Design of the Munin Distributed Shared Memory System

John B. Carter
Department of Computer Science
University of Utah
Salt Lake City, UT 84112

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.

Design of the Munin Distributed Shared Memory System

Proposed running head: Design of the Munin Distributed Shared Memory System

Contact author: *John B. Carter*
Department of Computer Science
University of Utah
Salt Lake City, UT 84112

Abstract

Software distributed shared memory (DSM) is a software abstraction of shared memory on a distributed memory machine. The key problem in building an efficient DSM system is to reduce the amount of communication needed to keep the distributed memories consistent. The Munin DSM system incorporates a number of novel techniques for doing so, including the use of multiple consistency protocols and support for multiple concurrent writer protocols. Due to these, and other, features, Munin is able to achieve high performance on a variety of numerical applications This paper contains a detailed description of the design and implementation of the Munin prototype, with special emphasis given to its novel *write shared* protocol. Furthermore, it describes a number of lessons that we learned from our experience with the prototype implementation that are relevant to the implementation of future DSMs.

List of Symbols

Boldface, *italics*, and `typewriter font` are used often.

Special symbols used: A number of mathematical symbols are used, including: slash (/), leftarrow ($\leftarrow$), left brace ($\{$), right brace ($\}$), and ellipsis (...).

1 Introduction

A software distributed shared memory (DSM) system provides the abstraction of a shared address space spanning the processors of a distributed memory multiprocessor. This abstraction simplifies the programming of distributed memory multiprocessors and allows parallel programs written for shared memory machines to be ported easily. The basic idea behind software DSM, which we will refer to as simply "DSM" throughout the rest of this paper, is to treat the local memory of a processor as if it were a coherent cache in a shared memory multiprocessor. DSM systems combine the best features of shared memory and distributed memory multiprocessors. They support the relatively simple and portable programming model of shared memory on physically distributed memory hardware, which is more scalable and less expensive to build than shared memory hardware. DSM would thus seem to be an ideal vehicle for making parallel processing widely available. However, although many DSM systems have been proposed and implemented [16, 15, 20, 3, 13, 10, 17, 6, 23, 5, 2, 19], DSM is not widely used. The reason for this is that it has proven difficult to achieve acceptable performance using DSM without requiring programmers to carefully restructure their shared-memory parallel programs to reflect the way that the DSM operates. For example, it is often necessary to decompose the shared data into small page-aligned pieces or to introduce new variables to reduce the amount of sharing. This restructuring can be as tedious and difficult as using message-passing directly or more so.

The challenge in building a DSM system is to achieve good performance over a wide range of programs without requiring programmers to restructure their shared memory parallel programs. The overhead of maintaining consistency in software and the high latency of sending messages make this difficult. The primary source of DSM overhead is the large amount of communication that is required to maintain consistency. Since DSMs use general-purpose networks (e.g., Ethernet) and operating systems to communicate, the latency of each message between nodes is high. Given this high cost of interprocessor communication, the performance challenge for DSM is to reduce the amount of communication performed during the execution of a DSM program, ideally to the same level as the amount of communication performed during the execution of an equivalent message passing program. DSM has not gained wide acceptance because previous DSM systems did not achieve this level of communication.

The Munin DSM system incorporates several techniques make DSM a more viable solution for distributed processing by substantially reducing the amount of communication required to maintain consistency compared to previous DSM systems. The innovative communication-reducing features supported by Munin include the use of *multiple consistency protocols*, which keep each shared variable consistent with a protocol well suited to its expected or observed access pattern, and support for *multiple concurrent writers*, which addresses the problem of false sharing by reducing the amount of unnecessary communication performed keeping falsely shared data consistent.

To determine the value of Munin's novel features, we evaluated the performance of a suite of seven scientific applications implemented using message passing, Munin DSM, and a conventional Ivy[16]-like DSM system. Munin's performance is within 5% of message passing for four out of the seven applications studied, and within 25% to 35% for the other three. Furthermore, for the five applications in which there was a moderate to high degree of sharing, the programs run under Munin achieved from 25% to over 100% higher speedups than their conventional DSM counterparts.

Munin's performance on a variety of application programs is evaluated elsewhere [9, 7, 8] – this paper concentrates on the *design* of Munin, especially its major data structures and the protocols used to maintain memory consistency. Also included are a number of lessons that we garnered from our experience with the prototype implementation that are relevant to the implementation of future DSMs.

The remainder of this paper is organized as follows. Section 2 presents an overview of Munin's runtime system. Section 3 describes Munin's multiple consistency protocols and the algorithms used to implement them. We summarize Munin's performance in Section 4, and draw conclusions in Section 5.

2 Overview of the Munin Runtime System

The core of the Munin system is the runtime library that contains the fault handling, thread support, synchronization, and other runtime mechanisms. It consists of approximately 8000 lines of C source code that create an 185-kilobyte library file that is linked into each Munin program. Each node of an executing Munin program consists of a collection of Munin runtime threads that handle consistency and synchronization operations, and one or more user threads performing the parallel computation. Munin programmers write parallel programs using threads, as they would on a uniprocessor or shared memory multiprocessor.

Figure 1 illustrates the organization of a Munin program during runtime. The Munin prototype was implemented on a collection of sixteen SUN 3/60 workstations running the V operating system [11]. On each participating node, the Munin runtime 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 are an *object directory* that maintains the state of the shared data being used by local user threads and the *delayed update queue* (DUQ), which manages Munin's software implementation of release consistency[14]. Munin installs itself as the default page fault handler for the Munin program so that the underlying V kernel will forward all memory exceptions to it for handling.

Each Munin node interacts with the V kernel to communicate with the other Munin nodes over the Ethernet and to manipulate the virtual memory system as part of maintaining the consistency of shared memory. Although the Munin prototype was implemented on the V system, it could be made to run on any operating system that allowed user programs to: manipulate its own virtual memory mappings, handle page faults at user level, create and destroy processes on remote nodes, exchange messages with remote processes, and access uniprocessor synchronization support (`P()` and `V()`). All of these requirements are present in current operating systems such as Mach, Chorus, and Unix.

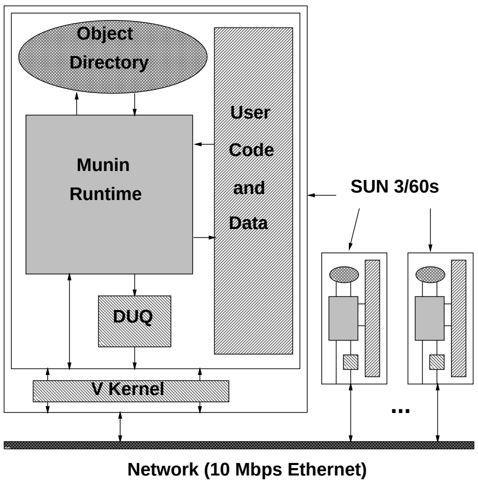

Figure 1 Munin Runtime Organization

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. In Munin, all variables on the same page are treated as a single "object" with the same consistency protocol. Programmers can guide placement of variables on to pages, but by default variables are placed on pages by themselves. The memory overhead of this solution is negligible for the applications studied as there were few distinct variables. When a Munin runtime thread cannot find an object directory entry in the local hash table, it requests a copy from the node where the computation started, the so-called the "root node," which contains all of the shared data at the start of the program. The fields of an entry in the directory include: a *lock* that provides exclusive access to the entry for a handler thread; a *start address* and *size* that act as keys for looking up a shared data item's directory entry; a *protocol* that specifies how the fault handler should service access faults on the data item; various *state bits* that characterize the dynamic state of the data, such as whether it is present locally and whether it is writable; a *copyset* that represents a "best guess" of the set of processors with copies of the data; and a *probable owner* that represents a "best guess" of the owner of the data.

The variable's *copyset* is the set of all nodes that the local node believes have a copy of the data. Nodes are added to a variable's copyset in several ways. If a remote node has a copy of the variable when the local node first accesses it, the reply message containing the data includes the remote node in the original copyset. Similarly, a node is added to the copyset when the local node handles a request from that node or the local node has been informed that the other node is caching the data as a side effect of some operation. It is possible for the copyset to include nodes no longer caching the data, because they have flushed their copy without informing the local node, 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 (the local copyset plus the copyset of the nodes in the local copyset plus ...) 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 that node informs the other nodes that it is no longer caching the data.

The *probable owner* is used to determine the identity of the Munin node that currently owns the data [16]. The owner node is used by the conventional and migratory protocols to arbitrate the decision of which node has 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 node willing to become the owner of last resort. This approach is analogous to the copy of last resort used in cache-only multiprocessors [25].

3 Multiple Consistency Protocols

Previous DSM systems have employed a single protocol to maintain the consistency of all shared data. The specific protocol varied from system to system, e.g., Ivy [16] supported a page-based emulation of a conventional hardware protocol while Emerald [15] used object-oriented language support to handle shared object invocations, but each system treated all shared data identically. This led to a situation where some programs could be handled effectively by a given DSM system, while others could not, depending on the way in which shared data was 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 [4] and others [12, 22, 1, 26, 24] 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. The results of those studies most relevant to the design of efficient DSM are summarized as follows:

1. A single mechanism cannot optimally support all data access patterns, and the most important distinction between the observed data access patterns is between those best handled via some form of *invalidate* protocol and those best handled via some form of *update* protocol [12, 22, 1, 26, 4, 24].

2. The number of characteristic sharing patterns is small and most shared data can be characterized as being accessed in one of these ways [26, 4].

3. Synchronization variables are accessed in an inherently different way than data variables, and are more sensitive to increased access latency [26, 4].

4. The characteristic access pattern of individual variables does not change frequently during execution [4, 24], so a static protocol selection policy suffices in most cases.

The first three results strongly suggest that a DSM system that supports a small number of consistency protocols will outperform conventional DSM systems that support a single static protocol. A small number of data access patterns characterize most accesses to shared data. Thus, it is feasible to support sufficient consistency protocols so that most individual shared variables can be kept consistent with a protocol that is well suited to the way that they are characteristically accessed, without the DSM system becoming overly complicated. Furthermore, the results indicate that at the very least three protocols should be supported: one invalidation-based, one update-based, and one for synchronization.

Munin supports four consistency protocols (*conventional*, *read-only*, *migratory*, and *write-shared*) plus a suite of synchronization protocols for locks, barriers, and condition variables. In addition to the consistency protocols provided, we provide a capability for users to install their own fault handlers and use Munin's facilities to create consistency protocols of their own. When writing a Munin program, the programmer annotates the declaration of shared variables with a sharing pattern to specify what protocol to use to keep it consistent, e.g., "`shared {write-shared} <C_type> <variable_name>`". If a variable is not annotated, the conventional protocol is used. Incorrect annotations may result in inefficient performance or in runtime errors that are detected by the Munin runtime system, but not in incorrect behavior if there is sufficient synchronization to satisfy the requirements of release consistency.

The consistency protocols all have roughly the same structure. Consistency operations generally are initiated when a user thread attempts to access data that is either not present or that has been protected by the runtime system so that accesses to it generates exceptions. The fault handler examines the exception message to determine the location and nature of the exception, which it uses to look up the data item in the object directory. It then performs the consistency operations appropriate to the data's protocol. After performing the consistency operations required to satisfy the fault, the fault handler resumes the user thread and waits to receive its next exception or remote request message. The remainder of this section describes the implementation of Munin's consistency mechanisms. More detailed descriptions, including pseudo-code of the algorithms, can be found elsewhere [7].

3.1 Conventional

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. We based Munin's *conventional* protocol on Ivy's distributed dynamic manager protocol [16]. This protocol is typical of what existing DSM systems provide [16, 13, 20], and is the *conventional DSM* protocol evaluated in our performance study.

We incorporated a simplified version of the freezing mechanism from Mirage [13] so that after a node acquires ownership of a conventional data item, it does not reply to requests from other nodes for a period of time (100 msecs). This mechanism guarantees that the node performing the write makes progress even in the face of heavy sharing. The performance of the conventional DSM was largely unaffected by the choice of the freeze time as long as it is above 10 msecs. Without the timeout mechanism, we observed several phenomena that severely hurt the performance of conventional data when there was a high degree of sharing. When two (or more) threads concurrently modify a single page, a frequent occurrence, the data ping pongs between the writers, with little progress made between faults. The freezing mechanism partially alleviates this problem by ensuring that progress is made no matter how much sharing is occurring. There were even problems when there was only a single writer, but multiple readers, of a single data item. When there were a large number of readers, it was often the case that by the time the writer finished invalidating all the replicas, one of the first nodes to be invalidated would have requested a new copy of the data to satisfy a read miss. Munin runtime threads have a higher scheduling priority than user threads to ensure that remote data requests do not starve, so the reload request was satisfied before the writer was resumed. This choice of priorities resulted in the data being read protected on the original node before the writer was able to complete the write that caused the original write fault, which caused the writer to fault anew. This result convinced us that a freezing mechanism akin to that found in Mirage should be incorporated into future software DSMs that support ownership-based invalidate protocols.

3.2 Read Only

Read-only data is writable only during initialization, which allows it to be initialized along with the rest of the program. The consistency protocol simply consists of replication on demand. A runtime error is generated and the system debugger is invoked if a thread attempts to write to read-only data. As noted earlier, if the programmer does not specify that a particular variable is shared, it is replicated when the worker nodes are created.

Thus, read-only objects could simply be not marked as shared. There are two reasons that programmers might wish to distinguish read-only shared data from non-shared data: (i) for debugging purposes, to detect unexpected writes to input data, and (ii) to conserve memory by loading on demand only the portion of the read-only data that a given node requires.

3.3 Migratory

For *migratory* data, a single thread performs multiple accesses to the data, including one or more writes, before another thread accesses the data [26, 4]. This access pattern is typical of shared data that is accessed only inside a critical section or via a work queue. For this type of data, it is generally best to migrate the data to a processor as soon as it accesses it the first time, regardless of whether the first access is a read or a write. 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. In addition, the Munin programmer can specify the logical connections between shared variables and the synchronization variables that protect them. This information is conveyed to the runtime system using the `AssociateDataAndSynch()` call, which suggests that Munin include a copy of the specified shared variable in the message that passes ownership of the specified synchronization variable. This pragma is particularly useful for associating migratory data accessed within a critical section with the lock controlling the critical section. It reduces the number of faults and messages needed to migrate the data, as in Clouds [20].

3.4 Write Shared

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 independent portions of the data. Because of the way that the data is laid out in memory, write-shared data often exhibits a high degree of sharing at a coarse granularity (e.g., a cache line or page), but no sharing at a word granularity – a phenomenon known as *false sharing*. One example of false sharing is when two independent shared variables reside on the same page of memory, each being modified by a different processor. Another example is when a shared array is laid out contiguously in a single page of memory and different processors are modifying disjoint parts of the array. An intelligent compiler or careful user can alleviate the false sharing in the first case by allocating unrelated variables on distinct pages, at the expense of using extra memory. However, the false sharing in the second case is unavoidable because the falsely shared data is part of a single contiguous array. We have observed that write-shared data is very common, and that its presence results in very poor DSM performance if it is handled by a conventional consistency protocol that communicates whenever a shared page is modified. 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. Any DSM system that expects to achieve acceptable performance *must* address the problem of false sharing.

The write-shared protocol is designed specifically to mitigate the effect of false sharing. It does so by supporting concurrent writers, by buffering writes until synchronization requires their propagation, by using an update-based consistency protocol, and by timing out and invalidating data that is not being used frequently. Unlike existing update protocols, the write-shared protocol *buffers* and *combines* update messages, as shown in Figure 2. The reason that Munin can buffer updates, rather than send them as soon as they are generated, is that it supports the *release consistency* memory model[14]. A detailed description of the release consistency model is beyond the scope of this paper, but, roughly, a shared memory implementation based upon release consistency can delay the point at which memory must be consistent until a processor performs a "release" operation (e.g., releases a lock or arrives at a barrier). In the case of Munin, this means that updates to shared data can be buffered and combined between release points. A single processor often performs a series of writes to a shared block within a critical section [4]. When this occurs, the write-shared protocol transmits a single update message containing all of the changes performed within the critical section to each node caching a copy of the data, rather than sending a stream of updates as each write occurs. This can lead to order of magnitude reductions in the number of messages required to maintain consistency compared to a conventional pipelined invalidate protocol. For DSM, where messages are expensive but where large messages are not much more expensive than small messages, combining updates is important.

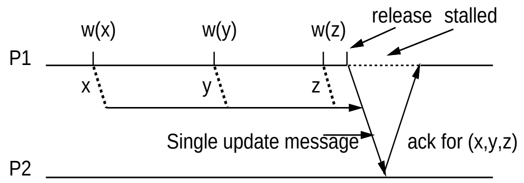

Figure 2 Buffering updates in Munin

The mechanism used to buffer and combine update messages, the *delayed update queue* (DUQ), is illustrated in Figure 3. The initial copyset of a write-shared data item is empty on all nodes, including on the root node. Before a write-shared variable (X) is modified, the page on which it resides is mapped so that the first write will cause a page fault, symbolized via the dashed box.

Read faults are handled as follows. If the data is present but has been read protected, such as the first time it is accessed after the node has received an update and the timeout mechanism has read protected it (see below), it simply remaps the page to be readable and marks the data as accessed as part of the update timeout mechanism. When a write-shared variable is first loaded on a node, it is made read-only so any attempts to modify it are detected. The one exception to this is if it is the only copy of the data in the system, in which case it can be mapped read-write until another node requests a copy, at which time the original copy is made read-only.

Write faults are handled similarly except when the data is being actively shared. In this case, the write-shared protocol invokes the DUQ mechanism, as illustrated in Figure 3a. After determining that the accessed variable (call it X) is write-shared, the write fault handler makes a copy of X (X_{twin}), and puts the data item's directory entry on the DUQ. It then maps the original copy of X to be read-write and resumes the faulted thread. Since the original copy of X is no longer write protected, all subsequent writes to X proceed with no consistency overhead. The key feature of the write fault handler is that it only communicates with other nodes if the data is not present. If the data is already present, but mapped read-only or supervisor-only, the handler only performs local operations. The handler does *not* need to get exclusive write access *nor* does it need to immediately propagate the modification to the other cached copies. This feature allows multiple nodes to concurrently modify a single data item without communicating for each write.

The server routine that handles remote requests for write-shared data is straightforward. It is irrelevant whether or not the requesting node faulted on a read or a write. Any node can respond to a request and not just the owner. If the node that satisfies a data request has a writable copy but not a twin of the data, the dirty copy of the data is sent to the requester and then remapped to be read-only so subsequent changes to the data are detected and eventually purged.

The trickiest part of the write shared protocol occurs when a local thread performs a release operation (releases a lock, arrives at a barrier, signals a condition variable, or terminates). At this time, all delayed modifications to write-shared data must be propagated to their remote copies before the local thread may proceed. These changes are propagated in phases, where each phase is responsible for propagating changes to a particular set of nodes. First, the DSM runtime determines if any updates have been buffered on the DUQ. If there are, the server walks down the DUQ and creates a differential encoding of every enqueued data item. In the example illustrated in Figure 3, it determines that X has been modified. It encodes the modifications by performing a word-by-word comparison of the data item (X) and its original contents (X_{twin})[1]. As it finds differences, it copies them to

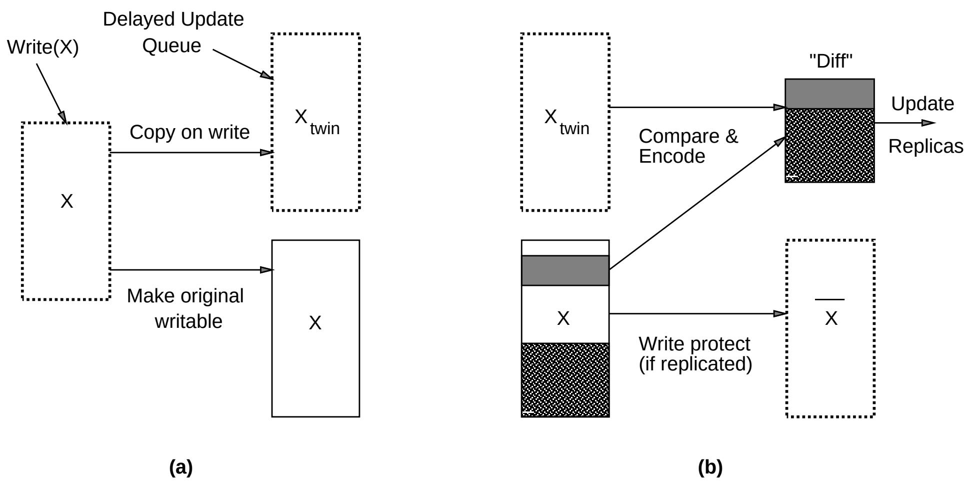

Figure 3 Delayed Update Queue Operation

[1]By detecting changes at the word level and not the byte level, we risk the possibility that two threads will modify different bytes of the same word, and the update mechanism will either signal a data race or overwrite one of the modifications. The applications that we examined had no byte-grained shared data, but if the

a buffer, prepending each sequence of updated words with their starting address and the run length. Because the "twin" is not needed after the modifications have been propagated, Munin uses the buffer that contained the twin to contain the encoding. By using eight bytes of scratch space in the message header and the portions of the twin array that have already been scanned, the encoding algorithm has the property that the "diff" is never larger than the twin, and it never overwrites parts of the twin that have not yet been examined as part of the encoding process. Thus, reuse of the twin buffer is safe.

After encoding all of the data items enqueued on the DUQ, the server transmits a descriptor message containing a list of the encoded data items to the remote nodes sharing any of the encoded data. Each of these recipients determines if it needs to receive updates to any of the data described in the descriptors, requests the encoded data that it is still caching, and replies with an indication of (i) whether it is still caching each data item and (ii) its copyset for each data item. The updating node continues to transmit update messages to any node thought to be caching a copy of any of the encoded data until it has sent updates to all such nodes. This process is guaranteed to terminate whenever either no new nodes are added to the copyset during a phase or all nodes have been updated. When a NACK is received, the node that sent the NACK is removed from the local copyset for the data items specified. After all the updates for a data item have been performed, X is write-protected if it is still replicated to ensure that subsequent writes are detected.

To incorporate updates, the receiving Munin worker thread examines the update message and determines if it is still caching any of the data specified. It is possible that the node has invalidated data that it once cached as a result of the update timeout mechanism (see below). If it still has a copy of any of the data described in the update message, it requests the corresponding encodings and sends a reply message to the updating node. The reply message includes an indication of which of the encoded data items the node is still caching and its copyset for each encoded item, whether it is still being cached or not. The receiving node then decodes the updates to extract the individual words that the sending node has modified. If the node is not caching any of the data described in the update message, it sends a reply message with NACKs and copysets for all of the data.

Incorporating an update normally entails simply traversing the encoding and copying the

---

user anticipates a problem, a runtime switch can change the granularity of comparison to the byte level, at the expense of increased encoding and decoding time.

modified sequences into the local copy of the data. However, the user can specify that the runtime system detect *dynamic data races*, which is useful for debugging complex parallel programs with synchronization bugs. If a node receives an update for a variable that it has modified, a clean copy of the variable will be present. The system can detect data races by performing a three-way comparison of the received update, the dirty version, and the clean version as it decodes the encoded updates. If, while performing the three-way comparison, it finds that all three copies of a particular word differ, it has detected a data race on that word and it generates an error message detailing what it has found.

This process is complicated somewhat by an *update timeout mechanism*. The goal of the timeout mechanism is to invalidate data that has grown stale so as to avoid unnecessary future updates. The timeout mechanism ensures that updates to an object are only accepted for a limited period of time after it was last accessed. When an update is incorporated, the data is temporarily mapped to be supervisor-only so that any access to it (read or write) by a local user thread will be detected. A timestamp is set in the data item's directory entry at this time. If the data is accessed before another update arrives, the subsequent fault simply remaps the data and resets a flag. However, if an update to the data is received and the node has not used the data since the last update, sufficient time has elapsed since the last update, and the data is not dirty, the update server invalidates the local copy. A *copy of last resort* for each write-shared variable is used to prevent all copies of the variable from being invalidated via the update timeout mechanism when several nodes send updates simultaneously. This copy cannot be invalidated unless the node is first able to find another node to take over this responsibility. The *prob_owner* chain is used to find this copy of last resort - the "owner" in the write-shared protocol is this copy.

A technique similar to the delayed update queue was used by the Myrias SPS multiprocessor [18]. It performed the *copy-on-write* and *diff* in hardware, but required a restricted form of parallelism to ensure correctness. Specifically, only one processor could modify a cache line at a time, and the only form of parallelism that could exploit this mechanism was a form of Fortran `doall` statement.

We considered implementing write detection by having the compiler add code to log writes to replicated data as part of the write, as is done in Emerald [15] and Midway [5]. However, although recent results indicate that compiler-based write detection can outperform VM-based detection [27], we chose not to explore this approach in the prototype because we did not want to modify the compiler and we are concerned with the portability constraints imposed by the requirement that DSM programmers use a special compiler. It is an attractive alternative for systems that do not support fast page fault handling, such as the iPSC-i860 hypercube. However, if the number of writes to a particular data item between DUQ flushes is high, as is often the case [4], this approach will perform relatively poorly because each write to a shared variable is slower.

4 Performance Summary

We present a summary of Munin's performance here to illustrate the value of Munin's design. More detailed evaluations appear elsewhere [9, 7, 8].

Seven application programs were used in the evaluation of Munin: Matrix Multiply (MULT), Finite Differencing (DIFF), Traveling Salesman Problem (both fine-grained, TSP-F, and coarse-grained, TSP-C), Quicksort (QSORT), Fast Fourier Transform (FFT), and Gaussian Elimination with Partial Pivoting (GAUSS). Three different versions of each application were written: a Munin DSM version, a conventional DSM version that used Munin's conventional protocol to implement a sequentially consistent memory (and thus measured the performance of a conventional DSM system such as Ivy), and a message passing version. The message passing programs represent best case implementations of the parallel programs. The DSM runtime system used the same general purpose message passing facilities provided by the operating system that the message passing programs used directly. Great care was taken to ensure that the inner loops of each computation, the problem decomposition, and the major data structures for each version were identical.

To evaluate the impact of Munin's design, two basic comparisons were performed. In the first comparison, the performance of the Munin versions of the programs was compared to the performance of equivalent message passing programs. This comparison measures the extent to which programs written using the shared memory model and run under Munin can achieve performance comparable to programs written using explicit message passing. Table 1 presents the speedup achieved by each version of the program and the percentage of the message passing speedup achieved by the Munin DSM version of the program for sixteen processors. For four of the applications (MULT, DIFF, TSP-C, and FFT), the Munin versions achieve over 95% of the speedup of their hand-coded message-passing equivalents, despite the fact that no significant restructuring of the original shared-memory programs was performed while porting them to run under Munin. For the other three applications (TSP-F, QSORT and GAUSS), the performance of the Munin variants is between 66% and 71% of their message-passing equivalents. Support for explicit RPC improves the performance of these applications to within 90% of their message passing equivalents [8, 9].

	Message Passing Speedup	Munin Speedup	How Close?
MULT	14.7	14.6	100%
DIFF	12.8	12.3	96%
TSP-C	13.2	12.6	96%
TSP-F	8.9	5.9	66%
QSORT	13.4	8.9	67%
FFT	8.6	8.2	95%
GAUSS	12.1	8.6	71%

Table 1 Munin vs Message Passing

	Munin Speedup	Conven- -tional Speedup	How Close?
MULT	14.6	14.5	99%
DIFF	12.3	8.4	68%
TSP-C	12.6	11.3	90%
TSP-F	5.9	4.7	80%
QSORT	8.9	4.1	46%
FFT	8.2	0.09	$< 0.1\%$
GAUSS	8.6	5.1	59%

Table 2 Conventional DSM vs Munin DSM

In the second comparison, the performance of the Munin versions of the programs was compared to the performance of the conventional DSM versions. This comparison identifies the types of programs that stand to benefit the most from Munin's novel features and the types of programs that are adequately supported with a conventional page-based invalidation-style DSM. Table 2 presents the speedup achieved by each version of the program and the percentage of the Munin DSM speedup achieved by the conventional DSM version of the program for sixteen processors. For the programs with large grained sharing (MULT and TSP-C), the conventional versions achieved performance within 10% of the speedup of their Munin counterparts. For DIFF, TSP-F, and GAUSS, the performance of the conventional versions was reduced to 59-80% of Munin. For QSORT, conventional performance was under 50% of Munin performance, while for FFT conventional performance was orders of magnitude worse.

These results are very promising and argue that Munin represents a significant step towards making DSM useful on a much wider spectrum of programs and programming styles. Specifically, conventional DSM performs well on programs with relatively little sharing or when the sharing is at a large granularity. However, its performance drops off quickly when there is much fine-grained sharing and becomes unacceptable when there is much concurrent write sharing or false sharing.

5 Conclusions

This paper contains a detailed description of the design and implementation of the Munin prototype with special emphasis given to its write shared protocol. Munin contains a number of mechanisms and implementation strategies designed to improve the performance of DSM, including support for concurrent writers, the use of distributed ownership protocols, and the provision of a specialized library of synchronization operations. Many of these features are also relevant to the design of future scalable shared memory hardware.

We learned a number of lessons from our experience with the prototype implementation that designers of future DSM systems should consider, including:

1. DSM can be made efficient without the use of unconventional programming languages, compilers, or operating system support.

2. Organizing the DSM runtime as a library package works well. In particular, this organization improves performance by reducing the number of context switches and the amount of cross-domain memory copying required to maintain consistency.

3. The implementation of delayed updates is subtle. The key to handling write-shared data efficiently is to minimize the amount of time spent purging the DUQ, so it is important to perform updates in parallel using multiple threads or multicast.

4. While it is easy to add consistency protocols if the server software is well designed, a small number of protocols is enough to handle the way that most shared data is accessed.

Munin's performance could be improved significantly by a number of factors: (a) lower latency OS operations (message passing, exception handling, and VM remapping), (b) a high bandwidth multicast network, and (c) fine-grained write detection hardware. In the prototype, OS overhead was a major source of overhead. In addition, while we carefully avoided the use of Ethernet's multicast capability to better model how Munin would work on a scalable network technology, the presence of multicast would greatly reduce the time needed to perform updates in the infrequent, but very expensive, situations where a large number of nodes are write sharing data, as in FFT. Finally, fine-grained write detection hardware[21] could eliminate the largest component of software overhead, diff creation. These issues are subjects of ongoing research at the University of Utah.

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] 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.

[3] H.E. Bal and A.S. Tanenbaum. Distributed programming with shared data. In *Proceedings of the 1988 International Conference on Computer Languages*, pages 82–91, October 1988.

[4] 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.

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

[6] 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.

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

[8] J.B. Carter, J.K. Bennett, and W. Zwaenepoel. Techniques for reducing consistency-related communication in distributed shared memory systems. *ACM Transactions on Computer Systems*. To appear.

[9] 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.

[10] 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.

[11] D.R. Cheriton and W. Zwaenepoel. The distributed V kernel and its performance for diskless workstations. In *Proceedings of the 9th ACM Symposium on Operating Systems Principles*, pages 129–140, October 1983.

[12] 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.

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

[14] 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.

[15] 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.

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

[17] 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.

[18] Myrias Corporation. System overview. Edmonton, Alberta, 1990.

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

[20] 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.

[21] S.K. Reinhardt, J.R. Larus, and D.A. Wood. Tempest and Typhoon: User-level shared memory. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 325–336, April 1994.

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

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

[24] 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.

[25] 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, December 1988.

[26] 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.

[27] M.J. Zekauskas, W.A. Sawdon, and B.N. Bershad. Software write detection for distributed shared memory. In *Proceedings of the First Symposium on Operating System Design and Implementation*, pages 87–100, November 1994.