Implementation and Performance of Munin

John B. Carter
Dept. of Comp. Sci.
Rice University
Houston, Texas 77251-1892

John K. Bennett
Dept. of Elec./Comp. Eng.
Rice University
Houston, Texas 77251-1892

Willy Zwaenepoel
Dept. of Comp. Sci.
Rice University
Houston, Texas 77251-1892

Appeared in Proceedings of the Thirteenth Symposium on Operating System Principles, pp. 152–164, October 1991.

Abstract

Munin is a distributed shared memory (DSM) system that allows shared memory parallel programs to be executed efficiently on distributed memory multiprocessors. Munin is unique among existing DSM systems in its use of *multiple consistency protocols* and in its use of *release consistency*. In Munin, shared program variables are annotated with their expected access pattern, and these annotations are then used by the runtime system to choose a consistency protocol best suited to that access pattern. Release consistency allows Munin to mask network latency and reduce the number of messages required to keep memory consistent. Munin's multi-protocol release consistency is implemented in software using a *delayed update queue* that buffers and merges pending outgoing writes. A sixteen-processor prototype of Munin is currently operational. We evaluate its implementation and describe the execution of two Munin programs that achieve performance within ten percent of message passing implementations of the same programs. Munin achieves this level of performance with only minor annotations to the shared memory programs.

This research was supported in part by the National Science Foundation under Grants CDA-8619893 and CCR-9010351, by the IBM Corporation under Research Agreement No. 20170041, and by a NASA Graduate Fellowship. Willy Zwaenepoel was on sabbatical at UT-Sydney and INRIA-Rocquencourt while a portion of this research was conducted.

1 Introduction

A 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 challenge in building a DSM system is to achieve good performance without requiring the programmer to deviate significantly from the conventional shared memory programming model. High memory latency and the high cost of sending messages make this difficult.

To meet this challenge, Munin incorporates two novel features. First, Munin employs *multiple consistency protocols*. Each shared variable declaration is annotated by its expected access pattern. Munin then chooses a consistency protocol suited to that pattern. Second, Munin is the first software DSM system that provides a *release-consistent* memory interface [19]. Roughly speaking, release consistency requires memory to be consistent only at specific synchronization points, resulting in a reduction of overhead and number of messages.

The Munin programming interface is the same as that of conventional shared memory parallel programming systems, except that it requires (i) all shared variable declarations to be annotated with their expected access pattern, and (ii) all synchronization to be visible to the runtime system. Other than that, Munin provides thread, synchronization, and data sharing facilities like those found in shared memory parallel programming systems [7].

We report on the performance of two Munin programs: Matrix Multiply and Successive Over-Relaxation (SOR). We have hand-coded the same programs on the same hardware using message passing, taking special care to ensure that the two versions of each program perform identical computations. Comparison between the Munin and the message passing versions has allowed us to assess the overhead associated with our approach. This comparison is encouraging: the performance of the Munin programs differs from that of the hand-coded message passing programs by at most ten percent, for configurations from one to sixteen processors.

The Munin prototype implementation consists of four parts: a simple preprocessor that converts the program annotations into a format suitable for use by the Munin runtime system, a modified linker that creates the shared memory segment, a collection of library routines that are linked into each Munin program, and operating system support for page fault handling and page table manipulation. This separation of functionality has resulted in a system that is largely machine- and language-independent, and we plan to port it to various other platforms and languages. The current prototype is implemented on a workstation-based distributed memory multiprocessor consisting of 16 SUN-3/60s connected by a dedicated 10 Mbps Ethernet. It makes use of a version of the V kernel [11] that allows user threads to handle page faults and to modify page tables.

A preliminary Munin design paper has been published previously [6], as well as some measurements on shared memory programs that corroborate the basic design [5]. This paper presents a refinement of the design, and then concentrates on the implementation of Munin and its performance.

2 Munin Overview

2.1 Munin Programming

Munin programmers write parallel programs using threads, as they would on a shared memory multiprocessor [7]. Munin provides library routines, `CreateThread()` and `DestroyThread()`, for this purpose. Any required user initialization is performed by a sequential `user_init()` routine, in which the programmer may also specify the number of threads and processors to be used. Similarly, there is an optional sequential `user_done()` routine that is run when the computation has finished. Munin currently does not perform any thread migration or global scheduling. User threads are run in a round robin fashion on the node on which they were created.

A Munin *shared object* corresponds to a single shared variable, although like Emerald [9], the programmer can specify that a collection of variables be treated as a single object or that a large variable be treated as a number of independent objects by the runtime system. By default, variables larger than a virtual memory page are broken into multiple page-sized objects. We use the term "object" to refer to an object as seen by the runtime system, i.e., a program variable, an 8-kilobyte (page-sized) region of a variable, or a collection of variables that share an 8-kilobyte page. Currently, Munin only supports statically allocated shared variables, although this limitation can be removed by a minor modification to the memory allocator. The programmer annotates the declaration of shared variables with a sharing pattern to specify the way in which the variable is expected to be accessed. These annotations indicate to the Munin runtime system what combination of protocols to use to keep shared objects consistent (see Section 2.3).

Synchronization is supported by library routines for the manipulation of locks and barriers. These library routines include `CreateLock()`, `AcquireLock()`, `ReleaseLock()`, `CreateBarrier()`, and `WaitAtBarrier()`. All synchronization operations must be explicitly visible to the runtime system (i.e., must use the Munin synchronization facilities). This restriction is necessary for release consistency to operate correctly (see Section 2.2).

2.2 Software Release Consistency

Release consistency was introduced by the DASH system. A detailed discussion and a formal definition can be found in the papers describing DASH [19, 23]. We summarize the essential aspects of that discussion.

Release consistency requires that each shared memory access be classified either as a *synchronization* access or an *ordinary* access.[1] Furthermore, each synchronization accesses must be classified as either a *release* or an *acquire*. Intuitively, release consistency requires the system to recognize synchronization accesses as special events, enforce normal synchronization ordering requirements,[2] and guarantee that the results of all writes performed by a processor prior to a release be propagated before a remote processor acquires the lock that was released.

More formally, the following conditions are required for ensuring release consistency:

1. Before an ordinary load or store is allowed to perform with respect to any other processor, all previous acquires must be performed.

2. Before a release is allowed to perform with respect to any other processor, all previous ordinary loads and stores must be performed.

3. Before an acquire is allowed to perform with respect to any other processor, all previous releases must be performed. Before a release is allowed to perform with respect to any other processor, all previous acquires and releases must be performed.

The term "all previous accesses" refers to all accesses by the same thread that precede the current access in program order. A load is said to have "performed with respect to another processor" when

[1]We ignore chaotic data [19] in this presentation.

[2]For example, only one thread can acquire a lock at a time, and a thread attempting to acquire a lock must block until the acquire is successful.

a subsequent store on that processor cannot affect the value returned by the load. A store is said to have "performed with respect to another processor" when a subsequent load by that processor will return the value stored (or the value stored in a later store). A load or a store is said to have "performed" when it has performed with respect to all other processors.

Previous DSM systems [3, 10, 17, 25, 27] are based on *sequential consistency* [22]. Sequential consistency requires, roughly speaking, that each modification to a shared object become visible immediately to the other processors in the system. Release consistency postpones until the next release the time at which updates must become visible. This allows updates to be buffered until that time, and avoids having to block a thread until it is guaranteed that its current update has become visible everywhere. Furthermore, if multiple updates need to go to the same destination, they can be coalesced into a single message. The use of release consistency thus allows Munin to mask memory access latency and to reduce the number of messages required to keep memory consistent. This is important on a distributed memory multiprocessor where remote memory access latency is significant, and the cost of sending a message is high.

To implement release consistency, Munin requires that all synchronization be done through system-supplied synchronization routines. We believe this is not a major constraint, as many shared memory parallel programming environments already provide efficient synchronization packages. There is therefore little incentive for programmers to implement separate mechanisms. Unlike DASH, Munin does not require that each individual shared memory access be marked.

Gharachorloo et al. [16, 19] have shown that a large class of programs, essentially programs with "enough" synchronization, produce the same results on a release-consistent memory as on a sequentially-consistent memory. Munin's multiple consistency protocols obey the ordering requirements imposed by release consistency, so, like DASH programs, Munin programs with "enough" synchronization produce the same results under Munin as under a sequentially-consistent memory. The experience with DASH and Munin indicates that almost all shared memory parallel programs satisfy this criterion. No modifications are necessary to these programs, other than making all synchronization operations utilize the Munin synchronization facilities.

2.3 Multiple Consistency Protocols

Several studies of shared memory parallel programs have indicated that no single consistency protocol is best suited for all parallel programs [5, 14, 15]. Furthermore, within a single program, different shared variables are accessed in different ways and a particular variable's access pattern can change during execution [5].

Munin allows a separate consistency protocol for each shared variable, tuned to the access pattern of that particular variable. Moreover, the protocol for a variable can be changed over the course of the execution of the program. Munin uses program annotations, currently provided by the programmer, to choose a consistency protocol suited to the expected access pattern of each shared variable.

The implementation of multiple protocols is divided into two parts: high-level sharing pattern annotations and low-level protocol parameters. The high-level annotations are specified as part of a shared variable declaration. These annotations correspond to the expected sharing pattern for the variable. The current prototype supports a small collection of these annotations that closely correspond to the sharing patterns observed in our earlier study of shared memory access patterns [5]. The low-level protocol parameters control specific aspects of the individual protocols, such as whether an object may be replicated or whether to use invalidation or update to maintain consistency. In Section 2.3.1, we discuss the low-level protocol parameters that can be varied under Munin, and in Section 2.3.2 we discuss the high-level sharing patterns supported in the current

Munin prototype.

2.3.1 Protocol Parameters

Munin's consistency protocols are derived by varying eight basic protocol parameters:

- **Invalidate or Update? (I)** This parameter specifies whether changes to an object should be propagated by invalidating or by updating remote copies.
- **Replicas allowed? (R)** This parameter specifies whether more than one copy of an object can exist in the system.
- **Delayed operations allowed? (D)** This parameter specifies whether or not the system may delay updates or invalidations when the object is modified.
- **Fixed owner? (FO)** This parameter directs Munin *not* to propagate ownership of the object. The object may be replicated on reads, but all writes must be sent to the owner, from where they may be propagated to other nodes.
- **Multiple writers allowed? (M)** This parameter specifies that multiple threads may concurrently modify the object with or without intervening synchronization.
- **Stable sharing pattern? (S)** This parameter indicates that the object has a stable access pattern, i.e., the same threads access the object in the same way during the entire execution of the program. If a different thread attempts to access the object, Munin generates a runtime error. For stable sharing patterns, Munin always sends updates to the same nodes. This allows updates to be propagated to nodes prior to these nodes requesting the data.
- **Flush changes to owner? (Fl)** This parameter directs Munin to send changes only to the object's owner and to invalidate the local copy whenever the local thread propagates changes.
- **Writable? (W)** This parameter specifies whether the shared object can be modified. If a write is attempted to a non-writable object, Munin generates a runtime error.

2.3.2 Sharing Annotations

Sharing annotations are added to each shared variable declaration, to guide Munin in its selection of the parameters of the protocol used to keep each object consistent. While these annotations are syntactically part of the variable's declaration, they are not programming language types, and as such they do not nest or cause compile-time errors if misused. Incorrect annotations may result in inefficient performance or in runtime errors that are detected by the Munin runtime system.

Read-only objects are the simplest form of shared data. Once they have 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 to write to a read-only object.

For **migratory** objects, a single thread performs multiple accesses to the object, including one or more writes, before another thread accesses the object [29]. Such an access pattern is typical of shared objects that are accessed only inside a critical section. The consistency protocol for migratory objects is to migrate the object to the new thread, provide it with read and write access (even if the first access is a read), and invalidate the original copy. This protocol avoids a write miss and a message to invalidate the old copy when the new thread first modifies the object.

Write-shared objects are concurrently written by multiple threads, without the writes being synchronized, because the programmer knows that the updates modify separate words of the object.

However, because of the way that objects are laid out in memory, there may be *false sharing.* False sharing occurs when two shared variables reside in the same consistency unit, such as a cache block or a virtual memory page. In systems that do not support multiple writers to an object, the consistency unit may be exchanged between processors even though the processors are accessing different objects.

Producer-consumer objects are written (produced) by one thread and read (consumed) by one or more other threads. The producer-consumer consistency protocol is to *replicate* the object, and to *update*, rather than invalidate, the consumer's copies of the object when the object is modified by the producer. This eliminates read misses by the consumer threads. Release consistency allows the producer's updates to be buffered until the producer releases the lock that protects the objects. At that point, all of the changes can be passed to the consumer threads in a single message. Furthermore, producer-consumer objects have *stable* sharing relationships, so the system can determine once which nodes need to receive updates of an object, and use that information thereafter. If the sharing pattern changes unexpectedly, a runtime error is generated.

Reduction objects are accessed via `Fetch_and_`Φ operations. Such operations are equivalent to a lock acquisition, a read followed by a write of the object, and a lock release. An example of a reduction object is the global minimum in a parallel minimum path algorithm, which would be maintained via a `Fetch_and_min`. Reduction objects are implemented using a fixed-owner protocol.

Result objects are accessed in phases. They are alternately modified in parallel by multiple threads, followed by a phase in which a single thread accesses them exclusively. The problem with treating these objects as standard write-shared objects is that when the multiple threads complete execution, they unnecessarily update the other copies. Instead, updates to result objects are sent back only to the single thread that requires exclusive access.

Conventional objects are replicated on demand and are kept consistent by requiring a writer to be the sole owner before it can modify the object. Upon a write miss, an invalidation message is transmitted to all other replicas. The thread that generated the miss blocks until it has the only copy in the system [24]. A shared object is considered conventional if no annotation is provided by the programmer.

The combination of protocol parameter settings for each annotation is summarized in Table 1.

New sharing annotations can be added easily by modifying the preprocessor that parses the Munin program annotations. For instance, we have considered supporting an invalidation-based protocol with delayed invalidations and multiple writers, essentially invalidation-based write-shared

Sharing Annotation	Parameter Settings							
	I	R	D	FO	M	S	Fl	W
Read-only	N	Y	-	-	-	-	-	N
Migratory	Y	N	-	N	N	-	N	Y
Write-shared	N	Y	Y	N	Y	N	N	Y
Producer-Consumer	N	Y	Y	N	Y	Y	N	Y
Reduction	N	Y	N	Y	N	-	N	Y
Result	N	Y	Y	Y	Y	-	Y	Y
Conventional	Y	Y	N	N	N	-	N	Y

Table 1 Munin Annotations and Corresponding Protocol Parameters

objects, but we have chosen not to implement such a protocol until we encounter a need for it.

2.4 Advanced Programming

For Matrix Multiply and Successive Over-Relaxation, the two Munin programs discussed in this paper, simply annotating each shared variable's declaration with a sharing pattern is sufficient to achieve performance comparable to a hand-coded message passing version. Munin also provides a small collection of library routines that allow the programmer to fine-tune various aspects of Munin's operation. These "hints" are optional performance optimizations.

In Munin, the programmer can specify the logical connections between shared variables and the synchronization objects that protect them [27]. Currently, this information is provided by the user using an `AssociateDataAndSynch()` call. If Munin knows which objects are protected by a particular lock, the required consistency information is included in the message that passes lock ownership. For example, if access to a particular object is protected by a particular lock, such as an object accessed only inside a critical section, Munin sends the new value of the object in the message that is used to pass lock ownership. This avoids one or more access misses when the new lock owner first accesses the protected data.

The `PhaseChange()` library routine purges the accumulated sharing relationship information (i.e., what threads are accessing what objects in a producer-consumer situation). This call is meant to support adaptive grid or sparse matrix programs in which the sharing relationships are stable for long periods of time between problem re-distribution phases. The shared matrices can be declared `producer-consumer`, which requires that the sharing behavior be stable, and `PhaseChange()` can then be invoked whenever the sharing relationships change.

`ChangeAnnotation()` modifies the expected sharing pattern of a variable and hence the protocol used to keep it consistent. This lets the system adapt to dynamic changes in the way a particular object is accessed. Since the sharing pattern of an object is an indication to the system of the consistency protocol that should be used to maintain consistency, the invocation of `ChangeAnnotation()` may require the system to perform some immediate work to bring the current state of the object up-to-date with its new sharing pattern.

`Invalidate()` deletes the local copy of an object, and migrates it elsewhere if it is the sole copy or updates remote copies with any changes that may have occurred.

`Flush()` advises Munin to flush any buffered writes immediately rather than waiting for a release.

`SingleObject()` advises Munin to treat a large (multi-page) variable as a single object rather than breaking it into smaller page-sized objects.

Finally, `PreAcquire()` advises Munin to acquire a local copy of a particular object in anticipation of future use, thus avoiding the latency caused by subsequent read misses.

3 Implementation

3.1 Overview

Munin executes a distributed directory-based cache consistency protocol [1] in software, in which each directory entry corresponds to a single object. Munin also implements locks and barriers, using a distributed queue-based synchronization protocol [20, 26].

During compilation, the sharing annotations are read by the Munin preprocessor, and an auxiliary file is created for each input file. These auxiliary files are used by the linker to create a shared data segment and a shared data description table, which are appended to the Munin executable file. The program is then linked with the Munin runtime library.

When the application program is invoked, the Munin root thread starts running. It initializes the shared data segment, creates Munin worker threads to handle consistency and synchronization functions, and registers itself with the kernel as the address space's page fault handler (as is done by Mach's external pagers [28]). It then executes the user initialization routine `user_init()`, spawns the number of remote copies of the program specified by `user_init()`, and initializes the remote shared data segments. Finally, the Munin root thread creates and runs the user root thread. The user root thread in turn creates user threads on the remote nodes.

Whenever a user thread has an access miss or executes a synchronization operation, the Munin root thread is invoked. The Munin root thread may call on one of the local Munin worker threads or a remote Munin root thread to perform the necessary operations. Afterwards, it resumes the user thread.

3.2 Data Object Directory

The data object directory within each Munin node maintains information about the state of the global shared memory. This directory is a hash table that maps an address in the shared address space to the entry that describes the object located at that address. The data object directory on the Munin root node is initialized from the shared data description table found in the executable file, whereas the data object directory on the other nodes is initially empty. When Munin cannot find an object directory entry in the local hash table, it requests a copy from the object's home node, which for statically defined objects is the root node. Object directory entries contain the following fields:

- **Start address** and **Size**: used as the key for looking up the object's directory entry in a hash table, given an address within the object.

- **Protocol parameter bits**: represent the protocol parameters described in Section 2.3.1.

- **Object state bits**: characterize the dynamic state of the object, e.g., whether the local copy is *valid*, *writable*, or *modified* since the last flush, and whether a *remote copy* of the object exists.

- **Copyset**: used to specify which remote processors have copies of the object that must be updated or invalidated. For a small system, such as our prototype, a bitmap of the remote processors is sufficient.[3]

- **Synchq** (optional): a pointer to the synchronization object that controls access to the object (see Section 2.4).

- **Probable owner** (optional): used as a "best guess" to reduce the overhead of determining the identity of the Munin node that currently owns the object [24]. The identity of the owner node is used by the ownership-based protocols (`migratory`, `conventional`, and `reduction`), and is also used when an object is locked in place (`reduction`) or when the changes to the object should be flushed only to its owner (`result`).

[3]This approach does not scale well to larger systems, but an earlier study of parallel programs suggests that a processor list is often quite short [29]. The common exception to this rule occurs when an object is shared by every processor, and a special `All_Nodes` value can be used to indicate this case.

- **Home node** (optional): the node at which the object was created. It is used for a few record keeping functions and as the node of last resort if the system ever attempts to invalidate all remote copies of an object.
- **Access control semaphore**: provides mutually exclusive access to the object's directory entry.
- **Links**: used for hashing and enqueueing the object's directory entry.

3.3 Delayed Update Queue

The *delayed update queue* (DUQ) is used to buffer pending outgoing write operations as part of Munin's software implementation of release consistency. A write to an object that allows delayed updates, as specified by the protocol parameter bits, is stored in the DUQ. The DUQ is flushed whenever a local thread releases a lock or arrives at a barrier.

Munin uses the virtual memory hardware to detect and enqueue changes to objects. Initially, and after each time that the DUQ is flushed, the shared objects handled by the DUQ are write-protected using the virtual memory hardware. When a thread first attempts to write to such an object, the resulting protection fault invokes Munin. The object's directory entry is put on the DUQ, write-protection is removed so that subsequent writes do not experience consistency overhead, and the faulting thread is resumed. If multiple writers are allowed on the object, a copy (*twin*) of the object is also made. This twin is used to determine which words within the object have been modified since the last update.

When a thread releases a lock or reaches a barrier, the modifications to the objects enqueued on the DUQ are propagated to their remote copies.[4] The set of remote copies is either immediately available in the *Copyset* in the data object directory, or it must be dynamically determined. The algorithm that we currently use to dynamically determine the *Copyset* is somewhat inefficient. We have devised, but not yet implemented, an improved algorithm that uses the owner node to collect *Copyset* information. Currently, a message indicating which objects have been modified locally is sent to all other nodes. Each node replies with a message indicating the subset of these objects for which it has a copy. If the protocol parameters indicate that the sharing relationship is stable, this determination is performed only once.

If an enqueued object does not have a twin (i.e., multiple writers are not allowed), Munin transmits updates or invalidations to nodes with remote copies, as indicated by the invalidate protocol parameter bit in the object's directory entry. If the object does have a twin, Munin performs a word-by-word comparison of the object and its twin, and run-length encodes the results of this "diff" into the space allocated for the twin. Each run consists of a count of identical words, the number of differing words that follow, and the data associated with those differing words. The encoded object is sent to the nodes that require updates, where the object is decoded and the changes merged into the original object. If a Munin node with a dirty copy of an object receives an update for that object, it incorporates the changes immediately. If a Munin node with a dirty copy of an object receives an invalidation request for that object and multiple writers are allowed, any pending local updates are propagated. Otherwise, a runtime error is generated.

This approach works well when there are multiple writes to an object between DUQ flushes, which allows the expense of the copy and subsequent comparison to be amortized over a large number of write accesses. Table 2 breaks down the time to handle updates to an 8-kilobyte object through the DUQ. This includes the time to handle a fault (including resuming the thread),

[4]This is a conservative implementation of release consistency, because the updates are propagated at the time of the

Component	One Word	All Words	Alternate Words
Handle Fault	2.01	2.01	2.01
Copy object	1.15	1.15	1.15
Encode object	3.07	4.79	6.57
Transmit object	1.72	12.47	12.47
Decode object	3.12	4.86	6.68
Reply	2.27	2.27	2.27
Total	13.34	27.55	31.15

Table 2 Time to Handle an 8-kilobyte Object through DUQ (msec.)

make a copy of the object, encode changes to the object, transmit them to a single remote node, decode them remotely, and reply to the original sender. We present the results for three different modification patterns. In the first pattern, a single word within the object has changed. In the second, every word in the object has changed. In the third, every other word has changed, which is the worst case for our run-length encoding scheme because there are a maximum number of minimum-length runs.

We considered and rejected two other approaches for implementing release consistency in software:

1. Force the thread to page fault on every write to a replicated object so that the modified words can be queued as they are accessed.

2. Have the compiler add code to log writes to replicated objects as part of the write.

The first approach works well if an object is only modified a small number of times between DUQ flushes, or if the page fault handling code can be made extremely fast. Since it is quite common for an object to be updated multiple times between DUQ flushes [5], the added overhead of handling multiple page faults makes this approach generally unacceptable. The second approach was used successfully by the Emerald system [9]. We chose not to explore this approach in the prototype because we have a relatively fast page fault handler, and we did not want to modify the compiler. This approach is an attractive alternative for systems that do not support fast page fault handling or modification of virtual memory mappings, such as the iPSC-i860 hypercube [12]. However, if the number of writes to a particular object between DUQ flushes is high, as is often the case [5], this approach will perform relatively poorly because each write to a shared object will be slowed. We intend to study this approach more closely in future system implementations.

3.4 Synchronization Support

Synchronization objects are accessed in a fundamentally different way than data objects [5], so Munin does *not* provide synchronization through shared memory. Rather, each Munin node interacts with the other nodes to provide a high-level synchronization service. Munin provides support

release, rather than being delayed until the release is performed (see Section 2.2).

for distributed locks and barriers. More elaborate synchronization objects, such as monitors and atomic integers, can be built using these basic mechanisms.

Munin employs a queue-based implementation of locks, which allows a thread to request ownership of a lock and then to block awaiting a reply without repeated queries. Munin uses a synchronization object directory, analogous to the data object directory, to maintain information about the state of the synchronization objects. For each lock, a queue identifies the user threads waiting for the lock, so a release-acquire pair can be performed with a single message exchange if the acquire is pending when the release occurs. The queue itself is distributed to improve scalability.

When a thread wants to acquire a lock, it calls `AcquireLock()`, which invokes Munin to find the lock in the synchronization object directory. If the lock is local and free, the thread immediately acquires the lock and continues executing. If the lock is not free, Munin sends a request to the probable lock owner to find the actual owner, possibly requiring the request to be forwarded multiple times. When the request arrives at the owner node, ownership is forwarded directly to the requester if the lock is free. Otherwise, the owner forwards the request to the thread at the end of the queue, which puts the requesting thread on the lock's queue. Each enqueued thread knows only the identity of the thread that follows it on the queue. When a thread performs an `Unlock()` and the associated queue is non-empty, lock ownership is forwarded directly to the thread at the head of the queue.

To wait at a barrier, a thread calls `WaitAtBarrier()`, which causes a message to be sent to the owner node, and the thread to be blocked awaiting a reply. When the Munin root thread on the owner node has received messages from the specified number of threads, it replies to the blocked threads, causing them to be resumed. For future implementations on larger systems, we envision the use of barrier trees and other more scalable schemes [21].

4 Performance

We have measured the performance of two Munin programs, Matrix Multiply and Successive Over-Relaxation (SOR). We have also hand-coded the same programs on the same hardware using the underlying message passing primitives. We have taken special care to ensure that the actual computational components of both versions of each program are identical. This section describes in detail the actions of the Munin runtime system during the execution of these two programs, and reports the performance of both versions of these programs. Both programs make use of the DUQ to mitigate the effects of false sharing and thus improve performance. They also exploit Munin's multiple consistency protocols to reduce the consistency maintenance overhead.

4.1 Matrix Multiply

The shared variables in Matrix Multiply are declared as follows:

```
shared read_only int input1[N][N];
shared read_only int input2[N][N];
shared result int output[N][N];
```

The `user_init()` routine initializes the input matrices and creates a barrier via a call to `CreateBarrier()`. After creating worker threads, the user root thread waits on the barrier by calling `WaitAtBarrier()`. Each worker thread computes a portion of the output matrix using a standard (sub)matrix multiply routine. When a worker thread completes its computation, it performs a `WaitAtBarrier()` call. After all workers reach the barrier, the program terminates. The program does not utilize any of the optimizations described in Section 2.4.

On the root node, the input matrices are mapped as read-only, based on the `read-only` annotation, and the output matrix is mapped as read-write, based on the `result` annotation. When a worker thread first accesses an input matrix, the resulting page fault is handled by the Munin root thread on that node. It acquires a copy of the object from the root node, maps it as read-only, and resumes the faulted thread. Similarly, when a remote worker thread first writes to the output matrix, a page fault occurs. A copy of that page of the output matrix is then obtained from the root node, and the copy is mapped as read-write. Since the output matrix is a `result` object, there may be multiple writers, and updates may be delayed. Thus, Munin makes a twin, and inserts the output matrix's object descriptor in the DUQ. When a worker thread completes its computation and performs a `WaitAtBarrier()` call, Munin flushes the DUQ. Since the output matrix is a `result` object, Munin sends the modifications only to the owner (the node where the root thread is executing), and invalidates the local copy.

Table 3 gives the execution times of both the Munin and the message passing implementations for multiplying two 400×400 matrices. The System time represents the time spent executing Munin code on the root node, while the User time is that spent executing user code. In all cases, the performance of the Munin version was within 10% of that of the hand-coded message passing version. Program execution times for representative numbers of processors are shown. The program behaved similarly for all numbers of processors from one to sixteen.

Since different portions of the output matrix are modified concurrently by different worker threads, there is false sharing of the output matrix. Munin's provision for multiple writers reduces the adverse effects of this false sharing. As a result, the data motion exhibited by the Munin version of Matrix Multiply is nearly identical to that exhibited by the message passing version. In the Munin version, after the workers have acquired their input data, they execute independently, without communication, as in the message passing version. Furthermore, the various parts of the output matrix are sent from the node where they are computed to the root node, again as in the message passing version. The only difference between the two versions is that in Munin the appropriate parts of the input matrices are paged in, while in the message passing version they are sent during initialization. The additional overhead present in the Munin version comes from the page fault handling and the copying, encoding, and decoding of the output matrix. In a DSM system that does not support multiple writers to an object, portions of the output matrix could "ping-pong" between worker threads.

The performance of Matrix Multiply can be optimized by utilizing one of the performance optimizations discussed in Section 2.4. If Munin is told to treat the first input array as a single object rather than breaking it into smaller page-sized objects, via a call to `SingleObject()`, the

# of	DM	Munin			%
Procs	Total	Total	System	User	Diff
1	753.15	—	—	—	—
2	378.74	382.15	1.21	376.24	0.9
4	192.21	196.92	2.45	191.26	2.5
8	101.57	105.73	5.82	97.31	4.1
16	66.31	72.41	8.51	54.19	9.2

Table 3 Performance of Matrix Multiply(sec.)

entire input array is transmitted to each worker thread when the array is first accessed. Overhead is lowered by reducing the number of access misses. This improves the performance of the Munin version to within 2% of of the hand-coded message passing version. Execution times reflecting this optimization are shown in Table 4.

4.2 Successive Over-Relaxation

SOR is used to model natural phenomena. An example of an SOR application is determining the temperature gradients over a square area, given the temperature values at the area boundaries. The basic SOR algorithm is iterative. The area is divided into a grid of points, represented by a matrix, at the desired level of granularity. Each matrix element corresponds to a grid point. During each iteration, each matrix element is updated to be the average of its four nearest neighbors. To avoid overwriting the old value of a matrix element before it is used, the program can either use a scratch array or only compute every other element per iteration (so-called "red-black" SOR). Both techniques work equally well under Munin. Our example employs the scratch array approach.

SOR is parallelized by dividing the area into sections and having a worker thread compute the values for each section. Newly computed values at the section boundaries must be exchanged with adjacent sections at the end of each iteration. This exchange engenders a producer-consumer relationship between grid points at the boundaries of adjacent sections.

In the Munin version of SOR, the user root thread creates a worker thread for each section. The matrix representing the grid is annotated as

```
shared producer_consumer int matrix [...]
```

The programmer is *not* required to specify the data partitioning to the runtime system. After each iteration, worker threads synchronize by waiting at a barrier. After all workers have completed all iterations, the program terminates. The Munin version of SOR did not utilize any of the optimizations described in Section 2.4.

A detailed analysis of the execution, which exemplifies how producer-consumer sharing is currently handled by Munin, follows:

- During the first compute phase, when the new average of the neighbors is computed in the scratch array, the nodes read-fault in copies of the pages of the matrix as needed.
- During the first copy phase, when the newly computed values are copied to the matrix, nodes write-fault, enqueue the appropriate pages on the DUQ, create twins of these pages, make the originals read-write, and resume.

# of	DM	Munin			%
Procs	Total	Total	System	User	Diff
1	753.15	—	—	—	—
2	378.74	380.51	0.34	376.16	0.5
4	192.21	194.27	0.57	190.15	1.1
8	101.57	102.84	0.87	97.21	1.3
16	66.31	67.21	1.26	54.18	1.4

Table 4 Performance of Optimized Matrix Multiply (sec.)

- When the first copy phase ends and the worker thread waits at the barrier, the sharing relationships between producer and consumer are determined as described in Section 3.3. Afterwards, any pages that have an empty *Copyset*, and are therefore private, are made locally writable, their twins are deleted, and they do not generate further access faults. In our SOR example, all non-edge elements of each section are handled in this manner.

- Since the sharing relationships of producer-consumer objects are stable, after all subsequent copy phases, updates to shared portions of the matrix (the edge elements of each section) are propagated only to those nodes that require the updated data (those nodes handling adjacent sections). At each subsequent synchronization point, the update mechanism automatically combines the elements destined for the same node into a single message.

Table 5 gives the execution times of both the Munin and the message passing implementation of 100 iterations of SOR on a 512×512 matrix, for representative numbers of processors. In all cases, the performance of the Munin version was within 10% of that of the hand-coded message passing version. Again, the program behaved similarly for all numbers of processors from one to sixteen.

Since the matrix elements that each thread accesses overlap with the elements that its neighboring threads access, the sharing is very fine-grained and there is considerable false sharing. After the first pass, which involves an extra phase to determine the sharing relationships, the data motion in the Munin version of SOR is essentially identical to the message passing implementation. The only extra overhead comes from the fault handling and from the copying, coding, and decoding of the shared portions of the matrix.

4.3 Effect of Multiple Protocols

We studied the importance of having multiple protocols by comparing the performance of the multi-protocol implementation with the performance of an implementation using only `conventional` or only `write-shared` objects. `Conventional` objects result in an ownership-based write-invalidate protocol being used, similar to the one implemented in Ivy [24]. We also chose `write-shared` because it supports multiple writers and fine-grained sharing.

The execution times for the unoptimized version of Matrix Multiply (see Table 4) and SOR, for the previous problem sizes and for 16 processors, are presented in Table 6. For Matrix Multiply, the use of `result` and `read_only` sped up the time required to load the input matrices and later purge the output matrix back to the root node and resulted in a 4.4% performance improvement

# of	DM	Munin			%
Procs	Total	Total	System	User	Diff
1	122.62	—	—	—	—
2	63.68	66.87	2.27	61.83	5.5
4	36.46	39.70	3.58	31.77	8.9
8	26.72	28.26	5.17	16.57	5.8
16	25.62	27.64	8.32	8.51	7.9

Table 5 Performance of SOR (sec.)

Protocol	Matrix Multiply	SOR
Multiple	72.41	27.64
Write-shared	75.59	64.48
Conventional	75.85	67.64

Table 6 Effect of Multiple Protocols (sec.)

over `write-shared` and a 4.8% performance improvement over `conventional`. For SOR, the use of `producer-consumer` reduced the consistency overhead, by removing the phase in which sharing relationships are determined for all but the first iteration. The resulting execution time was less than half that of the implementations using only `conventional` or `write-shared`. The execution time for SOR using `write-shared` can be improved by using an better algorithm for determining the *Copyset* (see Section 3.3).

4.4 Summary

For Matrix Multiply, after initialization, each worker thread transmits only a single result message back to the root node, which is the same communication pattern found in a hand-coded message passing version of the program. For SOR, there is only one message exchange between adjacent sections per iteration (after the first iteration), again, just as in the message passing version.

The common problem of false sharing of large objects (or pages), which can hamper the performance of DSM systems, is relatively benign under Munin because we do *not* enforce a single-writer restriction on objects that do not require it. Thus, intertwined access regions and non-page-aligned data are less of a problem in Munin than with other DSM systems. The overhead introduced by Munin in both Matrix Multiply and SOR, other than the determination of the sharing relationships after the first iteration of SOR, comes from the expense of encoding and decoding modified objects.

By adding only minor annotations to the shared memory programs, the resulting Munin programs executed almost as efficiently as the corresponding message passing versions. In fact, during our initial testing, the performance of the Munin programs was *better* than the performance of the message passing versions. Only after careful tuning of the message passing versions were we able to generate message passing programs that resulted in the performance data presented here. This anecdote emphasizes the difficulty of writing efficient message passing programs, and serves to emphasize the value of a DSM system like Munin.

5 Related Work

A number of software DSM systems have been developed [3, 8, 10, 17, 25, 27]. All, except Midway [8], use sequential consistency [22]. Munin's use of release consistency only requires consistency to be enforced at specific synchronization points, with the resulting reduction in latency and number of messages exchanged.

Ivy uses a single-writer, write-invalidate protocol, with virtual memory pages as the units of consistency [25]. The large size of the consistency unit makes the system prone to false sharing. In addition, the single-writer nature of the protocol can cause a "ping-pong" behavior between multiple writers of a shared page. It is then 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.

Clouds performs consistency management on a per-object basis, or in Clouds terminology, on a per-segment basis [27]. Clouds allows a segment to be locked by a processor, to avoid the "ping-pong" effects that may result from false sharing. Mirage also attempts to avoid these effects by locking a page with a certain processor for a certain Δ time window [17]. Munin's use of multiple-writer protocols avoids the adverse effects of false sharing, without introducing the delays caused by locking a segment to a processor.

Orca is also an object-oriented DSM system, but its consistency management is based on an efficient reliable ordered broadcast protocol [3]. For reasons of scalability, Munin does not rely on broadcast. In Orca, both invalidate and update protocols can be used. Munin also supports a wider variety of protocols.

Unlike the designs discussed above, in Amber the programmer is responsible for the distribution of data among processors [10]. The system does not attempt to automatically move or replicate data. Good speedups are reported for SOR running on Amber. Munin automates many aspects of data distribution, and still remains efficient by asking the programmer to specify the expected access patterns for shared data variables.

Linda provides a different abstraction for distributed memory programming: all shared variables reside in a tuple space, and the only operations allowed are atomic insertion, removal, and reading of objects from the tuple space [18]. Munin stays closer to the more familiar shared memory programming model, hopefully improving its acceptance with parallel programmers.

Midway [8] proposes a DSM system with *entry consistency*, a memory consistency model weaker than release consistency. The goal of Midway is to minimize communications costs by aggressively exploiting the relationship between shared variables and the synchronization objects that protect them.

Recently, designs for hardware distributed shared memory machines have been published [2, 23]. Our work is most related to the DASH project [23], from which we adapt the concept of release consistency. Unlike Munin, though, DASH uses a write-invalidate protocol for all consistency maintenance. Munin uses the flexibility of its software implementation to also attack the problem of read misses by allowing multiple writers to a single shared object and by using update protocols (`producer-consumer`, `write-shared`, `result`) and pre-invalidation (`migratory`) when appropriate. The APRIL machine takes a different approach in combatting the latency problem on distributed shared memory machines [2]. APRIL provides sequential consistency, but relies on extremely fast processor switching to overlap memory latency with computation.

A technique similar to the delayed update queue was used by the Myrias SPS multiprocessor [13]. It performed the *copy-on-write* and *diff* in hardware, but required a restricted form of parallelism to ensure correctness.

Munin's implementation of locks is similar to existing implementations on shared memory multiprocessors [20, 26].

An alternative approach for parallel processing on distributed memory machines is to have the compiler produce a message passing program starting from a sequential program, annotated by the programmer with data partitions [4, 30]. Given the static nature of compile-time analysis, these techniques appear to be restricted to numerical computations with statically defined shared memory access patterns.

6 Conclusions and Future Work

The objective of the Munin project is to build a DSM system in which shared memory parallel programs execute on a distributed memory machine and achieve good performance without the programmer having to make extensive modifications to the shared memory program. Munin's shared memory is different from "real" shared memory only in that it provides a release-consistent memory interface, and in that the shared variables are annotated with their expected access patterns. In the applications that we have programmed in Munin so far, the release-consistent memory interface has required no changes, while the annotations have proved to be only a minor chore. Munin programming has been easier than message passing programming. Nevertheless, we have achieved performance within 5–10 percent of message passing implementations of the same applications. We argue that this cost in performance is a small price to pay for the resulting reduction in program complexity.

Further work on Munin will continue to examine the tradeoff between performance and programming complexity. We are interested in examining whether memory consistency can be relaxed further, without necessitating more program modifications than release consistency. We are also considering more aggressive implementation techniques, such as the use of a *pending updates queue* to hold incoming updates, a dual to the delayed update queue already in use. We also wish to design higher-level interfaces to distributed shared memory in which the access patterns will be determined without user annotation. Another important issue is Munin's scalability in terms of processor speed, interconnect bandwidth, and number of processors. To explore this issue, we intend to implement Munin on suitable hardware platforms such as a Touchstone-class machine or a high-speed network of supercomputer workstations. In this vein, we are also studying hardware support for selected features of Munin.

Acknowledgements

Dave Johnson provided important guidance during the prototype implementation and his feedback throughout the project was very useful. Peter Ostrin developed and timed the distributed memory programs reported in this paper.

The authors wish to thank the anonymous referees for their many helpful comments. Brian Bershad, Elmootazbellah Elnozahy, and Matt Zekauskas also provided many suggestions for improving the paper.

References

[1] 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.

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

[4] V. Balasundaram, G. Fox, K. Kennedy, and U. Kremer. An interactive environment for data partitioning and distribution. In *Proceedings of the Fifth Distributed Memory Computing Conference*, Charleston, South Carolina, April 1990.

[5] 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.

[6] J.K. Bennett, J.B. Carter, and W. Zwaenepoel. Munin: Distributed shared memory based on type–specific memory coherence. In *Proceedings of the 1990 Conference on the Principles and Practice of Parallel Programming*, March 1990.

[7] 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.

[8] B.N. Bershad and M.J. Zekauskas. Shared memory parallel programming with entry consistency for distributed memory multiprocessors. Technical Report CMU-CS-91-170, Carnegie-Mellon University, September 1991.

[9] A. Black, N. Hutchinson, E. Jul, and H. Levy. Object structure in the Emerald system. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 78–86, October 1986.

[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 Twelfth 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 Ninth ACM Symposium on Operating Systems Principles*, pages 129–140, October 1983.

[12] Intel Corporation. i860 64-bit microprocessor programmer's manual. Santa Clara, California, 1990.

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

[14] 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.

[15] S.J. Eggers and R.H. Katz. The effect of sharing on the cache and bus performance of parallel programs. In *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Systems*, pages 257–270, April 1989.

[16] K. Gharachorloo et al. Performance evaluations of memory consistency models for shared-memory multiprocessors. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Systems*, April 1991.

[17] B.D. Fleisch and G.J. Popek. Mirage: A coherent distributed shared memory design. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, pages 211–23, December 1989.

[18] D. Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, January 1985.

[19] 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.

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

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

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

[23] 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.

[24] K. Li. *Shared Virtual Memory on Loosely Coupled Multiprocessors.* PhD thesis, Yale University, September 1986.

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

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

[27] U. Ramachandran, M. Ahamad, and M.Y. 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.

[28] R. Rashid, A. Tevanian, Jr. M. Young, D. Golub, R. Baron, D. Black, W. Bolosky, and J. Chew. Machine-independent virtual memory management for paged uniprocessor and multiprocessor architectures. *IEEE Transactions on Computers*, 37(8):896–908, August 1988.

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

[30] H.P. Zima, H.J. Bast, and M. Gerndt. Superb: A tool for semi-automatic SIMD/MIMD parallelization. *Parallel Computing*, 6:1–18, 1988.