-to-one association is feasible without forcing programmers to completely rewrite their programs.

Shasta [20] uses binary rewriting techniques to support the transparent interposition of software DSM on unmodified binaries. Load and store instructions are instrumented to test whether the shared data being accessed is locally present, and, for store instructions, writable. Shasta was able to support a version of Oracle Parallel Server on a workstation cluster, but its performance is currently poor due to the high overhead associated with checking every load and store to potentially shared data.

Cashmere [21] exploits DEC's Memory Channel hardware, which supports very high performance writes to remote memory, to reduce communication overhead, avoid the need for global directory locking, and avoid expensive TLB shootdowns. The resulting performance for Cashemere was quite good, ranging from 8 to 32 on 32 nodes, which emphasizes the importance of exploiting low latency networking technology whenever possible.

5 Conclusions

In this paper we have explored the notion that DSM systems can be made efficient without requiring heroic efforts to reduce the amount of communication induced by coherence management. We described the Quarks DSM system [15], which lacks many of the complex communication-reducing features of other modern DSM systems (e.g., lazy release consistency-based coherence protocols, cumbersome user-directed coherence, strict object semantics, etc.). Despite, or perhaps because of, its simplicity, Quarks performs comparably to these more sophisticated systems.

The source of Quarks' high performance is our attention to attacking the *computation overhead* of DSM. The time spent performing operations such as page fault handling, diff creation, synchronous I/O, memory protection manipulation and distributed garbage collection can seriously impact the rate at which useful user computation is performed. By carefully implementing Quarks in a way that avoids many of these sources of DSM overhead, and overlapping much of the rest with communication of synchronization stalls, we achieve high performance. As network performance continues to increase in dramatic fashion, we believe that attention to DSM computation overhead will become increasingly important.

As a side effect of its simplicity, Quarks is highly portable. It currently runs on five platforms (HP BSD, SunOS, Solaris, Linux, and DEC Unix), and ports are underway to other platforms. Source to Quarks can be obtained from `http://www.cs.utah.edu/projects/quarks`.

In the future, we plan to address three questions:

1. What communication optimizations continue to make sense in an environment with rapidly improving network performance? Although we have demonstrated that attacking communication overhead is only half the battle when implementing a high performance DSM systems, we believe that a limited number of optimizations geared towards reducing the number of I/O operations required to maintain consistency are warranted.

2. How can DSM technologies be employed to improve the performance, portability, and availability of *distributed applications* and services, such as distributed (clustered) file systems, distributed databases, and directory services [5]?

3. To what extent can highly tailored software DSM systems be made to perform comparably to dedicated hardware DSM machines, given modest amounts of hardware support? As part of the Avalanche multiprocessor project, we support a mixed message passing and DSM model as part of the base architecture. However, unlike dedicated DSM machines, we are implementing the DSM portion of the system as a combination of system-level software and specialized firmware in the communication infrastructure. Further work is necessary to evaluate how well this approach performs compared to both software DSM and fully custom hardware DSM.

References

[1] H. Bal, M. Kaashoek, and A. Tanenbaum. Orca: A language for parallel programming of distributed systems. *IEEE Transactions on Software Engineering*, pages 190–205, Mar. 1992.

[2] B. Bershad, M. Zekauskas, and W. Sawdon. The Midway distributed shared memory system. In *COMPCON '93*, pages 528–537, Feb. 1993.

[3] N. Boden et al. Myrinet – A gigabit-per-second local-area network. *IEEE MICRO*, 15(1):29–36, Feb. 1995.

[4] J. Carter, J. Bennett, and W. Zwaenepoel. Techniques for reducing consistency-related communication in distributed shared memory systems. *ACM Transactions on Computer Systems*, 13(3):205–243, Aug. 1995.

[5] J. Carter, A. Ranganathan, and S. Susarla. Building clustered services and applications using a global memory system. In *Submitted to The Eighteenth International Conference on Distributed Computing Systems*, 1998.

[6] S. Chandra, J. Larus, and A. Rogers. Where is time spent in message-passing and shared-memory programs? In *Proceedings of the 6th Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 61–73, Oct. 1994.

[7] J. Chase, F. Amador, E. Lazowska, H. Levy, and R. Littlefield. The Amber system: Parallel programming on a network of multiprocessors. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 147–158, Dec. 1989.

[8] A. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, and W. Zwaenepoel. Software versus hardware shared-memory implementation: A case study. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 106–117, May 1994.

[9] P. Dasgupta, et al., The design and implementation of the Clouds distributed operating system. *Computing Systems Journal*, 3, Winter 1990.

[10] M. Dubois, L. Barroso, J. Wang, and Y. Chen. Delayed consistency and its effects on the miss rate of parallel programs. In *Proceedings of Supercomputing'91*, pages 197–206, 1991.

[11] B. Fleisch and G. Popek. Mirage: A coherent distributed shared memory design. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 211–223, Dec. 1989.

[12] K. Gharachorloo, D. Lenoski, et al. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 15–26, Seattle, Washington, May 1990.

[13] K. Johnson, M. Kaashoek, and D. Wallach. CRL: High performance all-software distributed shared memory. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, 1995.

[14] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy consistency for software distributed shared memory. *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 13–21, 1992.

[15] D. Khandekar. Quarks: Distributed shared memory as a basic building block for complex parallel and distributed systems. Technical Report Master Thesis, University of Utah, March 1996.

[16] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. on Computers*, C-28(9):690–691, Sept. 1979.

[17] J. Laudon and D. Lenoski. The SGI Origin: A cc-NUMA highly scalable server. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 241-251, May 1997.

[18] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, Nov. 1989.

[19] S. Paikin, Lauria, and A. Chien. High performance messaging on workstations: Illinois fast messages (FM) for Myrinet. In *Proceedings of Supercomputing '88*, 1995.

[20] D. Scales and K. Gharachorloo. Towards transparent and efficient distributed shared memory. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, Oct. 1997.

[21] 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 16th ACM Symposium on Operating Systems Principles*, Oct. 1997.

[22] L. Stoller, R. Kuramkote, and M. Swanson. PAINT-PA instruction set interpreter. Technical Report UUCS-96-009, University of Utah - Computer Science Department, Sept. 1996.

[23] L. Stoller and M. Swanson. Direct deposit: A basic user-level protocol for carpet clusters. Technical Report UUCS-95-003, University of Utah - Computer Science Department, March 1995.

[24] J. Wilkes. Hamlyn - an interface for sender-based communication. Technical Report HPL-OSR-92-13, Hewlett-Packard Research Laboratory, Nov. 1992.

[25] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24–36, June 1995.