Techniques for Reducing Consistency-Related Communication in Distributed Shared Memory Systems

John B. Carter, John K. Bennett and Willy Zwaenepoel
Computer Systems Laboratory
Rice University
Houston, TX 77251-1892

Abstract

Distributed shared memory (DSM) is a software abstraction of shared memory on a distributed memory machine. One of the key problems in building an efficient DSM system is to reduce the amount of communication needed to keep the distributed memories consistent. In this paper we present four techniques for doing so: (1) software release consistency; (2) multiple consistency protocols; (3) write-shared protocols; and (4) an update-with-timeout mechanism. These techniques have been implemented in the Munin DSM system. We compare the performance of seven application programs on Munin, first to their performance when implemented using message passing and then to their performance when running on a conventional DSM system that does not embody the above techniques. On a 16-processor cluster of workstations, Munin's performance is within 5% of message passing for four out of the seven applications. For the other three, performance is within 29% to 33%. Detailed analysis of two of these three applications indicates that the addition of a function shipping capability would bring their performance within 7% of the message passing performance. Compared to a conventional DSM system, Munin achieves performance improvements ranging from a few to several hundred percent, depending on the application.

This research was supported in part by the National Science Foundation under Grants CDA-8619893, CCR-9010351, CCR-9116343, by the IBM Corporation under Research Agreement No. 20170041, by the Texas Advanced Technology Program under Grants 003604014 and 003604012, and by a NASA Graduate Fellowship.

1 Introduction

1.1 Background

There are two fundamental models for parallel programming and for building parallel machines: shared memory and distributed memory or message passing. The *shared memory model* is a direct extension of the conventional uniprocessor model wherein each processor is provided with the abstraction that there is but a single memory in the machine. A update to shared data therefore becomes visible to all the processors in the system. In contrast, in the *distributed memory model* there is no single shared memory. Instead, each processor has a private memory to which no other processor has direct access. The only way for processors to communicate is through explicit *message passing*.

Distributed memory machines are easier to build, especially for large configurations, because unlike shared memory machines they do not require complex and expensive hardware cache controllers [6, 36, 44, 48, 52]. The shared memory programming model is, however, more attractive since most application programmers find it difficult to program machines using a message passing paradigm that requires them to explicitly partition data and manage communication. Using a programming model that supports a global address space, an applications programmer can focus on algorithmic development rather than on managing partitioned data sets and communicating values.

A distributed shared memory (DSM) system provides a shared memory programming model on a distributed memory machine (see Figure 1). The system consists of the same hardware as that found in a distributed memory machine, with the addition of a software layer, represented by the dashed line, that provides the *abstraction* of a single shared memory. In practice, each memory remains physically independent, and all communication takes place through explicit message passing performed by the DSM software layer. DSM systems combine the best features of shared memory and distributed memory machines. They support the convenient shared memory programming model on distributed memory hardware, which is more scalable and less expensive to build. However, although many DSM systems have been proposed and implemented [4, 11, 12, 19, 27, 34, 7, 42, 46, 47, 49, 51], achieving good performance on DSM systems for a sizable class of applications has proven to be a major challenge.

To illustrate this challenge, Figure 2 shows how a conventional DSM system implements the

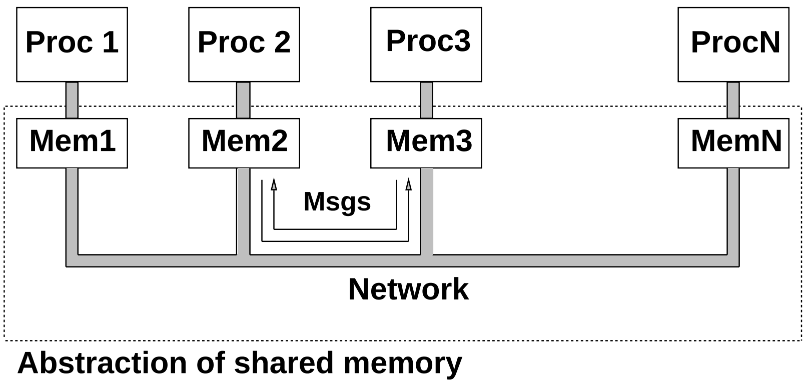

Figure 1 High-Level View of a Distributed Shared Memory System

shared memory abstraction [42]. The global shared address space is divided in pages, depicted by a shaded box in the global address space in Figure 2. The local memory of each processor is used as a cache on the global shared address space. The boxes at the bottom of Figure 2 represent the pages that are present in each processor's local memory. When Proc1 attempts to access a page of global virtual memory for which it does not have a copy, a page fault occurs. This page fault is handled by the DSM software. which retrieves a copy of the missing page from another node, in this case Proc3, and restarts Proc1. If the access is a read, then the page becomes replicated in read-only mode. If the access is a write, then all other copies of the pages are invalidated. Throughout the rest of this paper, the term *conventional DSM* [42] refers to a DSM system that employs a page-based write-invalidate consistency protocol, such as the one just described.

The primary source of overhead in a conventional DSM system is the large amount of communication that is required to maintain consistency, or, put another way, to maintain the shared memory abstraction. Ideally, the amount of communication for an application executing on a DSM system should be comparable to the amount of communication for the same application executing directly on the underlying message passing system. Conventional DSM systems have found it difficult to achieve this goal because of restrictive memory consistency models and inflexible consistency protocols. The *false sharing* problem is an example of this phenomenon. False sharing occurs when two threads on different machines concurrently update different shared data items that lie in the same virtual memory page. In conventional DSM systems, this false sharing can cause a page to "ping-pong" back and forth between different machines. In contrast, in a message passing system, each thread would independently update its own copy of the data, without unnecessary communication. Some of these problems can be overcome by carefully restructuring the shared memory programs to reflect the way that the DSM system operates. For example, one could decompose the shared data into small page-aligned pieces or one could introduce new variables to reduce the amount of false sharing. However, this restructuring can be as tedious and difficult as using message-passing directly.

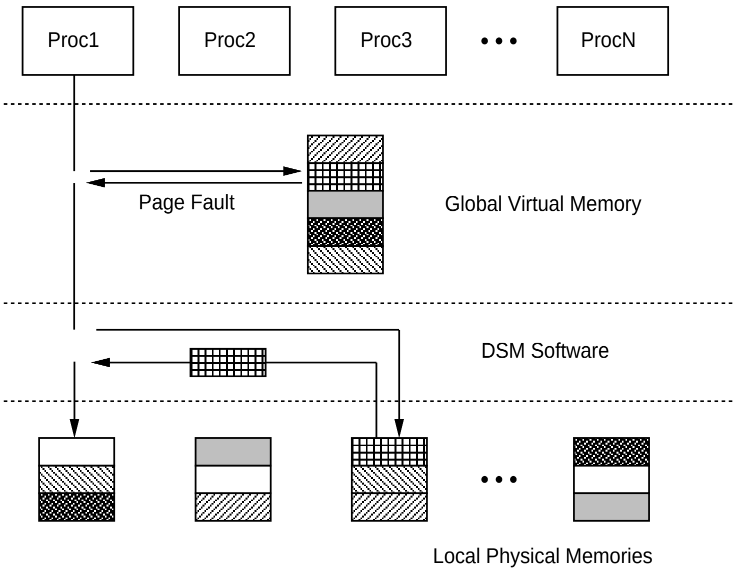

Figure 2 Conventional Distributed Shared Memory Implementation

1.2 Summary of Results

In this paper, we present the following four techniques for reducing the amount of communication needed for keeping the distributed memories consistent.

1. *Software release consistency* is a software implementation of release consistency [29], specifically aimed at reducing the number of messages required to maintain consistency in a software DSM system. Roughly speaking, release consistency requires memory to be consistent only at specific synchronization points.

2. *Multiple consistency protocols* are used to keep memory consistent in accordance with the observation that no single consistency protocol is the best for all applications, or even for all data items in a single application [8, 26].

3. *Write-shared protocols* address the problem of false sharing in DSM by allowing multiple processes to write concurrently into a shared page, with the updates being merged at the appropriate synchronization point, in accordance with the definition of release consistency.

4. An *update-with-timeout* mechanism, which is in essence an update protocol that causes remote copies of shared data to be updated rather than invalidated. However, copies that are not referenced during the last timeout interval are deleted, eliminating the need for further updates and thus reducing the total amount of communication.

These techniques have been incorporated in the Munin DSM system. Munin has been implemented on a network of SUN-3/60 workstations running the V-System [20]. The Munin programming interface is the same as that of conventional shared memory parallel programming systems, except that it requires (i) all synchronization to be visible to the runtime system, and (ii) all shared variables to be declared as such, and (optionally) annotated with the consistency protocol to be used. Other than that, Munin provides thread, synchronization, and data sharing facilities like those found in many shared memory parallel programming systems, e.g., Presto [10]. Munin does not currently support thread migration.

To evaluate the benefits of these optimizations, we measured the performance of seven shared memory parallel programs: Matrix Multiplication (MULT), Finite Differencing (DIFF), both a coarse-grained and a fine-grained version of the Traveling Salesman Problem (TSP-C and TSP-F), Quicksort (QSORT), Fast Fourier Transform (FFT), and Gaussian Elimination with partial pivoting (GAUSS). Three versions of each program were written, a message passing version, a Munin DSM version, and a conventional DSM version. The computational aspects of all three versions of each application were identical. The conventional DSM versions use a page-based write-invalidate protocol as described in Section 1.1.

Munin's performance is within 5% of message passing for MULT, DIFF, TSP-C, and FFT. For TSP-F, QSORT, and GAUSS, performance is within 29% to 33%. Detailed analysis of TSP-F and QSORT indicates that the addition of a function shipping capability would bring their performance within 7% of the message passing performance. Compared to a conventional DSM system, Munin achieves performance improvements ranging from a few percent for MULT to several hundred percent for FFT.

1.3 Outline of the Paper

The rest of this paper is organized as follows. Section 2 describes the techniques for reducing consistency-related communication. Section 3 summarizes some aspects of the implementation that are relevant to the performance evaluation. Section 4 describes the applications used in the evaluation, as well as the experimental methodology. Section 5 contains an overview of the results, followed by a program-by-program comparison of the performance of the Munin, message passing, and conventional DSM versions in Section 6. Section 7 attempts to isolate the benefits of the different techniques used to reduce consistency-related communication. Section 8 explores the additional performance benefits that could be achieved by the use of function shipping. Related work is discussed in Section 9. We conclude in Section 10.

2 Techniques for Reducing Communication

If DSM systems are to approach the performance of their message passing counterparts, then their communications requirements cannot be substantially higher than those of message passing systems. In this paper we explore four techniques for achieving this goal. These techniques are discussed in detail in this section.

2.1 Software Release Consistency

2.1.1 Release Consistency

Conventional DSM systems, as described in Section 1, employ the *sequential consistency* model [40] as the basis for their consistency protocols. Sequential consistency essentially requires that any update to shared data become visible to all other processors before the updating processor is allowed to issue another write to shared data [42]. This requirement imposes severe restrictions on possible performance optimizations. These restrictions have led to both theoretical [43] and empirical [8, 59] arguments that DSM systems based on sequential consistency require a substantial amount of communication and are thus inefficient. Therefore, we chose to explore a more relaxed notion of consistency in DSM systems.

Among the various relaxed memory models that have been developed [24, 29, 31, 43], we opted for the release consistency model developed as part of the DASH project [29, 41]. Release consistency exploits the fact that programmers use synchronization to separate accesses to shared variables by different threads. The system then only needs to guarantee that memory is consistent at (select) synchronization points. This ability to allow temporary, but harmless, inconsistencies is what gives release consistency its power. Consider for instance a program where all access to shared data is enclosed in critical sections. Release consistency guarantees that when a thread successfully acquires the critical section lock, it gains access to a version of shared data that includes all modifications made before the lock was last released. Similarly, for a program where all processes synchronize at a barrier, when a thread departs from the barrier, it is guaranteed to see all updates made by all other threads before they reached the barrier. In general, if a program is free of data races, or, in other words, if there is synchronization between all conflicting shared memory accesses, then the program generates the same results on a release consistent memory system as it would on a sequentially consistent memory system. Experience with release consistent memories indicates that because of the need to handle arbitrary thread preemption, most shared memory parallel programs are free of data races even when written assuming a sequentially consistent memory [16, 28].

More formally, the following constraints on the memory subsystem ensure release consistency:

1. Before an ordinary `read` or `write` is allowed to perform with respect to any other processor, all previous `acquire` accesses must be performed.

2. Before a `release` access is allowed to perform with respect to any other processor, all previous `read` and `write` accesses must be performed.

3. Synchronization accesses must be sequentially consistent with one another.

Lock acquires and releases map in the natural way on to `acquires` and `releases`. A barrier arrival is treated as a `release`, while a barrier departure is treated as an `acquire`. Release consistency relaxes the constraints of sequential consistency in three ways: (i) ordinary reads and writes can be buffered or pipelined between synchronization points, (ii) ordinary reads and writes following a release do not have to be delayed for the release to complete (i.e., a release only signals the state of *past* accesses to shared data), and (iii) an acquire access does not have to delay for previous ordinary reads and writes to complete (i.e., an acquire only controls the state of *future* accesses to shared data). The first point is the primary reason for release consistency's efficiency. Because ordinary reads and writes can be buffered or pipelined, a release consistent memory can mask much of the communication required to keep shared data consistent.

2.1.2 Buffered Update versus Pipelined Invalidate Release Consistency

The hardware implementation of release consistency in the DASH project *pipelines* invalidation messages caused by writes to shared data. This implementation is primarily geared towards masking the latency of writes, rather than reducing the number of messages sent. In a software DSM system, where the overhead of sending messages is very high, it is more important to reduce the frequency of communication than it is to mask latency by pipelining messages. For this reason, we developed an implementation of release consistency that *buffers* writes instead of pipelining them, as illustrated in Figures 3 and 4. These figures illustrate how writes to three shared variables (x, y, and z) within a critical section are handled by an implementation of release consistency that uses pipelining and an implementation that uses buffering, respectively. When a processor writes to several different replicated data items within a critical section, the pipelining scheme sends one message per write, while the buffering implementation buffers writes to shared data until the subsequent release, at which point it transmits the buffered writes. Ideally, the buffering implementation reduces the number of messages transmitted from one per write to one per critical section when there is a single replica of the shared data. The dashed line portion of the execution graph represents the delay that a processor experiences when releasing a lock. Because the buffering implementation delays all writes until the release point, it must transmit all buffered writes then, increasing the latency of releases. Nevertheless, the reduction in the number of messages far outweighs the effect of the higher release latencies.

Buffering and pipelining reduce the cost of writes, but have no effect on the cost of read misses. In software DSM systems, the cost of these read misses is very high, both in terms of communication

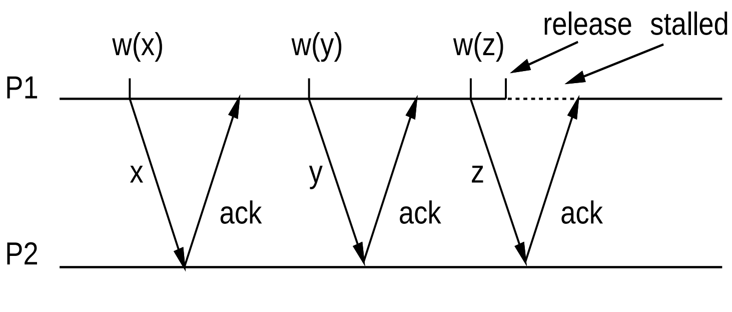

Figure 3 Pipelining Invalidations

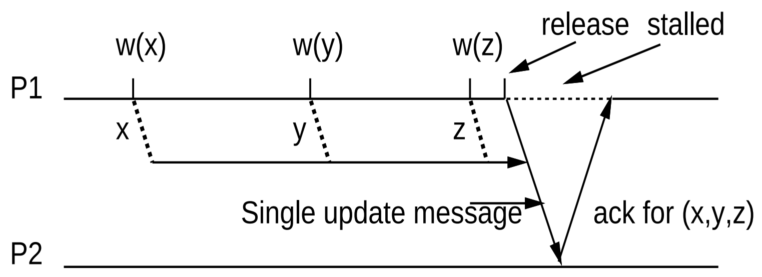

Figure 4 Buffering and Merging Updates

and in terms of the length of time that a thread stalls before resuming after a read miss. The impact of read misses can be partially mitigated by using an update protocol. Update protocols based on sequential consistency have fallen out of favor because of the large amount of communication required to send update messages for every write. An update protocol based on release consistency can, however, buffer writes, which reduces substantially the amount of communication required.

2.2 Multiple Consistency Protocols

Most DSM systems employ a single protocol to maintain the consistency of all shared data. The specific protocol varies from system to system. For instance, Ivy [42] supports a page-based write-invalidate protocol (see Section 1) while Emerald [34] uses object-oriented language support to handle shared object invocations. Each of these systems, however, treats all shared data the same way. The use of a single protocol for all shared data leads to a situation where some programs can be handled effectively by a given DSM system, while others cannot, depending on the way in which shared data is accessed by the program. To understand how shared memory programs characteristically access shared data, we studied the access behavior of a suite of shared memory parallel programs. The results of this study [8] and others [1, 23, 26, 50, 54, 56] support the notion that using the flexibility of a software implementation to support *multiple consistency protocols* can improve the performance of DSM. They also suggest the types of access patterns that should be supported: *conventional*, *read-only*, *migratory*, *write-shared*, and *synchronization*[1].

Conventional shared variables are replicated on demand and are kept consistent using an invalidation-based protocol that requires a writer to be the sole owner before it can modify the data. When a thread attempts to write to replicated data, a message is transmitted to invalidate all other copies of the data. The thread that generated the miss blocks until all invalidation messages are acknowledged. This single owner consistency protocol is typical of what existing DSM systems provide [21, 27, 42], and is what we use exclusively to represent a conventional DSM system in our performance evaluation.

Once *read-only* data has been initialized, no further updates occur. Thus, the consistency protocol simply consists of replication on demand. A runtime error is generated if a thread attempts

[1]The results of our original study indicated that there were eight basic access patterns (private, write-once, migratory, write-many, producer-consumer, result, read-mostly, and synchronization), but experience has made it clear that several of the protocols were redundant. Specifically, the *result* and *producer-consumer* access patterns were subcases of the *write-shared* access pattern.

to write to read-only data. Read-only data is provided as a special case of conventional data for debugging purposes.

Migratory data is accessed multiple times by a single thread, including one or more writes, before another thread accesses the data [8, 56]. This access pattern is typical of shared data that is accessed only inside a critical section or via a work queue. The consistency protocol for migratory data propagates the data to the next thread that accesses the data, provides the thread with read *and* write access (even if the first access is a read), and invalidates the original copy. This protocol avoids a write miss and a message to invalidate the old copy when the new thread first modifies the data.

Write-shared variables are frequently written by multiple threads concurrently, without intervening synchronization to order the accesses, because the programmer knows that each thread reads from and writes to different portions of the data. Because of the way that the data is laid out in memory, access to write-shared data suffers from the effects of *false sharing* if the DSM system attempts to keep these different portions of the data consistent at all times. This protocol is discussed in more detail in Section 2.3.

We support three types of *synchronization* variables: locks, barriers, and condition variables. Because synchronization variables are accessed in a fundamentally different way than normal data objects, it is important that synchronization *not* be provided through shared memory, but rather via a suite of synchronization library routines or similarly specialized implementation. Doing so reduces the number of messages required to implement synchronization, especially compared to conventional spinlock algorithms [5], and thereby reduces the amount of time that threads spend blocked at synchronization points.

2.3 Write-Shared Protocol

The write-shared protocol is designed specifically to mitigate the effect of false sharing, as discussed in Sections 1 and 2.2. False sharing is a particularly serious problem for DSM systems for two reasons: (i) the consistency units are large, so false sharing is very common, and (ii) the latencies associated with detecting modifications and communicating are large, so unnecessary faults and messages are particularly expensive. The write-shared protocol allows concurrent writers and buffers writes until synchronization requires their propagation. For instance, Figure 5 shows two processes concurrently updating a page. The updates are buffered locally, and when the processes arrive at a barrier, their updates are sent to the other processor.

In order to record the modifications to *write-shared* data, the DSM system initially write protects the pages containing the data. When a processor first writes to a page of write-shared data, the

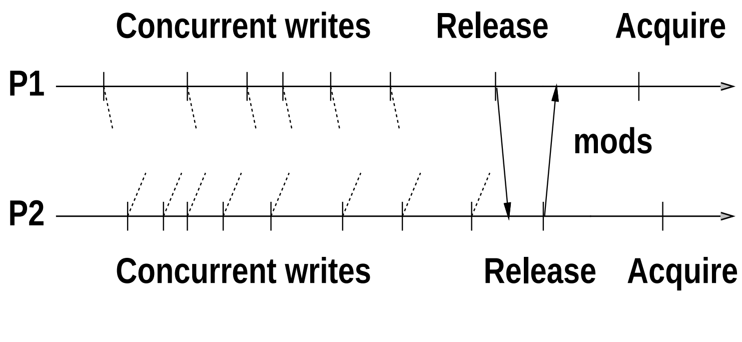

Figure 5 Concurrent Access in the Write-Shared Protocol

DSM software makes a copy of the page (a *twin*), and queues a record for the page in the delayed update queue (see Figure 6). The DSM them removes write protection on the shared data so that further writes can occur without any DSM intervention. At release time, the DSM system performs a word-by-word comparison of the page and its twin, and run-length encodes the results of this *diff* into the space allocated for the twin (see Figure 7). Each encoded update consists of a count of identical words, the number of differing words that follow, and the data associated with those differing words. Each node that has a copy of a shared object that has been modified is sent a list of the updates that are available. Nodes receiving update notifications request the updates they require[2], decode them, and merge the changes into their versions of the shared data, A runtime switch allows this comparison to be performed at the byte level, as opposed to the word level, if the data is more finely shared.

Another runtime switch can be set to check for conflicting updates to write-shared data. If this switch is set, then, when a *diff* arrives at a processor that has a dirty copy of the page, the DSM system checks whether any of the updates in the *diff* conflict with any of the local updates, and,

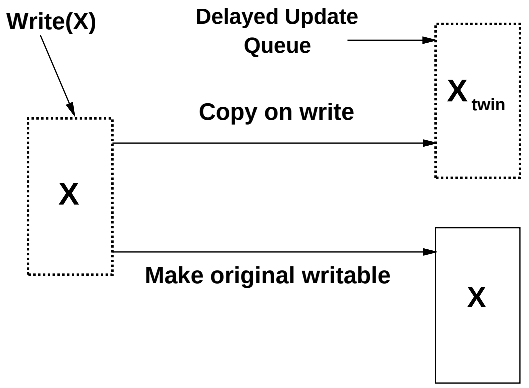

Figure 6 Write-Shared Protocol: Creating Twins

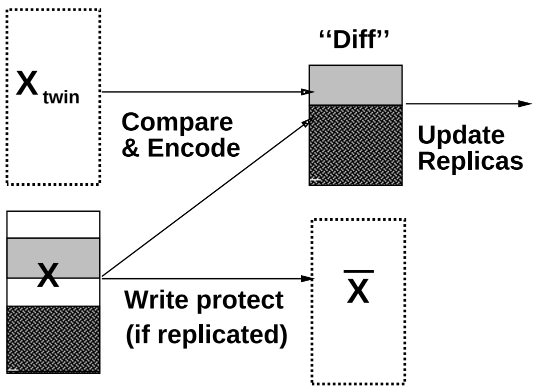

Figure 7 Write-Shared Protocol: Sending Out *Diffs*

[2]If all of the encoded updates fit into a single packet, they are sent directly in place of the list of available updates, thus eliminating unnecessary communication in the event that only a small amount of shared data has been modified.

if so, signals an error. The ability to detect conflicting updates allows Munin to support dynamic data race detection.

The results of several trace studies indicate that the overhead of buffering updates and transmitting them at synchronization points is small compared to the amount of overhead eliminated by reducing the amount of communication required to maintain consistency. The average number of different objects accessed between synchronization points is small [8], so the DSM system only needs to enqueue changes infrequently. Furthermore, when a shared data object needs to be updated, it usually resides on a small number of other processors [1, 50]. Thus, the number of update messages required to maintain consistency is small.

2.4 Update Timeout Mechanism

Update protocols suffer from the fact that updates to a particular data item are propagated to all of its replicas, including those that are no longer being used. In DSM systems, this problem becomes even more severe, because the main memories of the nodes in which the replicas are kept are very large and it takes a long time before a page gets replaced, if at all. Without special provisions, updates to these *stale* replicas can lead to a large number of unnecessary consistency messages, resulting in poor performance. This effect is a major reason that existing commercial multiprocessors use invalidation-based protocols. We address this problem with a timeout algorithm similar to the competitive snoopy caching algorithm devised by Karlin [35]. The goal of the update timeout mechanism is to invalidate replicas of a cached variable that have not been accessed recently upon receipt of an update.

An example of the problem and our solution are illustrated in Figure 8. In the figure the circles represent four processors in the system, and the arrows represent the source and destination of updates for a particular shared variable. In this example, exactly two processors access the variable at any given time. A dark shaded circle represents a processor that is actively using the shared variable, while a cross-hatched circle represents a processor that is not using the shared variable but that is still receiving updates to it. In Figure 8(a), only the two processors using the variable are sending or receiving updates involving the variable, which is the desired situation. In Figure 8(b), only two processors are using the variable, but a third is receiving updates to it because it recently used the variable and still has a copy of it in its memory. Without the use of an update timeout mechanism, this processor will continue to receive updates to the variable until the program terminates or the variable is invalidated because of memory capacity limits, even if it never again uses the variable. If, over the entire execution of the program, every processor accessed the variable at some point, then eventually every processor would receive updates for the variable, even though at most two processors were using it at a time. The timeout mechanism

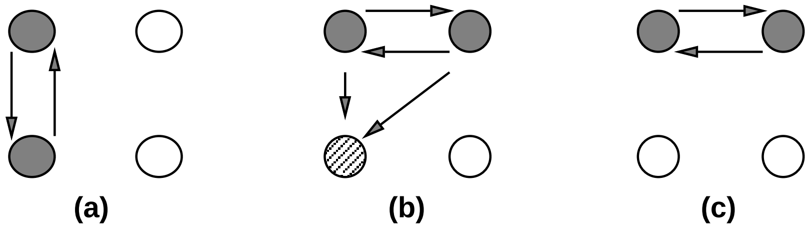

Figure 8 Update Timeout Mechanism

eliminates these logically unnecessary updates by invalidating stale replicas. Figure 8(c) illustrates the situation after the timeout mechanism has invalidated the stale replica that had been cached in the lower left processor. Only the two processors using the variable are sending or receiving updates involving the variable, which is the desired situation.

Munin's update timeout mechanism is implemented as follows. When receiving an update for a page for which no twin exists locally, the page is mapped such that it can only be accessed in supervisor mode, and the time of receipt of this update is recorded. A local access causes a fault, as a result of which protection is removed and the timestamp is reset. If the page is still in supervisor mode when another update arrives (meaning it has not been accessed locally since the first update), and a certain time window δ has expired (50 milliseconds in the prototype), then the page is invalidated, and a negative acknowledgement is sent to originator of the update, causing it to no longer to send updates to this processor.

The update timeout mechanisms works well in conjunction with the implementation of the write-shared protocol (see Section 2.3) whereby a node first sends out a lists of updates that it has available. The recipients of this message then select which updates they need to receive. No updates are requested for the pages that were invalidated by the timeout mechanism, thereby avoiding the propagation of these updates.

The use of update timeouts results in a hybrid update-invalidate protocol that allows Munin to gain the benefits of an update mechanism, i.e., the reduction in the number of read misses and subsequent high-latency (idle) reloads, while at the same time retaining the superior scalability of an invalidation protocol by limiting the extent to which stale copies of particular pages are updated.

3 The Munin DSM Prototype

The techniques described in Section 2 have been implemented in the Munin DSM system. Munin was evaluated on a network of SUN-3/60 workstations running the V-System [20] connected via an isolated 10 megabit per second Ethernet. This section provides a brief overview of aspects of the implementation of Munin that are relevant to its evaluation. A more detailed description of the Munin prototype appears elsewhere [15, 16].

3.1 Writing A Munin Program

Munin programmers write parallel programs using threads, as they would on many shared memory multiprocessors. All of the current applications were written in C. Synchronization is supported by library routines for the manipulation of locks, barriers and condition variables.

Munin currently supports only statically allocated shared variables. This is an artifact resulting from the fact that none of the current applications use dynamically allocated shared data. Support for dynamic allocated shared data could easily be added. The programmer annotates the declaration of shared variables to specify what protocol to use to keep shared data consistent, e.g., "`shared {protocol} <C_type> <variable_name>`". The keyword `shared` is required to specify that a variable will be shared among processes, although the `protocol` can be omitted. If the protocol is omitted, the conventional protocol is used. The choice of conventional as the default protocol was arbitrary, all of the shared data in the test programs was fully annotated. Incorrect annotations may result in inefficient performance or in runtime errors that are detected by the Munin runtime system, but not in incorrect results.

3.2 Compiling and Linking a Munin Program

A preprocessor filters the source code in search of shared variable declarations. For each such declaration, the preprocessor removes the Munin-specific "`shared {protocol}`" portion and adds an entry to an auxiliary file. After preprocessing, the source file is compiled with the regular compiler. The Munin linker reads the auxiliary file and relocates the shared variables to a shared segment. By default, the linker places each shared variable on a separate page. In addition, the Munin linker appends to the executable a shared segment symbol table that describes the layout of the shared memory and the protocols to be used for the shared data. These additions to Munin executables had a negligible impact on program size or startup costs.

3.3 Runtime Overview

Figure 9 illustrates the organization of a Munin program during runtime. On each participating node, the Munin library is linked into the same address space as the user program, and thus can access user data directly. The two major data structures used by the Munin runtime system are the *delayed update queue* (see Section 2), and the *object directory*, which maintains the state of the shared data being used by local user threads. A Munin system thread installs itself as the page fault handler for the Munin program. As a result, the underlying V kernel [20] forwards to this system thread all memory exceptions. The Munin thread also interacts with the V kernel to communicate with the other Munin nodes over the network, and to manipulate the virtual memory system as part of maintaining the consistency of shared memory. The prototype uses no

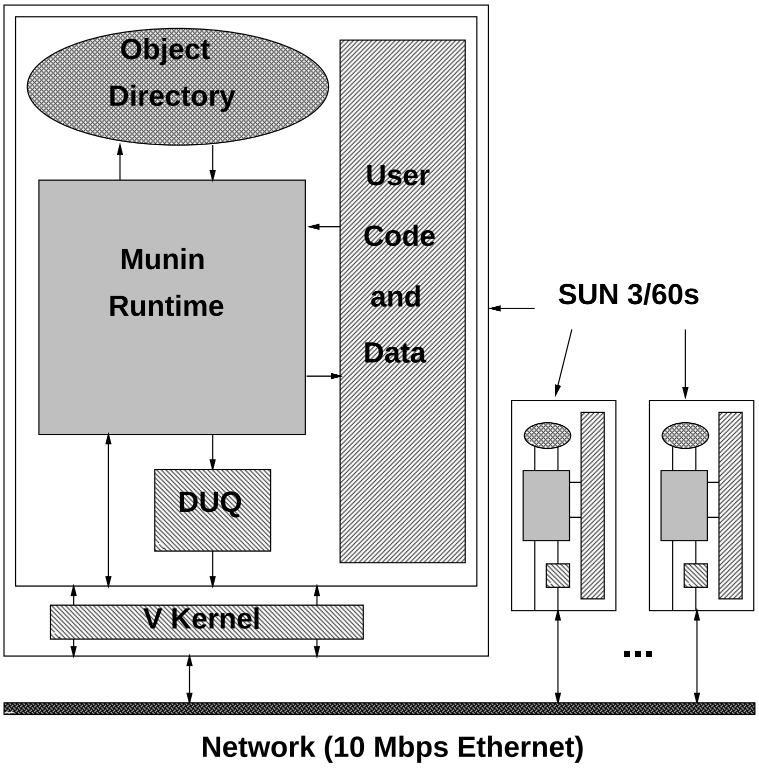

Figure 9 Munin Runtime Organization

features of V for which equivalent features are not commonly available on other platforms (e.g., Unix or Mach). In addition, we avoided using features that we felt might not be common on future workstation clusters, such as reference bits in the page table or a multicast capability on the network. For the update timeout mechanism, references are detected by mapping write-shared pages to supervisor mode so that the first reference to a page after an update to it is received results in a page fault. We thus maintain a reference bit and timestamp for each page without requiring hardware supported reference bits. Although the prototype runs on a collection of workstations connected via an Ethernet, the multicast capability of Ethernet is not used so that our results can be generalized to platforms without hardware multicast.

3.4 The Object Directory

On each node, the Munin runtime system maintains an *object directory* containing information on the state of each data item in the global shared memory, as shown in Figure 9. All shared variables on the same physical page are treated as a single object. Variables that are larger than a page, e.g., a large array, are treated as a number of independent page-sized objects. Munin uses variables rather than pages as the basic unit of granularity because this better reflects the way data is used and reduces the amount of false sharing between unrelated variables [8].

Munin's object directory is structured as a hash table that maps a virtual address to an entry that describes the data located at that address. The object directory on the Munin root node is initialized from the shared data description table located in the executable file. The object directory on the other nodes is loaded when the node is created, and then initialized to indicate that no shared data is present. When Munin cannot find an object directory entry in the local hash table, as occurs the first time a node other than the root node accesses a shared data item, it requests a copy from the root node. Directory entries include the following fields:

- **Lock**: provides exclusive access to the entry to serialize access to directory entries by different threads.
- **Start address** and **Size**: act as keys for looking up a shared data item's directory entry, given an address within the data.
- **Protocol**: specifies the protocol that the fault handler should use when servicing access faults on the data item.
- **State**: characterizes the dynamic state of the data, such as whether it is present locally and whether it is writable.
- **Copyset**: represents a "best guess" of the set of processors with copies of the data.
- **Probable owner**: represents a "best guess" of the owner of the data, which has different interpretations depending on the protocol being used to keep the data consistent.
- **Performance statistics**: collect runtime statistics used for performance tuning.

The data item's *copyset* is the set of all nodes that the local node believes have a copy of the data, either because a copy was present when the local node originally accessed the data, or the local node has handled requests from the other nodes for the data. It is possible for the copyset to include nodes no longer caching the data because they have deleted their copy, and it is also possible that nodes that are caching copies of the data are not in the copyset, because another node satisfied their load request. If a copy of the data resides locally, the transitive closure of the copysets is guaranteed to be a superset of the set of nodes that have a copy of the data, because an entry in a copyset is only removed when the corresponding node informs the other nodes that it is no longer caching the data. It is this property that ensures the correctness of the distributed ownership version of the update mechanism.

The *probable owner* is used to determine the identity of the Munin node that currently owns the data [42]. The owner node is used by the conventional and migratory protocols to arbitrate write access to the data. For the write-shared protocol, the owner node represents the copy of last resort that cannot be unilaterally purged from memory (e.g., as part of the update timeout mechanism), without finding another willing node to become the owner of last resort. This approach is analogous to the copy of last resort used in cache-only multiprocessors [55]. Munin implements a dynamic ownership protocol to distribute the task of data ownership across the nodes that use the data. When a shared data item is not owned by the local node, the information in the *probable owner* field acts as a "hint" to reduce the overhead of performing consistency operations.

The object directory maintenance protocol is designed to reduce the number of messages required to maintain the distributed object directory. Munin distributes the state information associated with write-shared data across the nodes that contain cached copies of the data. In many cases, this elimination of the notion of a static "owner" of data allows nodes to respond to requests completely locally. This approach also allows Munin to exploit locality of reference when maintaining directory information, since the need to maintain a single consistent directory entry, as has been proposed for most scalable shared-memory multiprocessors [3, 17, 18], is eliminated.

3.5 Synchronization Support

Synchronization objects are accessed in a fundamentally different way than ordinary data [8]. Thus, Munin provides efficient implementations of locks, barriers, and condition variables that directly use V's communication primitives rather than synchronizing through shared memory. More elaborate synchronization mechanisms, such as monitors and atomic integers, can be built using these basic mechanisms. Each Munin node maintains a synchronization object directory, analogous to the data object directory, containing state information for the synchronization data. All of Munin's synchronization primitives cause their invoking thread to block on an "acquire" and cause the local delayed update queue to be purged on a "release".

3.5.1 Locks

Munin employs a queue-based implementation of locks similar to existing implementations on shared memory multiprocessors [30, 45]. This allows a thread to request ownership of a lock and block awaiting a reply, without repeated queries. The system associates an ownership "token" and a distributed queue with each lock. A *probable owner* mechanism like that described above is used to locate the token or the end of the queue associated with the lock. The token migrates to nodes as they become owners, so no single node is responsible for maintaining the state of a given lock. This has the same benefits in terms of exploiting locality of reference, removing central bottlenecks, and reducing communications, as the distributed data ownership protocol. A frequent situation in which this scheme works to particular advantage is when a thread attempts to reacquire a lock for which it was the last owner. In this case, the thread finds the associated token to be available locally, and is thus able to acquire the lock immediately (without any message overhead). Similarly, if a small subset of threads continuously reuse the same lock, they communicate only with one another.

When the lock ownership token is unavailable locally, a message is sent along the probable owner chain to the last lock holder. If the lock is free (the token is available), the last lock holder forwards the token to the requester, which acquires the lock and continues executing. Otherwise, the thread that was at the end of the queue stores the locking thread's identity into a local data structure without replying. Each enqueued thread knows the identity of the thread that follows it on the queue, if any, so when a thread releases a lock and the associated queue is non-empty, lock ownership is forwarded directly to the next thread in the queue after all delayed updates are flushed in accordance with the requirements of release consistency.

3.5.2 Barriers

Barriers are used to simultaneously synchronize multiple threads. When a barrier is created, the user specifies the number of threads that must reach the barrier before it is lowered. When a thread wishes to wait at a barrier, it flushes any delayed updates, sends a message to the barrier manager thread (a well-known thread located on the root node), and awaits a response. When all of the threads have arrived at the barrier, the barrier manager replies to each waiting thread to let it resume. We considered using a distributed barrier mechanism similar to those designed for scalable multiprocessor systems [32], but for the small size of the prototype implementation, a simple centralized scheme was more practical and efficient. Unlike locks, which are point-to-point and which exhibit a high degree of locality that makes it beneficial to migrate ownership, barriers are most often used to synchronize all of the user threads in the program. In this case, locality of reference cannot be exploited, because single threads or small subsets of threads do not tend to access the barrier without intervening accesses by other threads. Thus, until the single barrier manager becomes a bottleneck, there is no reason to distribute barrier ownership.

3.5.3 Condition Variables

Munin's condition variables are essentially binary semaphores that also support a broadcast wakeup capability. Unlike locks, condition variables give threads the capability to synchronize indirectly. Any thread can perform a signal operation, while the lock protocol allows only the lock owner to release the lock. While it is possible to build this kind of mechanism using locks, we found it convenient to include condition variables as a primitive. In accordance with the requirements of the release consistency model, delayed modifications are flushed before the signal or broadcast message is forwarded to the condition manager thread.

4 Evaluation

4.1 Application Programs

Seven application programs were used in the evaluation. Three different versions of each application were written: a Munin DSM version, a conventional DSM version that used the conventional protocol for a sequentially consistent memory, and a message passing version. Great care was taken to ensure that the the "inner loops" of each computation, the problem decomposition, and the major data structures for each version were identical. Except where noted, all array elements are double precision floating point numbers. Both the DSM system and the message passing programs used V's standard communication mechanisms.

The DSM programs were originally written for a shared memory multiprocessor (a Sequent Symmetry). Our results may therefore be viewed as an indication of the possibility of "porting" shared memory programs to software DSM systems, but it should be recognized that better results may be obtained by tuning the programs to a particular DSM environment. Table 1 summarizes the seven application programs and problem sizes. An effort was made to select a suite of programs that would represent a relatively wide spectrum of shared memory parallel programs, varying in their parallelization techniques, granularity, degree and nature of sharing, and locality of shared data references. Matrix Multiply (MULT), Finite Differencing (DIFF), and Gaussian Elimination with partial pivoting (GAUSS) are numeric problems that statically distribute the data across the threads. MULT, DIFF, and GAUSS exhibit increasing degrees of sharing. FFT dynamically reallocates the data across threads, and exhibits an extremely high degree of sharing. The Traveling Salesman Problem (TSP) and Quicksort (QSORT) programs use the task queue model to dynamically allocate work to different threads. The granularity for TSP was varied (TSP-C and TSP-F access data at a coarse and fine grain, respectively). QSORT exhibits a high degree of false sharing in the array to be sorted. Small to moderate problem sizes were chosen so that the uniprocessor running times would be in the range of hundreds of seconds, and the sixteen processor running times would be on the order of tens of seconds. The uniprocessor running times represent sequential implementations of the programs with all synchronization and communication removed.

4.2 Experimental Methodology

For all three versions of each program, the sequential initialization routine is executed on the root node. Then the appropriate number of additional nodes are created, which for the DSM versions gives each node a copy of the non-shared data. The non-root nodes initialize themselves, and then synchronize with the root node by waiting at a barrier in the DSM versions and via an explicit message in the message passing versions. For the DSM versions, after the user thread on the root node creates the required worker threads, it reads the clock to get the initial value and then waits at the barrier, which causes the computation to begin. For the message passing versions, the root thread waits until all of the worker threads have sent it the initialization message. It then reads the initial clock value and sends a message to each of the workers to start computation. At this point, the workers read their inputs, via page faults for the DSM versions or via request messages for the message passing versions. Once all of the workers have completed, the root thread again reads its clock and calculates the total elapsed computation time.

In addition to execution times, the Munin runtime system gathers statistics on the number of

Program	Problem size
MULT	400 by 400 square matrices
DIFF	512 by 512 square matrices
TSP-C	18-city tours, recurse when 13 cities left
TSP-F	18-city tours, recurse when 12 cities left
QSORT	256K elements, recurse when less than 1024
FFT	32K elements
GAUSS	256 by 256 square matrices

Table 1 Programs and Problem Sizes Used in Experiments

faults, the amount of data transferred, and the amount of time stalled while performing various consistency operations. The message passing kernel collects similar data. Selected portions of these statistics are used throughout the analysis to highlight the reasons for observed performance differences between the different versions of the programs.

5 Overview of Results

The main results we report are the speedup of the various versions of the parallel programs over the sequential version, measured for 2 to 16 processors. Figures 10 through 16 show the speedup for each of the application programs as a function of the number of processors. Table 2 shows the speedup achieved on sixteen processors for the three versions of each application. The percentages in parentheses represent the percentage of the message passing speedup achieved by Munin, and the percentage of both the message passing and the Munin speedup achieved by the conventional DSM implementation. Tables 3 and 4 show the amount of communication required during execution of the programs on sixteen processors, both in terms of number of messages and kilobytes of data transmitted.

For MULT, DIFF, TSP-C, and FFT, the Munin versions achieved over 95% of the speedup of their hand-coded message passing equivalents, while for TSP-F, QSORT, and GAUSS the Munin

	Message Passing	Munin DSM	Conventional DSM
MULT	14.7	14.6 (100%)	14.5 (99%, 99%)
DIFF	12.8	12.3 (96%)	8.4 (66%, 68%)
TSP-C	13.2	12.6 (96%)	11.3 (86%, 90%)
TSP-F	8.9	6.0 (67%)	4.7 (53%, 80%)
QSORT	13.4	8.9 (67%)	4.1 (31%, 46%)
FFT	8.6	8.2 (95%)	0.1 (0%, 0%)
GAUSS	12.1	8.6 (71%)	5.1 (42%, 59%)

Table 2 Speedups Achieved (16 processors)

Program	Message Passing	Munin	Conventional
MULT	672	1567	1490
DIFF	14164	14646	35486
TSP-C	902	7870	7940
TSP-F	919	9776	10194
QSORT	667	31866	129428
FFT	9225	15322	1594952
GAUSS	14768	26034	32349

Table 3 Number of Messages for 16-Processor Execution

Program	Message Passing	Munin	Conventional
MULT	640	1384	1327
DIFF	8294	3645	26534
TSP-C	68	4163	4770
TSP-F	68	4989	5963
QSORT	524	14565	101007
FFT	9339	11621	1336317
GAUSS	4995	5526	7388

Table 4 Amount of Data (in Kilobytes) for 16-Processor Execution

programs achieved between 67% and 71%. For the programs with large grain sharing (MULT and TSP-C), the conventional versions achieved 99% and 90%, respectively, of the speedup of their Munin counterparts. For DIFF, TSP-F, QSORT, and GAUSS the performance of the conventional versions was reduced to 46-80% of Munin. For FFT, there was so much false sharing that the conventional version slowed down by a factor of ten when run on more than one processor.

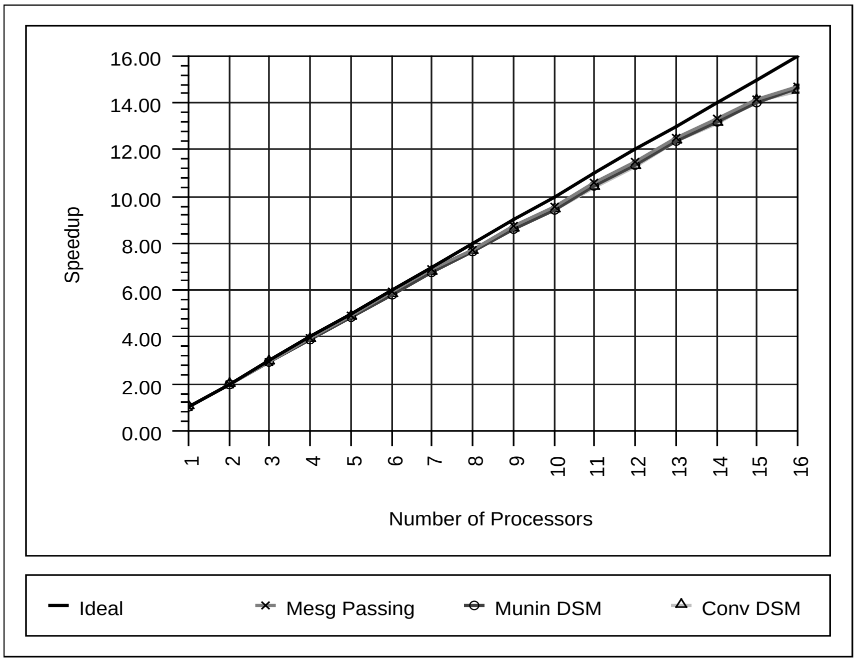

Figure 10 Matrix Multiplication (MULT)

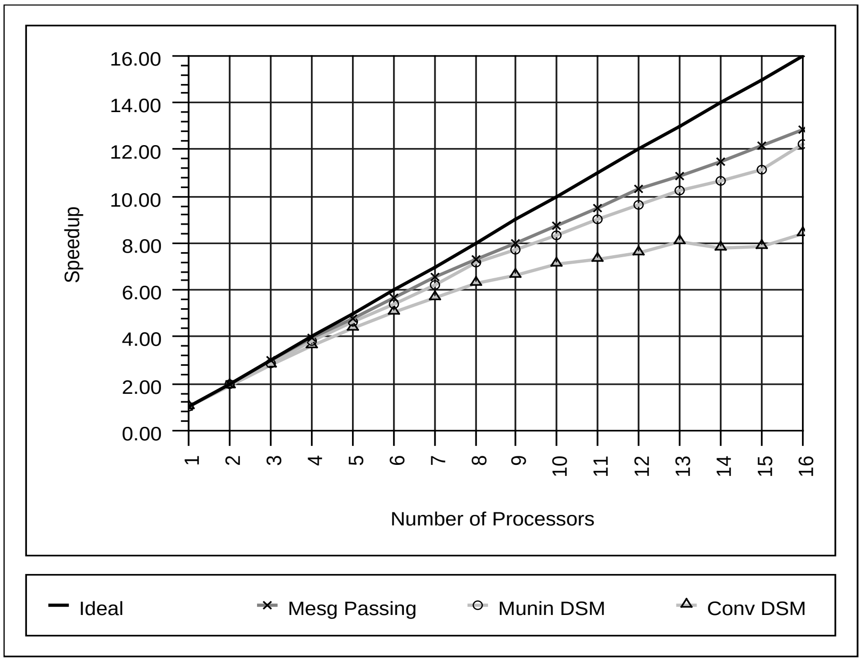

Figure 11 Finite Differencing (DIFF)

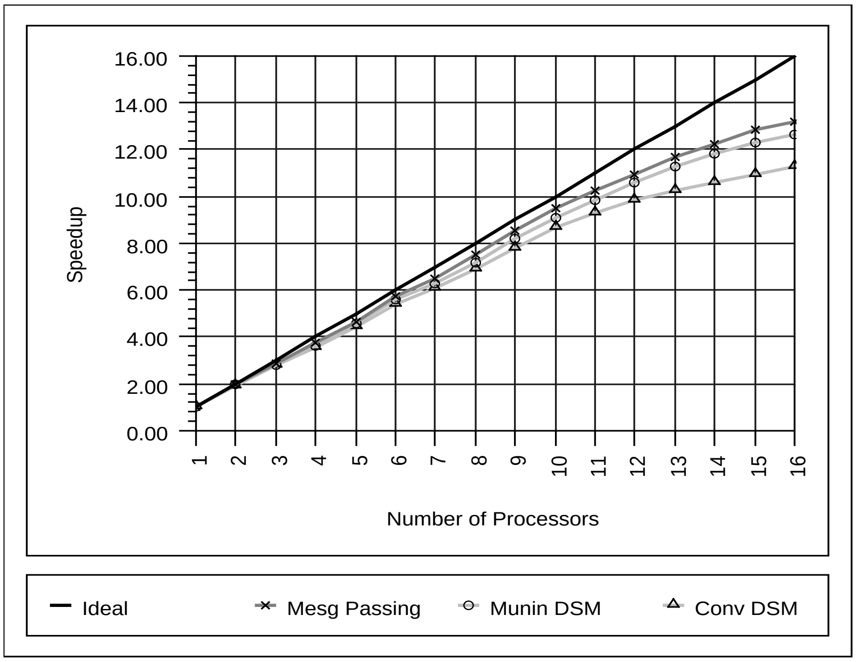

Figure 12 Coarse-Grained Traveling Salesman Problem (TSP-C)

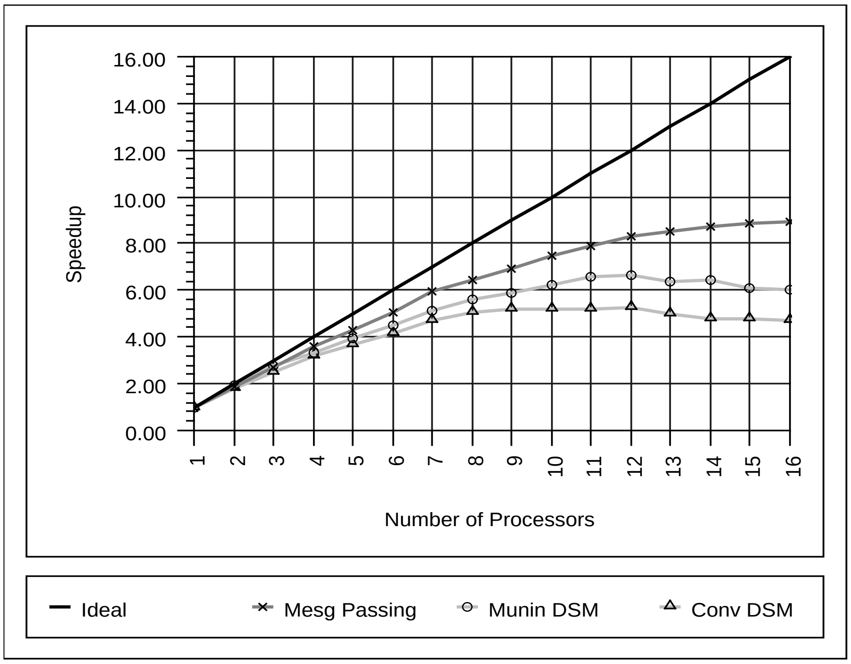

Figure 13 Fine-Grained Traveling Salesman Problem (TSP-F)

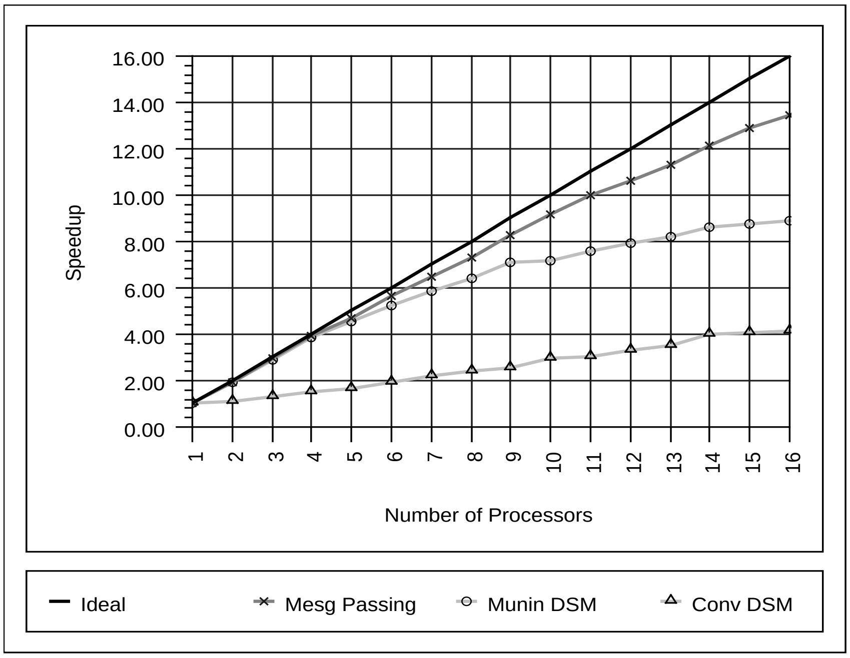

Figure 14 Quicksort (QSORT)

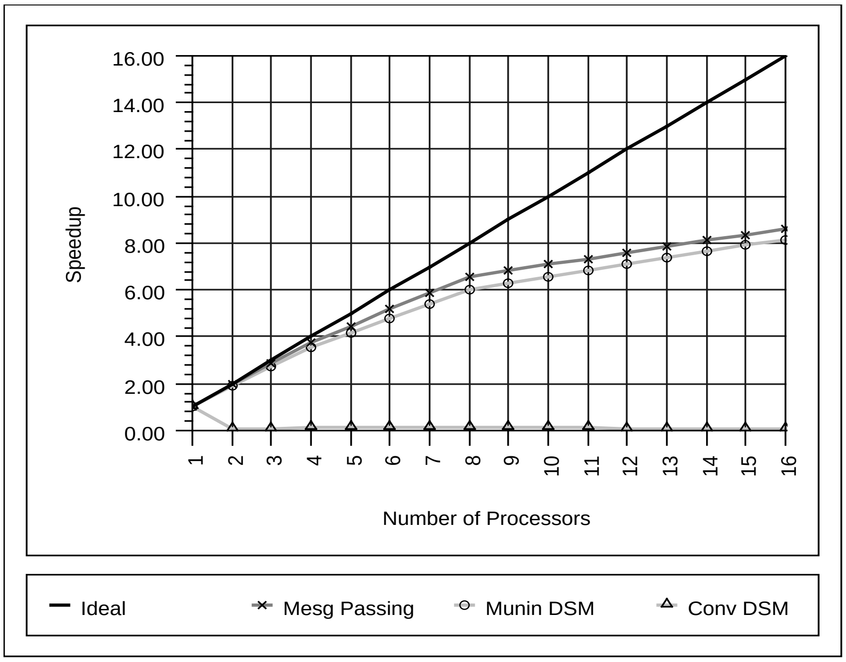

Figure 15 Fast Fourier Transform (FFT)

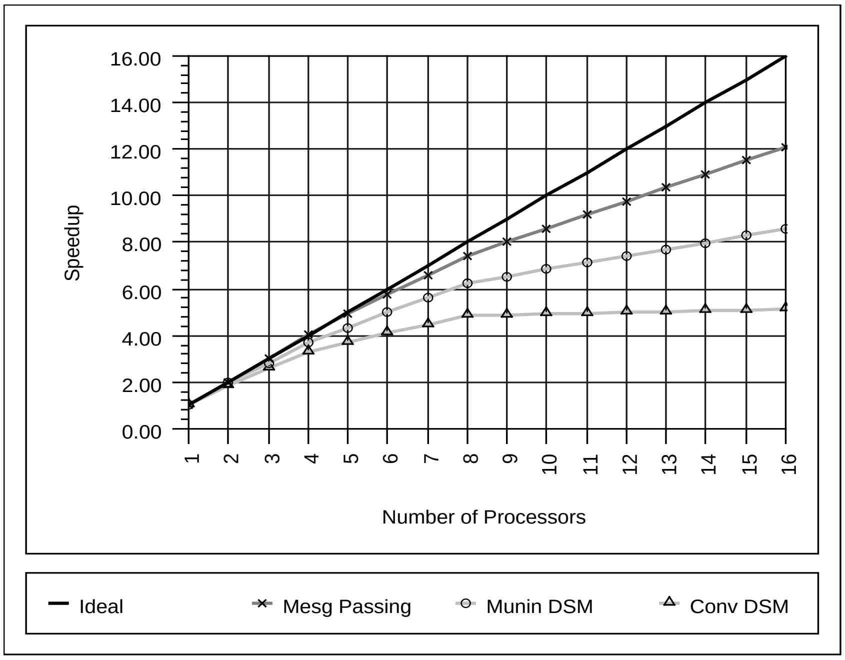

Figure 16 Gaussian Elimination with Partial Pivoting (GAUSS)

6 Detailed Analysis

In this section we analyze in detail, on a per-program basis, the reasons for the performance differences between the various versions of each program. Unless otherwise noted, the numbers in this section pertain to the 16-processor execution.

6.1 Matrix Multiply

Program Description

The problem is to multiply two N by N input arrays, and put the result in an N by N output array. Matrix Multiply is parallelized by giving each worker thread a number of contiguous rows of the output array to compute. After each worker thread has terminated, the root thread reads in the result array and terminates.

The DSM versions use a barrier to signal completion; each worker thread in the message passing version sends its result rows to the master when they have been computed. The Munin version declares the input arrays as `read_only` and the output array as `write_shared`.

Analysis

Matrix multiplication is almost completely compute-bound. As a result, the three versions achieved almost identical speedups (14.5 for conventional DSM, 14.6 for Munin, and 14.7 for message passing). In all cases, the cumulative computation time is roughly 900 seconds, while the cumulative communication time is roughly 4 seconds. Both the Munin and the conventional DSM versions perform approximately twice as much communications as the message passing version, because the DSM worker threads fault in the empty result array at the beginning of the computation, while the message passing worker threads simply initialize their portion of the result array in place. Also, in Munin, when a thread arrives at the final barrier, it updates any copies of a page in the result matrix that are cached by neighboring nodes due to false sharing. This results in the Munin version performing more communication than the conventional version. The Munin version still outperforms the conventional version because the extra communication is largely overlapped with computation, while the read misses experienced by the conventional version cause processors to stall. Nevertheless, compared to the overall execution time, the time spent communicating is minor, so both the conventional and Munin versions exhibit near linear speedup.

6.2 Finite Differencing

Program Description

During each iteration of the finite differencing algorithm, all elements of a matrix are updated to the average of their nearest neighbors (above, below, left, and right). To avoid overwriting the old value of a matrix element before it is used, an iteration is split up in two half-iterations. In the first half-iteration, the program uses a scratch array to compute the new values. In the second, it copies the scratch array back to the main matrix.

Each thread is assigned a number of contiguous rows to compute. The algorithm requires only those elements that lie directly along the boundary between two threads' subarrays to be communicated at the end of each iteration. In the Munin version, the matrix is declared as `write_shared`. In the DSM versions, the programmer is *not* required to specify the data partitioning to the runtime system - it is inferred at runtime based on the observed access pattern. After each half-iteration, the

DSM worker threads synchronize by waiting at a barrier. The message passing workers exchange results directly between neighboring nodes after each iteration.

Analysis

DIFF has a much smaller computation-to-communication ratio than MULT (see Tables 3 and 4), but the Munin version still performs within 5% of the message passing version (a speedup of 12.3 for Munin versus 12.8 for message passing). The reason for Munin's good performance is its use of software release consistency and the write-shared protocol. Together, these techniques result in the underlying communications patterns for the Munin version and the message passing version being nearly identical. When each thread first accesses a page of shared data, it gets a copy of the page. Thus, at the end of the first half-iteration, each node has a read-write copy of any pages for which it has the only copy, and a read-only copy of any pages that lie along a boundary. During the second half-iteration, during which each thread copies the new values from the scratch array to the shared array, each node creates a *diff* of its shared pages. When a thread arrives at the barrier after this half-iteration, it sends the *diff* directly to the appropriate neighbors before sending the barrier message to the barrier master. These diffs include all of the modified data on each boundary page, and not just the edge elements. Since the shared pages are still shared even after they are purged, they are write-protected again, so subsequent writes will be detected. For subsequent iterations, each node experiences a protection violation on the boundary pages, and then performs local operations (creating twins), except when exchanging the results. Thus, the data motion in the Munin version of DIFF is essentially identical to the message passing implementation – communication only occurs at the end of each iteration and only neighboring nodes exchange results. The only overhead comes from fault handling, and from copying, encoding, and decoding the shared portions of the matrix. As an aside, a curious phenomenon can be seen in Table 4: the Munin version of DIFF transmits less data than the message passing version. This is a result of the fact that Munin only transmits the words that have been modified during each iteration, while the message passing version ships the entire edge row. During the early iterations, many of the edge values have not yet been modified, and thus Munin does not transmit any new values for them. In practice, this extra transmitted data had a negligible effect on the running times. Rather, Munin's good performance derived from the fact that it transmits data only during synchronization and suffers no read misses (after the first iteration).

The conventional DSM version of DIFF achieved a speedup of only 8.4, compared to 12.3 for Munin. The conventional version suffers from (1) frequent read faults and reloads as a result of the invalidation protocol, and (2) blocking on write faults as a result of sequential consistency. The Munin version of DIFF creates and transmits diffs at the end of each iteration, which results in shared data being present before it is accessed during the next iteration. This eliminates read misses and reloads on the next iteration. In contrast, the conventional DSM implementation invalidates and reloads every shared page in its entirety on each iteration. In addition, write faults can be handled completely locally in Munin if the data is already present, which is the case for all but the first iteration. The local node simply makes a twin of the data. The conventional DSM implementation sends an invalidation message and waits for a response. The tradeoff is that synchronization under Munin is slowed down because memory needs to be made consistent before the synchronization operation can complete. However, the total time that the Munin worker threads spend blocked while waiting for memory to be made consistent (71.5 seconds) is far less than the time spent invalidating and reloading the data in the conventional version (a total of 356.1 seconds). The time spent invalidating and reloading seriously impacts execution time (356.1 seconds of a total execution time of 662.1 seconds).

6.3 Traveling Salesman Problem

Program Description

The Traveling Salesman Problem (TSP) takes as its input an array representing the distances between cities on a salesman's route, and computes the minimum length "tour" passing through each city exactly once. A *tour queue* maintains a number of partially evaluated tours. If the number of nodes remaining to complete the tour is below a threshold, 12 for TSP-F and 13 for TSP-C, the remainder of the tour is evaluated sequentially. If the number of nodes remaining is above this threshold, the partial tour is expanded by one node, and the new partial tours are entered on the tour queue. When a partial tour is removed from the queue, a lower bound on the remaining part of the tour is computed, and the tour is rejected if the sum of the current length and the lower bound is higher than the current best tour. This check is also performed before a potential new subtour is put on the task queue. The tour queue is a *priority queue* that orders the remaining subtours in the inverse order of a lower bound of their total length. Thus, the "most promising" subtours are evaluated first, which tends to prune uninteresting subtours more quickly. The major shared data structures of TSP are the current shortest tour and its length, an array of structures that represent partially evaluated tours, a priority queue that contains indices into the tour array of partially evaluated tours, and a stack of indices of unused tour array entries. TSP-C and TSP-F differ only in the problem granularity. TSP-C sequentially solves subtours of length 13 or less, while TSP-F sequentially solves subtours of length 12 or less. Depending on the particular input data set, the computation to communication ratio of TSP-C can be as much as ten times higher than that of TSP-F.

In the DSM versions, locks protect the priority queue, the current shortest tour, and its length. A condition variable is used to signal when there is work to be performed. Worker threads acquire the lock and continue to remove partial tours from the queue until a "promising" tour has been found that can be expanded sequentially, at which time the lock is released. In Munin, the priority queue and the stack of unused tours are declared `migratory`, while the other shared data structures are declared `write_shared`. For the message passing version, the *master* maintains a central priority queue that contains the indices of subtours to be solved. The *slaves* send request messages to the master, which responds either with a subtour to be solved sequentially, or an indication that there is no more work. Workers tell the master when they find a new global minimum, and the master is responsible for propagating it.

Analysis (Coarse Grain TSP)

The Munin version achieved a speedup of 12.6, within 5% of the 13.2 achieved by the message passing version. TSP-C is rather compute-bound: under 30 seconds of communication for the Munin version compared to a total execution time of 880 seconds. The performance difference between the message passing version and the Munin version comes from the cost of accessing the priority queue. In Munin, each time a thread tries to remove a tour from the queue, the queue data structure needs to be shipped to that thread. This behavior had two adverse effects on performance. First, worker threads cumulatively spent 62 seconds waiting on the task queue lock. Second, the Munin version shipped 4 megabytes of data, compared to only 900 kilobytes in the message passing version.

The difference in performance between the Munin and conventional DSM versions of TSP-C (a speedup of 12.6 for Munin versus 11.3 for conventional DSM) stems from (1) the use of a migratory protocol for the task queue, and (2) the use of an update, instead of an invalidate, protocol for the minimum tour length. The slightly higher overhead caused by loading and invalidating, rather than simply migrating, the task queue had the effect of causing more processors to idle themselves waiting for work. This was because access to the task queue was the primary bottleneck (a total of 94 seconds for the conventional version versus only 62 in the Munin version). The minimum tour length is an example of a shared data item for which an update protocol is better than an invalidate protocol, because it is read much more frequently than it is written. With the conventional protocol running on N processors, a thread that needs to update the minimum tour length typically sends $N-1$ invalidations and then waits for $N-1$ acknowledgements. All other threads in turn incur an access miss, and its associated latency, to obtain a new copy of the minimum tour length.

Analysis (Fine Grain TSP)

The Munin version of TSP-F achieved a speedup of 6.0, 33% less than the 8.9 speedup achieved by the message passing version. The reasons for the reduction in performance are the same as for TSP-C, but their relative importance is increased. In TSP-F, worker threads spent a cumulative 360 seconds waiting for the priority queue, and a total of 210 seconds performing useful computation. In addition, 9.2 megabytes of data were transmitted in the Munin version, compared to only 920 kilobytes for the message passing version. Similar arguments apply for the conventional DSM version, resulting in a speedup of only 4.7.

6.4 Quicksort

Program Description

Quicksort (QSORT) is a recursive sorting algorithm that operates by repeatedly partitioning an unsorted input lists into unsorted sublists such that all of the elements in one of the sublists are strictly greater than the elements of the other. The Quicksort algorithm is then recursively invoked on the two unsorted sublists. The base case of the recursion occurs when the lists are sufficiently small (1 kilobyte in our case), at which time they are sorted sequentially.

Quicksort is parallelized using a work queue that contains descriptors of unsorted sublists, from which worker threads continuously remove unsorted lists. In the DSM versions of QSORT, the major data structures are the array to be sorted, a task queue that contains range indices of unsorted subarrays, and a count of the number of worker threads blocked waiting for work. Like TSP, the task queue is declared to be `migratory`, while the array being sorted is declared to be `write_shared`. A lock protects the queue, and a condition variable is used to signal the presence of work to be performed. QSORT differs from TSP in that when QSORT releases control of the task queue, it may need to further subdivide the work by partitioning the subarray and placing the new subarrays back into the task queue. In contrast, TSP workers never relinquish control of the task queue until they have removed a subtour that can be solved sequentially. Therefore, the task queue in QSORT is accessed more frequently per unit of computation. Offsetting this is the fact that the threads in TSP hold the lock protecting the priority queue for a longer time as they perform the expansion.

For the message passing version of QSORT, the master maintains the work queue. The slaves send request messages to the master, which responds either with a sublist to be sorted sequentially or an indication that there is no more work. Along with the requests, the slaves ship the sorted results from their previous request, if any.

Analysis

The Munin version of QSORT achieves only 67% of the speedup of the message passing version (8.9 versus to 13.4). As with TSP-C and TSP-F, most of Munin's overhead comes from shipping the work queue each time a node tries to perform a queue insertion or deletion. Compounding this problem is the fact that the threads do not retain sole ownership of the work queue while subdividing the work into pieces sufficiently small to solve directly, so they repeatedly need to reacquire the task queue and partition their subarray until it contains at most 1024 elements. As a result, the threads spent a cumulative 842 seconds waiting on the task queue lock, out of a total execution time of 2160 seconds. Furthermore, the Munin version transmitted 23 megabytes of data, compared to 520 kilobytes for the message passing implementation.

For the conventional DSM version, speedup drops to 4.1. In addition to the cost of invalidating and reloading the task queue, rather than simply migrating it, the difference in performance between the conventional DSM version and the Munin version is primarily due to the presence of false sharing when two threads simultaneously attempt to sort subarrays that reside on the same page. As a result, communication goes from 23 megabytes in about 30,000 messages for the Munin version, to 110 megabytes in 231,000 messages for the conventional version.

6.5 Fast Fourier Transform

Program Description

The Fast Fourier Transform (FFT) program used in the evaluation is based on the Cooley-Tukey Radix 2 Decimation in Time algorithm [14]. It recursively subdivides the problem into its even and odd components, until the input is of length 2. For this base case, the output is an elementary function known as a Butterfly, a linear combination of its inputs. For an input array of size N, the FFT algorithm requires $\log_2 N$ passes. On pass K, the width of each butterfly is $N2^{-(K+1)}$. Thus, for the first pass, width of the butterfly is $N/2$, and each subsequent iteration the width of each butterfly halves. By starting with the wide butterflies, the result array is a permutation of the desired value, but this is rectified with an $O(N)$ cleanup phase.

If P processors are used to solve an N point FFT, where P is power of 2, then a reasonable initial decomposition of the work allows processor p to work with $x[p]$, $x[p+P]$, $x[p+2P]$, ..., $x[p+N-P]$. This allows all processors to perform the first $\log_2 N - \log_2 P$ passes without any inter-processor communication. Before executing the last $\log_2 P$ iterations, the processors exchange data and reallocate themselves to different (contiguous) subarrays, as illustrated in Figure 17. This figure illustrates an 8-element FFT being performed by two processors ($N = 8$, $P = 2$), where the dashed lines represent computations by p_0 and the dark lines represent computations by p_1.

Both the DSM and message passing programs are parallelized by dynamically allocating threads to data as described above. The array on which the FFT is being performed is declared to be `write_shared` in the Munin version. By carefully allocating processors to data as described above, it is possible to only reallocate the processors and exchange data at the end of the first $\log_2 N - \log_2 P$ phases. The DSM programs use a barrier to synchronize at this point. The DSM system automatically reallocates the data on demand. The message passing version manually encodes and shuffles the data, using a master process to collect and redistribute all of changes. This manual redistribution made the message passing version much harder to write than the DSM versions. The processor reallocation is built in to the algorithm itself.

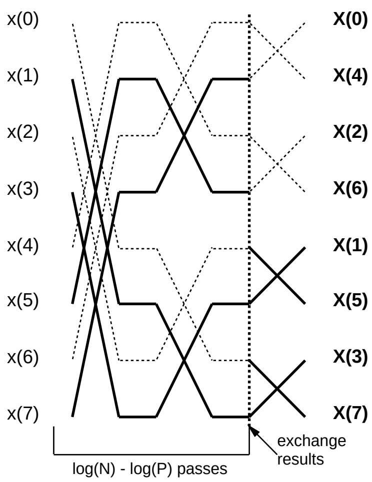

Figure 17 Parallel FFT: $N = 8$, $P = 2$

Analysis

The FFT algorithm used has a very high degree of sharing, which results in it being bus bandwidth limited to a speedup of approximately ten on a twenty processor, single-bus multiprocessor like the Sequent Symmetry. Because of the way that the data is distributed, every page is referenced (and modified) by every thread during the first $\log_2 N - \log_2 P$ iterations, the worst possible behavior for any DSM system. The conventional DSM version *slows down* by a factor of ten for two or more processors, while the Munin version achieved a speedup of 7.6 on sixteen processors. The cause for this dramatic difference in performance is Munin's ability to efficiently support multiple concurrent writers to a shared page of data. The message passing version of FFT performed slightly better (speedup of 8.8 on 16 processors) than the Munin version.

The conventional DSM implementation takes over 300,000 faults, requires 1.35 gigabytes of data to be shipped and 1.65 million messages to be transmitted, and cumulatively spends over 25000 seconds waiting for requests to be satisfied. While not devoid of overhead, the Munin version requires orders of magnitude less communication. It only takes 2168 faults and reloads a total of 12 megabytes of data. The primary source of overhead for the Munin program comes from sending out the lists updates during the data exchange phase after the first $\log_2 N - \log_2 P$ phases. At the beginning of the update phase, every processor is caching every page of shared data. This causes each processor to send a list of updates for every page to every other processor, which adds two seconds of synchronization overhead. Munin's update timeout mechanism keeps the processors from actually shipping most of the data to every node, resulting in the Munin version shipping only slightly more data than the message passing version.

6.6 Gaussian Elimination with Partial Pivoting

Program Description

Gaussian Elimination (GAUSS) decomposes a square matrix into upper and lower triangular submatrices by repeatedly eliminating the elements of the matrix under the diagonal, one column at a time. The basic algorithm for an N by N matrix is shown in Figure 18. For each iteration of the i-loop, the algorithm subtracts the appropriate multiple of the i^{th} row of the matrix from the rows

```
for i := 1 to N do
    for j := i+1 to N do
        for k = N+1 downto i do
            a[j][k] := a[j][k] - a[i][k]*a[j][i]/a[i][i];
```

Figure 18 Basic (w/o pivoting) Gaussian Elimination Algorithm

below it, so that the elements below the diagonal in the i^{th} column are zeroed. Partial pivoting improves the numerical stability of the basic algorithm by interchanging the i^{th} row with the row in the range $[i+1...N-1]$ containing the largest (in absolute value) element of the i^{th} column. Algorithmically, this involves inserting a phase between the i and j loops that searches the i^{th} column for the pivot element, and swapping that row and the i^{th} row.

We decomposed the computation by column so that the pivoting phase, which can be a synchronization bottleneck, can be performed on a single processor. Each thread gets roughly $\lfloor N/P \rfloor$ columns, striped across the matrix, and any extra columns are spread evenly across the worker threads. The computation itself involves N iterations, one per column, each iteration consisting of a pivoting phase and a computation phase.

The DSM versions are parallelized as follows. The shared data structures are the array on which the elimination is being performed, a vector into which the pivot row is copied, and an integer that contains the number of the pivot row – all of which are declared to be `write_shared` in the Munin version. Each iteration starts with a barrier. After the barrier falls, the thread responsible for the current column performs the necessary pivoting, sets a shared pivot row variable to indicate the row that needs to be pivoted with the current one, and copies the current column to a shared variable to be used by the other threads during the computation phase. A barrier is used to separate the pivoting and computation phases. After the barrier is passed, each thread performs the actual computation, which involves performing the local pivoting, followed by the elimination step shown in Figure 18.

The message-passing version works similarly, except that the barrier is replaced by messages from the slaves to the central master, and the pivot column and pivot row number are explicitly sent to the workers rather than faulted in asynchronously.

Analysis

The DSM versions of Gaussian Elimination require two barriers per iteration for synchronization. The Munin version achieves a speedup of 8.6, 71% of the message passing version's speedup of 12.1, on sixteen processors. The reason for this reduced performance is that the relatively small amount of work done per iteration, particularly during the latter stages of the algorithm, accentuates the overhead imposed by both the general purpose barrier mechanism, and the need to update shared data during synchronization. On average, each thread spends over 40 seconds waiting for barriers, which includes the time spent exchanging data.

The conventional DSM version of GAUSS achieves a speedup of 5.1 on sixteen processors, 42% of the message passing version. In addition to the synchronization issues noted in the Munin implementation, the conventional DSM implementation also suffers from frequent read misses caused by accesses to invalidated data. While the Munin implementation experiences 90 read misses, the conventional DSM implementation experiences 6780. This is caused by the use of an invalidation-based consistency protocol in the conventional DSM system. Since all of the modifications are made to shared data that is being actively shared (and constantly used) on all sixteen processors, the update-pruning advantage of an invalidation protocol is not relevant, while the increased number of read misses is a significant problem. Each thread stalls for an average of 50 seconds for read misses to be serviced. In addition, because the last thread to have its read miss satisfied must wait until fourteen other threads have successfully acquired their data, the computations tend to complete at noticeably different times. This causes the average time spent waiting at barriers to increase from 40 to 50 seconds. These two phenomena explain the lower performance of the conventional DSM implementation.

The performance times reported for the Munin version of all applications, including GAUSS, were with the update timeout mechanism enabled. For GAUSS, disabling the update timeout mechanism results in a slight performance advantage (a speedup of 8.9, instead of 8.6, on 16 processors). This is because, in GAUSS, all of the modified data is accessed every iteration, thus it is best to propagate the updates, and not selectively invalidate. In this case, the 50 millisecond default update timeout time was too short to ensure that no updates were timed out. Enabling the timeout mechanism thus resulted in unnecessary invalidations and subsequent reloads.

7 Effect of Communication Reduction Techniques

In this section we try to isolate the effects on performance of each of the techniques for reducing communications that were described in Section 2. This isolation is made somewhat difficult because of the synergistic effect on performance of using the techniques in conjunction with one another. In particular, write-shared protocols cannot be used in the absence of release consistency. Therefore, we first compare Munin's buffered write-update implementation of release consistency to a pipelined write-invalidate implementation of release consistency. Then we compare the use of multiple protocols versus using a single protocol, write-shared. Finally, we determine the value of the update timeout mechanism in connection with the update protocol.

7.1 Buffered Update versus Pipelined Invalidate Release Consistency

In Section 2.1.2 we described the motivation for using a buffered update protocol for implementing release consistency in software, and the advantages of doing so over using a pipelined invalidate protocol. To evaluate the impact of our decision, we implemented a pipelined write-invalidate consistency protocol and compared it to the buffered update protocol that is in normal use in Munin. In the pipelined write-invalidate protocol, a write fault causes ownership to be transferred to the faulting processor. Then invalidations are sent out in separate messages. Multiple invalidations can be outstanding concurrently, but no synchronization operation is allowed to complete until all outstanding invalidations have been acknowledged. We compared the performance of this implementation of release consistency with the Munin implementation using buffered-update and with the conventional DSM system. For MULT, TSP-C, TSP-F, and GAUSS there is little difference between the pipelined write-invalidate and buffered write-update implementations of release consistency. For DIFF and QSORT, the buffered write-update scheme performs 30% better for 16 processors, while for FFT it performs orders of magnitude better. For the latter three applications, the pipelined write-invalidate protocol performs slightly better than a conventional write-invalidate protocol. Figures 19 and 20 depict these results for DIFF and FFT. The performance of QSORT is similar to that of DIFF.

These results demonstrate that while the pipelined write-invalidate protocol offers some performance gain over a conventional sequentially consistent write-invalidate protocol in a software DSM system, a buffered write-update protocol outperforms both. Pipelining invalidations allows useful computation to be overlapped with invalidations, which reduces the cost of writes. However, it does not reduce the penalty associated with read misses, which are very expensive in a software DSM system. Furthermore, the pipelined-invalidate protocol suffers from false sharing, much in the same way that a conventional DSM system does. When read misses dominate, or when there is substantial false sharing, Munin's buffered update implementation is superior.

7.2 Multiple Consistency Protocols

The observation that no single consistency protocol is best-suited for all programs was discussed in Section 2.2. To evaluate the importance of Munin's support for multiple consistency protocols, we compared the performance of two versions of Munin: (i) a version in which multiple consistency protocols were used, and (ii) a version that labeled all shared data as write-shared, thus employing Munin's most versatile protocol. Figure 21 presents the results of this experiment for TSP-F; similar results were obtained for the other multiprotocol test programs (TSP-C and QSORT). For TSP-F, using multiple protocols leads to a 30% improvement in speedup for 16 processors. The reason is that the multiple protocol version of the program declares the task queue to be migratory, resulting in the advantages described in Section 2.2.

7.3 Update Timeout Mechanism

To test the value of the timeout mechanism in connection with the update protocol, we compared the performance of versions with and without the timeout enabled. For MULT, DIFF, and TSP-C there is no difference. For TSP-F and QSORT, the version with the timeout enabled is 10% and 15% faster for 16 processors, respectively. The difference is the largest for FFT. Speedup with

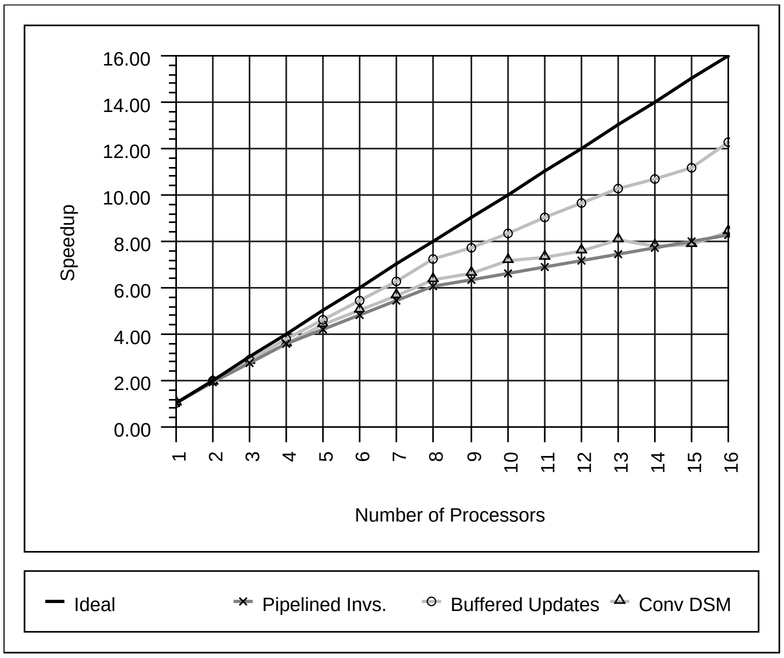

Figure 19 Buffered Write-Update RC versus Pipelined Write-Invalidate RC (DIFF)

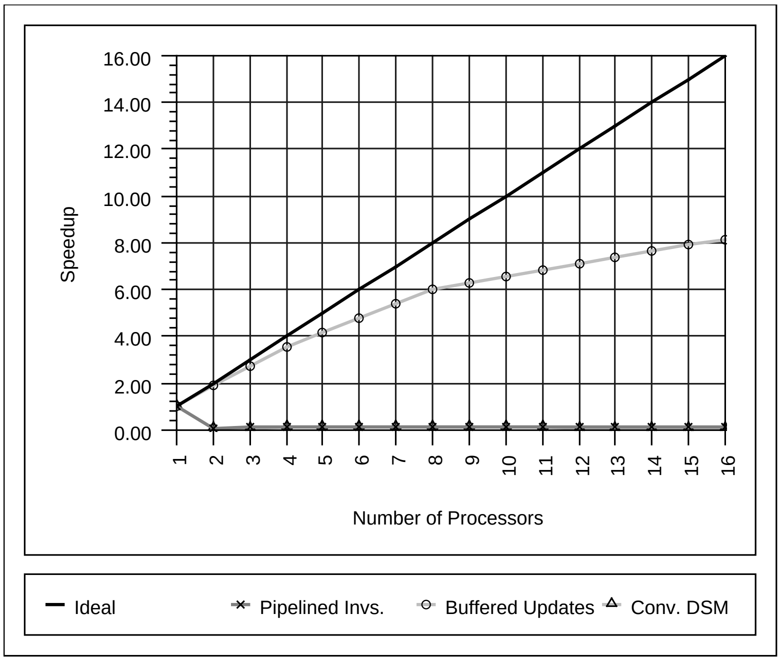

Figure 20 Buffered Write-Update RC versus Pipelined Write-Invalidate RC (FFT)

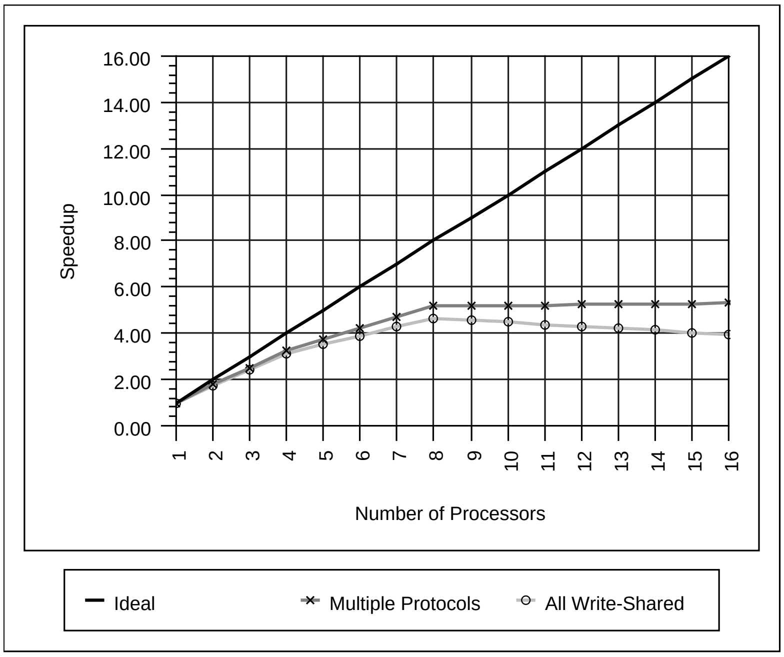

Figure 21 Multi-protocol versus All Write-Shared (TSP-F)

16 processors drops from 8.2 to 3.6 when the timeout was disabled (see Figure 22). Finally, for GAUSS, the timeout causes a 5% dropoff in performance for 16 processors.

In terms of the underlying DSM operation, without the timeout mechanism the 16-processor FFT sends 120,000 messages and 109 megabytes of data, while, with the timeout mechanism enabled, the 16-processor FFT sends only 48,000 messages and 78 megabytes of data. The reason that the amount of data shipped does not drop as dramatically as the number of messages is that, after a page of data has been speculatively invalidated, future accesses require an 8-kilobyte page to be transferred rather than just a diff.

The other two programs in which each processor's working set changes dynamically over the course of the program execution, TSP and QSORT, are also aided by the use of the timeout mechanism. For TSP, each page of the shared tour array tends to be used by many different processors over time, but each processor only uses it for a very short period of time, and only a few processors use a particular page at a time. Without the timeout mechanism, eventually almost every processor receives updates for almost every page. The shared sort array in QSORT exhibits a similar phenomenon.

With GAUSS, all of the modified data are accessed every iteration. The slight dropoff in performance for GAUSS is caused by the fact that the default update timeout time of 50 milliseconds is too short to ensure that no valid updates are timed out.

8 Function Shipping

For TSP-F and QSORT, the two programs that use the task queue model of parallelism and that have a significant amount of sharing, the Munin sixteen processor versions achieves speedups of

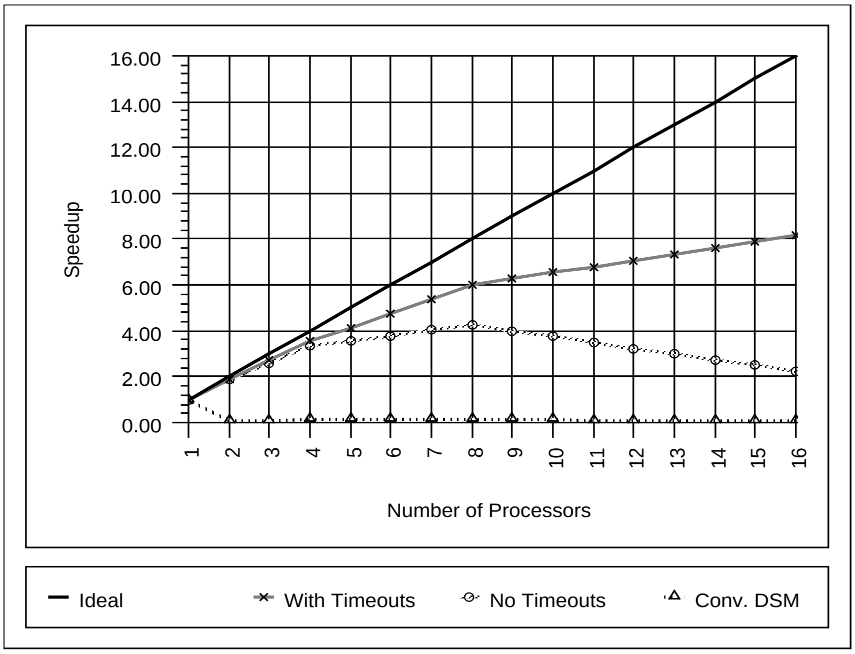

Figure 22 Effect of Update Timeout Mechanism on FFT

only 6.0 and 8.9, respectively, compared to 8.9 and 13.4 for the message passing versions. The conventional DSM versions performed even worse, achieving speedups of 4.7 and 4.1, respectively. As shown in Table 5, the major source of overhead for these DSM versions (with the exception of the conventional version of QSORT) is the amount of time spent waiting on the lock protecting the work queues. For the conventional version of QSORT, false sharing within the array being sorted is the dominant source of overhead.

These lock waiting times are large because the DSM versions must ship the work queue, a sizable data structure, to the acquiring thread before that thread can perform any operation on the work queue. In comparison, the actual time spent performing operations on the work queue is very small. The message passing versions do not suffer from this phenomenon, since the work queue is kept at the root node and worker threads perform remote procedure calls (RPCs), containing only a small amount of data, to the root node in order to operate on the queue.

In order to evaluate the feasibility and potential value of using a mixed data-shipping and function-shipping mechanism in a DSM system, we modified the DSM versions of TSP-F and QSORT such that the task queue remains attached to the root node, and all access to the task queue by other nodes is performed using RPC. These modifications were done in an ad hoc manner, but research is ongoing to extend Munin to support both DSM and function shipping in an integrated fashion. The results of function-shipping access to the task queue for the TSP-F and QSORT are shown in Figures 23 and 24. These figures show the speedups achieved by Munin and conventional DSM both with and without function shipping for the task queue.

For TSP-F, function shipping causes both DSM versions to perform almost as well as the message passing version (on 16 processors, a speedup of 9.1 for conventional DSM, 9.8 for Munin, and 10.6 for message passing). In contrast, without function shipping, Munin achieves a speedup of only 6.0, and the conventional DSM a speedup of only 4.7. For the Munin version without function shipping, communication is substantially more (9229 messages and 4989 kilobytes of data) than the Munin version with function shipping (3630 messages and 888 kilobytes of data). Perhaps more importantly, the reduced communication of the function shipping version nearly eliminates the time that threads are idle waiting for access to the task queue.

For QSORT, improvements are similar to those in TSP-F for the Munin version, but no improvement is achieved for the conventional DSM version. The addition of function-shipping for the task queue raises the 16-processor speedup for Munin from 8.9 to 12.9, compared to 13.4 for the message passing version. The conventional DSM version, both with and without function shipping for the task queue, achieves only a speedup of 4.1. As explained in Section 6, false sharing is the primary obstacle to good performance for the conventional version. While the average time waiting for locks is reduced from 13 seconds to below 1 second, the average time a process waits for fresh copies of data increases from 145 to 176 seconds, so the addition of function shipping has

Program	Average lock waiting time (per processor) (seconds)	Execution time (per processor) (seconds)
Munin TSP-F	19	32
Conventional TSP-F	22	45
Munin QSORT	53	135
Conventional QSORT	13	310

Table 5 Lock waiting times for TSP-F and QSORT

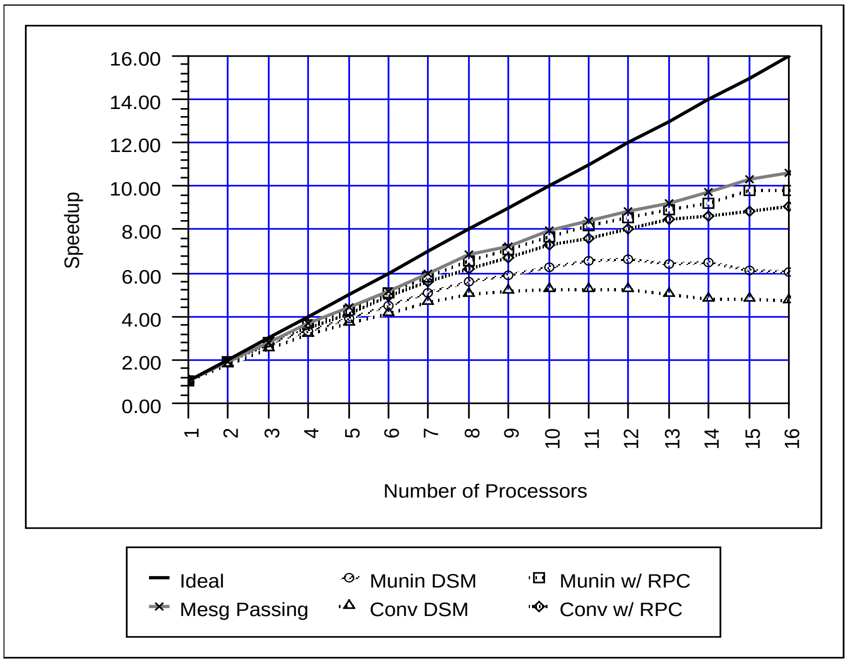

Figure 23 Effect of Function Shipping on Fine-grained TSP

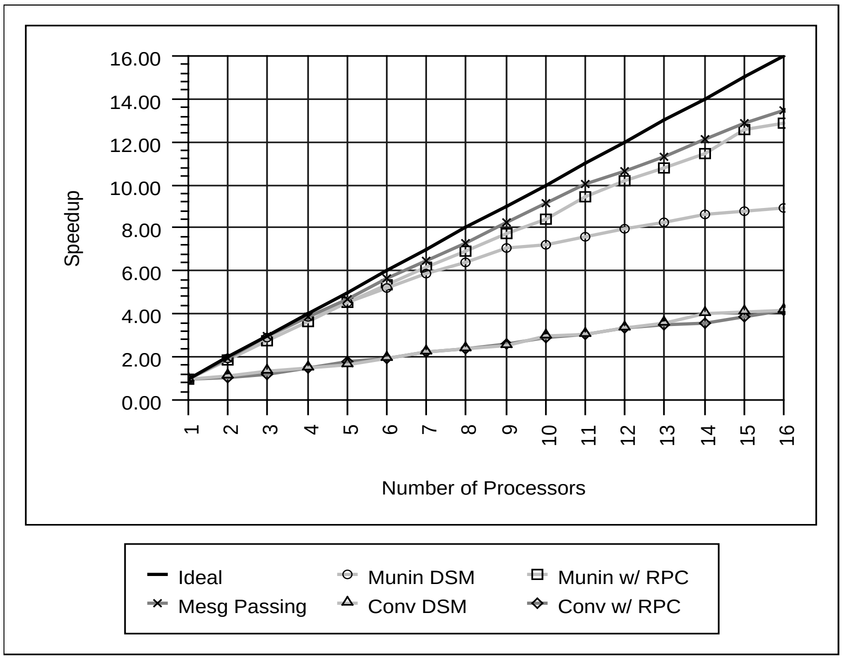

Figure 24 Effect of Function Shipping on Quicksort

no beneficial effects.

These experiments show that the addition of function shipping for accessing some shared data can significantly improve the performance of some programs. In addition, the QSORT experiment further illustrates the value of Munin's write-shared protocol for dealing with false sharing.

9 Related Work

This section compares our work with a number of existing software and hardware DSM systems, focusing on the mechanisms used by these other systems to reduce the amount of communication necessary to provide shared memory.

9.1 Software DSMs

Ivy was the first DSM system [42]. It uses a single-writer, write-invalidate protocol for all data, with virtual memory pages as the units of consistency. This protocol is used as the baseline conventional protocol in our experiments. The large size of the consistency unit and the single-writer protocol makes the system prone to large amounts of communication due to false sharing. It is up to the programmer or the compiler to lay out the program data structures in the shared address space such that false sharing is reduced. The directory management scheme in our implementation is largely borrowed from Ivy's dynamic distributed manager scheme.

Both Clouds [21] and Mirage [27] allow part of shared memory to be locked down at a particular processor. In Clouds, the programmer can request that a segment of shared memory be locked on a processor. In Mirage, a page remains at a processor for a certain Δ time window after it is modified by that processor. In both cases, the goal is to avoid extensive communication due to false sharing. The combination of software release consistency and write-shared protocols addresses the adverse effects of false sharing without introducing the delays caused by locking parts of shared memory to a processor.

Mether [46] supports a number of special shared memory segments in fixed locations in the virtual address space of each machine in the system. In an attempt to support efficient memory-based spinlocks, Mether supports several different shared memory segments, each with different protocol characteristics. Two segments are for small objects (up to 32 bytes), while two are for large objects (up to 8192 bytes). One of each pair is "demand-driven", which means that the memory is shipped when it is read, as in a conventional DSM. The other is "data-driven", which means that it is shipped when it is written. A thread that attempts to read the data will block until the next thread writes it. This latter form of data can support spinlocks and message-passing fairly effectively. Our support for multiple protocols is more general, without added cost, and Munin's separate synchronization package removes the need to support data-driven memory.

Lazy release consistency [38, 25], as used in TreadMarks [39], 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. Simulation studies indicate that lazy release consistency reduces the number of message required to maintain consistency, but the implementation is more expensive in terms of protocol and memory overhead. A comparative evaluation of the two approaches is the subject of an ongoing Ph.D. dissertation [37].

A variety of systems have sought to present an object-oriented interface to shared memory. We use the Orca [7] and the Midway system [11] as examples of this approach. Other noteworthy systems include Clouds [21], Emerald [34], and Amber [19]. In general, the object-oriented nature allows the compiler and the runtime system to carry out a number of powerful optimizations, but the programs have to be written in the particular object model supported.

The Orca language requires that (i) all access to objects is through well-defined per-object operations, (ii) only one operation on an object can be performed at a time, and (iii) there are no global variables or pointers. This programming model allows the compiler to detect all accesses to an object directly without the use of page faults. Programmers must, however, structure their programs so that objects are accessed in a way that does not limit performance. For example, an Orca implementation of DIFF requires that the edge elements be specified as shared buffers - the entire array should not be declared as a single object. However, once a program has been structured appropriately, Orca can transparently choose whether to replicate an object or force all accesses to be made via RPCs to a master node. If it chooses to replicate an object, it can support both invalidate and update consistency protocols. It remains to be seen how well Orca's optimizations can be integrated into a less restrictive language. On an orthogonal issue, Orca's consistency management uses an efficient, reliable, ordered broadcast protocol. For reasons of scalability, Munin does not rely on broadcast, although support for efficient multicast could improve the performance of some aspects of Munin.

Midway [11] 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. For example, the programmer of an entry consistent DIFF program would have to hand decompose the shared array to exploit the power of entry consistency. The designers of Midway recognized this problem and gave programmers the ability to increase and decrease the strength of the consistency model supported. Thus, programs for which the data-synchronization association required by entry consistency is convenient can exploit its flexibility, while programs for which this association is inconvenient can use either release consistency (when adequate synchronization is performed) or sequential consistency. Unlike Munin, Midway exploits the power of a sophisticated compiler. The Midway compiler inserts code around data accesses so that the Midway runtime system can determine whether a particular shared variable is present before it is accessed. Thus, Midway is able to detect access violations without taking page faults, which eliminates the time spent handling interrupts.

9.2 Hardware DSMs

Recently, a number of designs for hardware distributed shared memory machines have been published [2, 9, 13, 22, 41, 57, 58]. We limit our discussion to those systems that are most related to the work presented in this paper.

We have adopted from the DASH project [41] the concept of release consistency. The differences between DASH's implementation of release consistency and Munin's implementation of release consistency were explained in detail in Section 2.1. DASH uses a write-invalidate protocol for all consistency maintenance. We instead use the flexibility of its software implementation to also attack the problem of read misses by using update protocols and migration when appropriate. The GalacticaNet system [57] also demonstrated that support for an update-based protocol that exploits the flexibility of a relaxed consistency protocol can improve performance by reducing the number of read misses and attendant processor stalls. The GalacticaNet design includes a provision to time out updates to stale data, which is shown to have a significant effect on performance when there is a large number of processors.

The APRIL machine addresses the problem of high latencies in distributed shared memory multiprocessors in a different way [2]. APRIL provides sequential consistency, but relies on extremely fast processor switching to overlap memory latency with computation. For APRIL to be successful at reducing the impact of read misses, there must be several threads ready to run on each processor. Because APRIL performs many low-level consistency operations in very fast trap handling software, it would be possible to adopt several of our techniques to their hardware cache consistency mechanism.

The Data Diffusion Machine (DDM) [55] and the KSR-1 [13] use hardware DSM to support an *all-cache* model for memory, wherein the entire memory of the system is treated as a large cache of the global virtual address space, similar to the way our software DSM treats memory. The KSR-1 supports a fairly conventional directory-based write-invalidate consistency protocol, which limits its scalability when there is frequent write sharing.

10 Conclusions and Directions for Further Work

Distributed shared memory (DSM) systems provide a shared memory abstraction on hardware with physically distributed memory. This approach is appealing because it combines the desirable features of distributed and shared memory machines: Distributed memory machines are easier to build, but shared memory provides a more convenient programming model. It has, however, proven difficult to achieve performance on DSM systems that is comparable to what can be achieved with hand-coded message passing programs. In particular, conventional DSM implementations have suffered from excessive amounts of communications engendered by sequential consistency and false sharing.

In this paper we have presented and evaluated a number of techniques to reduce the amount of communication necessary to maintain consistency. In particular, we replaced sequential consistency by release consistency as our choice of consistency model. We developed a buffered, update-based implementation of release consistency, suitable for software systems. The update protocol has a timeout feature, preventing large numbers of unnecessary updates to copies of pages that are no longer in use. Furthermore, we allow the use of multiple protocols to maintain consistency. Of particular interest among these protocols is the write-shared protocol that allows several processes to write to a page concurrently, with the individual modifications merged at a later point according to the requirements of release consistency.

We have implemented these techniques in the Munin DSM system. The resulting system runs on a network of workstations and provides an interface that is very close to a conventional shared memory programming system. For data-race free programs, release consistent memory produces the same results as sequentially consistent memory. All synchronization operations must be performed through system-supplied primitives, and shared variables may optionally be annotated with the desired consistency protocol. For the applications that we have looked at, these requirements proved to be a very minor burden.

The use of these techniques has substantially broadened the class of applications for which DSM on a network of workstations is a viable vehicle for parallel programming. For very coarse-grained applications conventional DSM performs satisfactorily. However, as the granularity of parallelism decreases, conventional DSM performance falls behind, while Munin's performance continues to track that of hand-coded message passing. The addition of a function shipping ability further improves the performance of DSM.

Hardware technology has improved dramatically since the experiments reported here were performed, and there are no signs that the current rate of performance improvement will abate soon. In particular, both processor and network speeds have improved by a factor of fifteen to twenty in the past four years. Interprocessor communication is still a high latency operation, but there are indications that latencies can be improved by an order of magnitude through careful protocol implementation [33, 53]. At the same time, DRAM latencies are improving very slowly, so some form of cache will be present on essentially all future high-performance platforms. Finally, hardware DSM systems are becoming increasingly common. An important issue to address is the applicability of the techniques introduced in this paper to future DSM system, both hardware and software.

We believe that there are two basic requirements that DSM systems, hardware or software, must satisfy to provide acceptably high performance. Both the *latency* and the *frequency* of processor-stalling DSM operations (e.g., cache misses or synchronization events) must be kept low. It appears that despite improvements in networking and operating system designs, the latency of remote operations will slowly increase compared to processor cycle times. However, because memory speeds are not increasing very rapidly, the ratio of remote memory access to local memory access (not satisfied by the cache) will decrease. This observation would seem to indicate that a simple implementation of DSM that ships entire pages (or cache lines) on demand and uses invalidation to maintain consistency would suffice as processor and network technology improves. We believe that this will not be the case because of our second requirement for efficient DSM: a low *frequency* of processor-stalling DSM operations. As processor cycle times continue to decrease dramatically, it is becoming increasingly important to avoid stalling the processor. As described in Section 7.1, using a conventional invalidation-based consistency protocol can increase the number of high-latency read misses dramatically. Also, as the size of memories and caches increase, page and cache line sizes are also increasing, which indicates that false sharing will become an increasingly important problem. These observations indicates that some form of update protocol that supports multiple concurrent writers, such as Munin's write-shared protocol, will be useful in future DSM systems.

Our current DSM work focuses on novel techniques required to implement DSM on current high-performance platforms, with faster processors and networks than the ones used for the experiments in this paper. In particular, we are studying a more aggressive implementation of release consistency, lazy release consistency, and compiler techniques to further optimize performance. We are also studying the value of the techniques described here in the context of a distributed shared memory multiprocessor.

References

[1] A. Agarwal and A. Gupta. Memory-reference characteristics of multiprocessor applications under MACH. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 215–225, June 1988.

[2] A. Agarwal, B.-H. Lim, D. Kranz, and J. Kubiatowicz. APRIL: A processor architecture for multiprocessing. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 104–114, May 1990.

[3] A. Agarwal, R. Simoni, J. Hennessy, and M. Horowitz. An evaluation of directory schemes for cache coherence. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 280–289, June 1988.

[4] M. Ahamad, P.W. Hutto, and R. John. Implementing and programming causal distributed shared memory. In *Proceedings of the 11th International Conference on Distributed Computing Systems*, pages 274–281, May 1991.

[5] T.E. Anderson. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, January 1990.

[6] J. Archibald and J.-L. Baer. Cache coherence protocols: Evaluation using a multiprocessor simulation model. *ACM Transactions on Computer Systems*, 4(4):273–298, November 1986.

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

[8] J.K. Bennett, J.B. Carter, and W. Zwaenepoel. Adaptive software cache management for distributed shared memory architectures. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 125–134, May 1990.

[9] J.K. Bennett, S. Dwarkadas, J.A. Greenwood, and E. Speight. Willow: A scalable shared memory multiprocessor. In *Proceedings of Supercomputing '92*, pages 336–345, November 1992.

[10] B.N. Bershad, E.D. Lazowska, and H.M. Levy. PRESTO: A system for object-oriented parallel programming. *Software: Practice and Experience*, 18(8):713–732, August 1988.

[11] B.N. Bershad, M.J. Zekauskas, and W.A. Sawdon. The Midway distributed shared memory system. In *COMPCON '93*, pages 528–537, February 1993.

[12] R. Bisiani and M. Ravishankar. PLUS: A distributed shared-memory system. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 115–124, May 1990.

[13] H. Burkhardt, S. Frank, B. Knobe, and J. Rothnie. Overview of the KSR1 computer system. Technical Report KSR-TR-9002001, Kendall Square Research, February 1992.

[14] C. S. Burrus and T. W. Parks. DFT/FFT and Convolution Algorithms. *Wiley-Interscience*, 1985.

[15] J.B. Carter. *Efficient Distributed Shared Memory Based On Multi-Protocol Release Consistency.* PhD thesis, Rice University, August 1993.

[16] J.B. Carter, J.K. Bennett, and W. Zwaenepoel. Implementation and performance of Munin. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 152–164, October 1991.

[17] L. Censier and P. Feautrier. A new solution to coherence problems in multicache systems. *IEEE Transactions on Computers*, C-27(12):1112–1118, December 1978.

[18] D. Chaiken, J. Kubiatowicz, and A. Agarwal. LimitLESS directories: A scalable cache coherence scheme. In *Proceedings of the 4th Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 224–234, April 1991.

[19] J.S. Chase, F.G. Amador, E.D. Lazowska, H.M. Levy, and R.J. 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, December 1989.

[20] D.R. Cheriton. The V distributed system. *Communications of the ACM*, 31(3):314–333, March 1988.

[21] P. Dasgupta, R.C. Chen, S. Menon, M. Pearson, R. Ananthanarayanan, U. Ramachandran, M. Ahamad, R. LeBlanc Jr., W. Applebe, J.M. Bernabeu-Auban, P.W. Hutto, M.Y.A. Khalidi, and C.J. Wilekn-loh. The design and implementation of the Clouds distributed operating system. *Computing Systems Journal*, 3, Winter 1990.

[22] G.S. Delp, A.S. Sethi, and D.J. Farber. An analysis of MemNet: An experiment in high-speed shared-memory local networking. In *Proceedings of the Sigcomm '88 Symposium*, pages 165–174, August 1988.

[23] C. Dubnicki and T. LeBlanc. Adjustable block size coherent caches. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 170–180, May 1992.

[24] M. Dubois and C. Scheurich. Memory access dependencies in shared-memory multiprocessors. *IEEE Transactions on Computers*, 16(6):660–673, June 1990.

[25] S. Dwarkadas, P. Keleher, A.L. Cox, and W. Zwaenepoel. Evaluation of release consistent software distributed shared memory on emerging network technology. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 144–155, June 1993.

[26] S.J. Eggers and R.H. Katz. A characterization of sharing in parallel programs and its application to coherency protocol evaluation. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 373–383, May 1988.

[27] 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, December 1989.

[28] K. Gharachorloo, A. Gupta, and J. Hennessy. Performance evaluations of memory consistency models for shared-memory multiprocessors. In *Proceedings of the 4th Symposium on Architectural Support for Programming Languages and Operating Systems*, April 1991.

[29] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. 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.

[30] J. R. Goodman, M. K. Vernon, and P.J. Woest. Efficient synchronization primitives for large-scale cache-coherent multiprocessor. In *Proceedings of the 3rd Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 64–75, April 1989.

[31] J.R. Goodman. Cache consistency and sequential consistency. Technical Report CS-1006, University of Wisconsin-Madison, February 1991.

[32] D. Hensgen, R. Finkel, and U. Manber. Two algorithms for barrier synchronization. *International Journal of Parallel Programming*, 17(1):1–17, January 1988.

[33] D.B. Johnson and W. Zwaenepoel. The Peregrine high-performance rpc system. *Software: Practice and Experience*, 23(2):201–221, February 1993.

[34] E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.

[35] A.R. Karlin, M.S. Manasse, L. Rudolph, and D.D. Sleator. Competitive snoopy caching. In *Proceedings of the 16th Annual IEEE Symposium on the Foundations of Computer Science*, pages 244–254, 1986.

[36] R. Katz, S. Eggers, D. Wood, C.L. Perkins, and R. Sheldon. Implementing a cache consistency protocol. In *Proceedings of the 12th Annual International Symposium on Computer Architecture*, pages 276–283, June 1985.

[37] P. Keleher. *Distributed Shared Memory Using Lazy Release Consistency.* PhD thesis, Rice University, 1994.

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

[39] P. Keleher, S. Dwarkadas, A. Cox, and W. Zwaenepoel. Treadmarks: Distributed shared memory on standard workstations and operating systems. In *Proceedings of the 1994 Winter Usenix Conference*, pages 115–131, January 1994.

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

[41] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. The directory-based cache coherence protocol for the DASH multiprocessor. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 148–159, May 1990.

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

[43] R.J. Lipton and J.S. Sandberg. PRAM: A scalable shared memory. Technical Report CS-TR-180-88, Princeton University, September 1988.

[44] T. Lovett and S. Thakkar. The Symmetry multiprocessor system. In *Proceedings of the 1988 International Conference on Parallel Processing*, pages 303–310, August 1988.

[45] J.M. Mellor-Crummey and M.L. Scott. Synchronization without contention. In *Proceedings of the 4th Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 269–278, April 1991.

[46] R.G. Minnich and D.J. Farber. The Mether system: A distributed shared memory for SunOS 4.0. In *Proceedings of the Summer 1989 USENIX Conference*, pages 51–60, June 1989.

[47] B. Nitzberg and V. Lo. Distributed shared memory: A survey of issues and algorithms. *IEEE Computer*, 24(8):52–60, August 1991.

[48] M. Papamarcos and J. Patel. A low overhead coherence solution for multiprocessors with private cache memories. In *Proceedings of the 11th Annual International Symposium on Computer Architecture*, pages 348–354, May 1984.

[49] U. Ramachandran, M. Ahamad, and Y.A. Khalidi. Unifying synchronization and data transfer in maintaining coherence of distributed shared memory. Technical Report GIT-CS-88/23, Georgia Institute of Technology, June 1988.

[50] R.L. Sites and A.Agarwal. Multiprocessor cache analysis using ATUM. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 186–195, June 1988.

[51] M. Stumm and S. Zhou. Algorithms implementing distributed shared memory. *IEEE Computer*, 24(5):54–64, May 1990.

[52] C.P. Thacker, L.C. Stewart, and E.H. Satterthewaite, Jr. Firefly: A multiprocessor workstation. *IEEE Transactions on Computers*, 37(8):909–920, August 1988.

[53] A.C. Thekkath and H. Levy. Limits to low-latency communications on high-speed networks. *acm Transactions on Computer Systems*, 11(2):179–203, May 1993.

[54] J.E. Veenstra and R.J. Fowler. A performance evaluation of optimal hybrid cache coherency protocols. In *Proceedings of the 5th Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 149–160, September 1992.

[55] D.H.D. Warren and S. Haridi. The Data Diffusion machine - A shared virtual memory architecture for parallel execution of logic programs. In *Proceedings of the 1988 International Conference on Fifth Generation Computer Systems*, pages 943–952, Tokyo, Japan, December 1988.

[56] W.-D. Weber and A. Gupta. Analysis of cache invalidation patterns in multiprocessors. In *Proceedings of the 3rd Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 243–256, April 1989.

[57] A. Wilson and R. LaRowe. Hiding shared memory reference latency on the GalacticaNet distributed shared memory architecture. *Journal of Parallel and Distributed Computing*, 15(4):351–367, August 1992.

[58] L.D. Wittie, G. Hermannsson, and A. Li. Eager sharing for efficient massive parallelism. In *1992 International Conference on Parallel Processing*, pages 251–255, St. Charles, IL, August 1992.

[59] R.N. Zucker and J.-L. Baer. A performance study of memory consistency models. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 2–12, May 1992.